# A Precise Type Analysis of Logic Programs

Lunjin Lu
Department of Computer Science and Engineering
The Oakland University
Rochester, MI 48309

## ABSTRACT

This paper presents a new type analysis for logic programs. The type information in a set of substitutions is described by a disjunction of variable typings each of which maps a variable to a non-deterministic regular type. The use of non-deterministic regular types, set union and intersection operators, and disjunctions of variable typings makes the new type analysis more precise than those found in the literature. Experimental results on the performance of the new analysis are given together with comparable results from an existing type analysis. The fundamental problem of checking the emptiness of non-deterministic regular types is more complex in the new analysis. The experimental results, however, show that careful use of tabling reduces the effect to an average of 15% of execution time on a set of benchmarks.

## 1. INTRODUCTION

Types play an important role in programming languages. They make programs easier to understand and help detect errors. Types have been introduced into logic programming in the forms of type checking [14, 36, 39, 9, 11, 33, 40] and type analysis [31, 41, 15, 16, 12, 37, 26, 5, 27]. The reader is referred to [35] for a collection of papers on types in logic programming.

### 1.1 Related Work

Type analyses infer types from a program. Most type analyses infer descriptive types[12, 15, 16, 31, 40, 41, 7]. A descriptive type is a description of a set of terms by an expression in a type language. Descriptive types are useful in compiler optimizations. On the other hand, they are not intuitive and difficult for the programmer to understand. There have been research into prescriptive type analysis [18, 21, 1, 22, 4, 26, 37, 5, 27, 23] where types of function symbols are provided by the programmer.

Horiuchi and Kanamori [18, 21], Kanamori and Kawamura [22], and Lu [26, 27] infer prescriptive types of Prolog programs using top-down abstract interpretation frameworks. Barbuti and Giacobazzi [1] infer polymorphic types of Horn clause logic programs using a bottom-up abstract interpretation technique. Codish and Demoen [4] apply abstract compilation [17] to infer type dependencies by associating each type with an incarnation of the abstract domain Prop [29]. The incarnations of Prop define meanings of types and capture interactions between types. Codish and Lagoon [5] improve [4] by augmenting abstract compilation with ACI-unification. An associative, commutative and idempotent operator $\oplus$ is introduced to form the type of a term from the types of its subterms. Saglam and Gallagher [37] infer a combination of prescriptive and descriptive types by extending [12] that infers descriptive types by approximating a logic program with a unary regular logic program [40]. Levi and Spoto [23] use Heyting completion to construct abstract domains for type analysis.

### 1.2 Contributions

Existing prescriptive type analyses are not precise for two reasons. Firstly, the types they infer are deterministic. This is obvious for [21, 18, 1, 22, 6, 26, 27] where a function symbol can only be typed by exactly one type rule. (Type rules will be formally introduced in section 3.) During abstract compilation of [4, 5], each concrete term is compiled into a type term. Specifically, a complex term $f(t_1, \cdots, t_n)$ is compiled into $type(f(t_1, \cdots, t_n)) = \alpha(f)(\oplus_{j \notin \pi(f)} type(t_j)) \oplus \oplus_{i \in \pi(f)} type(t_i)$ where $\alpha$ is a function from the set of function symbols to the set of type constructors and $\pi(f) \subseteq \{1, \cdots, n\}$. The abstract function $\alpha$ prevents a function symbol from being overloaded. Therefore, types in [4, 5] are deterministic. Saglam and Gallagher [37] infer types in two stages - type reduction and regular approximation. The type reduction stage uses type definitions provided by the programmer to reduce terms in the program into type terms. The fact that this process is determinate implies that prescriptive types in [37] are deterministic although descriptive types in [37] are non-deterministic. Levi and Spoto [23] do not explicitly state if they allow non-deterministic types. The examples they give in [23] involve only deterministic types. The restriction to deterministic type rules allows fast computation of the type of a term from those of its variables since at each level there is at most one applicable type rule. However, it causes imprecision in the analysis because of the limited power of the type language. The same restriction also prevents many natural typings. For instance, these two type rules $float ::= +(integer, float)$ and $float ::= +(float, float)$ violate the restriction.

$append([\,], L, L).$
$append([H|L_1], L_2, [H|L_3]) \quad \leftarrow append(L_1, L_2, L_3).$ ⓑ

$\leftarrow$ ⓐ append$(Xs, Ys, Zs)$.ⓒ

**Figure 1: Logic program** *append*

Secondly, the type languages in existing prescriptive type analyses lack set union and set intersection operators. This is obvious for [21, 18, 1, 22, 6, 26, 27, 23]. The $\oplus$ operator in [5] is not equivalent to the set union operator as $\oplus$ is assumed to be distributive over type constructors, for instance, $list(atom \oplus float) = list(atom) \oplus list(float)$. The set union of two types is approximated by a supertype of the two types in [12, 37]. The following example illustrates how lack of set operators affects analysis precision. Let ⓐ $: Q \rightsquigarrow$ ⓑ $: R$ denote that if $Q$ holds at the program point ⓐ then $R$ holds whenever the execution reaches the program point ⓑ. Let $X \mapsto R$ stand for "$X$ is of type $R$".

EXAMPLE 1.1. *Take the program and the initial goal in Figure 1 as an example. Let type constructors nat and list() be defined as usual (see Example 3.1). The analyses presented in [3, 18, 19, 21, 22, 1, 4, 26, 27] are not able to infer*

$$\text{ⓐ} : \{Xs \mapsto list(nat),\ Ys \mapsto list(list(nat))\}$$
$$\rightsquigarrow \quad \text{ⓑ} : \{Zs \mapsto list(nat \sqcup list(nat))\}$$

*where $\sqcup$ is interpreted as set union. Rather, they infer* ⓐ $: \{Xs \mapsto list(nat), Ys \mapsto list(list(nat))\} \rightsquigarrow$ ⓑ $: \{Zs \mapsto list(\mathbf{1})\}$ *where $\mathbf{1}$ is the universal type that denotes the set of all terms. Lack of set union operator in type expressions forces an over approximation when the least upper bound of types is needed. The analysis in [5] infers* ⓐ $: \{Xs \mapsto list(nat), Ys \mapsto list(list(nat))\} \rightsquigarrow$ ⓑ $: \{Zs \mapsto list(nat \oplus list(nat))\}$ *and the analysis in [37] infers* ⓐ $: \{Xs \mapsto list(nat), Ys \mapsto list(list(nat))\} \rightsquigarrow$ ⓑ $: \{Zs \mapsto list(nat\ union\ list(nat))\}$. *Note that $\oplus$ and "union" are not set union, they are approximations to set union.* ∎

This paper presents a prescriptive type analysis that infers non-deterministic regular types. [1] It produces a more precise result than existing prescriptive type analyses. The contributions of our work are as follows. Firstly, we design a new type analysis for logic programs. For the sake of analysis precision, we choose a type language that consists of regular types and allows set union and set intersection operators. We describe type information in a set of substitutions by a set of variable typings that are mappings from variables to types. Set operators allow us to express the results of type analyses without adding type rules. Adding type rules compromises the benefit of a prescriptive type analysis because added type rules may not be comprehensible to the programmer. Secondly, we observe that the number of different regular types whose emptiness must be checked during an analysis is small while the total number of their occurrences is high. We give experimental results with a prototype implementation. It shows that tabling [38] reduces the time increase of the new type analysis with respect to a deterministic type analysis to 15%.

[1] We shall omit the word "non-deterministic" hereafter.

## 1.3 Outline of the paper

The remainder of the paper is organized as follows. In section 2, we briefly describe the abstract interpretation framework in which we present the new type analysis. Section 3 is devoted to types, their definitions and denotations. Section 4 presents the new type analysis. In section 5, we present a prototype implementation of the new type analysis and some experimental results. Section 6 concludes. The appendix contains proofs and auxiliary lemmas.

# 2. PRELIMINARIES

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

## 2.1 Notation

Let $\mathsf{func}$ be a set of *function symbols*, $\mathsf{pred}$ a set of *predicate symbols* and $\mathcal{V}$ a denumerable set of variables. Let $\mathcal{U} \subseteq \mathcal{V}$. $\mathsf{Term}(\mathsf{func}, \mathcal{U})$ denotes the set of *terms* that can be constructed from $\mathsf{func}$ and $\mathcal{U}$. $\mathsf{Atom}(\mathsf{func}, \mathsf{pred}, \mathcal{U})$ is the set of *atoms* that is constructible from $\mathsf{pred}$ and $\mathsf{Term}(\mathsf{func}, \mathcal{U})$. $\mathsf{Term} \stackrel{def}{=} \mathsf{Term}(\mathsf{func}, \mathcal{V})$ and $\mathsf{Atom} \stackrel{def}{=} \mathsf{Atom}(\mathsf{pred}, \mathsf{func}, \mathcal{V})$. Let $\theta$ be a *substitution*. $dom(\theta)$ denotes the domain of $\theta$. $\theta\restriction\mathcal{U}$ denotes the restriction of $\theta$ to $\mathcal{U}$.

An *equation* is a formula of the form $l = r$ where either $l, r \in \mathsf{Term}$ or $l, r \in \mathsf{Atom}$. The set of all equations is denoted as $\mathsf{Eqn}$. For a set of equations $E \in \wp(\mathsf{Eqn})$, $mgu : \wp(\mathsf{Eqn}) \mapsto Sub \cup \{fail\}$ returns either a most general unifier for $E$ if $E$ is unifiable or $fail$ otherwise, where $Sub$ is the set of idempotent substitutions. $mgu(\{l = r\})$ is sometimes written as $mgu(l, r)$. $\circ$ is the function composition operator defined as $f \circ g \stackrel{def}{=} \lambda x.f(g(x))$. Let $\theta \circ fail \stackrel{def}{=} fail$ and $fail \circ \theta \stackrel{def}{=} fail$ for any $\theta \in Sub \cup \{fail\}$. $eq(\theta)$ denotes the set of equations in solved form corresponding to $\theta$. $eq(fail) \stackrel{def}{=} fail$.

We will use a fixed renaming substitution $\Psi$ which is a one-to-one mapping from $\mathcal{V}$ to $\mathcal{V}'$ - another denumerable set of variables that is disjoint with $\mathcal{V}$. $\Psi$ is called a tagging substitution [32]. Let $\mathcal{V}_P$ be the set of program variables of interest. $\mathcal{V}_P$ is usually the set of variables in the program. We sometimes use Church's lambda notation for functions, so that a function $f$ will be denoted by $\lambda x.f(x)$.

## 2.2 Abstract Interpretation Framework

The abstract interpretation framework in [25] is based on a collecting semantics of normal logic programs that is defined in terms of two operators on $\langle\wp(Sub), \subseteq\rangle$. One operator is the set union $\cup$ and the other is $UNIFY$ defined as follows. Let $a_1, a_2 \in \mathsf{Atom}$ and $\Theta_1, \Theta_2 \in \wp(Sub)$.

$$\begin{aligned} & UNIFY(a_1, \Theta_1, a_2, \Theta_2) \\ \stackrel{def}{=} \;& \{unify(a_1, \theta_1, a_2, \theta_2) \neq fail \mid \theta_1 \in \Theta_1 \wedge \theta_2 \in \Theta_2\} \end{aligned}$$

where $unify(a_1, \theta_1, a_2, \theta_2) \stackrel{def}{=} mgu((\Psi(\theta_1))(\Psi(a_1)), \theta_2(a_2)) \circ \theta_2$.

Specializing the framework for a program analysis consists in designing an abstract domain $\langle ASub, \sqsubseteq\rangle$, a concretization function $\gamma : ASub \mapsto \wp(Sub)$, and an abstract operator $AUNIFY$ such that, for any $a_1, a_2 \in \mathsf{Atom}(\mathsf{func}, \mathsf{pred}, \mathcal{V}_P)$ and any $\pi_1, \pi_2 \in ASub$,

$$UNIFY(a_1, \gamma(\pi_1), a_2, \gamma(\pi_2)) \subseteq \gamma(AUNIFY(a_1, \pi_1, a_2, \pi_2))$$

since we always have $\gamma(\pi_1) \cup \gamma(\pi_2) \subseteq \gamma(\pi_1 \sqcup \pi_2)$ where $\sqcup$ is the least upper bound operator on $\langle ASub, \sqsubseteq\rangle$.

## 3. TYPES

The type language in a type system decides which sets of terms are types. To be useful, the set of types should be closed under set intersection, set union and set complement operations. The decision problems such as the emptiness of a type, inclusion of a type in another, and equivalence of two types should be decidable. Regular term languages [13], also called regular types, satisfy these conditions and have been widely used as types.

A type is represented by a ground term constructed from a ranked alphabet Cons and $\{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}$, called type constructors. It is assumed that $(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}) \cap \mathsf{func} = \emptyset$. Thus, a type expression is a term in $\mathsf{Term}(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}, \emptyset)$. The denotations of type constructors in Cons are determined by type definitions whilst $\sqcap, \sqcup, \mathbf{1}$ and $\mathbf{0}$ have fixed denotations. Let $\mathsf{RT} \stackrel{def}{=} \mathsf{Term}(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}, \emptyset)$.

Several equivalent formalisms such as tree automata [13], regular term grammars [13, 10], and regular unary logic programs [40] have been used to define regular types. We define types by type rules [10]. A type parameter is a variable that ranges over RT. A type rule is of the form $c(\beta_1, \cdots, \beta_m) ::= f(\tau_1, \cdots, \tau_n)$ where $c \in \mathsf{Cons}, f \in \mathsf{func}$, $\beta_1, \cdots, \beta_m$ are different type parameters, and each $\tau_j$, for $1 \leq j \leq n$, is either in $\mathsf{Para}_m$ with $\mathsf{Para}_m = \{\beta_1, \cdots, \beta_m\}$ or of the form $d(\beta'_1, \cdots, \beta'_k)$ with $\beta'_1, \cdots, \beta'_k$ being different type parameters in $\mathsf{Para}_m$ and $d \in \mathsf{Cons}$. A type parameter may appear in $\tau_i$ and $\tau_j$ for different $1 \leq i, j \leq n$. The restriction on the form of $\tau_i$ can be relaxed by allowing $\tau_i$ to be an arbitrary term in $\mathsf{Term}(\mathsf{Cons}, \mathsf{Para}_m)$. However, this would not increase the power of the type definition formalism, as shown in [10]. The restriction that every type parameter in the righthand side of a type rule must occur in the lefthand side has been used in all type definition formalisms. Note that overloading of function symbols is permitted as a function symbol can appear in the righthand sides of many type rules. We denote by $\Delta$ the set of all type rules and define $\mathsf{Para} \stackrel{def}{=} \bigcup_{c \in \mathsf{Cons}} \mathsf{Para}_{arity(c)}$. Type rules are similar to type definitions used in typed logic programming languages Mercury and Gödel. A poly-type is a term in $\mathsf{Term}(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}, \mathsf{Para})$. Note that a poly-type may contain type parameters.

EXAMPLE 3.1. *Let* $\mathsf{func} = \{0, s(), nil, cons(,)\}$, $\mathsf{Cons} = \{nat, even, odd, list()\}$. *The* $\Delta$ *below defines natural numbers, even numbers, odd numbers, and lists.*

$$\Delta = \left\{ \begin{array}{ll} nat ::= 0, & nat ::= s(nat), \\ even ::= 0, & even ::= s(odd), \\ odd ::= s(even), & \\ list(\beta) ::= nil, & \\ list(\beta) ::= cons(\beta, list(\beta)) & \end{array} \right\}$$

∎

A type valuation is a set $\mathcal{K} = \{\beta_1 \mapsto R_1, \cdots, \beta_n \mapsto R_n\}$ where $\beta_i \in \mathsf{Para}$ and $R_i \in \mathsf{RT}$ and $\beta_i \neq \beta_j$ for $i \neq j$. $dom(\mathcal{K}) = \{\beta_1, \cdots, \beta_n\}$ is the domain of $\mathcal{K}$. The application of $\mathcal{K}$ to a poly-type is to replace simultaneously each occurrence of $\beta_i$ in that poly-type with $R_i$. For instance, $\{\beta_1 \mapsto list(nat), \beta_2 \mapsto nat\}(list(\beta_1)) = list(list(nat))$. $\mathcal{K}(\beta) \stackrel{def}{=} \mathbf{0}$ for any $\beta \notin dom(\mathcal{K})$. Let TVal denote the set of all type valuations.

Given $\Delta$, the set of terms denoted by a type is defined as follows.

$$\begin{array}{rcl} [\![\mathbf{1}]\!]_\Delta & \stackrel{def}{=} & \mathsf{Term} \\ [\![\mathbf{0}]\!]_\Delta & \stackrel{def}{=} & \emptyset \\ [\![R_1 \sqcap R_2]\!]_\Delta & \stackrel{def}{=} & [\![R_1]\!]_\Delta \cap [\![R_2]\!]_\Delta \\ [\![R_1 \sqcup R_2]\!]_\Delta & \stackrel{def}{=} & [\![R_1]\!]_\Delta \cup [\![R_2]\!]_\Delta \\ [\![c(R_1, \cdots, R_m)]\!]_\Delta & \stackrel{def}{=} & \end{array}$$

$$\left\{ f(t_1, \cdots, t_n) \;\middle|\; \begin{array}{l} \exists (c(\beta_1, \cdots, \beta_m) ::= f(\tau_1, \cdots, \tau_n)) \in \Delta. \\ \forall 1 \leq i \leq n. \\ t_i \in [\![\{\beta_j \mapsto R_j \mid 1 \leq j \leq m\}(\tau_i)]\!]_\Delta \end{array} \right\}$$

$[\![\cdot]\!]_\Delta$ gives fixed denotations to $\sqcap, \sqcup, \mathbf{1}$ and $\mathbf{0}$. $\sqcap$ and $\sqcup$ are interpreted by $[\![\cdot]\!]_\Delta$ as set intersection and set union respectively. $\mathbf{1}$ denotes Term and $\mathbf{0}$ the empty set. *Every type in* RT *denotes a regular term language* [28].

EXAMPLE 3.2. *Let* $\Delta$ *be that in Example 3.1. Then*

$$[\![nat]\!]_\Delta = \{0, s(0), s(s(0)), \cdots\}$$

$$[\![list(\mathbf{0})]\!]_\Delta = \{nil\}$$

$$[\![list(\mathbf{1})]\!]_\Delta = \{nil, cons(x, nil), \cdots\}$$

*where* $x \in \mathcal{V}$. ∎

A type is atomic if its main constructor is in $\mathsf{Cons} \cup \{\mathbf{1}, \mathbf{0}\}$. A type is conjunctive if it is of the form $\sqcap_{1 \leq i \leq k} A_j$ with each $A_j$ being atomic. Two types $R_1$ and $R_2$ are equivalent iff $[\![R_1]\!]_\Delta = [\![R_2]\!]_\Delta$. It is obvious that any type $R$ is equivalent to another type $\sqcup_{1 \leq i \leq m} C_i$ with each $C_i$ being a conjunctive type. We call $\sqcup_{1 \leq i \leq m} C_i$ a disjunctive normal form of $R$ and denote $\sqcup_{1 \leq i \leq m} C_i$ by DNF($R$).

## 4. NEW TYPE ANALYSIS

This section presents the new type analysis in terms of a domain of abstract substitutions and an abstract unification operator required by the abstract interpretation framework in [25]. It can be easily adapted to other abstract interpretation frameworks of logic programs such as those in [30, 34, 2, 20].

## 4.1 Abstract Domain

A set of substitutions has been described in existing prescriptive type analyses [18, 21, 22, 26, 27] by a variable typing that maps each variable in $\mathcal{V}_P$ into a type. The denotation of a variable typing is given by $\gamma_{\mathsf{VT}} : (\mathcal{V}_P \mapsto \mathsf{RT}) \mapsto \wp(Sub)$ as follows.

$$\gamma_{\mathsf{VT}}(\mu) \stackrel{def}{=} \{\theta \mid \forall x \in \mathcal{V}_P.(\theta(x) \in [\![\mu(x)]\!]_\triangle)\}$$

For instance, $\gamma_{\mathsf{VT}}(\{x \mapsto nat, y \mapsto list(nat)\}) = \{\theta \mid \theta(x) \in [\![nat]\!]_\triangle \wedge \theta(y) \in [\![list(nat)]\!]_\triangle\}$. Using single variable typing as an abstract substitution is a source of imprecision in those existing prescriptive type analyses that are based on top-down abstract interpretation frameworks[18, 21, 22, 26, 27]. We intend to describe a set of substitutions using a set of variable typings that is interpreted as the set union of the denotations of the variable typings.

EXAMPLE 4.1. *For instance, letting* $\mathcal{S} = \{\{x \mapsto nat, y \mapsto list(nat)\}, \{x \mapsto list(nat), y \mapsto nat\}\}$, $\mathcal{S}$ *denotes* $\{\theta \mid \theta(x) \in [\![nat]\!]_\triangle \wedge \theta(y) \in [\![list(nat)]\!]_\triangle\} \cup \{\sigma \mid \sigma(x) \in [\![list(nat)]\!]_\triangle \wedge \sigma(y) \in [\![nat]\!]_\triangle\}$. ∎

There may be many sets of variable typings that denote the same set of substitution. Let $\mathcal{S} = \{\{x \mapsto list(\mathbf{1})\}, \{x \mapsto list(nat)\}\}$. $\mathcal{S}$ has the same denotation as one of its proper subset $\mathcal{S}' = \{\{x \mapsto list(\mathbf{1})\}\}$. We follow the normal practice in program analysis of identifying those abstract elements that have the same denotation. Let $\preceq$ on $\wp(\mathcal{V}_P \mapsto \mathsf{RT})$ be defined as $\mathcal{S}_1 \preceq \mathcal{S}_2 \stackrel{def}{=} (\bigcup_{\mu \in \mathcal{S}_1} \gamma_{\mathsf{VT}}(\mu)) \subseteq (\bigcup_{\nu \in \mathcal{S}_2} \gamma_{\mathsf{VT}}(\nu))$. Define $\approx$ on $\wp(\mathcal{V}_P \mapsto \mathsf{RT})$ by $(\mathcal{S}_1 \approx \mathcal{S}_2) \stackrel{def}{=} (\mathcal{S}_1 \preceq \mathcal{S}_2) \wedge (\mathcal{S}_2 \preceq \mathcal{S}_1)$. $\approx$ is an equivalence relation on $\wp(\mathcal{V}_P \mapsto \mathsf{RT})$ induced by $\preceq$. The equivalence classes with respect to $\approx$ are abstract substitutions in the new type analysis. Thus, the abstract domain is $\langle \mathsf{DVT}, \tilde{\sqsubseteq} \rangle$ where

$$\mathsf{DVT} \stackrel{def}{=} \wp(\mathcal{V}_P \mapsto \mathsf{RT})_{/\approx} \qquad \text{and} \qquad \tilde{\sqsubseteq} \stackrel{def}{=} \preceq_{/\approx}$$

$\langle \mathsf{DVT}, \tilde{\sqsubseteq} \rangle$ is a complete lattice. We use $[\mathcal{S}]$ to denote the equivalence class of $\mathcal{S}$ with respect to $\approx$. The least upper bound operator and the greatest lower bound operator are respectively $[\mathcal{S}_1] \tilde{\cup} [\mathcal{S}_2] = [\mathcal{S}_1 \cup \mathcal{S}_2]$ and $[\mathcal{S}_1] \tilde{\cap} [\mathcal{S}_2] = [\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow}]$ where $\mathcal{S}_i^{\downarrow} \stackrel{def}{=} \{\mu \mid (\mu \in (\mathcal{V}_P \mapsto \mathsf{RT})) \wedge \exists \nu \in \mathcal{S}_i.(\gamma_{\mathsf{VT}}(\mu) \subseteq \gamma_{\mathsf{VT}}(\nu))\}$. The infimum is $[\emptyset]$ and the supremum $[\{x \mapsto \mathbf{1} \mid x \in \mathcal{V}_P\}]$.

The concretization function $\gamma_{\mathsf{DVT}} : \mathsf{DVT} \mapsto \wp(Sub)$ is defined as follows.

$$\gamma_{\mathsf{DVT}}([\mathcal{S}]) \stackrel{def}{=} \bigcup_{\mu \in \mathcal{S}} \gamma_{\mathsf{VT}}(\mu)$$

LEMMA 4.2. $\gamma_{\mathsf{DVT}}(\mathsf{DVT})$ *is a Moore family.* ∎

The abstract domain is not simply a disjunctive completion of that of any existing prescriptive type analysis because the underlying domain of types is more powerful than those used in any existing prescriptive type analysis.

## 4.2 Outline of Abstract Unification

The abstract unification for the new type analysis closely simulates concrete unification. The abstract unification is reduced to computing a set of variable typings that describes the set of those substitutions that satisfy both a set of equations in solved form and an initial set of variable typings. This is done by propagating type information in the initial set of variable typings via the equations in two steps. The first step uses type information about the variables on the lefthand sides of the equations to derive more type information about the variables occurring in the terms on the righthand sides. We call this step downward propagation as it involves recursively computing types of the top-level subterms of a term from a type of the term. The second step uses type information about the variables occurring in the terms on the righthand sides to derive more type information about the variables on the lefthand sides. We call the second step upward propagation as it involves recursively computing a type of a term from types of its top-level subterms.

## 4.3 Ancillary Operators

We now define some ancillary operators that will be used to define the abstract unification operator. Let $\mathcal{V}_P^+ = \mathcal{V}_P \cup \Psi(\mathcal{V}_P)$. Many of ancillary operators will work with variable typings that map variables in $\mathcal{V}_P^+$ to regular types in $\mathsf{RT}$. $\approx$ and $\gamma_{\mathsf{DVT}}$ extend naturally to variable typings on $\mathcal{V}_P^+$.

### 4.3.1 *Greatest lower bound of two sets of variable typings*

The definition of $\tilde{\cap}$ is not constructive in that the downward closure of a set of variable typings $\mathcal{S}$ can be of infinite size. For instance, letting $\mathcal{S} = \{\{x \mapsto list(\mathbf{1})\}\}$, $\{x \mapsto list^k(nat)\}$ for any $k \geq 1$ is in $\mathcal{S}^{\downarrow}$. We now define an operator that, given representatives of two equivalence classes with respect to $\approx$, computes a representative of the greatest lower bound of the two equivalence classes. Let $\mathcal{S}_1, \mathcal{S}_2 \in \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT})$. Define

$$\begin{aligned} & \mathcal{S}_1 \bigotimes \mathcal{S}_2 \\ = \; & \{\{x \mapsto (\mu(x) \sqcap \nu(x)) \mid x \in \mathcal{V}_P^+\} \mid \mu \in \mathcal{S}_1 \wedge \nu \in \mathcal{S}_2\} \end{aligned}$$

Note that $\sqcap$ is a type constructor.

LEMMA 4.3. $\gamma_{\mathsf{DVT}}([\mathcal{S}_1 \bigotimes \mathcal{S}_2]) = \gamma_{\mathsf{DVT}}([\mathcal{S}_1] \tilde{\cap} [\mathcal{S}_2])$.

### 4.3.2 *Downward Propagation*

The function $down : \wp(\mathsf{Eqn}) \times \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT}) \mapsto \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT})$ defined below accomplishes the downward propagation step. For each variable typing $\mu$ in $\mathcal{S}$ and each equation $x = t$ in $E$, $down(E, \mathcal{S})$ computes a set of variable typings from $\mu(x)$ and $t$ and conjoins it with $\{\mu\}$ using $\bigotimes$. Its main functionality is done by $vts\_cv(R, t)$ that, given a type $R$ and a term $t$, computes a set of variable typings $\mathcal{S}'$ such that $\{\theta \mid \theta(t) \in [\![R]\!]_\triangle\} \subseteq \gamma_{\mathsf{DVT}}([\mathcal{S}'])$.

$$down(E, \mathcal{S}) \stackrel{def}{=} \bigcup_{\mu \in \mathcal{S}} (\{\mu\} \otimes \bigotimes_{(x=t) \in E} vts\_cv(\mu(x), t)) \qquad (1)$$

Let $Id^+ = \lambda x \in \mathcal{V}_P^+.\mathbf{1}$. Then $vts\_cv(R,t)$ is defined as follows.

$$
\begin{array}{rcl}
vts\_cv(R_1 \sqcap R_2, t) & \stackrel{def}{=} & vts\_cv(R_1,t) \bigotimes vts\_cv(R_2,t) \\
vts\_cv(R_1 \sqcup R_2, t) & \stackrel{def}{=} & vts\_cv(R_1,t) \cup vts\_cv(R_2,t) \\
vts\_cv(A,x) & \stackrel{def}{=} & \{Id^+[x \mapsto A]\} \\
vts\_cv(\mathbf{1},t) & \stackrel{def}{=} & \{Id^+\}
\end{array}
$$

$$
vts\_cv(c(R_1,\cdots,R_m), f(t_1,\cdots,t_n)) \stackrel{def}{=}
\bigcup_{(c(\beta_1,\cdots,\beta_m)::=f(\tau_1,\cdots,\tau_n))\in\Delta} \left( \begin{array}{l} let\ \mathcal{K}=\{\beta_j \mapsto R_j \mid 1\le j\le m\} \\ in \\ \bigotimes_{1\le i\le n} vts\_cv(\mathcal{K}(\tau_i),t_i) \end{array} \right)
$$

The first two definitions of $vts\_cv(R,t)$ deal with set operators and the last three deal with atomic types. The third and the fourth definitions are straightforward. In the fifth definition, for each type rule $c(\beta_1,\cdots,\beta_m)::=f(\tau_1,\cdots,\tau_n)$ in $\Delta$ with $c/m$ and $f/n$, a set of variable typings $\bigotimes_{1\le i\le n} vts\_cv(\mathcal{K}(\tau_i),t_i)$ is computed. If there is no type rule for $c/m$ and $f/n$, $f(t_1,\cdots,t_n)$ can not be of type $R$ and an empty set of variable typings is returned. If there are more than one type rules for the fixed $c/m$ and $f/n$, the results from these type rules are combined together by $\bigcup$.

Example 4.4. *Let $\mathcal{V}_P^+ = \{x,y\}$, $\mathcal{S} = \{\{x \mapsto \mathbf{1}, y \mapsto (list(nat)\sqcup nat)\}\}$ and $\Delta$ be that in Example 3.1. We have $vts\_cv(list(nat), cons(x,nil)) = \{\{x \mapsto nat, y \mapsto \mathbf{1}\}\}$ and $vts\_cv(nat, cons(x,nil)) = \emptyset$. So,*

$$vts\_cv(list(nat)\sqcup nat, cons(x,nil)) = \{\{x \mapsto nat, y \mapsto \mathbf{1}\}\}$$

*and*

$$
\begin{array}{l}
down(\{y = cons(x,nil)\}, \mathcal{S}) \\
\quad = \left\{ \left\{ \begin{array}{c} x \mapsto \mathbf{1}, \\ y \mapsto (list(nat)\sqcup nat) \end{array} \right\} \right\} \otimes \{\{x \mapsto nat, y \mapsto \mathbf{1}\}\} \\
\quad = \{\mu\}
\end{array}
$$

*where $\mu = \{x \mapsto nat, y \mapsto (list(nat)\sqcup nat)\}$.* ∎

### 4.3.3 Computing a covering type valuation

We introduce a special type valuation $\bowtie$ and define $\bowtie(\tau) \stackrel{def}{=} \mathbf{1}$ for any poly-type. A poly-type is a general type if it is either in $\mathsf{Para}$ or of the form $d(\beta_1,\cdots,\beta_k)$ with $\beta_1,\cdots,\beta_k$ being different type parameters in $\mathsf{Para}$ and $d \in \mathsf{Cons}$. The type of $f(t_1,\cdots,t_n)$ will be computed from those of $t_1,\cdots,t_n$ by applying the type rules for $f/n$. The role of $\bowtie$ is to indicate that types of $t_1,\cdots,t_n$ do not satisfy type conditions in a type rule, and the type of $f(t_1,\cdots,t_n)$ from the type rule is $\mathbf{1}$ in this case. Let $\mathcal{K}_1,\mathcal{K}_2 \in \mathsf{TVal}$. Define

$$
\mathcal{K}_1 \mathring{\vee} \mathcal{K}_2 \stackrel{def}{=} \left\{ \begin{array}{ll} \bowtie & if\ (\mathcal{K}_1 = \bowtie) \vee (\mathcal{K}_2 = \bowtie); \\ \{\beta \mapsto \mathcal{K}_1(\beta)\sqcup\mathcal{K}_2(\beta) \mid \beta \in dom(\mathcal{K}_1)\cup dom(\mathcal{K}_2)\} & otherwise. \end{array} \right.
$$

$$
\mathcal{K}_1 \mathring{\wedge} \mathcal{K}_2 \stackrel{def}{=} \left\{ \begin{array}{ll} \mathcal{K}_2 & if\ (\mathcal{K}_1 = \bowtie); \\ \mathcal{K}_1 & if\ (\mathcal{K}_2 = \bowtie); \\ \{\beta \mapsto \mathcal{K}_1(\beta)\sqcap\mathcal{K}_2(\beta) \mid \beta \in dom(\mathcal{K}_1)\cap dom(\mathcal{K}_2)\} & otherwise. \end{array} \right.
$$

Given a conjunctive type $C$ and a general type $\tau$, $tv\_cv(C,\tau)$ is a type valuation $\mathcal{K}$ such that $[\![C]\!]_\Delta \subseteq [\![\mathcal{K}(\tau)]\!]_\Delta$.

$$
\begin{array}{lcl}
tv\_cv(\sqcap_{1\le i\le k} A_i, \tau) & \stackrel{def}{=} & \mathring{\bigwedge}_{1\le i\le k} tv\_cv(A_i,\tau) \\
tv\_cv(A,\beta) & \stackrel{def}{=} & \{\beta \mapsto A\}
\end{array}
$$

$$
tv\_cv(c(R_1,\cdots,R_m), d(\beta_1,\cdots,\beta_k)) \stackrel{def}{=} \left\{ \begin{array}{ll} \{\beta_j \mapsto R_j \mid 1\le j\le m\} & if\ c/m = d/k; \\ \bowtie & otherwise. \end{array} \right.
$$

### 4.3.4 Computing a type of a term

Given conjunctive types $C_1,\cdots,C_n$, $ctype(f,\langle C_1,\cdots,C_n\rangle)$ is a conjunctive type that contains the terms $f(s_1,\cdots,s_n)$ for all $s_i \in [\![C_i]\!]_\Delta$. $ctype(f,\langle C_1,\cdots,C_n\rangle)$ is computed as follows. For each type rule $\tau::=f(\tau_1,\cdots,\tau_n)$, a type valuation $\mathcal{K} = \mathring{\bigvee}_{1\le i\le n} tv\_cv(C_i,\tau_i)$ is computed. $\mathcal{K}(\tau)$ is an atomic type that contains the terms $f(s_1,\cdots,s_n)$ for all $s_i \in [\![C_i]\!]_\Delta$. The intersection of the atomic types $\mathcal{K}(\tau)$ from all the type rules $\tau::=f(\tau_1,\cdots,\tau_n)$ for $f$ also contains the terms $f(s_1,\cdots,s_n)$ for all $s_i \in [\![C_i]\!]_\Delta$ and defines $ctype(f,\langle C_1,\cdots,C_n\rangle)$.

$$
ctype(f,\langle C_1,C_2,\cdots,C_n\rangle) \stackrel{def}{=}
\sqcap_{(\tau::=f(\tau_1,\cdots,\tau_n))\in\Delta} \left( \begin{array}{l} let\ \mathcal{K} = \mathring{\bigvee}_{1\le i\le n} tv\_cv(C_i,\tau_i) \\ in \\ \mathcal{K}(\tau) \end{array} \right)
$$

### 4.3.5 Upward Propagation

The function $up : \wp(\mathsf{Eqn}) \times \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT}) \mapsto \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT})$ performs the upward propagation step. The core functionality of $up$ is performed by $type$ that, for a given term $t$ and a variable typing $\mu$, computes a type $R$ such that $\theta(t) \in [\![R]\!]_\Delta$ for all $\theta \in \gamma_{\mathsf{VT}}(\mu)$.

$$
up(E,\mathcal{S}) \stackrel{def}{=}
\bigcup_{\mu\in\mathcal{S}} \left\{ \lambda x \in \mathcal{V}_P^+. \left( \begin{array}{l} if\ \exists t.(x=t)\in E \\ then\ \mu(x)\sqcap type(t,\mu) \\ else\ \mu(x) \end{array} \right) \right\}
$$

The definition of $type(t,\mu)$ below ensures that $type(t,\mu)$ is in disjunctive normal form. If $t$ is a variable then $type(t,\mu)$ is the disjunctive normal form of $\mu(t)$. Otherwise, $t = f(t_1,\cdots,t_n)$ for some $n \ge 0$. $type(t_i,\mu) = \sqcup_{1\le j\le k_i} C_{i,j}$ for each $1 \le i \le n$ is computed. Then, for each tuple $C_{1,j_1},\cdots,C_{n,j_n}$ with $1 \le j_i \le k_i$, a conjunctive type $ctype(f,\langle C_{1,j_1},\cdots,C_{n,j_n}\rangle)$ is computed. These conjunctive types are then combined together by $\sqcup$. Let $R$ be a type in disjunctive normal form and $C$ a conjunctive type. $C \hat{\in} R$ denotes that $C$ is a conjunct in $R$.

$$
\begin{array}{lcl}
type(x,\mu) & \stackrel{def}{=} & \mathrm{DNF}(\mu(x)) \\
type(f(t_1,\cdots,t_n),\mu) & \stackrel{def}{=} & \\
\multicolumn{3}{l}{\quad \sqcup_{C_1 \hat{\in} type(t_1,\mu),\cdots,C_n \hat{\in} type(t_n,\mu)} ctype(f,\langle C_1,\cdots,C_n\rangle)}
\end{array}
$$

EXAMPLE 4.5. *Let $\mathcal{S}$ and $\mu$ be those in Example 4.4. By the definition of tv_cv, $\mathcal{K}_1 = tv\_cv(nat, \beta) = \{\beta \mapsto nat\}$ and $\mathcal{K}_2 = tv\_cv(list(\mathbf{0}), list(\beta)) = \{\beta \mapsto \mathbf{0}\}$. By the definitions of ctype and type, we have, $type(x, \mu) = nat$ and $type(nil, \mu) = list(\mathbf{0})$. So,*

$$\begin{aligned} & type(cons(x, nil), \mu) \\ = \; & ctype(cons, \langle nat, list(\mathbf{0})\rangle) \\ = \; & (\mathcal{K}_1 \overset{\circ}{\vee} \mathcal{K}_2)(list(\beta)) \\ = \; & list(nat) \end{aligned}$$

*By the definition of up,*

$$\begin{aligned} & up(\{y = cons(x, nil)\}, \{\mu\}) \\ = \; & \{\mu[y \mapsto (list(nat) \sqcup nat) \sqcap list(nat)]\} \\ \approx \; & \{x \mapsto nat, y \mapsto list(nat)\} \end{aligned}$$

∎

## 4.4 Abstract Unification

Algorithm 4.6 defines the abstract unification operator for the new type analysis. Given $[\mathcal{S}_1], [\mathcal{S}_2] \in \mathsf{DVT}$ and $a_1, a_2 \in \mathsf{Atom}(\mathsf{func}, \mathsf{pred}, \mathcal{V}_P)$, it computes $AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2])$ as follows. $\Psi$ is first applied to $a_1$ and $\mathcal{S}_1$ to obtain $\Psi(a_1)$ and $\Psi(\mathcal{S}_1)$, and $E_0 = eq \circ mgu(\Psi(a_1), a_2)$ is then computed. If $E_0 = fail$ then the algorithm returns $[\emptyset]$ - the infimum of $\langle \mathsf{DVT}, \tilde{\subseteq}\rangle$. Otherwise, the algorithm continues. $\Psi(\mathcal{S}_1)$ and $\mathcal{S}_2$ are combined to obtain $\mathcal{S}_0' = \Psi(\mathcal{S}_1) \Cap \mathcal{S}_2$ so that a substitution satisfying $\mathcal{S}_0'$ satisfies both $\Psi(\mathcal{S}_1)$ and $\mathcal{S}_2$. Note that $\mathcal{S}_0' \in \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT})$. $solve(E_0, \mathcal{S}_0')$ is then called to compute a set $\mathcal{S}_1'$ of variable typings that describes the set of the substitutions satisfying $E_0$ and $\mathcal{S}_0'$ and finally each variable typing in $\mathcal{S}_1'$ is restricted to $\mathcal{V}_P$ by calling $rest(\mathcal{S}_1')$. $solve(E_0, \mathcal{S}_0')$ first performs the downward propagation step and then the upward propagation step.

ALGORITHM 4.6. *Let $a_1, a_2 \in \mathsf{Atom}(\mathsf{func}, \mathsf{pred}, \mathcal{V}_P)$ and $[\mathcal{S}_1], [\mathcal{S}_2] \in \mathsf{DVT}$.*

$$\begin{aligned} & AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2]) \\ & \qquad \stackrel{def}{=} \left\{ \begin{array}{ll} let & E_0 = eq \circ mgu(\Psi(a_1), a_2) \; in \\ if & E_0 \neq fail \\ then & [rest \circ solve(E_0, \Psi(\mathcal{S}_1) \Cap \mathcal{S}_2)] \\ else & [\emptyset] \end{array} \right. \\ & \mathcal{S}_1 \Cap \mathcal{S}_2 \stackrel{def}{=} \{\mu \cup \nu \mid \mu \in \mathcal{S}_1 \wedge \nu \in \mathcal{S}_2\} \\ & rest(\mathcal{S}) \stackrel{def}{=} \{\mu \uparrow \mathcal{V}_P \mid \mu \in \mathcal{S}\} \\ & solve(E, \mathcal{S}) \stackrel{def}{=} up(E, down(E, \mathcal{S})) \end{aligned}$$

EXAMPLE 4.7. *Let $\mathcal{V}_P = \{x\}$, $\Psi(x) = y$, $a_1 = p(x)$, $a_2 = p(cons(x, nil))$, $\mathcal{S}_1 = \{\{x \mapsto (list(nat) \sqcup nat)\}\}$, $\mathcal{S}_2 = \{\{x \mapsto \mathbf{1}\}\}$. Then $E_0 = \{y = cons(x, nil)\}$ and $\Psi(\mathcal{S}_1) \Cap \mathcal{S}_2 = \mathcal{S}$ with $\mathcal{S}$ being that in Example 4.4. By examples 4.4 and 4.5, $solve(E_0, \mathcal{S}) = up(E_0, down(E_0, \mathcal{S})) = up(E_0, \{\mu\}) = \{\{x \mapsto nat, y \mapsto list(nat)\}\}$ with $\mu$ being that in Example 4.5.* ∎

The following theorem states that $AUNIFY$ abstracts $UNIFY$ with respect to $\gamma_{\mathsf{DVT}}$, establishing the correctness of the new type analysis.

THEOREM 4.8. *For any $[\mathcal{S}_1], [\mathcal{S}_2] \in \mathsf{DVT}$ and any $a_1, a_2 \in \mathsf{Atom}(\mathsf{func}, \mathsf{pred}, \mathcal{V}_P)$,*

$$\begin{aligned} & UNIFY(a_1, \gamma_{\mathsf{DVT}}([\mathcal{S}_1]), a_2, \gamma_{\mathsf{DVT}}([\mathcal{S}_2])) \\ & \quad \subseteq \gamma_{\mathsf{DVT}}(AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2])) \end{aligned}$$

∎

# 5. IMPLEMENTATION

We have implemented the new type analysis and the abstract interpretation framework in SWI-Prolog. Both are implemented as meta-interpreters using ground representations for program variables. The implementation of new type analysis also supports the primitive types *integer*, *float*, *number*, *atom*, *string* and *atomic* with their usual denotations in Prolog. [2]

EXAMPLE 5.1. *The following is the* intersect *program that computes the intersection of two lists and the result of the new type analysis of the program. Abstract substitutions are displayed as comments. The abstract substitution at the program point immediately before the query is input whilst all other abstract substitutions are output. Sets are represented by lists. $V \mapsto T$ is written as $V/T$, $\sqcup$ as or, $\sqcap$ as and, and integer as int. The predicate* member *is omitted.*

```
:-  %[[X/list(atom or float),Y/list(atom or int)]]
  intersect(X,Y,Z).
    %[[X/list(atom or float),Y/list(atom or int),
    %  Z/list(atom)]]

intersect([],L,[]).
    %[[L/list(atom or int)]]
intersect([X|Xs],Ys,[X|Zs]) :-
    %[[X/atom,Xs/list(atom or float),
    %  Ys/list(atom or int)],
    % [X/float,Xs/list(atom or float),
    %  Ys/list(atom or int)]]
  member(X,Ys),
    %[[X/atom,Xs/list(atom or float),
    %  Ys/list(atom or int)]]
  intersect(Xs,Ys,Zs).
    %[[X/atom,Xs/list(atom or float),
    %  Ys/list(atom or int),
    %  Zs/list(atom)]]
intersect([X|Xs],Ys,Zs) :-
    %[[X/atom,Xs/list(atom or float),
    %  Ys/list(atom or int)],
    % [X/float,Xs/list(atom or float),
    %  Ys/list(atom or int)]]
  \+ member(X,Ys),
    %[[X/float,Xs/list(atom or float),
    %  Ys/list(atom or int)],
    % [X/atom,Xs/list(atom or float),
    %  Ys/list(atom or int)]]
  intersect(Xs,Ys,Zs).
    %[[X/float,Xs/list(atom or float),
    %  Ys/list(atom or int),
    %  Zs/list(atom)],
```

[2]The implementation can be obtained from http://www.cs.waikato.ac.nz/~lunjin/LPANE-1-0.zip

```
% [X/atom,Xs/list(atom or float),
%  Ys/list(atom or int),
%  Zs/list(atom)]]
```

*The result shows that the intersection of a list containing atoms and float numbers and another list containing atoms and integer numbers is a list of atoms. This is precise because the type* $(atom \sqcup float) \sqcap (atom \sqcup integer)$ *is equivalent to the type atom. Without the set operators* $\sqcap$ *and* $\sqcup$ *in their type languages, existing prescriptive type analyses [18, 21, 22, 1, 4, 5, 12, 37, 26, 27, 23] cannot produce a result as precise as the above.* ∎

In the remainder of this section, we extend the language of regular types so that decision problems in the new type analysis can be reduced to checking the emptiness of regular types in the extended language. We then show the effect of tabling the emptiness checks of regular types and compare the new type analysis with a deterministic type analysis in terms of accuracy and time performance.

## 5.1 Regular Types with Complement Operator

The language of regular types we used for our new type analysis does not contain the complement operator $\sim$. There are two reasons for not including $\sim$. Firstly, if the two input abstract substitutions of $AUNIFY$ do not contain $\sim$ then its output abstract substitution does not contain $\sim$ either. That is, abstract unification does not introduce $\sim$. Nor does the least upper bound operator $\tilde{\cup}$. Secondly, inclusion of $\sim$ in input abstract substitutions would make $vts\_cv$ much more complex. It would be necessary to approximate a type $\sim R$ with a type formed of atomic types, and $\sqcap$ and $\sqcup$ operators before applying type rules to recursively compute types of the top-level subterms of a term from a type of that term. This approximation may need be done recursively as $\sim$ may occur in $R$. However, the introduction of $\sim$ will enable us to reduce decision problems in the new type analysis to checking the emptiness of regular types. We thus restrict the use of $\sim$ so that it occurs only in types encountered in the decision problems.

The extended language of regular types is $\mathsf{Term}(\mathsf{Cons} \cup \{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\}, \emptyset)$ with $\sim$ being interpreted as follows.

$$[\![\sim R]\!]_\Delta \stackrel{def}{=} \mathsf{Term} - [\![R]\!]_\Delta$$

We have

$$\begin{aligned} &[\![list(nat \sqcap \sim even)]\!]_\Delta \\ &\quad = \{cons(s(0), nil), cons(s(s(s(0))), nil), \cdots\} \end{aligned}$$

for $\Delta$ in Example 3.1. We denote by $\mathsf{eRT}$ the extended language. It is obvious that $\mathsf{RT} \subseteq \mathsf{eRT}$. Let $R \in \mathsf{eRT}$ and define $etype(R) \stackrel{def}{=} ([\![R]\!]_\Delta = \emptyset)$. $etype(R)$ is true iff $R$ denotes the empty set of terms. It is well known that the emptiness of regular types is decidable [13].

The detection of the least fixpoint of the abstract semantics for the new type analysis and the detection of redundancy in a set of variable typings are both reduced to checking the emptiness of regular types in $\mathsf{eRT}$.

## 5.2 Tabling

The new type analysis is strictly more powerful than deterministic type analyses as it employs more powerful types. However, operations in the new type analysis are also more complex than those in deterministic type analyses. An abstract substitution in a deterministic type analysis is a variable typing where each variable is associated with a deterministic type. In contrast to this, an abstract substitution in the new type analysis is a set of variable typings mapping variables to non-deterministic regular types. Both deterministic type analyses and the new type analysis reduce operations on abstract substitutions to operations on types. The equality of two abstract substitutions in deterministic type analyses can be done in linear time [18, 21, 22, 26] while the same operation in the new type analysis takes exponential time because the decidability of the emptiness of a regular type takes exponential time [13]. This indicates that the new type analysis could be much more time consuming than deterministic type analyses.

As we show later, there is a high degree of repetition in emptiness checks during the analysis of a program. Making use of this observation, we have reduced time increase to 15% on average using a simple tabling technique. The algorithm for deciding the emptiness of a regular type given a set of type rules is described in [28]. $etype(R)$ is called with $R$ being a ground term representing a regular type. We memoize each call to $etype(R)$ and its success or failure by asserting a fact $\$etype\_tabled(R, Ans)$. $\$etype\_tabled(R, yes)$ (resp. $\$etype\_tabled(R, no)$) denotes the fact that $etype(R)$ has been called before and $etype(R)$ succeeded (resp. failed). The tabled version $etype\_tabled(R)$ of $etype(R)$ first checks if a fact $\$etype\_tabled(R, Ans)$ exists. If so, $etype\_tabled(R)$ succeeds or fails immediately. Otherwise, it calls $etype(R)$ and memoizes its success or failure.

## 5.3 Experimental Results

We now present some experimental results with the prototype implementation. The experiments were done with a 5x86 IBM compatible PC running Windows95 and SWI-Prolog-3.2.1.

### *5.3.1 Time Performance*

Table 1 shows the execution time of the new type analysis on a set of programs that are often used to benchmark analyses of logic programs. Each row except the last one corresponds to a test case which is specified by the first three columns. The first and the second columns contain the name and the size of the program. The third gives the goal. Its input abstract substitution is a singleton set consisting of the variable typing that maps each variable to $\mathbf{1}$. These test cases will be used in subsequent tables where only the program names are given. The fourth column gives analysis time in seconds for the test case. The time is obtained by running the analyzer ten times on the test case and averaging the times from these runs. Timing data in other tables are also obtained in this way.

$nat/0$ and $list/1$ are defined in Example 3.1, and $dictionary/2$ is defined by the following type rule.

$$dictionary(\alpha, \beta) ::= dict(\alpha, \beta)$$

**Table 1: Time Performance**

| Program | Lines | Goal | Time (sec) |
|---|---|---|---|
| chat | 883 | chat_parser | 18.16 |
| cs_r | 277 | pgenconfig(C) | 11.76 |
| dnf | 77 | go | 3.07 |
| nandc | 110 | play(R) | 4.55 |
| neural | 341 | go | 5.18 |
| read | 394 | read(X, Y) | 12.40 |
| ronp | 90 | puzzle(X) | 1.06 |
| semi | 161 | go(N, T) | 5.36 |
| SUM | 2333 | | 61.58 |

**Table 2: Repetition of Emptiness Checks**

| Program | Total Chks | Diff. Chks | Degree of Repetit'n | Check Time |
|---|---|---|---|---|
| chat | 23532 | 27 | 871.55 | 1.82 |
| cs_r | 25962 | 69 | 376.26 | 3.87 |
| dnf | 5823 | 8 | 727.87 | 0.42 |
| nandc | 38355 | 83 | 462.10 | 2.81 |
| neural | 24500 | 192 | 127.60 | 2.20 |
| read | 21200 | 67 | 316.41 | 2.37 |
| ronp | 3190 | 33 | 96.66 | 0.24 |
| semi | 16506 | 60 | 275.10 | 1.73 |
| AVE. | | | 406.69 | |

**Table 3: Effect of Tabulation**

| | With Tabling | | Without Tabling | |
|---|---|---|---|---|
| Program | Analy. | Chk | Analy. | Chk |
| chat | 18.16 | 1.82 | 26.55 | 8.73 |
| cs_r | 11.76 | 3.87 | 84.46 | 80.95 |
| dnf | 3.07 | 0.42 | 9.55 | 7.37 |
| nandc | 4.55 | 2.81 | 22.72 | 21.77 |
| neural | 5.18 | 2.20 | 43.09 | 41.67 |
| read | 12.40 | 2.37 | 24.63 | 14.18 |
| ronp | 1.06 | 0.24 | 2.72 | 1.72 |
| semi | 5.36 | 1.73 | 19.87 | 17.10 |
| SUM | 61.58 | 15.49 | 233.62 | 193.52 |

**Table 4: Comparison with Det. Type Analysis**

| Program | Det. | New | Ratio | Relative Accuracy |
|---|---|---|---|---|
| chat | 19.71 | 18.16 | 1.08 | yes |
| cs_r | 7.86 | 11.76 | 0.66 | yes |
| dnf | 2.91 | 3.07 | 0.94 | no |
| nandc | 1.30 | 4.55 | 0.28 | yes |
| neural | 3.21 | 5.18 | 0.61 | yes |
| read | 11.99 | 12.40 | 0.96 | yes |
| ronp | 0.91 | 1.06 | 0.86 | no |
| semi | 5.00 | 5.36 | 0.93 | yes |
| AVE. | | | 0.85 | |

The table shows that it takes an average of 0.026 seconds for the analyzer to process one line of the program. The table also shows that the average speed per program line does not increase as the size of the program increases.

### *5.3.2 Repetition of Emptiness Checks*

Table 2 shows that there is a high degree of repetition in emptiness checks that occur during the analysis of a program. Each test case corresponds to a row of the table. The first column of the row is the name of the program, the second column is the total number of emptiness checks that occur during the analysis of the test case. The third column gives the number of different regular types of which emptiness is checked for the test case. The fourth column gives the average degree of emptiness checks that is the ratio of the second and the third columns. The fifth column gives the time spent on emptiness checks. While the total number of emptiness checks can be very large for a test case, the number of different emptiness checks is small, implying a high degree of repetition in emptiness checks. Depending on the test case, the average repetition of the emptiness checks ranges from 96 to 871. The weighted repetition average is 406. This prompts the use of tabling to reduce the time on emptiness checks.

### *5.3.3 Effect of Tabulation*

Table 3 illustrates the effect of tabling in the new type analysis. It gives statistics for the new type analyzer with and without tabling. For both runs, we measured the analysis time and the time on emptiness checks. The table shows that tabling emptiness checks reduces the time spent on them by more than 12.49 times and the analysis time by 3.79 times.

### *5.3.4 Comparison with a Deterministic Type Analysis*

Table 4 compares the time performance of the new type analysis and a deterministic type analysis [21]. We have chosen to compare the new type analysis with [21] because both analyses are based on top-down abstract interpretation frameworks and it is therefore easy to automatically perform the comparison. The deterministic type analysis is done with an input abstract substitution that contains the same type information as that for the new type analysis. The input abstract substitution for the new type analysis in each test case is a singleton set of a variable typing. The corresponding abstract substitution for the deterministic type analysis is the variable typing. For each test case, the table gives the analysis time on the deterministic type analysis, the analysis time on the new type analysis and their ratio. It also indicates whether or not the new type analysis gives more precise result than the deterministic type analysis. The relative performance of the two analyzers varies with the test case. On average, the deterministic type analysis takes 85 percent of the analysis time of the new type analysis. This illustrates that using regular types rather than deterministic types does not substantially increase analysis time. We note that tabling subtyping checks would not increase efficiency of the deterministic type analysis because a subtyping check is a linear operation for the deterministic type analysis. Out of the 8 test cases, the new type analysis produces a more precise result than the deterministic type analysis for 6 test cases.

## 6. CONCLUSION

We have presented a new prescriptive type analysis. It is strictly more precise than those presented in the literature as it uses non-deterministic regular types, allows set union and set intersection in type expressions, and uses a set of variable typings to describe type information in a set of substitutions. A prototype implementation has shown that the new type analysis is more precise on actual programs with only a small increase in analysis time.

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

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

[4] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. *Lecture Notes in Computer Science*, 864:281–297, 1994.

[5] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, June 1996.

[6] M.M. Corsini and K. Musumbu. Type inference: a new approach. *Journal of Theoretical Computer Science*, 119(1):23–38, 1993.

[7] A. Cortesi, B. Le Charlier, and P. van Hentenryck. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–208, 1995.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth annual ACM symposium on Principles of programming languages*, pages 238–252. The ACM Press, 1977.

[9] P.W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *Journal of Logic Programming*, 14(1-2):31–69, 1992.

[10] 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.

[11] T. Fruhwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309. The IEEE Computer Society Press, 1991.

[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] F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, 1984.

[14] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Journal of Theoretical Computer Science*, 89(1):63–106, 1991.

[15] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the seventh Annual ACM Symposium on Principles of Programming Languages*, pages 197–209. The ACM Press, 1990.

[16] N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–155. The MIT Press, 1992.

[17] M. Hermenegildo, R. Warren, and S.K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(1, 2, 3 and 4):349–366, 1992.

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

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

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

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

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

[23] G. Levi and F. Spoto. An experiment in domain refinement: type domains and type representations for logic programs. *Lecture Notes in Computer Science*, 1490:152–169, 1998.

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

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

[26] L. Lu. Type analysis of logic programs in the presence of type definitions. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based program manipulation*, pages 241–252. The ACM Press, 1995.

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

[28] L. Lu and J. Cleary. An emptiness algorithm for regular types with set operators. Technical report, Department of Computer Science, The University of Waikato, November 1998. http://xxx.lanl.gov/ps.LO/cs/9811015.

[29] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. Ritter, editor, *Information Processing'89*. North-Holland, 1989.

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

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

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

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

[34] U. Nilsson. Towards a framework for the abstract interpretation of logic programs. In *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82. Springer-Verlag, 1988.

[35] F. Pfenning, editor. *Types in logic programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[36] U.S. Reddy. Types for logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 836–40. The MIT Press, 1990.

[37] H. Saglam and J.P. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-017, Department of Computer Science, University of Bristol, July 1995.

[38] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[39] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphically typed logic programs. In K. Furukawa, editor, *Logic Programming. Proceedings of the Eighth International Conference*, pages 379–93. The MIT Press, 1991.

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

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

# APPENDIX

## A. PROOFS

PROOF OF LEMMA 4.2: $\gamma_{\mathsf{DVT}}([\{x \mapsto \mathbf{1} \mid x \in \mathcal{V}_P\}]) = \wp(Sub)$. Let $[\mathcal{S}_1], [\mathcal{S}_2] \in \mathsf{DVT}$. Then $[\mathcal{S}_1]\tilde{\sqcap}[\mathcal{S}_2] \in \mathsf{DVT}$. Furthermore,

$$\begin{aligned}
\gamma_{\mathsf{DVT}}([\mathcal{S}_1]\tilde{\sqcap}[\mathcal{S}_2]) &= \gamma_{\mathsf{DVT}}([\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow}]) \\
&= (\bigcup_{\mu \in \mathcal{S}_1^{\downarrow}} \gamma_{\mathsf{VT}}(\mu)) \cap (\bigcup_{\nu \in \mathcal{S}_2^{\downarrow}} \gamma_{\mathsf{VT}}(\nu)) \\
&= (\bigcup_{\mu \in \mathcal{S}_1} \gamma_{\mathsf{VT}}(\mu)) \cap (\bigcup_{\nu \in \mathcal{S}_2} \gamma_{\mathsf{VT}}(\nu)) \\
&= \gamma_{\mathsf{DVT}}([\mathcal{S}_1]) \cap \gamma_{\mathsf{DVT}}([\mathcal{S}_2])
\end{aligned}$$

This completes the proof of the lemma. ∎

PROOF OF LEMMA 4.3: We first prove $\gamma_{\mathsf{DVT}}([\mathcal{S}_1 \bigotimes \mathcal{S}_2]) \subseteq \gamma_{\mathsf{DVT}}([\mathcal{S}_1]\tilde{\sqcap}[\mathcal{S}_2])$. Let $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}_1 \bigotimes \mathcal{S}_2])$. Then there is $\rho$ in $(\mathcal{S}_1 \bigotimes \mathcal{S}_2)$ such that $\theta \in \gamma_{\mathsf{VT}}(\rho)$. This implies that there are a $\mu$ in $\mathcal{S}_1$ and a $\nu \in \mathcal{S}_2$ such that $\rho = \lambda x \in \mathcal{V}_P.(\mu(x) \sqcap \nu(x))$. We have $\gamma_{\mathsf{VT}}(\rho) \subseteq \gamma_{\mathsf{VT}}(\mu)$ and $\gamma_{\mathsf{VT}}(\rho) \subseteq \gamma_{\mathsf{VT}}(\nu)$, implying $\rho \in \mathcal{S}_1^{\downarrow}$ and $\rho \in \mathcal{S}_2^{\downarrow}$. Therefore, $\rho \in (\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow})$. $\theta \in \gamma_{\mathsf{VT}}(\rho)$ implies that $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow}])$ and $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}_1]\tilde{\sqcap}[\mathcal{S}_2])$.

We now prove $\gamma_{\mathsf{DVT}}([\mathcal{S}_1 \bigotimes \mathcal{S}_2]) \supseteq \gamma_{\mathsf{DVT}}([\mathcal{S}_1]\tilde{\sqcap}[\mathcal{S}_2])$. Let $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}_1]\tilde{\sqcap}[\mathcal{S}_2])$. Then $\theta \in \gamma_{\mathsf{VT}}(\rho)$ for some $\rho \in (\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow})$ by the definition of $\tilde{\sqcap}$. There are a $\mu \in \mathcal{S}_1$ and a $\nu \in \mathcal{S}_2$ such that $\gamma_{\mathsf{VT}}(\rho) \subseteq \gamma_{\mathsf{VT}}(\mu)$ and $\gamma_{\mathsf{VT}}(\rho) \subseteq \gamma_{\mathsf{VT}}(\nu)$, implying $\forall x \in \mathcal{V}_P.(\theta(x) \in [\![\mu(x) \sqcap \nu(x)]\!]_{\Delta})$. Thus, $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}_1 \bigotimes \mathcal{S}_2])$ by the definition of $\bigotimes$. This completes the proof of the lemma. ∎

PROOF OF THEOREM 4.8: The proof of the theorem requires the proofs of the correctness of the downward propagation function *down* in lemma A.2 and the upward propagation function *up* in lemma A.6. Both lemmas A.2 and A.6 follow this proof.

Let $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}_1]$, $\sigma \in \gamma_{\mathsf{DVT}}([\mathcal{S}_2]$ Let $E_0 = eq \circ mgu(\Psi(a_1), a_2)$. We have $\zeta \in \gamma_{\mathsf{DVT}}([\mathcal{S}]) \leftrightarrow (\zeta \uparrow \mathcal{V}_P) \in \gamma_{\mathsf{DVT}}([rest(\mathcal{S})])$ for any $\zeta \in \mathsf{Sub}$ and $\mathcal{S} \in \wp(\mathcal{V}_P^{+} \mapsto \mathsf{RT})$. Therefore,

$$\begin{aligned}
& unify(a_1, \theta, a_2, \sigma) \in \gamma_{\mathsf{DVT}}(AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2])) \\
\leftrightarrow \quad & mgu(\{(\Psi(\theta))(\Psi(a_1)) = \sigma(a_2)\}) \circ \sigma \\
& \quad \in \gamma_{\mathsf{DVT}}(AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2]) \\
\leftrightarrow \quad & mgu(\eta(E_0)) \circ \eta \in \gamma_{\mathsf{DVT}}(AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2])) \\
& \text{with } \eta = \sigma \cup \Psi(\theta)
\end{aligned}$$

By lemmas A.2 and A.6,

$$mgu(\eta(E_0)) \circ \eta \in \gamma_{\mathsf{DVT}}([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \bigcap \mathcal{S}_2))])$$

since $\eta \in \gamma_{\mathsf{DVT}}(\Psi(\mathcal{S}_1) \bigcap \mathcal{S}_2)$. So,

$$unify(a_1, \theta, a_2, \sigma) \in \gamma_{\mathsf{DVT}}(AUNIFY(a_1, [\mathcal{S}_1], a_2, [\mathcal{S}_2]))$$

This completes the proof of the theorem. ∎

Before proving the correctness of the downward propagation function *mdown*, we first prove the correctness of *vts_cv* that given a type $R$ and a term $t$ computes a set of variable types $vts\_cv(R,t)$ that describes all substitutions that instantiate $t$ to a term in $R$.

LEMMA A.1. *$\{\theta \mid \theta(t) \in [R]_\triangle\} \subseteq \gamma_{\mathsf{DVT}}([vts\_cv(R,t)])$ for any $R \in \mathsf{RT}$ and $t \in \mathsf{Term}(\mathsf{func}, \mathcal{V}_P^+)$.*

PROOF. *Let $\mathsf{N}$ denote the set of natural numbers. Define $h : \mathsf{Term}(\mathsf{func}, \mathcal{V}_P^+) \mapsto \mathsf{N}$ as follows. $h(x) \stackrel{def}{=} 0$ for all $x \in \mathcal{V}_P^+$ and $h(f(t_1,\cdots,t_n)) \stackrel{def}{=} 1 + max\{h(t_i) \mid 1 \le i \le n\}$. Define $h : \mathsf{RT} \mapsto \mathsf{N}$ in the same way. Note $h(R) \ge 1$ for any $R \in \mathsf{RT}$. Let $\langle x^1, y^1\rangle \prec \langle x^2, y^2\rangle \stackrel{def}{=} (x^1 < x^2) \vee (x^1 = x^2) \wedge (y^1 < y^2)$. The proof is done by induction on $\langle h(R), h(t)\rangle$.*

*Basis. $\langle h(R), h(t)\rangle = \langle 1, 0\rangle$. Then $t \in \mathcal{V}_P^+$ and $vts\_cv(R,t) = \{Id^+[t \mapsto R]\}$. The lemma holds since $\{\theta \mid \theta(t) \in [R]_\triangle\} = \gamma_{\mathsf{DVT}}([vts\_cv(R,t)])$.*

*Induction. Assume that the lemma holds for all $R' \in \mathsf{RT}$ and $t' \in \mathsf{Term}(\mathsf{func}, \mathcal{V}_P^+)$ such that $\langle h(R'), h(t')\rangle \prec \langle h(R), h(t)\rangle$. Either (1) $h(R) > 1$ or (2) $h(R) = 1$.*

*Consider the case (1). Either (i) $R = R_1 \sqcap R_2$ or (ii) $R = R_1 \sqcup R_2$ or (iii) $R = c(R_1,\cdots,R_m)$ for some $m \ge 1$. The cases (i) and (ii) are immediate. Consider the case (iii). By the definition of $[\![\cdot]\!]_\triangle$, $\theta(f(t_1,\cdots,t_n)) \in [R]_\triangle$ implies that there is a type rule $c(\beta_1,\cdots,\beta_m) ::= f(\tau_1,\cdots,\tau_n)$ in $\Delta$ such that $\theta(t_i) \in [\![\mathcal{K}(\tau_i)]\!]_\triangle$ where $\mathcal{K} = \{\beta_j \mapsto R_j \mid 1 \le j \le m\}$. We have $h(\mathcal{K}(\tau_i)) \le h(c(R_1,\cdots,R_m))$ and $h(t_i) < h(f(t_1,\cdots,t_n))$. By the induction hypothesis,*

$$\theta \in \gamma_{\mathsf{DVT}}([vts\_cv(\mathcal{K}(\tau_i), t_i)])$$

*for all $1 \le i \le n$. By lemmas 4.2 and 4.3,*

$$\theta \in \gamma_{\mathsf{DVT}}([\bigotimes_{1 \le i \le n} vts\_cv(\mathcal{K}(\tau_i), t_i)])$$

*By the definition of $vts\_cv$, we have*

$$\theta \in \gamma_{\mathsf{DVT}}([vts\_cv(c(R_1,\cdots,R_m), f(t_1,\cdots,t_n))])$$

*Thus, the lemma holds for the case (1).*

*Now consider the case (2). $t = f(t_1,\cdots,t_n)$. The proof is the same as that for the case (1).(iii). This completes the proof of the lemma.* $\square$

The following lemma states the correctness of the downward propagation function *down*.

LEMMA A.2. *Let $\mathcal{S}' = down(E, \mathcal{S})$. Then $mgu(\theta(E)) \circ \theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}'])$ for all $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}])$.*

PROOF. *Let $\mathcal{S}_\mu = \{\mu\} \otimes \bigotimes_{(x=t)\in E} vts\_cv(\mu(x), t)$. It suffices to prove that $mgu(\sigma(E)) \circ \sigma \in \gamma_{\mathsf{DVT}}([\mathcal{S}_\mu])$ for all $\sigma \in \gamma_{\mathsf{VT}}(\mu)$. $mgu(\sigma(E)) \circ \sigma \in \gamma_{\mathsf{VT}}(\mu)$ as the denotation of any type in $\mathsf{RT}$ is closed under substitution. By lemma A.1, we have $mgu(\sigma(E)) \circ \sigma \in \gamma_{\mathsf{DVT}}([vts\_cv(\mu(x), t)])$ for any $(x = t)$ in $E$. So, $mgu(\sigma(E)) \circ \sigma \in \gamma_{\mathsf{DVT}}([\mathcal{S}_\mu])$. This completes the proof of the lemma.* $\square$

The following lemma states that, given a general type $\tau$ and a conjunctive type $C$, $tv\_cv(C,\tau)$ is a type valuation that instantiate $\tau$ to a supertype of $C$.

LEMMA A.3. *Let $\tau$ be a general type, $C$ be a conjunctive type and $\mathcal{K} = tv\_cv(C,\tau)$. Then $[C]_\triangle \subseteq [\![\mathcal{K}(\tau)]\!]_\triangle$.*

PROOF. *Let $C$ be atomic and $C = c(R_1,\cdots,R_m)$. Either (a) $\tau \in \mathsf{Para}$ or (b) $\tau = d(\beta_1,\cdots,\beta_k)$ with $\beta_1,\cdots,\beta_k$ being different type parameters in $\mathsf{Para}$. In the case (a), we have $\mathcal{K} = \{\tau \mapsto C\}$. The lemma holds because $\mathcal{K}(\tau) = C$. Consider the case (b), if $c/m = d/k$ then $\mathcal{K} = \{\beta_j \mapsto R_j \mid 1 \le j \le m\}$ by the definition of $tv\_cv$ and we have $\mathcal{K}(\tau) = C$. Otherwise, $\mathcal{K} = \bowtie$ by the definition of $tv\_cv$ and $\mathcal{K}(\tau) = \mathbf{1}$ by the definition of $\bowtie$. So, the lemma holds in the case (b).*

*Now suppose $C = A_1 \sqcap \cdots \sqcap A_k$. Let $\mathcal{K}_i = tv\_cv(A_i, \tau)$. We have $\mathcal{K} = \mathring{\bigwedge}_{1 \le i \le k} \mathcal{K}_i$. We have $[A_i]_\triangle \subseteq [\![\mathcal{K}_i(\tau)]\!]_\triangle$. So,*

$$\begin{array}{rcl} [C]_\triangle & = & [\![\sqcap_{1 \le i \le k} A_i]\!]_\triangle \\ & \subseteq & \cap_{1 \le i \le k} [\![\mathcal{K}_i(\tau)]\!]_\triangle \\ & = & [\![(\mathring{\bigwedge}_{1 \le i \le k} \mathcal{K}_i)(\tau)]\!]_\triangle \\ & = & [\![\mathcal{K}(\tau)]\!]_\triangle \end{array}$$

*This completes the proof of the lemma.* $\square$

The lemma below states that $ctype(f, \langle C_1,\cdots,C_n\rangle)$ is a conjunctive type that contains all terms $f(s_1,\cdots,s_n)$ such that $s_i$ is of type $C_i$.

LEMMA A.4. *Let $C_1,\cdots,C_n$ be conjunctive types. Then*

$$\begin{array}{l} \{f(s_1,\cdots,s_n) \mid \wedge_{1 \le i \le n} (s_i \in [\![C_i]\!]_\triangle)\} \\ \quad \subseteq \ [\![ctype(f, \langle C_1,\cdots,C_n\rangle)]\!]_\triangle \end{array}$$

PROOF. *Let $\tau ::= f(\tau_1,\cdots,\tau_n)$ be a type rule in $\Delta$. By lemma A.3 and the definition of $\mathring{\bigvee}$, we have*

$$[\![C_i]\!]_\triangle \subseteq [\![(\mathring{\bigvee}_{1 \le i \le n} tv\_cv(C_i, \tau_i))(\tau_i)]\!]_\triangle$$

*Therefore,*

$$\begin{array}{l} \{f(s_1,\cdots,s_n) \mid \wedge_{1 \le i \le n} (s_i \in [\![C_i]\!]_\triangle)\} \\ \quad \subseteq \ [\![(\mathring{\bigvee}_{1 \le i \le n} tv\_cv(C_i, \tau_i))(\tau)]\!]_\triangle \end{array}$$

*by the definition of $[\![\cdot]\!]_\triangle$. This is true of every type rule in $\Delta$ of the form $\tau ::= f(\tau_1,\cdots,\tau_n)$. So,*

$$\begin{array}{rl} & \{f(s_1,\cdots,s_n) \mid \wedge_{1 \le i \le n} (s_i \in [\![C_i]\!]_\triangle)\} \\ \subseteq & \cap_{(\tau ::= f(\tau_1,\cdots,\tau_n)) \in \Delta} [\![(\mathring{\bigvee}_{1 \le i \le n} tv\_cv(C_i, \tau_i))(\tau)]\!]_\triangle \\ = & [\![\sqcap_{(\tau ::= f(\tau_1,\cdots,\tau_n)) \in \Delta} (\mathring{\bigvee}_{1 \le i \le n} tv\_cv(C_i, \tau_i))(\tau)]\!]_\triangle \\ = & [\![ctype(f, \langle C_1,\cdots,C_n\rangle)]\!]_\triangle \end{array}$$

*This completes the proof of the lemma.* $\square$

This lemma says that, given $t \in \mathsf{Term}(\mathsf{func}, \mathcal{V}_P^+)$ and a variable typing $\mu \in (\mathcal{V}_P^+ \mapsto \mathsf{RT})$, $mlt(t, \mu)$ is a type that contains the instances of $t$ under all the substitutions described by $\mu$.

LEMMA A.5. *Let $t \in \mathsf{Term}(\mathsf{func}, \mathcal{V}_P^+)$ and a variable typing $\mu \in (\mathcal{V}_P^+ \mapsto \mathsf{RT})$. $\theta(t) \in [\![type(t, \mu)]\!]_\Delta$ for all $\theta \in \gamma_{\mathsf{VT}}(\mu)$.*

PROOF. *The proof is done by induction on $h(t)$.*

*Basis.* $h(t) = 0$. *Then $t \in \mathcal{V}_P^+$ and $type(t, \mu) = \mathrm{DNF}(\mu(t))$. The lemma holds immediately.*

*Induction.* $h(t) > 0$. *Let $t = f(t_1, \cdots, t_n)$ and $R_i = type(t_i, \mu)$ and $\theta \in \gamma_{\mathsf{VT}}(\mu)$. By the induction hypothesis, we have $\theta(t_i) \in [\![R_i]\!]_\Delta$ for all $1 \leq i \leq n$. By the definition of $type(\cdot, \cdot)$ and lemma A.4, we have*

$$\begin{aligned} &\theta(f(t_1, \cdots, t_n)) \\ &\in [\![\sqcup_{C_1 \hat{\in} type(t_1, \mu), \cdots, C_n \hat{\in} type(t_n, \mu)} ctype(f, \langle C_1, \cdots, C_n \rangle)]\!]_\Delta \\ &= [\![type(f(t_1, \cdots, t_n), \mu)]\!]_\Delta \end{aligned}$$

*This completes the proof of the lemma.* □

The correctness of the upward propagation function *up* is given by the following lemma.

LEMMA A.6. *Let $\mathcal{S} \in \wp(\mathcal{V}_P^+ \mapsto \mathsf{RT})$ and $E \in \wp(\mathsf{Eqn})$. $mgu(\theta(E)) \circ \theta \in \gamma_{\mathsf{DVT}}([up(E, \mathcal{S})])$ for all $\theta \in \gamma_{\mathsf{DVT}}([\mathcal{S}])$.*

PROOF. *Let*

$$\mu' = \lambda x \in \mathcal{V}_P^+ . \begin{pmatrix} if\ \exists t.(x = t) \in E \\ then\ \mu(x) \sqcap type(t, \mu) \\ else\ \mu(x) \end{pmatrix}$$

*It suffices to prove that $mgu(\sigma(E)) \circ \sigma \in \gamma_{\mathsf{VT}}(\mu')$ for all $\sigma \in \gamma_{\mathsf{VT}}(\mu)$. By lemma A.5, $\sigma(t) \in [\![type(t, \mu)]\!]_\Delta$. We have $(mgu(\sigma(E)) \circ \sigma)(x) \in [\![type(t, \mu)]\!]_\Delta$ for all $x$ and $t$ such that $(x = t) \in E$. Therefore, $(mgu(\sigma(E)) \circ \sigma) \in \gamma_{\mathsf{VT}}(\mu')$. This completes the proof of the lemma.* □