Improving Precision of Type Analysis Using Non-Discriminative Union

LUNJIN LU

Oakland University, Rochester, MI 48309, USA.
(*e-mail:* lunjin@acm.org)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

This paper presents a new type analysis for logic programs. The analysis is performed with *a priori* type definitions; and type expressions are formed from a fixed alphabet of type constructors. Non-discriminative union is used to join type information from different sources without loss of precision. An operation that is performed repeatedly during an analysis is to detect if a fixpoint has been reached. This is reduced to checking the emptiness of types. Due to the use of non-discriminative union, the fundamental problem of checking the emptiness of types is more complex in the proposed type analysis than in other type analyses with *a priori* type definitions. The experimental results, however, show that use of tabling reduces the effect to a small fraction of analysis time on a set of benchmarks.

KEYWORDS: Type analysis, Non-discriminative union, Abstract interpretation, Tabling

1 Introduction

Types play an important role in programming. They make programs easier to understand and help detect errors. There has been much research into types in logic programming. A type checker requires the programmer to declare types for each predicate in the program and verifies if the program is consistent with the declared types (Aiken and Lakshman 1994; Dart and Zobel 1992a; Fages and Coquery 2001; Frühwirth et al. 1991; Mycroft and O'Keefe 1984; Reddy 1990; Yardeni et al. 1991; Yardeni and Shapiro 1991). A type analysis derives types for the predicates or literals in the program from the text of the program (Gallagher and Puebla 2002; Charatonik and Podelski 1998; Codish and Lagoon 2000; Gallagher and de Waal 1994; Heintze and Jaffar 1990; Heintze and Jaffar 1992; Lu 1998; Mishra 1984; Saglam and Gallagher 1995; Zobel 1987).

This paper presents a new type analysis that infers types with *a priori* type definitions which determine possible types and their meanings. Types are formed from type constructors from a fixed alphabet. This is in contrast to those type analyses that generate type definitions during analysis. Both kinds of type analysis are useful. An analysis that generates type definitions may be favored in compile-time

optimizations and program transformations whilst an analysis with *a priori* type definitions may be preferred in interactive programming tools such as debuggers because inferred types are easier for the programmer to understand.

A number of factors compromise the precision of previous type analyses with *a priori* type definitions. Firstly, they only allow deterministic type definitions. A function symbol cannot occur more than once in the definition of the same type. A type then denotes a tree language recognized by a deterministic top-down tree automaton (Comon et al. 2002) and hence called a deterministic type. The restriction to deterministic type definitions allows fast propagation of type information. However, it 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 $float \rightarrowtriangle +(integer, float)$ and $float \rightarrowtriangle +(float, float)$ violate the restriction. Some previous work even disallows function overloading (Horiuchi and Kanamori 1988; Kanamori and Horiuchi 1985; Kanamori and Kawamura 1993), which makes it hard to support built-in types. For instance, Prolog has built-in type *atom* that denotes the set of atoms. Without function overloading, atoms such as [ ] cannot be a member of another type, say *list*. Secondly, the type languages in previous type analyses with *a priori* type definitions do not include set union as a type constructor. The denotation of the join of two types can be larger than the set union of their denotations. For instance, the join of $list(integer)$ and $list(float)$ is $list(number)$. Let or be a type constructor that is interpreted as set union. Then $list(number)$ is a super-type of $\mathsf{or}(list(integer), list(float))$ since the list [1, 2.5] belongs to the former but not the latter. Should non-deterministic type definitions be allowed, there is also a need to use set intersection as a type constructor as explained in Section 2. Finally, previous type analyses with *a priori* type definitions describe a set of substitutions by a single variable typing which maps variables of interest into types. The least upper bound of two variable typings is performed point-wise, effectively severing type dependency between variables.

Our type analysis aims to improve precision by eliminating the above mentioned factors. It supports non-deterministic type definitions, uses a type language that includes set union and intersection as type constructors and describes a set of substitutions by a set of variable typings. All these help improve analysis precision. On the other hand, they all incur performance penalty. However, experimental results with a prototype implementation show that tabling (Warren 1992) reduces the time increase to a small fraction on a suite of benchmark programs. Our type analysis is presented as an abstract domain together with a few primitive operations on the domain. The domain is presented for an abstract semantics that is Nilsson's abstract semantics (Nilsson 1988) extended to deal with negation and built-in predicates. The primitive operations on the domain can be easily adapted to work with other abstract semantics such as (Bruynooghe 1991).

The remainder of the paper is organized as follows. Section 2 provides motivation behind our work with some examples and Section 3 briefly presents the abstract semantics along with basic concepts and notations used in the remainder of the paper. Section 4 is devoted to types — their definitions and denotations. Section 5 presents the abstract domain and Section 6 the abstract operations. In Section 7,

we present a prototype implementation of our type analysis and some experimental results. Section 8 compares our type analysis with others and Section 9 concludes. An appendix contains proofs.

2 Motivation

This section provides motivation behind our type analysis via examples. The primary operations for propagating type information are informally illustrated; and the need for using set union and intersection as type constructors is highlighted.

Example 2.1
This example demonstrates the use of set union as a type constructor. Consider the following program and type rules.

$$\begin{array}{rcl} p(Z) & \leftarrow & ② X = a\ ③, Y = 2.5\ ④\ , Z = cons(X, cons(Y, nil))\ ⑤. \\ & \leftarrow & ①\ p(Z)\ ⑥.\ \%\ \text{query} \end{array}$$

$$\begin{array}{rcl} list(\beta) & \rightarrowtriangle & nil \\ list(\beta) & \rightarrowtriangle & cons(\beta, list(\beta)) \end{array}$$

The two type rules define lists. They state that a term is of type $list(\beta)$ iff it is either nil or of the form $cons(X, Y)$ such that X is of type β and Y of type $list(\beta)$. Type rules are formally introduced in Section 4. The program has been annotated with circled numbers to identify relevant program points for the purpose of exposition.

The type analysis can be thought of as an abstract execution that mimics the concrete (normal) execution of the program. A program state in the concrete execution is replaced with an abstract one that describes the concrete state. The abstract states are type constraints.

Suppose that no type information is given at program point ① — the start point of the execution. This is described by the type constraint $\mu_1 = true$. The execution reaches program point ② with the abstract state $\mu_2 = true$. The abstract state at program point ③ is $\mu_3 = (X \in atom)$ which states that X is of type $atom$. The abstract state at program point ④ is $\mu_4 = (X \in atom) \wedge (Y \in float)$. The abstract execution of $Z = cons(X, cons(Y, nil))$ in μ_4 obtains the abstract state μ_5 at program point ⑤. The computation of μ_5 needs some explanation. The two terms that are unified have the same type after the unification. Since μ_4 does not constrain Z, there is no type information propagated from Z to either X or Y. The type for Z in μ_5 equals the type of $cons(X, cons(Y, nil))$ in μ_4 which is computed in a bottom-up manner. To compute the type for nil, we apply the type rule for $nil/0$. The type rule states that nil is of type $list(\beta)$ for any β. Thus, the most precise type for nil is $list(\mathbf{0})$ where the type $\mathbf{0}$ denotes the empty set of terms. We omit the process of computing the type $list(float)$ for $cons(Y, nil)$ in μ_4 since it is similar to the following. To compute the type for $cons(X, cons(Y, nil))$, we apply the type rule for $cons/2$. The right hand side of the type rule is $cons(\beta, list(\beta))$. We first find the smallest value for β such that β is greater than or equal to $atom$ — the type for X in

μ_4 and the smallest value for β such that $list(\beta)$ is greater than or equal to $list(float)$ — the type for $cons(Y, nil)$ in μ_4. Those two values are respectively $atom$ and $float$ and their least upper bound is $\mathsf{or}(atom, float)$. Replacing β with $\mathsf{or}(atom, float)$ in the left hand side of the type rule gives the most precise type $list(\mathsf{or}(atom, float))$ for $cons(X, cons(Y, nil))$ in μ_4. Conjoining $Z \in list(\mathsf{or}(atom, float))$ with μ_4 results in $\mu_5 = ((X \in atom) \wedge (Y \in float) \wedge (Z \in list(\mathsf{or}(atom, float))))$. The abstract state at program point ⑥ is $\mu_6 = (Z \in list(\mathsf{or}(atom, float)))$ which is obtained from μ_5 by projecting out type constraints on X and Y.

The existence of the type constructor or helps avoid approximations. Without it, the least upper bound of $atom$ and $float$ is $\mathbf{1}$ which denotes the set of all terms. Note that the collection of type rules is fixed during analysis. ∎

When two or more type rules are associated with a single function symbol, there is also a need to use set intersection as a type constructor. The following example illustrates this point.

Example 2.2
Suppose that types are defined by the following four type rules.

$$\begin{aligned} list(\beta) &\rightarrowtriangle nil \\ list(\beta) &\rightarrowtriangle cons(\beta, list(\beta)) \\ tree(\beta) &\rightarrowtriangle nil \\ tree(\beta) &\rightarrowtriangle node(tree(\beta), \beta, tree(\beta)) \end{aligned}$$

Consider the problem of computing the type for $cons(X, nil)$ in the abstract state $\mu = (X \in integer)$.

There are two type rules for $nil/0$. The type rule $list(\beta) \rightarrowtriangle nil$ states that nil belongs to $list(\beta)$ for any β. The most precise type for nil that can be inferred from this rule is $list(\mathbf{0})$. Similarly, the most precise type for nil that can be inferred from the type rule $tree(\beta) \rightarrowtriangle nil$ is $tree(\mathbf{0})$. Thus, the most precise type for nil is $\mathsf{and}(list(\mathbf{0}), tree(\mathbf{0}))$ where and is a type constructor that denotes set intersection.

To compute the type for $cons(X, nil)$, we apply the type rule for $cons/2$. Its right hand side is $cons(\beta, list(\beta))$. We first find the smallest value for β such that β is greater than or equal to $integer$ — the type for X in μ. The value is $integer$. We then find the smallest value for β such that $list(\beta)$ is greater than or equal to $\mathsf{and}(list(\mathbf{0}), tree(\mathbf{0}))$ — the type for nil in μ. This is done by matching $list(\beta)$ with $list(\mathbf{0})$ and with $tree(\mathbf{0})$ and intersecting values for β obtained from these two matches. The first match results in $\mathbf{0}$. The second match is unsuccessful and produces $\mathbf{1}$ since we are computing an upper approximation. The intersection of these two types is $\mathsf{and}(\mathbf{0}, \mathbf{1})$ which is equivalent to $\mathbf{0}$. The join of the two smallest values $integer$ and $\mathbf{0}$ for β is $\mathsf{or}(integer, \mathbf{0})$ which is equivalent to $integer$. Finally, the type $list(integer)$ for $cons(X, nil)$ is obtained by substituting $integer$ for β in the left hand side of the type rule.

Without and in the type language, a choice must be made between $list(\mathbf{0})$ and $tree(\mathbf{0})$ as the type for nil. Though these types are equivalent to $\mathsf{and}(list(\mathbf{0}), tree(\mathbf{0}))$, the choice made could complicate the ensuing computation. Should $tree(\mathbf{0})$ be chosen, we would need to find the smallest value for β such that $list(\beta)$ is greater than

or equal to $tree(\mathbf{0})$. This could only be solved by applying an algorithm for solving type inclusion constraints. The presence of and allows us to avoid that. ∎

For the purpose of improving the precision of analysis, there is also a need for disjunction at the level of abstract states. The following example illustrates this point.

Example 2.3
Consider the following program

$$\begin{aligned} p(X) &\leftarrow q(X,Y),\ ①\ ... \\ q(1,2). & \\ q(a,b). & \\ &\leftarrow p(X).\ \%\ \text{query} \end{aligned}$$

When the execution reaches program point ①, X and Y are both of type *integer* or they are both of type *atom*. This is described by a type constraint $((X \in integer) \wedge (Y \in integer)) \vee ((X \in atom) \wedge (Y \in atom))$. Without disjunction at the level of abstract states, we would have to replace the type constraint with a less precise one: $((X \in \mathsf{or}(integer, atom)) \wedge (Y \in \mathsf{or}(integer, atom)))$. ∎

3 Preliminaries

The reader is assumed to be familiar with the terminology of logic programming (Lloyd 1987) and that of abstract interpretation (Cousot and Cousot 1977). We consider a subset of Prolog which contains definite logic programs extended with negation as failure and some built-in predicates.

3.1 Basic Concepts

We sometimes use Church's lambda notation for functions, so that a function f defined $f(x) = e$ will be denoted $\lambda x.e$. Let A and B be sets. Then $A \mapsto B$ is the set of total functions from A to B and $A \rightarrowtail B$ is the set of partial functions from A to B. The function composition $\circ$ is defined $f \circ g = \lambda x.f(g(x))$. Let D be a set. A sequence over D is either ϵ or $d \bullet \vec{d}$ where $d \in D$ and $\vec{d}$ is a sequence over D. The infix operator $\bullet$ associates to the right and prepends an element to a sequence to form a longer sequence. The set of all sequences over D is denoted D^*. Let $\vec{d} = d_1 \bullet d_2 \bullet \cdots \bullet d_n \bullet \epsilon$. We will sometimes write $\vec{d}$ as $d_1, d_2, \cdots, d_n$. The dimension $\|\vec{d}\|$ of $\vec{d}$ is n. Let $E \subseteq D$ and $S \subseteq D^*$. The set extension of $\bullet$ is defined as $E \bullet S = \{d \bullet \vec{d} \mid d \in E \wedge \vec{d} \in S\}$.

3.2 Abstract Interpretation

A semantics of a program is given by an interpretation $\langle (C, \sqsubseteq_C), \mathcal{C} \rangle$ where $(C, \sqsubseteq_C)$ is a complete lattice and $\mathcal{C}$ is a monotone function on $(C, \sqsubseteq_C)$. The semantics is defined as the least fixed point *lfp* $\mathcal{C}$ of $\mathcal{C}$. The concrete semantics of the program

is given by the concrete interpretation $\langle (C, \sqsubseteq_C), \mathcal{C}\rangle$ while an abstract semantics is given by an abstract interpretation $\langle (A, \sqsubseteq_A), \mathcal{A}\rangle$. The correspondence between the concrete and the abstract domains is formalized by a Galois connection (α, γ) between $(C, \sqsubseteq_C)$ and $(A, \sqsubseteq_A)$. A Galois connection between A and C is a pair of monotone functions $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ satisfying $\forall c \in C.(c \sqsubseteq_C \gamma \circ \alpha(c))$ and $\forall a \in A.(\alpha \circ \gamma(a) \sqsubseteq_A a)$. The function α is called an abstraction function and the function γ a 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 (Cousot and Cousot 1992). The abstraction and concretization functions in a Galois connection uniquely determine each other; and a complete meet-morphism $\gamma : A \mapsto C$ induces a Galois connection (α, γ) with $\alpha(c) = \sqcap_A\{a \mid c \sqsubseteq_C \gamma(a)\}$. A function $\gamma : A \mapsto C$ is a complete meet-morphism iff $\gamma(\sqcap_A X) = \sqcap_C\{\gamma(x) \in X\}$ for any $X \subseteq A$. Thus, an analysis can be formalized as a tuple $(\langle (C, \sqsubseteq_C), \mathcal{C}\rangle, \gamma, \langle (A, \sqsubseteq_A), \mathcal{A}\rangle)$ such that $\langle (C, \sqsubseteq_C), \mathcal{C}\rangle$ and $\langle (A, \sqsubseteq_A), \mathcal{A}\rangle$ are interpretations, γ is a complete meet-morphism from $(C, \sqsubseteq_C)$ to $(A, \sqsubseteq_A)$, and $\forall a \in A.(\mathcal{C} \circ \gamma(a) \sqsubseteq_C \gamma \circ \mathcal{A}(a))$.

3.3 Logic Programs

Let Σ be a set of function symbols, Π a set of predicate symbols and Var a denumerable set of variables. Each function or predicate symbol has an arity which is a non-negative integer. We write $f/n \in \Sigma$ for an n-ary function symbol f in Σ and $p/n \in \Pi$ for an n-ary predicate symbol p in Π. Let $V \subseteq \mathsf{Var}$. The set of all terms over Σ and V, denoted $\mathsf{Term}(\Sigma, V)$, is the smallest set satisfying: (i) $V \subseteq \mathsf{Term}(\Sigma, V)$; and (ii) if $\{t_1, \cdots, t_n\} \subseteq \mathsf{Term}(\Sigma, V)$ and $f/n \in \Sigma$ then $f(t_1, \cdots, t_n) \in \mathsf{Term}(\Sigma, V)$. The set of all atoms that can be constructed from Π and $\mathsf{Term}(\Sigma, V)$ is denoted $\mathsf{Atom}(\Pi, \Sigma, V)$; $\mathsf{Atom}(\Pi, \Sigma, V) = \{p(t_1, \cdots, t_n) \mid (p/n \in \Pi) \wedge (\{t_1, \cdots, t_n\} \subseteq \mathsf{Term}(\Pi, \Sigma, V))\}$. Let $\mathsf{Term} = \mathsf{Term}(\Sigma, \mathsf{Var})$ and $\mathsf{Atom} = \mathsf{Atom}(\Pi, \Sigma, \mathsf{Var})$ for abbreviation. The set Term contains all terms and the set Atom all atoms. The negation of an atom $p(t_1, \cdots, t_n)$ is written $\neg p(t_1, \cdots, t_n)$. A literal is either an atom or the negation of an atom. The set of all literals is denoted $\mathsf{Literal}$. Let Bip denote the set of calls to built-in predicates. Note that $\mathsf{Bip} \subseteq \mathsf{Atom}$.

A clause C is a formula of the form $H \leftarrow L_1, \cdots, L_n\blacksquare$ where $H \in \mathsf{Atom} \cup \{\square\}$ and $L_i \in \mathsf{Literal}$ for $1 \leq i \leq n$. H is called the head of the clause and $L_1, \cdots, L_n\blacksquare$ the body of the clause. Note that $\square$ denotes the empty head and $\blacksquare$ denotes the empty body. A query is a clause whose head is $\square$. A program is a set of clauses of which one is a query. The query initiates the execution of the program.

Program states which exist during the execution of a logic program are called substitutions. A substitution θ is a mapping from Var to Term such that $dom(\theta) = \{x \mid (x \in \mathsf{Var}) \wedge (\theta(x) \neq x)\}$ is finite. The set $dom(\theta)$ is called the domain of θ. Let $dom(\theta) = \{x_1, \cdots, x_n\}$. Then θ is written as $\{x_1 \mapsto \theta(x_1), \cdots, x_n \mapsto \theta(x_n)\}$. A substitution θ is idempotent if $\theta \circ \theta = \theta$. The set of idempotent substitutions is denoted Sub; and the identity substitution is denoted ϵ. Let $Sub_{fail} = Sub \cup \{fail\}$ and extend $\circ$ by $\theta \circ fail = fail$ and $fail \circ \theta = fail$ for any $\theta \in Sub_{fail}$.

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

An equation is a formula of the form $l = r$ where either $l, r \in \mathsf{Term}$ or $l, r \in \mathsf{Atom}$. The set of all equations is denoted Eqn. For a set of equations E, $mgu : \wp(\mathsf{Eqn}) \mapsto Sub_{fail}$ returns either a most general unifier for E if E is unifiable or *fail* otherwise. Let $mgu(l, r)$ stand for $mgu(\{l = r\})$. Define $eq(\theta) = \{x = \theta(x) \mid x \in dom(\theta)\}$ for $\theta \in Sub$ and $eq(fail) = fail$.

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

We assume that there is a function $sys : \mathsf{Bip} \times Sub \mapsto \wp(Sub)$ that models the behavior of built-in predicates. The set $sys(p(t_1, \cdots, t_n), \theta)$ consists of all those substitutions $\sigma \circ \theta$ such that σ is a computed answer to $\theta(p(t_1, \cdots, t_n))$.

Let V_P be the set of variables in the program and $\mathsf{Atom}_P = \mathsf{Atom}(\Pi, \Sigma, V_P)$. Define $uf : \mathsf{Atom}_P \times Sub \times \mathsf{Atom}_P \times Sub \mapsto Sub_{fail}$ by

$$uf(a_1, \theta, a_2, \omega) = let\ \rho \in \mathsf{Ren}(\theta(a_1), \omega(a_2))\ in\ mgu(\rho(\theta(a_1)), \omega(a_2)) \circ \omega$$

The operation $uf(a_1, \theta, a_2, \omega)$ models both procedure-call and procedure-exit operations. In a procedure-call operation, a_1 and θ are the call and the program state before the call, a_2 is the head of the clause that is used to resolve with the call and ω the identity substitution ϵ. In a procedure-exit operation, a_2 and ω are the call and the program state before the call, a_1 is the head of the clause that was used to resolve with the call and θ is the program state after the execution of the body of the clause. A renaming is applied to the call in a procedure-call operation whilst in a procedure-exit operation it is the head of the clause that is renamed.

3.4 Abstract Semantics

The new type analysis is presented as an abstract domain with four abstract operations. The domain and the operations are designed for an abstract semantics in (Nilsson 1988) extended with supports for negation-as-failure and built-in predicates. The extended abstract semantics is a special case of an abstract semantics in (Lu 2003) where a formal presentation can be found. The adaptation of the analysis to other abstract semantics such as (Bruynooghe 1991) is straightforward since they require abstract operations with similar functionalities.

The abstract semantics is parameterized by an abstract domain $\langle ASub^\flat, \sqsubseteq^\flat \rangle$. The elements in $ASub^\flat$ are called abstract substitutions since they are properties of substitutions. The abstract domain is related to the collecting domain $\langle \wp(Sub), \subseteq \rangle$ via a concretization function $\gamma : ASub^\flat \mapsto \wp(Sub)$. We say that an abstract substitution π describes a set of substitutions Θ iff $\Theta \subseteq \gamma(\pi)$. As usual, the abstract domain and the concretization function are required to satisfy the following conditions.

C1: $< ASub^\flat, \sqsubseteq^\flat >$ is a complete lattice with least upper bound operation $\sqcup^\flat$;
C2: $\gamma(ASub^\flat)$ is a Moore family where $\gamma(X) = \bigcup\{\gamma(x) \mid x \in X\}$.

We informally present the abstract semantics using the following program as a running example.

$$
\begin{array}{lll}
\texttt{diff}(X,L,K) & \leftarrow & ①\ \texttt{member}(X,L),\ ②\ \neg\texttt{member}(X,K)\ ③\\
\texttt{diff}(X,L,K) & \leftarrow & ④\ \texttt{member}(X,K),\ ⑤\ \neg\texttt{member}(X,L)\ ⑥\\
\texttt{member}(X,[X|L]) & \leftarrow & ⑦\\
\texttt{member}(X,[H|L]) & \leftarrow & ⑧\ \texttt{member}(X,L)\ ⑨\\
 & \leftarrow & ⑩\ Y=[a,b]\ \ ⑪\ Z=[1,2]\ ⑫\ \texttt{diff}(X,Y,Z)\ ⑬
\end{array}
$$

The intended interpretation for $\texttt{member}(X,L)$ is that X is a member of list L. The intended interpretation for $\texttt{diff}(X,L,K)$ is that X is in L or K but not in both. For brevity of exposition, let $A = \texttt{member}(X,L)$; $B = \texttt{member}(X,K)$; $C = \texttt{member}(X,[X|L])$; $D = \texttt{member}(X,[H|L])$; $E = \texttt{diff}(X,L,K)$ and $F = \texttt{diff}(X,Y,Z)$. The atom in the literal to the right of a program point p is denoted $\mathbb{A}(p)$. For instance, $\mathbb{A}(2) = \mathbb{A}(4) = B$. Let $\mathbb{H}(p)$ denote the head of the clause with which p is associated. For instance, $\mathbb{H}(1) = \mathbb{H}(2) = E$. Let p^- be the point to the left of p if p^- exists. For instance, $2^- = 1$ whilst 1^- is undefined.

The abstract semantics associates each textual program point with an abstract substitution. The abstract substitution describes all the substitutions that may be obtained when the execution reaches the program point. The abstract semantics is the least solution to a system of data flow equations - one for each program point. The system is derived from the control flow graph of the program whose vertices are the textual program points. Let Pt be the set of the textual program points. An edge from vertex p to vertex q in the graph is denoted $q \leftharpoonup p$; and it indicates that the execution may reach q immediately after it reaches p.

Consider the example program. We have $\mathsf{Pt} = \{1, \cdots, 13\}$. The program point $\iota = 10$ is called the initial program point since it is where the execution of the program is initiated. The abstract substitution at $\iota = 10$ is an analysis input, denoted π_ι, and it does not change during analysis. Thus, the data flow equation for program point 10 is $X^\flat(10) = \pi_\iota$ where $X^\flat$ is a mapping from program points to abstract substitutions. The data flow equations for other program points are derived by considering four kinds of control flow that may arise during program execution. The first kind models the execution of built-in calls. For instance, the control may flow from program point 10 to program point 11 by executing $Y = [a, b]$. The data flow equation for program point 11 is $X^\flat(11) = Sys^\flat(Y = [a, b], X^\flat(10))$ where the transfer function $Sys^\flat : \mathsf{Bip} \times ASub^\flat \mapsto ASub^\flat$ emulates the execution of a built-in call. Let Pt^{bip} be the set of all the program points that follow the built-in calls in the program. We have $\mathsf{Pt}^{bip} = \{11, 12\}$ for the example program. Another kind of control flow models negation-as-failure. The transfer function for this kind of control flow is the identity function. For instance, the control may flow from program point 2 to program point 3 since member(X,K) may fail, which yields this data flow equation $X^\flat(3) = X^\flat(2)$. Denote by Pt^{nf} the set of all the program points that follow negative literals. We have $\mathsf{Pt}^{nf} = \{3, 6\}$ for the example program.

The third kind of control flow arises when a procedure-call is performed. For instance, the control may flow from program point 1 to program point 8. The description of data that flow from program point 1 to program point 8 is expressed

as $Uf^\flat(A, X^\flat(1), D, Id^\flat)$ where $Id^\flat$ is an abstract substitution that describes $\{\epsilon\}$. Note that A is the call and D the head of the clause to which program point 8 belongs. The control may also flow to program point 8 from program points 4, 8, 2 and 5. The control flows from program point 5 to program point 8 when the negated sub-goal `member`(K,L) is executed. The descriptions of data that flow to program point 8 from those five source program points are merged together using the least upper bound operation $\sqcup^\flat$ on $ASub^\flat$, yielding the following data flow equation.

$$\begin{aligned} X^\flat(8) \;=\; & Uf^\flat(A, X^\flat(1), D, Id^\flat) \sqcup^\flat Uf^\flat(B, X^\flat(4), D, Id^\flat) \sqcup^\flat Uf^\flat(A, X^\flat(8), D, Id^\flat) \\ & \sqcup^\flat Uf^\flat(B, X^\flat(2), D, Id^\flat) \sqcup^\flat Uf^\flat(A, X^\flat(5), D, Id^\flat) \end{aligned}$$

The transfer function $Uf^\flat : \mathsf{Atom}_P \times ASub^\flat \times \mathsf{Atom}_P \times ASub^\flat \mapsto ASub^\flat$ approximates $Uf : \mathsf{Atom}_P \times \wp(Sub) \times \mathsf{Atom}_P \times \wp(Sub) \mapsto \wp(Sub)$ defined

$$Uf(a_1, \Theta_1, a_2, \Theta_2) = \{uf(a_1, \theta_1, a_2, \theta_2) \neq fail \mid \theta_1 \in \Theta_1 \wedge \theta_2 \in \Theta_2\}$$

which is the set extension of uf. Denote by Pt^{call} the set of program points that are reached via procedure-calls. We have $\mathsf{Pt}^{call} = \{1, 4, 7, 8\}$ for the example program.

The fourth kind of control flow arises when a procedure exits. For instance, the control may flow from program point 3 to program point 13. The description of data that flow from program point 3 to program point 13 is expressed by $Uf^\flat(E, X^\flat(3), F, X^\flat(12))$ where E is the head of the clause to which program point 3 belongs and F the call that invoked the clause. The only other control flow to program point 13 is from program point 6. Thus, the data flow equation for program point 13 is $X^\flat(13) = Uf^\flat(E, X^\flat(3), F, X^\flat(12)) \sqcup^\flat Uf^\flat(E, X^\flat(6), F, X^\flat(12))$. Let Pt^{ret} be the set of program points that are reached via procedure-exits. For the example program, we have $\mathsf{Pt}^{ret} = \{2, 5, 9, 13\}$.

Let $\mathsf{Edge}^\jmath = \{q \leftarrow p \mid q \in \mathsf{Pt}^\jmath\}$ where $\jmath \in \{call, ret, nf, bip\}$. Note that $\mathsf{Edge}^\jmath$ is the set of control flows that sink in $\mathsf{Pt}^\jmath$. The data flow equation has the following general form.

$$X^\flat(q) = \begin{cases} \pi_\iota & \text{if } q = \iota \\ \sqcup^\flat\{Uf^\flat(\mathbb{A}(p), X^\flat(p), \mathbb{H}(q), Id^\flat) \mid q \leftarrow p \in \mathsf{Edge}\} & \text{if } q \in \mathsf{Pt}^{call} \\ \sqcup^\flat\{Uf^\flat(\mathbb{H}(q), X^\flat(q), \mathbb{A}(p^-), X^\flat(p^-)) \mid q \leftarrow p \in \mathsf{Edge}\} & \text{if } q \in \mathsf{Pt}^{ret} \\ X^\flat(q^-) & \text{if } q \in \mathsf{Pt}^{nf} \\ Sys^\flat(\mathbb{A}(q^-), X^\flat(q^-)) & \text{if } q \in \mathsf{Pt}^{bip} \end{cases}$$

where π_ι is the input abstract substitution. The least solution to the system of data flow equations is a correct analysis if, in addition to C1 and C2, the following local safety requirements are met.

C3: $\{\epsilon\} \subseteq \gamma(Id^\flat)$;
C4: $Sys(a, \gamma(\pi)) \subseteq \gamma(Sys^\flat(a, \pi))$ for any $a \in \mathsf{Bip}$ with $vars(a) \subseteq V_P$ and $\pi \in ASub^\flat$; and
C5: $Uf(a_1, \gamma(\pi_1), a_2, \gamma(\pi_2)) \subseteq \gamma \circ Uf^\flat(a_1, \pi_1, a_2, \pi_2)$ for any $\pi_1, \pi_2 \in ASub^\flat$, any $a_1, a_2 \in \mathsf{Atom}_P$.

Note that the condition C2 implies that $\sqcup^\flat$ safely abstracts $\cup$ with respect to γ. The

operation $Uf^\flat$ is called abstract unification since it mimics the normal unification operation whilst $Sys^\flat$ is called abstract built-in execution operation.

The complete system of data flow equations for the example program is as follows.

$$
\begin{array}{rcl}
X^\flat(1) & = & Uf^\flat(F, X^\flat(12), E, Id^\flat) \\
X^\flat(2) & = & Uf^\flat(C, X^\flat(7), A, X^\flat(1)) \sqcup^\flat Uf^\flat(D, X^\flat(9), A, X^\flat(1)) \\
X^\flat(3) & = & X^\flat(2) \\
X^\flat(4) & = & Uf^\flat(F, X^\flat(12), E, Id^\flat) \\
X^\flat(5) & = & Uf^\flat(C, X^\flat(7), B, X^\flat(4)) \sqcup^\flat Uf^\flat(D, X^\flat(9), B, X^\flat(4)) \\
X^\flat(6) & = & X^\flat(5) \\
X^\flat(7) & = & Uf^\flat(A, X^\flat(1), C, Id^\flat) \sqcup^\flat Uf^\flat(B, X^\flat(4), C, Id^\flat) \sqcup^\flat \\
 & & Uf^\flat(A, X^\flat(8), C, Id^\flat) \sqcup^\flat Uf^\flat(B, X^\flat(2), C, Id^\flat) \sqcup^\flat \\
 & & Uf^\flat(A, X^\flat(5), C, Id^\flat) \\
X^\flat(8) & = & Uf^\flat(A, X^\flat(1), D, Id^\flat) \sqcup^\flat Uf^\flat(B, X^\flat(4), D, Id^\flat) \sqcup^\flat \\
 & & Uf^\flat(A, X^\flat(8), D, Id^\flat) \sqcup^\flat Uf^\flat(B, X^\flat(2), D, Id^\flat) \sqcup^\flat \\
 & & Uf^\flat(A, X^\flat(5), D, Id^\flat) \\
X^\flat(9) & = & Uf^\flat(C, X^\flat(7), A, X^\flat(8)) \sqcup^\flat Uf^\flat(D, X^\flat(9), A, X^\flat(8)) \\
X^\flat(10) & = & \pi_\iota \\
X^\flat(11) & = & Sys^\flat(Y = [a, b], X^\flat(10)) \\
X^\flat(12) & = & Sys^\flat(Z = [1, 2], X^\flat(11)) \\
X^\flat(13) & = & Uf^\flat(E, X^\flat(3), F, X^\flat(12)) \sqcup^\flat Uf^\flat(E, X^\flat(6), F, X^\flat(12))
\end{array}
$$

The remainder of the paper presents our type analysis as an abstract domain and four abstract operations as required by the above abstract semantics. We begin with the type language and type definitions.

4 Types

The type language in a type system decides which sets of terms are types. A type is syntactically a ground term constructed from a ranked alphabet Cons and $\{\mathsf{and}, \mathsf{or}, \mathbf{1}, \mathbf{0}\}$ where and and or are binary and $\mathbf{1}$ and $\mathbf{0}$ are nullary. Elements of $\mathsf{Cons} \cup \{\mathsf{and}, \mathsf{or}, \mathbf{1}, \mathbf{0}\}$ are called type constructors. It is assumed that $(\mathsf{Cons} \cup \{\mathsf{and}, \mathsf{or}, \mathbf{1}, \mathbf{0}\}) \cap \Sigma = \emptyset$. The set of types is $\mathsf{Type} = \mathsf{Term}(\mathsf{Cons} \cup \{\mathsf{and}, \mathsf{or}, \mathbf{1}, \mathbf{0}\}, \emptyset)$. The denotations of type constructors in Cons are determined by type definitions whilst $\mathsf{and}, \mathsf{or}, \mathbf{1}$ and $\mathbf{0}$ have fixed denotations.

4.1 Type Rules

Types are defined by type rules. A type parameter is a variable from Para. A type scheme is either a type parameter or of the form $c(\beta_1, \cdots, \beta_m)$ where $c \in \mathsf{Cons}$ and $\beta_1, \cdots, \beta_m$ are different parameters. Let Schm be the set of all type schemes. A type

rule is of the form $c(\beta_1, \cdots, \beta_m) \twoheadrightarrow f(\tau_1, \cdots, \tau_n)$ where $c \in \mathsf{Cons}$, $f/n \in \Sigma$, $\beta_1, \cdots, \beta_m$ are different type parameters, and τ_j is a type scheme with type parameters from $\{\beta_1, \cdots, \beta_m\}$. Note that every type parameter in the right-hand side of a type rule must occur in the left-hand side. Overloading of function symbols is permitted since a function symbol can appear in the right-hand sides of two or more type rules. Let Δ be the set of all type rules. We assume that each function symbol occurs in at least one type rule and that each type constructor occurs in at least one type rule. Type rules are similar to type definitions used in typed logic programming languages Mercury (Somogyi et al. 1996) and Gödel (Hill and Lloyd 1994).

Example 4.1
Let $\Sigma = \{0, s(), [\,], [\mid], void, tr(,,)\}$ and $\mathsf{Cons} = \{nat, even, odd, list(), tree()\}$. The following set of type rules will be used in examples throughout the paper.

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

Type rules in Δ define natural numbers, even numbers, odd numbers, lists and trees. ∎

4.2 Denotations of Types

A (ground) type substitution is a member of $\mathsf{TSub} = (\mathsf{Para} \rightarrowtail \mathsf{Type}) \cup \{\top, \bot\}$. The application of a type substitution to a type scheme is defined as follows. $\top(\tau) = \mathbf{1}$ and $\bot(\tau) = \mathbf{0}$ for any type scheme τ. Let $\Bbbk \in (\mathsf{Para} \rightarrowtail \mathsf{Type})$. Define $\Bbbk(\beta) = \mathbf{0}$ for each $\beta \notin dom(\Bbbk)$ where $dom(\Bbbk)$ is the domain of $\Bbbk$. Then $\Bbbk(\tau)$ is obtained by replacing each β in τ with $\Bbbk(\beta)$. For instance, $\{\beta_1 \mapsto list(nat), \beta_2 \mapsto nat\}(list(\beta_1)) = list(list(nat))$.

Definition 4.2
The meaning of a type is defined by a function $[\![\cdot]\!]_\Delta : \mathsf{Type} \mapsto \wp(\mathsf{Term})$.

$$\begin{array}{rcl} [\![\mathbf{1}]\!]_\Delta & = & \mathsf{Term} \\ [\![\mathbf{0}]\!]_\Delta & = & \emptyset \\ [\![\mathsf{and}(R_1, R_2)]\!]_\Delta & = & [\![R_1]\!]_\Delta \cap [\![R_2]\!]_\Delta \\ [\![\mathsf{or}(R_1, R_2)]\!]_\Delta & = & [\![R_1]\!]_\Delta \cup [\![R_2]\!]_\Delta \\ [\![c(R_1, \cdots, R_m)]\!]_\Delta & = & \\ \multicolumn{3}{r}{\bigcup_{(c(\beta_1,\cdots,\beta_m) \twoheadrightarrow f(\tau_1,\cdots,\tau_n)) \in \Delta} \left(\begin{array}{l} let\ \Bbbk = \{\beta_j \mapsto R_j \mid 1 \le j \le m\} \\ in \\ \{f(t_1, \cdots, t_n) \mid \forall 1 \le i \le n . t_i \in [\![\Bbbk(\tau_i)]\!]_\Delta\} \end{array} \right)} \end{array}$$

∎

The function $[\![\cdot]\!]_\Delta$ gives fixed denotations to $\mathsf{and}, \mathsf{or}, \mathbf{1}$ and $\mathbf{0}$. Type constructors and and or are interpreted as set intersection and set union respectively. The type constructor $\mathbf{1}$ denotes Term and $\mathbf{0}$ the empty set. We say that a term t is in a

type R iff $t \in [\![R]\!]_\Delta$. Set inclusion and $[\![\cdot]\!]_\Delta$ induce a pre-order $\sqsubseteq$ on types: $(R_1 \sqsubseteq R_2) = ([\![R_1]\!]_\Delta \subseteq [\![R_2]\!]_\Delta)$ and an equivalence relation $\equiv$ on types: $(R_1 \equiv R_2) = (R_1 \sqsubseteq R_2) \wedge (R_2 \sqsubseteq R_1)$.

Example 4.3
Continuing with Example 4.1, we have

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

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

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

where $x \in \mathsf{Var}$. Observe that $\mathsf{or}(list(even), list(odd)) \not\equiv list(nat)$ since $[0, s(0)] \in [\![list(nat)]\!]_\Delta$ and $[0, s(0)] \notin [\![list(even)]\!]_\Delta$ and $[0, s(0)] \notin [\![list(odd)]\!]_\Delta$. ∎

The type constructors and and or will sometimes be written as infix operators, i.e., $\mathsf{and}(R_1, R_2)$ is written as $(R_1\ \mathsf{and}\ R_2)$ and $\mathsf{or}(R_1, R_2)$ as $(R_1\ \mathsf{or}\ R_2)$. A type is atomic if its main constructor is neither and nor or. A type is conjunctive if it is of the form $\mathsf{and}_{1 \le i \le k} A_i$ where each A_i is atomic. By an obvious analogy to propositional logic, for any type R, there is a type of the form $\mathsf{or}_{1 \le i \le m} C_i$ such that each C_i is conjunctive and $R \equiv \mathsf{or}_{1 \le i \le m} C_i$. We call $\mathsf{or}_{1 \le i \le m} C_i$ a disjunctive normal form of R.

A term in a type may contain variables. This lemma states that types are closed under instantiation.

Lemma 4.4
Let $R \in \mathsf{Type}$ and $t \in \mathsf{Term}$. If $t \in [\![R]\!]_\Delta$ then $\sigma(t) \in [\![R]\!]_\Delta$ for any $\sigma \in Sub$. ∎

Type rules in Δ are production rules for a context-free tree grammar (Comon et al. 2002; Gécseg and Steinby 1984). The complement of the denotation of a type is not necessarily closed under instantiation. For an instance, let Δ be defined as in Example 4.1, $x \in \mathsf{Var}$ and $\sigma = \{x \mapsto s(0)\}$. Observe that $x \notin [\![nat]\!]_\Delta$ and $\sigma(x) \in [\![nat]\!]_\Delta$. Since $x \in \mathsf{Term} \setminus [\![nat]\!]_\Delta$ and $\sigma(x) \notin \mathsf{Term} \setminus [\![nat]\!]_\Delta$, $\mathsf{Term} \setminus [\![nat]\!]_\Delta$ is not closed under instantiation and cannot be denoted by a type in Type. The example shows that the family of types is not closed under complement. This explains why set complement is not a type constructor.

Types have also been defined using tree automata (Gécseg and Steinby 1984; Comon et al. 2002), regular term grammars (Dart and Zobel 1992b; Smaus 2001; Lagoon and Stuckey 2001), and regular unary logic programs (Yardeni and Shapiro 1991). A type defined in such a formalism denotes a regular set of ground terms. The meaning function $[\![\cdot]\!]_\Delta$ interprets a type as a set of possible non-ground terms; in particular, it interprets $\mathbf{1}$ as the set of all terms. Type rules are used to propagate type information during analysis. Let x be of type nat and y of type $list(atom)$. Then the type rule $list(\beta) \rightarrowtriangle [\beta|list(\beta)]$ is used to infer that $[x|y]$ is of type $list(\mathsf{or}(nat, atom))$. The type parameter β is not only used as a placeholder but also used in folding heterogeneous types precisely via non-discriminated union operator.

4.3 Type Sequences

During propagation of type information, it is necessary to work with type sequences. A type sequence expression is an expression consisting of type sequences of the same dimension and constructors and and or. Note that constructors and and or are overloaded. The dimension of the type sequence expression is defined to be that of a type sequence in it. Let $R \in \mathsf{Type}$, $\vec{R} \in \mathsf{Type}^*$ and $\mathbf{E}_1$ and $\mathbf{E}_2$ be type sequence expressions. We extend $[\![\cdot]\!]_\Delta$ to type sequence expressions as follows.

$$\begin{aligned} [\![\epsilon]\!]_\Delta &= \{\epsilon\} \\ [\![R \bullet \vec{R}]\!]_\Delta &= [\![R]\!]_\Delta \bullet [\![\vec{R}]\!]_\Delta \\ [\![\mathbf{E}_1 \text{ and } \mathbf{E}_2]\!]_\Delta &= [\![\mathbf{E}_1]\!]_\Delta \cap [\![\mathbf{E}_2]\!]_\Delta \\ [\![\mathbf{E}_1 \text{ or } \mathbf{E}_2]\!]_\Delta &= [\![\mathbf{E}_1]\!]_\Delta \cup [\![\mathbf{E}_2]\!]_\Delta \end{aligned}$$

The relations $\sqsubseteq$ and $\equiv$ on types carry over naturally to type sequence expressions. An occurrence of $\vec{\mathbf{0}}$ (respectively $\vec{\mathbf{1}}$) stands for the type sequence of $\mathbf{0}$'s (respectively $\mathbf{1}$'s) with a dimension appropriate for the occurrence.

5 Abstract Domain

Abstract substitutions in our type analysis are type constraints represented as a set of variable typings which are mappings from variables to types. A variable typing represents the conjunction of primitive type constraints of the form $x \in R$. For instance, the variable typing $\{x \mapsto nat, y \mapsto even\}$ represents the type constraint $(x \in nat) \wedge (y \in even)$. The restriction of a variable typing μ to a set V of variables is defined as

$$\mu \uparrow V = \lambda x.(\text{if } x \in V \text{ then } \mu(x) \text{ else } \mathbf{1})$$

The denotation of a variable typing is given by $\gamma_{\mathsf{VT}} : (V_P \mapsto \mathsf{Type}) \mapsto \wp(Sub)$ defined

$$\gamma_{\mathsf{VT}}(\mu) = \{\theta \mid \forall x \in V_P.(\theta(x) \in [\![\mu(x)]\!]_\Delta)\}$$

For instance, $\gamma_{\mathsf{VT}}(\{x \mapsto nat, y \mapsto list(nat)\}) = \{\theta \mid \theta(x) \in [\![nat]\!]_\Delta \wedge \theta(y) \in [\![list(nat)]\!]_\Delta\}$. The denotation of a set of variable typings is the set union of the denotations of its elements.

Example 5.1
For instance, letting $\mathcal{S} = \{\{x \mapsto nat, y \mapsto list(nat)\}, \{x \mapsto list(nat), y \mapsto nat\}\}$, $\mathcal{S}$ denotes $\{\theta \mid \theta(x) \in [\![nat]\!]_\Delta \wedge \theta(y) \in [\![list(nat)]\!]_\Delta\} \cup \{\theta \mid \theta(x) \in [\![list(nat)]\!]_\Delta \wedge \theta(y) \in [\![nat]\!]_\Delta\}$. ∎

There may be many sets of variable typings that denote the same set of substitutions. Firstly, two different type expressions in Type may denote the same set of terms. For instance, $[\![nat \text{ and } list(\mathbf{1})]\!]_\Delta = [\![\mathbf{0}]\!]_\Delta$ using Δ in Example 4.1. Secondly, an element of a set of variable typings may have a smaller denotation than another. For an example, let $\mathcal{S} = \{\{x \mapsto list(\mathbf{1})\}, \{x \mapsto list(nat)\}\}$. Then $\mathcal{S}$ has the same denotation as one of its proper subset $\mathcal{S}' = \{\{x \mapsto list(\mathbf{1})\}\}$. Those abstract elements that have the same denotation are identified. Let $\preccurlyeq$ on $\wp(V_P \mapsto \mathsf{Type})$ be defined

as $\mathcal{S}_1 \preccurlyeq \mathcal{S}_2 = (\bigcup_{\mu\in\mathcal{S}_1} \gamma_{\mathsf{VT}}(\mu)) \subseteq (\bigcup_{\nu\in\mathcal{S}_2} \gamma_{\mathsf{VT}}(\nu))$. It is a pre-order and induces an equivalence relation $\approx$ on $\wp(V_P \mapsto \mathsf{Type})$: $(\mathcal{S}_1 \approx \mathcal{S}_2) = (\mathcal{S}_1 \preccurlyeq \mathcal{S}_2) \wedge (\mathcal{S}_2 \preccurlyeq \mathcal{S}_1)$. The equivalence classes with respect to $\approx$ are abstract substitutions. Thus, the abstract domain is $\langle ASub^\flat, \sqsubseteq^\flat\rangle$ where

$$\begin{aligned} ASub^\flat &= \wp(V_P \mapsto \mathsf{Type})_{/\approx} \\ \sqsubseteq^\flat &= \preccurlyeq_{/\approx} \end{aligned}$$

$\langle ASub^\flat, \sqsubseteq^\flat\rangle$ is a complete lattice. Its join and meet operators are respectively $[\mathcal{S}_1]_\approx \sqcup^\flat [\mathcal{S}_2]_\approx = [\mathcal{S}_1 \cup \mathcal{S}_2]_\approx$ and $[\mathcal{S}_1]_\approx \sqcap^\flat [\mathcal{S}_2]_\approx = [\mathcal{S}_1^\downarrow \cap \mathcal{S}_2^\downarrow]_\approx$ where $\mathcal{S}_i^\downarrow = \{\mu \in (V_P \mapsto \mathsf{Type}) \mid \exists \nu \in \mathcal{S}_i.(\gamma_{\mathsf{VT}}(\mu) \subseteq \gamma_{\mathsf{VT}}(\nu))\}$. The infimum is $[\emptyset]_\approx$ and the supremum $[\{x \mapsto \mathbf{1} \mid x \in V_P\}]_\approx$. The concretization function $\gamma : ASub^\flat \mapsto \wp(Sub)$ is defined

$$\gamma([\mathcal{S}]_\approx) = \bigcup_{\mu\in\mathcal{S}} \gamma_{\mathsf{VT}}(\mu)$$

The following lemma states that γ satisfies the safety requirement C2.

Lemma 5.2
$\gamma(ASub^\flat)$ is a Moore family. ∎

The definition of $\sqcap^\flat$ is not constructive since the downward closure of a set of variable typings $\mathcal{S}$ can be infinite. For instance, letting $\mathcal{S} = \{\{x \mapsto list(\mathbf{1})\}\}$, $\{x \mapsto list^k(nat)\}$ is in $\mathcal{S}^\downarrow$ for any $k \geq 1$. The following operator $\otimes : \wp(V_P \mapsto \mathsf{Type}) \times \wp(V_P \mapsto \mathsf{Type}) \mapsto \wp(V_P \mapsto \mathsf{Type})$ computes effectively the meet of abstract substitutions.

$$\mathcal{S}_1 \otimes \mathcal{S}_2 = \{\{x \mapsto (\mu(x) \text{ and } \nu(x)) \mid x \in V_P\} \mid \mu \in \mathcal{S}_1 \wedge \nu \in \mathcal{S}_2\}$$

If $\mathcal{S}_1$ and $\mathcal{S}_2$ are finite representatives of two abstract substitutions then $\mathcal{S}_1 \otimes \mathcal{S}_2$ is a finite representative of the meet of the abstract substitutions, which is stated in this lemma.

Lemma 5.3
$\gamma([\mathcal{S}_1 \otimes \mathcal{S}_2]_\approx) = \gamma([\mathcal{S}_1]_\approx \sqcap^\flat [\mathcal{S}_2]_\approx)$. ∎

We will use a fixed renaming substitution Ψ such that $V_P \cap \Psi(V_P) = \emptyset$ and define $V_P' = V_P \cup \Psi(V_P)$. The relation $\approx$, the functions γ_{VT} and γ and the operator $\otimes$ extend naturally to sets of variable typings over V_P'. Let μ be a variable typing, $\mathcal{S}$ a set of variable typings and θ a substitution. We say that θ satisfies μ if $\theta \in \gamma_{\mathsf{VT}}(\mu)$; and we say that θ satisfies $\mathcal{S}$ if $\theta \in \gamma([\mathcal{S}]_\approx)$.

The conditions C1 and C2 are satisfied. C1 holds because $\langle ASub^\flat, \sqsubseteq^\flat, \sqcup^\flat\rangle$ is a complete lattice. C2 is implied by Lemma 5.2.

6 Abstract Operations

The design of our type analysis is completed with four abstract operations required by the abstract semantics given in Section 3. One operation is $\sqcup^\flat$ which is the least upper bound on $\langle ASub^\flat, \sqsubseteq^\flat\rangle$. Let $Id^\flat = [\{\lambda x \in V_P.\mathbf{1}\}]_\approx$. The operation $Id^\flat$ obviously satisfies the condition $C3$ and thus safely abstracts $\{\epsilon\}$ with respect to γ.

Since abstract built-in execution operation $Sys^\flat$ makes use of ancillary operations for abstract unification operation $Uf^\flat$, we present $Uf^\flat$ before $Sys^\flat$.

6.1 Outline of Abstract Unification

The abstract unification operator $Uf^\flat$ takes two atoms and two abstract substitutions and computes an abstract substitution. The computation is reduced to solving a constraint that consists of a set of equations in solved form E and a set of variable typings $\mathcal{S}_i$. The solution to the constraint is a set of variable typings $\mathcal{S}_o$. In order to ensure that $Uf^\flat$ safely abstracts Uf, $\mathcal{S}_o$ is required to describe the set of all those substitutions that satisfy both E and $\mathcal{S}_i$. Let $E = \{x_1 = t_1, \cdots, x_n = t_n\}$. The set $\mathcal{S}_o$ is computed in two steps. In the first step, type information about x_i is used to derive more type information about the variables in t_i. This is a downward propagation since type information is propagated from a term to its sub-terms. The second step propagates type information in the opposite direction. It derives more type information about x_i from type information about the variables in t_i.

For an illustration, let $E = \{x = [w], y = [w]\}$ and $S_i = \{\mu\}$ where $\mu = \{w \mapsto \mathbf{1}, x \mapsto \mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{float}), y \mapsto \mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{integer})\}$. During the downward propagation step, more type information for w is derived from type information for both x and y. Since $\mu(x) = \mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{float})$ and $x = [w]$, $[w]$ is of type $\mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{float})$. Since there is only one type rule for $[\cdot|\cdot]$: $\mathit{list}(\beta) \rightarrowtriangle [\beta|\mathit{list}(\beta)]$, we deduce that w is of type $(\mathit{atom}\ \mathsf{or}\ \mathit{float})$. Similarly, we deduce that w is of type $(\mathit{atom}\ \mathsf{or}\ \mathit{integer})$ since $\mu(y) = \mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{integer})$ and $y = [w]$. So, w is of type $((\mathit{atom}\ \mathsf{or}\ \mathit{float})\ \mathsf{and}\ (\mathit{atom}\ \mathsf{or}\ \mathit{integer}))$ that is equivalent to atom. The derived type atom for w is used to strengthen μ into $\nu = \{w \mapsto \mathit{atom}, x \mapsto \mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{float}), y \mapsto \mathit{list}(\mathit{atom}\ \mathsf{or}\ \mathit{integer})\}$. During the upward propagation step, more type information for both x and y is derived type information for w. Note that $[w]$ is an abbreviation for $[w|[\,]]$. By applying the type rule $\mathit{list}(\beta) \rightarrowtriangle [\,]$, we infer that $[\,]$ is of type $\mathit{list}(\mathbf{0})$. Since $\nu(w) = \mathit{atom}$, we derive that $[w]$ is of type $\mathit{list}(\mathit{atom})$ by applying the type rule $\mathit{list}(\beta) \rightarrowtriangle [\beta|\mathit{list}(\beta)]$. We deduce that both x and y are of type $\mathit{list}(\mathit{atom})$ since $x = [w]$ and $y = [w]$. The derived type $\mathit{list}(\mathit{atom})$ for x and y is used to strengthen ν, resulting in this singleton set of variable typing $S_o = \{\{w \mapsto \mathit{atom}, x \mapsto \mathit{list}(\mathit{atom}), y \mapsto \mathit{list}(\mathit{atom})\}\}$. Both the downward and upward propagation steps in the preceding example produce a single output variable typing from an input variable typing. In more general cases, both steps may yield multiple output variable typings from an input variable typing. We now present in details these two steps.

6.2 Downward Propagation

Downward propagation requires propagating a type R downwards (the structure of) a term $t \in \mathsf{Term}(\Sigma, V'_P)$. Let $\Theta = \{\theta \mid \theta(t) \in [\![R]\!]_\Delta\}$. Propagation of R downwards t calculates a set of variable typings $\mathcal{S}$ (computed as $\mathit{vts}(R, t)$) such that $\Theta \subseteq \gamma([\mathcal{S}]_\approx)$, that is, $\mathcal{S}$ describes the set of all those substitutions that instantiate t to a term of type R. This is done by a case analysis. If $R = \mathbf{1}$ then $\Theta = \mathit{Sub}$ since $\theta(t)$ is in R for

any $\theta \in Sub$. Put $\mathcal{S} = \{\lambda y \in V'_P.\mathbf{1}\}$. Then $\mathcal{S}$ satisfies the condition that $\Theta \subseteq \gamma([\mathcal{S}]_{\approx})$. If $t \in V'_P$ then $\mathcal{S} = \{\lambda y \in V'_P.(\text{if } y = t \text{ then } R \text{ else } \mathbf{1})\}$ satisfies the condition that $\Theta \subseteq \gamma([\mathcal{S}]_{\approx})$. Consider the case $R = (R_1 \text{ or } R_2)$. We have $\Theta = \Theta_1 \cup \Theta_2$ where $\Theta_1 = \{\theta_1 \mid \theta_1(t) \in [\![R]\!]_\Delta\}$ and $\Theta_2 = \{\theta_2 \mid \theta_2(t) \in [\![R_2]\!]_\Delta\}$. We propagate the types R_1 and R_2 downwards t separately, obtaining two sets of variable typings $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\Theta_1 \subseteq \gamma([\mathcal{S}_1]_{\approx})$ and $\Theta_2 \subseteq \gamma([\mathcal{S}_2]_{\approx})$. Put $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$. Then the condition that $\Theta \subseteq \gamma([\mathcal{S}]_{\approx})$ is satisfied. For the case $R = R_1 \text{ and } R_2$, $\mathcal{S} = \mathcal{S}_1 \otimes \mathcal{S}_2$ satisfies the condition that $\Theta \subseteq \gamma([\mathcal{S}]_{\approx})$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ are obtained as above. Consider the remaining case $R = c(R_1, \cdots, R_2)$ and $t = f(t_1, \cdots, t_n)$. Assume that there are k type rules $\Upsilon^1, \cdots, \Upsilon^k$ for c/m and f/n and Υ^j is $c(\beta^j_1, \cdots, \beta^j_m) \twoheadrightarrow f(\tau^j_1, \cdots, \tau^j_n)$. By the definition of $[\![\cdot]\!]_\Delta$, $\Theta = \bigcup_{1 \le j \le k} \Theta_j$ where

$$\begin{array}{rcl}
\Theta_j & = & \{\theta \mid \theta(f(t_1, \cdots, t_n)) \in \{f(s_1, \cdots, s_n) \mid \forall 1 \le i \le n.(s_i \in [\![\kappa^j(\tau^j_i))]\!]_\Delta\}\} \\
& = & \{\theta \mid f(\theta(t_1), \cdots, \theta(t_n)) \in \{f(s_1, \cdots, s_n) \mid \forall 1 \le i \le n.(s_i \in [\![\kappa^j(\tau^j_i))]\!]_\Delta\}\} \\
& = & \{\theta \mid \forall 1 \le i \le n.(\theta(t_i) \in [\![\kappa^j(\tau^j_i))]\!]_\Delta \\
& = & \Theta^j_1 \cap \Theta^j_2 \cap \cdots \cap \Theta^j_n
\end{array}$$

and $\kappa^j = \{\beta^j_1 \mapsto R_1, \cdots, \beta^j_m \mapsto R_m\}$ and $\Theta^j_i = \{\theta \mid \theta(t_i) \in [\![\kappa^j(\tau^j_i)]\!]_\Delta\}$. We obtain $\mathcal{S}$ as follows. We first propagate type $\kappa^j(\tau^j_i)$ downwards term t_i, obtaining a set of variable typings $\mathcal{S}^j_i$. We have that $\Theta^j_i \subseteq \gamma([\mathcal{S}^j_i]_{\approx})$. We then calculate $\mathcal{S}^j = \mathcal{S}^j_1 \otimes \cdots \otimes \mathcal{S}^j_n$ for the type rule Υ^j. The set $\mathcal{S}^j$ satisfies the condition that $\Theta^j \subseteq \gamma([\mathcal{S}^j]_{\approx})$. Finally, we compute $\mathcal{S} = \mathcal{S}^1 \cup \cdots \cup \mathcal{S}^k$. Since $\Theta^j \subseteq \gamma([\mathcal{S}^j]_{\approx})$ and $\Theta = \bigcup_{1 \le j \le k} \Theta^j$, $\mathcal{S}$ satisfies the condition that $\Theta \subseteq \gamma([\mathcal{S}]_{\approx})$. In summary, $\mathcal{S} = vts(R, t)$ where $vts : \mathsf{Type} \times \mathsf{Term}(\Sigma, V'_P) \mapsto \wp(V'_P \mapsto \mathsf{Type})$ is defined

$$\begin{array}{rcl}
vts(\mathbf{1}, t) & = & \{\lambda y \in V'_P.\mathbf{1}\} \\
vts(R, x) & = & \{\lambda y \in V'_P.(\text{if } y = x \text{ then } R \text{ else } \mathbf{1})\} \\
vts((R_1 \text{ and } R_2), t) & = & vts(R_1, t) \otimes vts(R_2, t) \\
vts((R_1 \text{ or } R_2), t) & = & vts(R_1, t) \cup vts(R_2, t) \\
vts(c(R_1, \cdots, R_m), f(t_1, \cdots, t_n)) & = & \\
\multicolumn{3}{r}{\bigcup_{(c(\beta_1, \cdots, \beta_m) \twoheadrightarrow f(\tau_1, \cdots, \tau_n)) \in \Delta} \left(\begin{array}{l} let\ \Bbbk = \{\beta_j \mapsto R_j \mid 1 \le j \le m\} \\ in \\ \bigotimes_{1 \le i \le n} vts(\Bbbk(\tau_i), t_i) \end{array} \right)}
\end{array}$$

where $x \in V'_P$, $f/n \in \Sigma$ and $c/m \in \mathsf{Cons}$. The first one applies when there are multiple applicable alternatives.

The following lemma states that $vts(R, t)$ describes all the substitutions that instantiate t to a term of type R.

Lemma 6.1
For any $R \in \mathsf{Type}$ and $t \in \mathsf{Term}(\Sigma, V'_P)$, $\{\theta \mid \theta(t) \in [\![R]\!]_\Delta\} \subseteq \gamma([vts(R, t)]_{\approx})$. ∎

We now consider the overall downward propagation given a set of variable typings $\mathcal{S}$ and a set of equations in solved form $E = \{x_1 = t_1, \cdots, x_n = t_n\}$. Each variable typing μ in $\mathcal{S}$ is processed separately as follows. We first propagate the type $\mu(x_i)$ downwards t_i. This results in a set of variable typings $vts(\mu(x_i), t_i)$ which describes all the substitutions that instantiate t_i to a term of type $\mu(x_i)$. We then calculate

$\mathcal{S}_\mu = vts(\mu(x_1), t_1) \otimes \cdots \otimes vts(\mu(x_n), t_n)$. The set $\mathcal{S}_\mu$ describes all the substitutions that instantiate t_i to a term of type $\mu(x_i)$ for all $1 \leq i \leq n$. We finally conjoin $\mathcal{S}_\mu$ with $\{\mu\}$, obtaining $\{\mu\} \otimes \mathcal{S}_\mu$ which describes all the substitutions that satisfy both μ and E. After each variable typing in $\mathcal{S}$ is processed, results from different variable typings are joined together using set union. The overall downward propagation function $down : \wp(\mathsf{Eqn}) \times \wp(V'_P \mapsto \mathsf{Type}) \mapsto \wp(V'_P \mapsto \mathsf{Type})$ is defined

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

Example 6.2
Let $V'_P = \{x, y\}$, $\mathcal{S} = \{\{x \mapsto \mathbf{1}, y \mapsto (list(nat)\ \mathsf{or}\ nat)\}\}$ and Δ be that in Example 4.1. We have $vts(list(nat), [x|[\,]]) = \{\{x \mapsto nat, y \mapsto \mathbf{1}\}\}$ and $vts(nat, [x|[\,]]) = \emptyset$. So,

$$vts(list(nat)\ \mathsf{or}\ nat, [x|[\,]]) = \{\{x \mapsto nat, y \mapsto \mathbf{1}\}\}$$

and

$$\begin{aligned} & down(\{y = [x|[\,]]\}, \mathcal{S}) \\ &= \{\{x \mapsto \mathbf{1}, y \mapsto (list(nat)\ \mathsf{or}\ nat)\}\} \otimes \{\{x \mapsto nat, y \mapsto \mathbf{1}\}\} \\ &= \{\mu\} \end{aligned}$$

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

The following lemma states the correctness of downward propagation.

Lemma 6.3
Let $\mathcal{S}' = down(E, \mathcal{S})$. Then $mgu(\theta(E)) \circ \theta \in \gamma([\mathcal{S}']_{\approx})$ for all $\theta \in \gamma([\mathcal{S}]_{\approx})$. ∎

6.3 Upward Propagation

We now consider upward propagation of type information. The key step in upward propagation is to compute a type for a term from those of its variables. We first consider how a type rule $\tau \rightarrowtriangle f(\tau_1, \cdots, \tau_n)$ can be applied to compute a type of $f(t_1, \cdots, t_n)$ from types of its top-level sub-terms $t_1, \cdots, t_n$. Let R_i be the type of t_i. A simplistic approach would compute a type substitution $\Bbbk$ such that $\langle R_1, \cdots, R_n \rangle \sqsubseteq \Bbbk(\langle \tau_1, \cdots, \tau_n \rangle)$ and then return $\Bbbk(\tau)$ as the type of t. However, this leads to loss of precision. Consider the term $[x|y]$ and the type rule $list(\beta) \rightarrowtriangle [\beta|list(\beta)]$. Let the types of x and y be $(even\ \mathsf{or}\ odd)$ and $list(\mathbf{0})$. Then the minimal type substitution $\Bbbk$ such that $\langle even\ \mathsf{or}\ odd, list(\mathbf{0}) \rangle \sqsubseteq \Bbbk(\langle \beta, list(\beta) \rangle)$ is $\Bbbk = \{\beta \mapsto (even\ \mathsf{or}\ odd)\}$. We would obtain $\Bbbk(list(\beta)) = list(even\ \mathsf{or}\ odd)$ as a type of $[x|y]$. A more precise type of $[x|y]$ is $(list(even)\ \mathsf{or}\ list(odd))$. We first compute a set of type substitutions $\mathcal{K}$ such that $\langle R_1, \cdots, R_n \rangle \sqsubseteq \mathsf{or}_{\Bbbk \in \mathcal{K}} \Bbbk(\langle \tau_1, \cdots, \tau_n \rangle)$ and then return $\mathsf{or}_{\Bbbk \in \mathcal{K}} \Bbbk(\tau)$ as a type of $f(t_1, \cdots, t_n)$. Continue with the above example. Let $\mathcal{K} = \{\{\beta \mapsto even\}, \{\beta \mapsto odd\}\}$. Then $\langle even\ \mathsf{or}\ odd, list(\mathbf{0}) \rangle \sqsubseteq \mathsf{or}_{\Bbbk \in \mathcal{K}} \Bbbk(\langle \beta, list(\beta) \rangle)$. We obtain $(list(even)\ \mathsf{or}\ list(odd))$ as a type of $[x|y]$.

Definition 6.4
Let $\tau \in \mathsf{Schm}$, $\vec{\tau} \in \mathsf{Schm}^*$, $R \in \mathsf{Type}$, $\vec{R} \in \mathsf{Type}^*$ and $\mathcal{K} \in \wp(\mathsf{TSub})$. We say that $\mathcal{K}$ is a cover for R and τ iff $R \sqsubseteq \mathsf{or}_{\Bbbk \in \mathcal{K}} \Bbbk(\tau)$. We say that $\mathcal{K}$ is a cover for $\vec{R}$ and $\vec{\tau}$ iff $\vec{R} \sqsubseteq \mathsf{or}_{\Bbbk \in \mathcal{K}} \Bbbk(\vec{\tau})$. ∎

Calculating a cover for a type and a type scheme is a key task in upward propagation of type information. Before defining a function that does the computation, we need some operations on type substitutions.

6.3.1 Operations on Type Substitutions

We first introduce an operation for calculating an upper bound of two type substitutions. It is the point-wise extension of or when both of its operands are mappings from type parameters to types. Define $\curlyvee : \mathsf{TSub} \times \mathsf{TSub} \mapsto \mathsf{TSub}$ as follows.

$$\Bbbk_1 \curlyvee \Bbbk_2 = \begin{cases} \top, & \texttt{if } (\Bbbk_1 = \top) \vee (\Bbbk_2 = \top); \\ \Bbbk_2, & \texttt{else if } (\Bbbk_1 = \bot); \\ \Bbbk_1, & \texttt{else if } (\Bbbk_2 = \bot); \\ \{\beta \mapsto (\Bbbk_1(\beta) \mathsf{ or } \Bbbk_2(\beta)) \mid \beta \in dom(\Bbbk_1) \cup dom(\Bbbk_2)\}, & \texttt{otherwise}. \end{cases}$$

An operation $\curlywedge : \mathsf{TSub} \times \mathsf{TSub} \mapsto \mathsf{TSub}$ that calculates a lower bound of type substitutions is defined dually:

$$\Bbbk_1 \curlywedge \Bbbk_2 = \begin{cases} \bot, & \texttt{if } (\Bbbk_1 = \bot) \vee (\Bbbk_2 = \bot); \\ \Bbbk_2, & \texttt{else if } (\Bbbk_1 = \top); \\ \Bbbk_1, & \texttt{else if } (\Bbbk_2 = \top); \\ \{\beta \mapsto (\Bbbk_1(\beta) \mathsf{ and } \Bbbk_2(\beta)) \mid \beta \in dom(\Bbbk_1) \cap dom(\Bbbk_2)\}, & \texttt{otherwise}. \end{cases}$$

The following lemma states that the operations $\curlyvee$ and $\curlywedge$ indeed compute upper and lower bounds of two type substitutions respectively.

Lemma 6.5
For any $\tau \in \mathsf{Schm}$ and any $\Bbbk_1, \Bbbk_2 \in \mathsf{TSub}$,

(a) $(\Bbbk_1(\tau) \mathsf{ or } \Bbbk_2(\tau)) \sqsubseteq (\Bbbk_1 \curlyvee \Bbbk_2)(\tau)$; and
(b) $(\Bbbk_1(\tau) \mathsf{ and } \Bbbk_2(\tau)) \equiv (\Bbbk_1 \curlywedge \Bbbk_2)(\tau)$.

∎

While the type substitution operation is a meet homomorphism according to Lemma 6.5.(b), it is not a join homomorphism. For an instance, let $\tau = \mathit{list}(\beta)$, $\Bbbk_1 = \{\beta \mapsto \mathit{nat}\}$ and $\Bbbk_2 = \{\beta \mapsto \mathit{list}(\mathit{nat})\}$. Then $\Bbbk_1 \curlyvee \Bbbk_2 = \{\beta \mapsto (\mathit{nat} \mathsf{ or } \mathit{list}(\mathit{nat}))\}$, $(\Bbbk_1 \curlyvee \Bbbk_2)(\tau) = \mathit{list}(\mathit{nat} \mathsf{ or } \mathit{list}(\mathit{nat}))$, and $\Bbbk_1(\tau) \mathsf{ or } \Bbbk_2(\tau) = \mathit{list}(\mathit{nat}) \mathsf{ or } \mathit{list}(\mathit{list}(\mathit{nat}))$. Observe that $(\Bbbk_1(\tau) \mathsf{ or } \Bbbk_2(\tau)) \not\equiv (\Bbbk_1 \curlyvee \Bbbk_2)(\tau)$ since the term $[0, [0]]$ has type $\mathit{list}(\mathit{nat} \mathsf{ or } \mathit{list}(\mathit{nat}))$ but it does not have type $(\mathit{list}(\mathit{nat}) \mathsf{ or } \mathit{list}(\mathit{list}(\mathit{nat})))$.

Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be sets of type substitutions. We say that $\mathcal{K}_1$ and $\mathcal{K}_2$ are equivalent, denoted as $\mathcal{K}_1 \cong \mathcal{K}_2$, iff $(\mathsf{or}_{\Bbbk \in \mathcal{K}_1} \Bbbk(\tau)) \equiv (\mathsf{or}_{\Bbbk \in \mathcal{K}_2} \Bbbk(\tau))$ for any type scheme τ. Define $\bigvee, \bigwedge : \wp(\mathsf{TSub}) \times \wp(\mathsf{TSub}) \mapsto \wp(\mathsf{TSub})$ as the set extensions of $\curlyvee$ and $\curlywedge$ respectively:

$$\begin{aligned}\mathcal{K}_1 \bigvee \mathcal{K}_2 &= \{\Bbbk_1 \curlyvee \Bbbk_2 \mid \Bbbk_1 \in \mathcal{K}_1 \wedge \Bbbk_2 \in \mathcal{K}_2\}\\ \mathcal{K}_1 \bigwedge \mathcal{K}_2 &= \{\Bbbk_1 \curlywedge \Bbbk_2 \mid \Bbbk_1 \in \mathcal{K}_1 \wedge \Bbbk_2 \in \mathcal{K}_2\}\end{aligned}$$

Example 6.6
Let $\mathcal{K}_1 = \{\{\beta_1 \mapsto \mathit{tree}(\mathit{nat}), \beta_2 \mapsto \mathit{nat}\}, \{\beta_1 \mapsto \mathit{list}(\mathit{nat}), \beta_2 \mapsto \mathit{nat}\}\}$ and $\mathcal{K}_2 = \{\{\beta_1 \mapsto \mathit{list}(\mathit{even}), \beta_2 \mapsto \mathit{even}\}\}$. Since $\mathit{even} \sqsubseteq \mathit{nat}$ and $\mathit{list}(\mathit{even}) \sqsubseteq \mathit{list}(\mathit{nat})$, we have

$$\begin{aligned}\mathcal{K}_1 \bigvee \mathcal{K}_2 &= \left\{ \begin{array}{c} \{\beta_1 \mapsto \mathit{tree}(\mathit{nat}) \text{ or } \mathit{list}(\mathit{even}), \beta_2 \mapsto \mathit{nat} \text{ or } \mathit{even}\}, \\ \{\beta_1 \mapsto \mathit{list}(\mathit{nat}) \text{ or } \mathit{list}(\mathit{even}), \beta_2 \mapsto \mathit{nat} \text{ or } \mathit{even}\} \end{array} \right\}\\ &\cong \left\{ \begin{array}{c} \{\beta_1 \mapsto \mathit{tree}(\mathit{nat}) \text{ or } \mathit{list}(\mathit{even}), \beta_2 \mapsto \mathit{nat}\}, \\ \{\beta_1 \mapsto \mathit{list}(\mathit{nat}), \beta_2 \mapsto \mathit{nat}\} \end{array} \right\}\end{aligned}$$

We also have

$$\begin{aligned}\mathcal{K}_1 \bigwedge \mathcal{K}_2 &= \left\{ \begin{array}{c} \{\beta_1 \mapsto (\mathit{tree}(\mathit{nat}) \text{ and } \mathit{list}(\mathit{even})), \beta_2 \mapsto (\mathit{nat} \text{ and } \mathit{even})\}, \\ \{\beta_1 \mapsto (\mathit{list}(\mathit{nat}) \text{ and } \mathit{list}(\mathit{even})), \beta_2 \mapsto (\mathit{nat} \text{ and } \mathit{even})\} \end{array} \right\}\\ &\cong \left\{ \begin{array}{c} \{\beta_1 \mapsto (\mathit{tree}(\mathit{nat}) \text{ and } \mathit{list}(\mathit{even})), \beta_2 \mapsto \mathit{even}\}, \\ \{\beta_1 \mapsto \mathit{list}(\mathit{even}), \beta_2 \mapsto \mathit{even}\} \end{array} \right\}\\ &\cong \{\{\beta_1 \mapsto \mathit{list}(\mathit{even}), \beta_2 \mapsto \mathit{even}\}\}\end{aligned}$$

since $(\mathit{tree}(\mathit{nat}) \text{ and } \mathit{list}(\mathit{even})) \equiv \mathbf{0}$. ∎

A cover for a type sequence and a type scheme sequence can be computed compositionally according to the following lemma.

Lemma 6.7
Let $\mathcal{K}_1, \mathcal{K}_2 \in \wp(\mathsf{TSub})$, $R \in \mathsf{Type}$, $\tau \in \mathsf{Schm}$, $\vec{R} \in \mathsf{Type}^*$ and $\vec{\tau} \in \mathsf{Schm}^*$ such that $\|\vec{R}\| = \|\vec{\tau}\|$. If $R \sqsubseteq \mathsf{or}_{\Bbbk_1 \in \mathcal{K}_1} \Bbbk_1(\tau)$ and $\vec{R} \sqsubseteq \mathsf{or}_{\Bbbk_2 \in \mathcal{K}_2} \Bbbk_2(\vec{\tau})$ then $R \bullet \vec{R} \sqsubseteq \mathsf{or}_{\Bbbk \in (\mathcal{K}_1 \curlyvee \mathcal{K}_2)} \Bbbk(\tau \bullet \vec{\tau})$. ∎

6.3.2 Calculating a Cover

We now consider how to compute a cover $\mathcal{K}$ for a type R and a type scheme τ. In the case $R = \mathbf{1}$, $\mathcal{K} = \{\top\}$ is a cover since $\top(\tau) = \mathbf{1}$; and $\mathcal{K} = \{\bot\}$ is a cover in the case $R = \mathbf{0}$ since $\bot(\tau) = \mathbf{0}$. Consider the case $R = (R_1 \text{ or } R_2)$, a cover $\mathcal{K}_j$ can be recursively computed for R_j and τ for $j = 1, 2$. We have that $R_j \sqsubseteq \mathsf{or}_{\Bbbk \in \mathcal{K}_j} \Bbbk(\tau)$ and hence that $(R_1 \text{ or } R_2) \sqsubseteq \mathsf{or}_{\Bbbk \in (\mathcal{K}_1 \cup \mathcal{K}_2)} \Bbbk(\tau)$. So, the union of $\mathcal{K}_1$ and $\mathcal{K}_2$ is a cover for R and τ. Consider the case $R = (R_1 \text{ and } R_2)$. A cover $\mathcal{K}_j$ can be recursively computed for R_j and τ for $j = 1, 2$. Let $\mathcal{K} = \mathcal{K}_1 \bigwedge \mathcal{K}_2 = \{\Bbbk_1 \curlywedge \Bbbk_2 \mid \Bbbk_1 \in \mathcal{K}_1 \wedge \Bbbk_2 \in \mathcal{K}_2\}$. Then $\mathsf{or}_{\Bbbk \in \mathcal{K}}(\tau) = \mathsf{or}_{\Bbbk_1 \in \mathcal{K}_1 \wedge \Bbbk_2 \in \mathcal{K}_2}(\Bbbk_1 \curlywedge \Bbbk_2)(\tau)$. By Lemma 6.5.(b), $\mathsf{or}_{\Bbbk \in \mathcal{K}}(\tau) = \mathsf{or}_{\Bbbk_1 \in \mathcal{K}_1 \wedge \Bbbk_2 \in \mathcal{K}_2}(\Bbbk_1(\tau) \text{ and } \Bbbk_2(\tau))$ and hence $\mathsf{or}_{\Bbbk \in \mathcal{K}}(\tau) = (\mathsf{or}_{\Bbbk_1 \in \mathcal{K}_1} \Bbbk_1(\tau)) \text{ and } (\mathsf{or}_{\Bbbk_2 \in \mathcal{K}_2} \Bbbk_2(\tau))$. So, $\mathcal{K} = \mathcal{K}_1 \bigwedge \mathcal{K}_2$ is a cover for R and τ. In the case R is atomic and τ is a type parameter, $\mathcal{K} = \{\{\tau \mapsto R\}\}$ is a cover for R and τ. In the remaining case, $R = c(R_1, \cdots, R_m)$ and $\tau = d(\beta_1, \cdots, \beta_k)$. If $c/m = d/k$

then $\{\{\beta_j \mapsto R_j \mid 1 \leq j \leq m\}\}$ is a cover. Otherwise $\{\top\}$ is a cover. In summary, the function that computes a cover is $cover : \mathsf{Type} \times \mathsf{Schm} \mapsto \wp(\mathsf{TSub})$ defined

$$\begin{array}{rcl} cover(\mathbf{1}, \tau) & = & \{\top\} \\ cover(\mathbf{0}, \tau) & = & \{\bot\} \\ cover((R_1 \text{ or } R_2), \tau) & = & cover(R_1, \tau) \cup cover(R_2, \tau) \\ cover((R_1 \text{ and } R_2), \tau) & = & cover(R_1, \tau) \curlywedge cover(R_2, \tau) \\ cover(R, \beta) & = & \{\{\beta \mapsto R\}\} \\ cover(c(R_1, \cdots, R_m), d(\beta_1, \cdots, \beta_k)) & = & \\ \multicolumn{3}{r}{\begin{cases} \mathit{if}\ (c/m) = (d/k)\ \mathit{then}\ \{\{\beta_j \mapsto R_j \mid 1 \leq j \leq m\}\} \\ \mathit{else}\ \{\top\} \end{cases}} \end{array}$$

Example 6.8
Let Cons be given in Example 4.1. Then,

$$\begin{array}{rcl} & & cover((list(nat) \text{ and } tree(even)), list(\beta)) \\ & = & cover(list(nat), list(\beta)) \curlywedge cover(tree(even), list(\beta)) \\ & \cong & \{\{\beta \mapsto nat\}\} \curlywedge \{\top\} \\ & = & \{\{\beta \mapsto nat\}\} \end{array}$$

∎

The following lemma states that $cover(R, \tau)$ is a cover for R and τ.

Lemma 6.9
Let $\tau \in \mathsf{Schm}$, $R \in \mathsf{Type}$ and $\mathcal{K} = cover(R, \tau)$. Then $R \sqsubseteq \mathsf{or}_{\mathbb{k} \in \mathcal{K}} \mathbb{k}(\tau)$. ∎

6.3.3 Computing a Type

The type of a term t is computed from those of its variables in a bottom-up manner. The types of the variables are given by a variable typing μ. For a compound term $t = f(t_1, \cdots, t_n)$, a type R_i is first computed from t_i and μ for each $1 \leq i \leq n$. Each type rule for f/n is applied to compute a type of t. Types resulting from all type rules for f/n are conjoined using and. The result is a type of t since conjunctions of two or more types of t is also a type of t. For a type rule $\tau \twoheadrightarrow f(\tau_1, \cdots, \tau_n)$, a cover $\mathcal{K}_i$ for R_i and τ_i is computed for each $1 \leq i \leq n$. Joining covers for $1 \leq i \leq n$ obtains a cover $\mathcal{K}$ for $\langle R_1, \cdots, R_n \rangle$ and $\langle \tau_1, \cdots, \tau_n \rangle$. The type that is computed from the type rule is $\mathsf{or}_{\mathbb{k} \in \mathcal{K}} \mathbb{k}(\tau)$. Define $type : \mathsf{Term}(\Sigma, V'_P) \times (V'_P \mapsto \mathsf{Type}) \mapsto \mathsf{Type}$ by

$$\begin{array}{rcl} type(x, \mu) & = & \mu(x) \\ type(f(t_1, \cdots, t_n), \mu) & = & \mathsf{and}_{\tau \twoheadrightarrow f(\tau_1, \cdots, \tau_n) \in \Delta} (\mathsf{or}_{\mathbb{k} \in (\Upsilon_{1 \leq i \leq n}\, cover(type(t_i, \mu), \tau_i))} \mathbb{k}(\tau)) \end{array}$$

Example 6.10
Let $\mu = \{x \mapsto nat, y \mapsto (list(nat) \text{ or } nat)\}$, $\mathbb{k}_1 = \{\beta \mapsto nat\}$ and $\mathbb{k}_2 = \{\beta \mapsto \mathbf{0}\}$. By the definition of $cover$, $cover(nat, \beta) = \{\mathbb{k}_1\}$ and $cover(list(\mathbf{0}), list(\beta)) = \{\mathbb{k}_2\}$. By the definition of $type$, $type(x, \mu) = nat$ and $type([\,], \mu) = list(\mathbf{0})$. So, $type([x|[\,]], \mu) = (\mathbb{k}_1 \curlyvee \mathbb{k}_2)(list(\beta)) = list(nat)$. ∎

The following lemma says that $type(t, \mu)$ is a type that contains all the instances of t under the substitutions described by μ.

Lemma 6.11
Let $t \in \mathsf{Term}(\Sigma, V'_P)$ and $\mu \in (V'_P \mapsto \mathsf{Type})$. Then $\theta(t) \in [\![type(t, \mu)]\!]_\Delta$ for all $\theta \in \gamma_{\mathsf{VT}}(\mu)$. ∎

6.3.4 Upward Propagation

We are now ready to present the overall upward propagation. For a set $\mathcal{S}$ of variable typings and a set E of equations in solved form, upward propagation strengthens each variable typing μ in $\mathcal{S}$ as follows. For each equation $x = t$ in E, $type(t, \mu)$ is a type of x if variables occurring in t satisfy μ. The overall upward propagation is performed by a function $up : \wp(\mathsf{Eqn}) \times \wp(V'_P \mapsto \mathsf{Type}) \mapsto \wp(V'_P \mapsto \mathsf{Type})$ defined

$$up(E, \mathcal{S}) = \bigcup_{\mu \in \mathcal{S}} \left\{ \lambda x \in V'_P. \left(\begin{array}{l} if\ \exists t.(x = t) \in E \\ then\ (\mu(x)\ \mathsf{and}\ type(t, \mu)) \\ else\ \mu(x) \end{array} \right) \right\}$$

Example 6.12
Continue with Example 6.10. We have

$$\begin{array}{rcl} up(\{y = [x|[\,]]\}, \{\mu\}) & = & \{\mu[y \mapsto ((list(nat)\ \mathsf{or}\ nat)\ \mathsf{and}\ list(nat))]\} \\ & \approx & \{\{x \mapsto nat, y \mapsto list(nat)\}\} \end{array}$$

∎

The correctness of upward propagation is ensured by this lemma.

Lemma 6.13
Let $\mathcal{S} \in \wp(V'_P \mapsto \mathsf{Type})$ and $E \in \wp(\mathsf{Eqn})$. Then $mgu(\theta(E)) \circ \theta \in \gamma([up(E, \mathcal{S})]_\approx)$ for all $\theta \in \gamma([\mathcal{S}]_\approx)$. ∎

6.4 Abstract Unification

Algorithm 6.14 defines the abstract unification operation $Uf^\flat$. Given two atoms $a_1, a_2 \in \mathsf{Atom}_P$ and two abstract substitutions $[\mathcal{S}_1]_\approx, [\mathcal{S}_2]_\approx \in ASub^\flat$, it first applies the renaming substitution Ψ to a_1 and $\mathcal{S}_1$ and computes $E_0 = eq \circ mgu(\Psi(a_1), a_2)$. If $E_0 = fail$, it returns $[\emptyset]_\approx$ – the smallest abstract substitution which describes the empty set of substitutions. Otherwise, it calculates $\mathcal{S}'_0 = \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2$ where $\biguplus : (\Psi(V_P) \mapsto \mathsf{Type}) \times (V_P \mapsto \mathsf{Type}) \mapsto (V'_P \mapsto \mathsf{Type})$. A variable typing represents a conjunctive type constraint. If μ and ν have disjoint domains then $\mu \cup \nu$ represents the conjunction of μ and ν. The first operand of $\biguplus$ is a set of variable typings over $\Psi(V_P)$ and the second operand a set of variable typings over V_P. The result of $\biguplus$ describes the set of all the substitutions that satisfy both of its two operands. Thus $\mathcal{S}'_0$ describes the set of all the substitutions that satisfy both $\Psi(\mathcal{S}_1)$ and $\mathcal{S}_2$. Note that $\mathcal{S}'_0 \in \wp(V'_P \mapsto \mathsf{Type})$. The abstract unification operation then calls a function $solve : \mathsf{Eqn} \times \wp(V'_P \mapsto \mathsf{Type}) \mapsto \wp(V'_P \mapsto \mathsf{Type})$ to perform downward

and upward propagations. The result $\mathcal{S}_1'$ is $solve(E_0, \mathcal{S}_0')$ which describes the set of all the substitutions that satisfy both E_0 and $\mathcal{S}_0'$. Finally, it calls a function $rest : \wp(V_P' \mapsto \mathsf{Type}) \mapsto \wp(V_P \mapsto \mathsf{Type})$ to restrict each variable typing in $\mathcal{S}_1'$ to V_P.

Algorithm 6.14

$$
\begin{array}{rcl}
Uf^\flat(a_1, [\mathcal{S}_1]_\approx, a_2, [\mathcal{S}_2]_\approx) & = & \left\{ \begin{array}{ll} let & E_0 = eq \circ mgu(\Psi(a_1), a_2)\ in \\ if & E_0 \neq fail \\ then & [rest \circ solve(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2)]_\approx \\ else & [\emptyset]_\approx \end{array} \right. \\
\mathcal{S}_1 \biguplus \mathcal{S}_2 & = & \{\mu \cup \nu \mid \mu \in \mathcal{S}_1 \wedge \nu \in \mathcal{S}_2\} \\
rest(\mathcal{S}) & = & \{\mu \uparrow V_P \mid \mu \in \mathcal{S} \wedge \forall x \in V_P'.(\mu(x) \not\equiv \mathbf{0})\} \\
solve(E, \mathcal{S}) & = & up(E, down(E, \mathcal{S}))
\end{array}
$$

The function *rest* removes those variable typings that denote the empty set of substitutions and projects the remaining variable typings onto V_P.

Example 6.15
Let $V_P = \{x\}$, $\Psi(x) = y$, $a_1 = p(x)$, $a_2 = p([x|[\,]])$, $\mathcal{S}_1 = \{\{x \mapsto (list(nat)\ \mathsf{or}\ nat)\}\}$, and $\mathcal{S}_2 = \{\{x \mapsto \mathbf{1}\}\}$. Then $E_0 = \{y = [x|[\,]]\}$ and $\Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2 = \mathcal{S}$ with $\mathcal{S}$ being that in Example 6.2. By Examples 6.2 and 6.12,

$$
\begin{array}{rcl}
solve(E_0, \mathcal{S}) & = & up(E_0, down(E_0, \mathcal{S})) = up(E_0, \{\mu\}) \\
& = & \{\{x \mapsto nat, y \mapsto list(nat)\}\}
\end{array}
$$

with μ given in Example 6.12. ∎

The following theorem states that $Uf^\flat$ safely abstracts Uf with respect to γ.

Theorem 6.16
For any $[\mathcal{S}_1]_\approx, [\mathcal{S}_2]_\approx \in ASub^\flat$ and any $a_1, a_2 \in \mathsf{Atom}_P$,

$$
Uf(a_1, \gamma([\mathcal{S}_1]_\approx), a_2, \gamma([\mathcal{S}_2]_\approx)) \quad \subseteq \quad \gamma(Uf^\flat(a_1, [\mathcal{S}_1]_\approx, a_2, [\mathcal{S}_2]_\approx))
$$

∎

6.5 Abstract Built-in Execution Operation

For each built-in, it is necessary to specify an operation that transforms an input abstract substitution to an output abstract substitution. These operations are given in Table 1 where abstract substitutions are displayed as sets of variable typings. The primitive types *integer*, *float*, *number*, *string*, *atom* and *atomic* have their usual denotations in Prolog. Observe that $number = (integer\ \mathsf{or}\ float)$ and $atomic = (number\ \mathsf{or}\ atom)$.

Unification t_1=t_2 is modeled by $\lambda\mathcal{S}.solve(mgu(t_1, t_2), \mathcal{S})$. Let θ be the program state before the execution of t_1=t_2 and assume that θ satisfies $\mathcal{S}$. The program state after the execution of t_1=t_2 is $mgu(\theta(t_1), \theta(t_2)) \circ \theta$ and satisfies $solve(mgu(t_1, t_2), \mathcal{S})$. Built-ins such as </2 succeed only if their arguments satisfy certain type constraints. Such type constraints are conjoined with the input abstract substitution to obtain the output abstract substitution. For instance, the execution of t_1<t_2 in an input

program state θ succeeds only if θ instantiates both t_1 and t_2 to numbers. So, the abstract operation for t_1<t_2 is $f_3 = \lambda\mathcal{S}.(\mathcal{S} \otimes vts(number, t_1) \otimes vts(number, t_2))$ where *vts* defined in Section 6.2 is extended to deal with built-in types. The extension is straightforward and omitted. For another instance, $format(t_1)$ succeeds only if t is an atom, or a list of character codes or a string in its input program state. The above type constraint is obtained as $vts(atom \text{ or } list(integer) \text{ or } string, t_1)$. The type $list(integer)$ describes lists of character codes since character codes are integers. The type checking built-ins such as $atom/1$ are modeled in the same way. Built-ins such as `@</2` do not instantiate their arguments or check types of their arguments. They are modeled by the identity function $\lambda\mathcal{S}.\mathcal{S}$. The built-in $fail/0$ never succeeds and hence is modeled by the constant function that always returns $\emptyset$.

Consider a built-in to which a call $p(t_1, \cdots, t_n)$ will definitely instantiate t_i to a term of type R_i upon success. The type R_i can be propagated downwards t_i, resulting in a set of variable typings. The input abstract substitution can be strengthened by this set of variable typings to give the output abstract substitution. For an instance, consider $name(t_1, t_2)$. Upon success, t_1 is either an atom or an integer and t_2 is a string. So, $name(t_1, t_2)$ is modeled by $\lambda\mathcal{S}.(\mathcal{S} \otimes vts(atom \text{ or } integer, t_1) \otimes vts(string, t_2))$. The built-ins $length(t_1, t_2)$ and $compare(t_1, t_2, t_3)$ fall into this category.

Consider the built-in $var(t)$. The execution of $var(t)$ succeeds in a program state θ iff $\theta(t)$ is a variable. All types that contains variables are equivalent to $\mathbf{1}$. Thus, the built-in $var(t)$ is modeled by $\lambda\mathcal{S}.\{\mu \mid \mu \in \mathcal{S} \wedge type(t, \mu) \equiv \mathbf{1}\}$. The output abstract substitution contains only those variable typings in which t has no type smaller than $\mathbf{1}$. The built-in $nonvar(t)$ is modeled by the identity function $\lambda\mathcal{S}.\mathcal{S}$ since a term being a non-variable does not provide any information about its type unless non-freeness is defined as a type. So is the built-in $ground(t)$ since a term being ground says nothing about its type unless groundness is defined as a type. The operation for the built-in $compound(t)$ makes use of the property that a compound term is not atomic. It removes from the input abstract substitution any variable typing in which t is atomic.

7 Implementation

We have implemented a prototype of our type analysis in SWI-Prolog. The prototype is a meta-interpreter using ground representations for program variables. The prototype supports the primitive types *integer*, $float$, *number*, *string*, *atom* and *atomic* with their usual denotations in Prolog.

7.1 Examples

Example 7.1
The following is the intersect program that computes the intersection of two lists and its analysis result. Lists are defined in Example 4.1. Abstract substitutions are displayed as comments. The abstract substitution associated with the entry point

Predicate	*Operation*
$abort$, $fail$, $false$	$\lambda\mathcal{S}.\emptyset$
!, t_1@<t_2, t_1@>t_2, t_1=<@t_2, t_1@>=t_2, t_1\==t_2, t_1\=t_2, $display(t_1)$, $ground(t_1)$, $listing$, $listing(t_1)$, nl, $nonvar(t_1)$, $portray\_clause(t_1)$, $print(t_1)$, $read(t_1)$, $repeat$, $true$, $write(t_1)$, $writeq(t_1)$	$\lambda\mathcal{S}.\mathcal{S}$
$compound(t)$	$\lambda\mathcal{S}.(\{\mu \mid \mu \in \mathcal{S} \wedge type(t,\mu) \not\sqsubseteq atomic\})$
$atom(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(atom, t))$
$atomic(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(atomic, t))$
$float(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(float, t))$
$erase(t)$, $integer(t)$, $tab(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(integer, t))$
$number(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(number, t))$
$put(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(atom \text{ or } integer, t))$
$string(t)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(string, t))$
$var(t)$	$\lambda\mathcal{S}.\{\mu \mid \mu \in \mathcal{S} \wedge type(t,\mu) \equiv \mathbf{1}\}$
t_1=t_2, t_1==t_2	$\lambda\mathcal{S}.solve(mgu(t_1,t_2),\mathcal{S})$
$format(t_1)$, $format(t_1,t_2)$, $format(t_0,t_1,t_2)$	f_4
t_1<t_2, $t_1 > t_2$, t_1=<t_2, t_1>=t_2, t_1=:=t_2, t_1=\=t_2, $is(t_1,t_2)$	f_3
$length(t_1,t_2)$	f_1
$compare(t_1,t_2,t_3)$	$\lambda\mathcal{S}.(\mathcal{S} \otimes vts(atom, t_1))$
$name(t_1,t_2)$	f_2

Table 1. *Abstract operations for built-ins where* $f_1 = \lambda\mathcal{S}.(\mathcal{S} \otimes vts(list(\mathbf{1}), t_1) \otimes vts(integer, t_2))$, $f_2 = \lambda\mathcal{S}.(\mathcal{S} \otimes vts(atom \text{ or } integer, t_1) \otimes vts(string, t_2))$, $f_3 = \lambda\mathcal{S}.(\mathcal{S} \otimes vts(number, t_1) \otimes vts(number, t_2))$, *and* $f_4 = \lambda\mathcal{S}.(\mathcal{S} \otimes vts(atom \text{ or } list(integer) \text{ or } string, t_1))$.

of the query is an analysis input whilst all other abstract substitutions are analysis outputs. Sets are displayed as lists. A binding $V \mapsto T$ is written as V/T, or as *or* and and as *and*. The code for the predicate member/2 is omitted.

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

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

```
    % [X/float,Xs/list(atom or float),Ys/list(atom or integer)]]
  \+ member(X,Ys),
    %[[X/float,Xs/list(atom or float),Ys/list(atom or integer)],
    % [X/atom,Xs/list(atom or float),Ys/list(atom or integer)]]
  intersect(Xs,Ys,Zs).
    %[[X/float,Xs/list(atom or float),Ys/list(atom or integer),
    %  Zs/list(atom)],
    % [X/atom,Xs/list(atom or float),Ys/list(atom or integer),
    %  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 $((atom \text{ or } float) \text{ and } (atom \text{ or } integer))$ is equivalent to the type *atom*. Without the set operators and and or in their type languages, previous type analyses with *a priori* type definitions cannot produce a result as precise as the above. ∎

Example 7.2
The following is a program $p/1$. The analysis result is displayed with the typing binding $x \mapsto \mathbf{1}$ omitted for any variable x.

```
p([]). % [[]]

p([X|Y]) :-
    % [[]]
  integer(X),
    % [[X/integer]]
  p(Y).
    % [[X/integer, Y/list(or(atom, integer))]]

p([X|Y]) :-
    % [[]]
  atom(X),
    % [[X/atom]]
  p(Y).
    % [[X/atom, Y/list(or(atom, integer))]]

:-  % [[]]
   p(U).
    % [[U/list(or(atom, integer))]]
```

The result captures precisely type information in the success set of the program, that is, U is a list consisting of integers and atoms upon success of p(U).[1] ∎

[1] This example was provided by an anonymous referee of a previous version of this paper.

During analysis of a program, the analyzer repeatedly checks if two sets of variable typings are equivalent and if a set of variable typings contains redundant elements. Both of these decision problems are reduced to checking if a given type denotes the empty set of terms.

7.2 Emptiness of Types

Type rules in Δ are production rules for a context-free tree grammar in restricted form (Gécseg and Steinby 1984). According to (Lu and Cleary 1998), if $\mathbf{1}$ denotes the set of all ground terms instead of all terms then each type denotes a regular tree language. We now show how an algorithm in (Lu and Cleary 1998) can be used for checking the emptiness of types. We first extend the type language with the complement operator $\sim$ and define $\mathsf{eType} = \mathsf{Term}(\mathsf{Cons} \cup \{\sim, \mathsf{and}, \mathsf{or}, \mathbf{1}, \mathbf{0}\}, \emptyset)$. Observe that $\mathsf{Type} \subset \mathsf{eType}$ and that $[\cdot]_\Delta$ is not defined for elements in $\mathsf{eType} \setminus \mathsf{Type}$. Since the algorithm in (Lu and Cleary 1998) was developed for checking the emptiness of types that denote sets of ground terms, we need to justify its application by closing the gap between the two different semantics of types. This is achieved by extending the signature Σ with an extra constant ϱ ($\varrho \in \Sigma$) that is used to encode variables in terms. Use of extended signatures in analysis of logic programs can be traced to (Gallagher et al. 1995) where extra constants are used to encode non-ground terms. In fact, by introducing an infinite set of extra constants one can obtain an isomorphism between the set of all terms in the original signature and the set of the ground terms in the extended signature.

Definition 7.3
The meaning of a type in eType is given by a function $\langle\!\langle\cdot\rangle\!\rangle_\Delta : \mathsf{eType} \mapsto \wp(\mathsf{Term}(\Sigma \cup \{\varrho\}, \emptyset)$.

$$
\begin{array}{rcl}
\langle\!\langle \mathbf{1} \rangle\!\rangle_\Delta & = & \mathsf{Term}(\Sigma \cup \{\varrho\}, \emptyset) \\
\langle\!\langle \mathbf{0} \rangle\!\rangle_\Delta & = & \emptyset \\
\langle\!\langle \sim R \rangle\!\rangle_\Delta & = & \langle\!\langle \mathbf{1} \rangle\!\rangle_\Delta \setminus \langle\!\langle R \rangle\!\rangle_\Delta \\
\langle\!\langle \mathsf{and}(R_1, R_2) \rangle\!\rangle_\Delta & = & \langle\!\langle R_1 \rangle\!\rangle_\Delta \cap \langle\!\langle R_2 \rangle\!\rangle_\Delta \\
\langle\!\langle \mathsf{or}(R_1, R_2) \rangle\!\rangle_\Delta & = & \langle\!\langle R_1 \rangle\!\rangle_\Delta \cup \langle\!\langle R_2 \rangle\!\rangle_\Delta \\
\langle\!\langle c(R_1, \cdots, R_m) \rangle\!\rangle_\Delta & = & \\
\multicolumn{3}{r}{\bigcup_{(c(\beta_1,\cdots,\beta_m) \twoheadrightarrow f(\tau_1,\cdots,\tau_n)) \in \Delta} \left(\begin{array}{l} let\ \Bbbk = \{\beta_j \mapsto R_j \mid 1 \le j \le m\} \\ in \\ \{f(t_1, \cdots, t_n) \mid \forall 1 \le i \le n . t_i \in \langle\!\langle \Bbbk(\tau_i) \rangle\!\rangle_\Delta\} \end{array} \right)}
\end{array}
$$

∎

There are two differences between $\langle\!\langle\cdot\rangle\!\rangle_\Delta$ and $[\cdot]_\Delta$. Firstly, $\sim$ is interpreted as set complement under $\langle\!\langle\cdot\rangle\!\rangle_\Delta$ whilst it has no denotation under $[\cdot]_\Delta$. Type constructor $\sim$ can be interpreted as set complement by $\langle\!\langle\cdot\rangle\!\rangle_\Delta$ because $\langle\!\langle R \rangle\!\rangle_\Delta$ is a regular tree language for any $R \in \mathsf{eType}$ (Lu and Cleary 1998). It cannot be interpreted as set complement by $[\cdot]_\Delta$ because the complement of $[\![R]\!]_\Delta$ is not closed under instantiation. Secondly, the universal type $\mathbf{1}$ denotes $\mathsf{Term}(\Sigma, \mathsf{Var})$ in $[\cdot]_\Delta$ whilst it denotes $\mathsf{Term}(\Sigma \cup \{\varrho\}, \emptyset)$ in $\langle\!\langle\cdot\rangle\!\rangle_\Delta$. An implication is that a type denotes a set of terms closed under instantiation under $[\cdot]_\Delta$ whilst it denotes a set of ground terms under $\langle\!\langle\cdot\rangle\!\rangle_\Delta$.

Let $\chi : \mathsf{Term}(\Sigma, \mathsf{Var}) \mapsto \mathsf{Term}(\Sigma \cup \{\varrho\}, \emptyset)$ be defined $\chi(x) = \varrho$ for all $x \in \mathsf{Var}$ and $\chi(f(t_1, \cdots, t_n)) = f(\chi(t_1), \cdots, \chi(t_n))$. The function $\chi(\cdot)$ transforms a term into a ground term by replacing all variables in the term with the same constant ϱ. The following theorem states that, given a term t and a type R, the membership of t in $[\![R]\!]_\Delta$ is equivalent to that of $\chi(t)$ in $\langle\!\langle R \rangle\!\rangle_\Delta$.

Theorem 7.4
For any term t in $\mathsf{Term}(\Sigma, \mathsf{Var})$ and any type R in Type, $t \in [\![R]\!]_\Delta$ iff $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$. ∎

As a consequence, checking the emptiness of a type under $[\![\cdot]\!]_\Delta$ can be reduced to checking the emptiness of the type under $\langle\!\langle \cdot \rangle\!\rangle_\Delta$, and vice versa. Therefore, whether $[\![R]\!]_\Delta = \emptyset$ can be decided by employing the algorithm developed in (Lu and Cleary 1998) that checks if $\langle\!\langle R \rangle\!\rangle_\Delta = \emptyset$. The following corollary of the theorem allows us to reduce a type inclusion test under $[\![\cdot]\!]_\Delta$ to a type inclusion test under $\langle\!\langle \cdot \rangle\!\rangle_\Delta$.

Corollary 7.5
For any $R_1, R_2 \in \mathsf{Type}$, $[\![R_1]\!]_\Delta \subseteq [\![R_2]\!]_\Delta$ iff $\langle\!\langle R_1 \rangle\!\rangle_\Delta \subseteq \langle\!\langle R_2 \rangle\!\rangle_\Delta$. ∎

In order to reduce the decision problems to the emptiness of types, we need to extend the syntax for type sequence expressions with the operator $\sim$ and $\langle\!\langle \cdot \rangle\!\rangle_\Delta$ to type sequences. The expression $\sim\mathbf{E}$ is a type sequence expression whenever $\mathbf{E}$ is a type sequence expression. Let R be a type in eType, $\vec{R}$ a type sequence in eType^*, $\mathbf{E}_1$ and $\mathbf{E}_2$ be type sequence expressions. Define $\langle\!\langle \epsilon \rangle\!\rangle_\Delta = \{\epsilon\}$, $\langle\!\langle R \bullet \vec{R} \rangle\!\rangle_\Delta = \langle\!\langle R \rangle\!\rangle_\Delta \bullet \langle\!\langle \vec{R} \rangle\!\rangle_\Delta$, $\langle\!\langle \mathbf{E}_1 \ \mathsf{and} \ \mathbf{E}_2 \rangle\!\rangle_\Delta = \langle\!\langle \mathbf{E}_1 \rangle\!\rangle_\Delta \cap \langle\!\langle \mathbf{E}_2 \rangle\!\rangle_\Delta$, $\langle\!\langle \mathbf{E}_1 \ \mathsf{or} \ \mathbf{E}_2 \rangle\!\rangle_\Delta = \langle\!\langle \mathbf{E}_1 \rangle\!\rangle_\Delta \cup \langle\!\langle \mathbf{E}_2 \rangle\!\rangle_\Delta$ and $\langle\!\langle \sim \mathbf{E} \rangle\!\rangle_\Delta = \langle\!\langle \vec{\mathbf{1}} \rangle\!\rangle_\Delta - \langle\!\langle \mathbf{E} \rangle\!\rangle_\Delta$. It can be shown that both Theorem 7.4 and Corollary 7.5 carry over to type sequence expressions that do not contain $\sim$.

Set inclusion and $\langle\!\langle \cdot \rangle\!\rangle_\Delta$ induces an equivalence between types and type sequence expressions. Let $R_1 \doteq R_2$ iff $\langle\!\langle R_1 \rangle\!\rangle_\Delta = \langle\!\langle R_2 \rangle\!\rangle_\Delta$ and $\mathbf{E}_1 \doteq \mathbf{E}_2$ iff $\langle\!\langle \mathbf{E}_1 \rangle\!\rangle_\Delta = \langle\!\langle \mathbf{E}_2 \rangle\!\rangle_\Delta$. The following function eliminates the complement operator $\sim$ over type sequence expressions.

$$
\begin{aligned}
push(\sim(\mathsf{or}_{i\in I}\mathbf{E}_i)) &= \mathsf{and}_{i\in I} push(\sim\mathbf{E}_i) \\
push(\sim(\mathsf{and}_{i\in I}\mathbf{E}_i)) &= \mathsf{or}_{i\in I} push(\sim\mathbf{E}_i) \\
push(\sim(R_1, R_2, \cdots, R_k)) &= \mathsf{or}_{1\le l\le k}(\underbrace{\mathbf{1}, \cdots, \mathbf{1}}_{l-1}, \sim R_l, \underbrace{\mathbf{1}, \cdots, \mathbf{1}}_{k-l}) \quad \text{for } k \ge 1
\end{aligned}
$$

It follows from De Morgan's law and the definition of $\langle\!\langle \cdot \rangle\!\rangle_\Delta$ that $push(\sim\mathbf{E}) \doteq \sim\mathbf{E}$. Note that the complement operator $\sim$ does not apply to any type sequence expression in $push(\sim\mathbf{E})$; it only applies to type expressions. Let $R \in \mathsf{eType}$ and define $etype(R) = (\langle\!\langle R \rangle\!\rangle_\Delta = \emptyset)$. The formula $etype(R)$ is true iff $R \doteq \mathbf{0}$ is true. By Theorem 7.4, if $R \in \mathsf{Type}$ then $etype(R)$ is true iff $R \equiv \mathbf{0}$ is true.

7.3 Equivalence between Sets of Variable Typings

An indispensable operation in a static analyzer is to check if a fixpoint has been reached. This operation reduces to checking if two sets of variable typings denote

the same set of concrete substitutions. This equivalence test is reduced to checking emptiness of types as follows. Let $V_P = \{x_1, \cdots, x_k\}$ and $\mathcal{S}_1, \mathcal{S}_2 \in \wp(V_P \mapsto \mathsf{Type})$. By definition, $\mathcal{S}_1 \approx \mathcal{S}_2$ iff $\bigcup_{\mu\in\mathcal{S}_1} \gamma_{\mathsf{VT}}(\mu) \subseteq \bigcup_{\nu\in\mathcal{S}_2} \gamma_{\mathsf{VT}}(\nu)$ and $\bigcup_{\nu\in\mathcal{S}_2} \gamma_{\mathsf{VT}}(\nu) \subseteq \bigcup_{\mu\in\mathcal{S}_1} \gamma_{\mathsf{VT}}(\mu)$. Suppose $\mathcal{S}_1 = \{\mu_1, \mu_2, \cdots, \mu_m\}$ and $\mathcal{S}_2 = \{\nu_1, \nu_2, \cdots, \nu_n\}$. We construct $\vec{R}_1, \vec{R}_2, \cdots, \vec{R}_m$ and $\vec{T}_1, \vec{T}_2, \cdots, \vec{T}_n$ as follows. $\vec{R}_i = \langle \mu_i(x_1), \mu_i(x_2), \cdots, \mu_i(x_k)\rangle$ and $\vec{T}_j = \langle \nu_j(x_1), \nu_j(x_2), \cdots, \nu_j(x_k)\rangle$. Then $\bigcup_{\mu\in\mathcal{S}_1} \gamma_{\mathsf{VT}}(\mu) \subseteq \bigcup_{\nu\in\mathcal{S}_2} \gamma_{\mathsf{VT}}(\nu)$ is true iff $\mathsf{or}_{1\leq i}\vec{R}_i \sqsubseteq \mathsf{or}_{1\leq j}\vec{T}_j$ is true. By Corollary 7.5, $\mathsf{or}_{1\leq i}\vec{R}_i \sqsubseteq \mathsf{or}_{1\leq j}\vec{T}_j$ is true iff $(\mathsf{or}_{1\leq i}\vec{R}_i)$ and $\sim(\mathsf{or}_{1\leq j}\vec{T}_j) \doteq \vec{\mathbf{0}}$ is true. The latter can be reduced to emptiness of types as shown in (Lu and Cleary 1998).

Example 7.6
Let Δ be given as in Example 4.1, $V_P = \{x, y\}$, $\mathcal{S}_1 = \{\mu_1, \mu_2\}$ and $\mathcal{S}_2 = \{\mu_3\}$ where

$$\begin{aligned}
\mu_1 &= \{x \mapsto \mathit{list}(\mathit{even}), y \mapsto \mathit{list}(\mathit{nat})\}\\
\mu_2 &= \{x \mapsto \mathit{list}(\mathit{odd}), y \mapsto \mathit{list}(\mathit{nat})\}\\
\mu_3 &= \{x \mapsto \mathit{list}(\mathit{even}) \ \mathsf{or}\ \mathit{list}(\mathit{odd}), y \mapsto \mathit{list}(\mathit{nat})\}
\end{aligned}$$

The truth value of $(\gamma_{\mathsf{VT}}(\mu_1) \cup \gamma_{\mathsf{VT}}(\mu_2)) \subseteq \gamma_{\mathsf{VT}}(\mu_3)$ is decided by testing emptiness of types as follows. Let

$$\begin{aligned}
\vec{R}_1 &= \langle \mathit{list}(\mathit{even}), \mathit{list}(\mathit{nat})\rangle\\
\vec{R}_2 &= \langle \mathit{list}(\mathit{odd}), \mathit{list}(\mathit{nat})\rangle\\
\vec{T}_1 &= \langle \mathit{list}(\mathit{even}) \ \mathsf{or}\ \mathit{list}(\mathit{odd}), \mathit{list}(\mathit{nat})\rangle
\end{aligned}$$

Then $(\gamma_{\mathsf{VT}}(\mu_1) \cup \gamma_{\mathsf{VT}}(\mu_2)) \subseteq \gamma_{\mathsf{VT}}(\mu_3)$ iff $(\vec{R}_1 \ \mathsf{or}\ \vec{R}_2)$ and $\sim\vec{T}_1 \doteq \vec{\mathbf{0}}$ which, by replacing $\sim\vec{T}_1$ with $\mathit{push}(\sim\vec{T}_1)$ and distributing and over or, is equivalent to the conjunction of the following formulas.

$$\begin{aligned}
\langle \mathit{list}(\mathit{even}), \mathit{list}(\mathit{nat})\rangle \ \mathsf{and}\ \langle \mathbf{1}, \sim\mathit{list}(\mathit{nat})\rangle &\doteq \vec{\mathbf{0}}\\
\langle \mathit{list}(\mathit{odd}), \mathit{list}(\mathit{nat})\rangle \ \mathsf{and}\ \langle \mathbf{1}, \sim\mathit{list}(\mathit{nat})\rangle &\doteq \vec{\mathbf{0}}\\
\langle \mathit{list}(\mathit{even}), \mathit{list}(\mathit{nat})\rangle \ \mathsf{and}\ \langle \sim\mathit{list}(\mathit{even}) \ \mathsf{and}\ \sim\mathit{list}(\mathit{odd}), \mathbf{1}\rangle &\doteq \vec{\mathbf{0}}\\
\langle \mathit{list}(\mathit{odd}), \mathit{list}(\mathit{nat})\rangle \ \mathsf{and}\ \langle \sim\mathit{list}(\mathit{even}) \ \mathsf{and}\ \sim\mathit{list}(\mathit{odd}), \mathbf{1}\rangle &\doteq \vec{\mathbf{0}}
\end{aligned}$$

The first of the above holds iff either $\mathit{list}(\mathit{even}) \doteq \mathbf{0}$ or $\mathit{list}(\mathit{nat})$ and $\sim\mathit{list}(\mathit{nat}) \doteq \mathbf{0}$, both of which are emptiness tests on types. Since $\mathit{list}(\mathit{nat})$ and $\sim\mathit{list}(\mathit{nat}) \doteq \mathbf{0}$, the first formula is decided to be true. The other three can be decided to be true similarly. Therefore, $(\gamma_{\mathsf{VT}}(\mu_1) \cup \gamma_{\mathsf{VT}}(\mu_2)) \subseteq \gamma_{\mathsf{VT}}(\mu_3)$ holds. In a similar way, we can show that $\gamma_{\mathsf{VT}}(\mu_3) \subseteq (\gamma_{\mathsf{VT}}(\mu_1) \cup \gamma_{\mathsf{VT}}(\mu_2))$ holds. So, $\mathcal{S}_1 \approx \mathcal{S}_2$. ∎

7.4 Redundancy Removal

For the sake of an efficient implementation, an abstract substitution $[\mathcal{S}]_{\approx}$ should be represented by a set of variable typings that does not contain redundancy. A set of variable typings can be redundant in two ways. Firstly, a variable typing μ in $\mathcal{S}$ may denote the empty set of substitutions i.e., $\mu(x) \equiv \mathbf{0}$ for some $x \in V_P$.

Secondly, a variable typing μ in $\mathcal{S}$ can be subsumed by other variable typings in that $\gamma_{\mathsf{VT}}(\mu) \subseteq \bigcup_{\nu \in \mathcal{S} \wedge \nu \neq \mu} \gamma_{\mathsf{VT}}(\nu)$. In both cases, $\mathcal{S} \setminus \{\mu\}$ and $\mathcal{S}$ denote the same set of substitutions and μ can be removed from $\mathcal{S}$. Suppose $V_P = \{x_1, \cdots, x_k\}$. The detection of $\gamma_{\mathsf{VT}}(\mu) = \emptyset$ reduces to $etype(\mu(x_1)) \vee \cdots \vee etype(\mu(x_k))$ while the detection of $\gamma_{\mathsf{VT}}(\mu) \subseteq \bigcup_{\nu \in \mathcal{S} \wedge \nu \neq \mu} \gamma_{\mathsf{VT}}(\nu)$ can be reduced to checking emptiness of types as in Section 7.3.

Example 7.7
Let Δ be given Example 4.1, $V_P = \{x, y\}$, $\mathcal{S} = \{\mu_1, \mu_2, \mu_3\}$ where

$$\begin{aligned} \mu_1 &= \{x \mapsto list(even), y \mapsto list(nat)\} \\ \mu_2 &= \{x \mapsto list(odd), y \mapsto list(nat)\} \\ \mu_3 &= \{x \mapsto list(nat), y \mapsto list(nat)\} \end{aligned}$$

We now show how μ_1 is decided to be redundant in $\mathcal{S}$. Let

$$\begin{aligned} \vec{R}_1 &= \langle list(even), list(nat)\rangle \\ \vec{R}_2 &= \langle list(odd), list(nat)\rangle \\ \vec{R}_3 &= \langle list(nat), list(nat)\rangle \end{aligned}$$

Then $\gamma_{\mathsf{VT}}(\mu_1) \subseteq \gamma_{\mathsf{VT}}(\mu_2) \cup \gamma_{\mathsf{VT}}(\mu_3)$ holds iff $\vec{R}_1$ and $\sim(\vec{R}_2$ or $\vec{R}_3) \doteq \vec{\mathbf{0}}$ holds iff $\vec{R}_1$ and $\sim\vec{R}_2$ and $\sim\vec{R}_3 \doteq \vec{\mathbf{0}}$ holds. The latter, after replacing $\sim\vec{R}_2$ and $\sim\vec{R}_3$ with $push(\sim\vec{R}_2)$ and $push(\sim\vec{R}_3)$ respectively and distributing and over or, is equivalent to the conjunction of the following formulas.

$$\begin{aligned} \langle list(even), list(nat)\rangle \text{ and } \langle \mathbf{1}, \sim list(nat)\rangle \text{ and } \langle \mathbf{1}, \sim list(nat)\rangle &\doteq \vec{\mathbf{0}} \\ \langle list(even), list(nat)\rangle \text{ and } \langle \mathbf{1}, \sim list(nat)\rangle \text{ and } \langle \sim list(nat), \mathbf{1}\rangle &\doteq \vec{\mathbf{0}} \\ \langle list(even), list(nat)\rangle \text{ and } \langle \sim list(odd), \mathbf{1}\rangle \text{ and } \langle \mathbf{1}, \sim list(nat)\rangle &\doteq \vec{\mathbf{0}} \\ \langle list(even), list(nat)\rangle \text{ and } \langle \sim list(odd), \mathbf{1}\rangle \text{ and } \langle \sim list(nat), \mathbf{1}\rangle &\doteq \vec{\mathbf{0}} \end{aligned}$$

Each of the above can be decided to be true by testing emptiness of types as in Example 7.6. Therefore, μ_1 is redundant in $\mathcal{S}$. In a similar way, μ_2 is decided to be redundant in $\mathcal{S} \setminus \{\mu_1\}$. So, $\mathcal{S} \approx \{\mu_3\}$. That μ_3 is not redundant in $\mathcal{S}$ is decided similarly. ∎

7.5 Tabling

The operations in our type analysis are complex because of non-deterministic type definitions and non-discriminative union at both the level of types and the level of abstract substitutions. The equality of two abstract substitutions in an analysis without these features can be done in linear time (Horiuchi and Kanamori 1988; Kanamori and Horiuchi 1985; Kanamori and Kawamura 1993; Lu 1995). The same operation is exponential in our type analysis because deciding the emptiness of a type is exponential. This indicates that our type analysis could be much more time consuming.

As shown later, there is a high degree of repetition in emptiness checks during

Table 2. *Time Performance*

Program	Program Points	Goal	Time
browse	103	q	110
cs_r	277	pgenconfig(C)	661
disj_r	132	top(K)	171
dnf	77	go	200
kalah	228	play(G, R)	590
life	100	life(MR, MC, LC, SFG)	89
meta	89	interpret(G)	50
neural	341	go	250
nbody	375	go(M, G)	281
press	318	test_press(X, Y)	161
serialize	37	go(S)	80
zebra	43	zebra(E, S, J, U, N, Z, W)	40
	Sum = 2120		Sum = 2683

the analysis of a program. Making use of this observation, we have reduced time increase to 15% on average using a simple tabling technique. We memoize each call to $etype(R)$ and its success or failure by asserting a fact $\$etype\_tabled(R, Ans)$. The fact $\$etype\_tabled(R, yes)$ (resp. $\$etype\_tabled(R, no)$) indicates that $etype(R)$ has been called before and $etype(R)$ succeeded (resp. failed). The tabled version of $etype(R)$ is $etype\_tabled(R)$. It first checks if a fact $\$etype\_tabled(R, Ans)$ exists. If so, the call $etype\_tabled(R)$ succeeds or fails immediately. Otherwise, it calls $etype(R)$ and memoizes its success or failure.

We now present some experimental results with the prototype analyzer. The experiments were done with a Pentium (R) 4 CPU 2.26 GHz running GNU/Linux and SWI-Prolog-5.2.13.

7.5.1 Time Performance

Table 2 shows analysis time on a suite of benchmark programs. Each row except the last one corresponds to a test case. The first three columns contain the name, the size of the program in terms of the number of program points and the top-level goal. The top abstract substitution which contains no type information is used as the input abstract substitution for each test case. These test cases will be used in subsequent tables where only the program names are given. The fourth column gives analysis time in milliseconds. The time is obtained by running the analyzer ten times on the test case and averaging analysis time from these runs. Timing data in other tables are also obtained in this way. The table shows that the analyzer takes an average of 1.27 milliseconds per program point.

7.5.2 Repetition of Emptiness Checks

Table 3 shows that there is a high degree of repetition in emptiness checks during analysis. Each test case corresponds to a row of the table. The first column of the row is the name of the program, the second is the total number of emptiness checks that occur during analysis. The third column gives the number of different types

Table 3. *Repetition of Emptiness Checks*

Program	Total Checks	Different Checks	Degree of Repetition
browse	3050	64	47.65
cs_r	23846	53	449.92
disj_r	4500	37	121.62
dnf	6290	9	698.88
kalah	31182	86	362.58
life	3277	24	136.54
meta	468	13	36.00
neural	7985	131	60.95
nbody	8567	39	219.66
press	1734	23	75.39
serialize	2019	37	54.56
zebra	947	22	43.04
			Ave.=192.23

that are checked for emptiness. The fourth column gives the average repetition of emptiness checks, which is the ratio of the second and the third columns. While the total number of emptiness checks can be very large for a test case, the number of different emptiness checks is small, exhibiting a high degree of repetition in emptiness checks. The repetition of the emptiness checks ranges from 36.00 to 698.88. The weighted repetition average is about 192.23. This motivated the use of tabling to reduce the time spent on emptiness checks.

7.5.3 Effect of Tabulation

Table 4 illustrates the effect of tabling. Statistics are obtained by running the analyzer with and without tabling. For both experiments, we measured analysis time and time spent on emptiness checks. The table shows that tabling reduces analysis time to $\frac{1}{2.6}$. The table also gives the proportion of analysis time that is spent on emptiness checks. An average of 53% of analysis time is spent on emptiness checks without tabling while only a negligible portion of analysis time is spent on emptiness checks with tabling.

7.6 Cost and Effect of Precision Improvement Features

The precision improvement features in our type analysis all incur some performance penalty. In order to evaluate the effect of these features, we also implemented a simplified type analysis. The simplified analysis is obtained by removing the precision improvement features from the full-fledged analysis. In the simplified analysis, type expressions do not contain the constructors or or and; an abstract substitution is simply a variable typing; and non-deterministic type definitions are disallowed. Function overloading is still allowed. Abstract operations are simplified accordingly. For instance, since ($list(list(nat))$ or $list(nat)$) is not in the type language of the simplified analysis, the least upper bound of $list(list(nat))$ and $list(nat)$ is $list(\mathbf{1})$.

Table 4. *Effect of Tabulation*

Program	With Tabling			Without Tabling		
	Analysis Time	Check Time	Proportion	Analysis Time	Check Time	Proportion
browse	110	10	0.09	269	129	0.47
cs_r	661	0	0	1700	891	0.52
disj_r	171	0	0	359	157	0.43
dnf	200	0	0	581	363	0.62
kalah	590	0	0	1939	1124	0.57
life	89	0	0	210	117	0.55
meta	50	0	0	60	21	0.35
neural	250	30	0.12	731	469	0.64
nbody	281	0	0	620	314	0.50
press	161	0	0	250	29	0.11
serialize	80	0	0	179	82	0.45
zebra	40	0	0	70	31	0.44
	Sum=2683		Ave.=0.01	Sum=6968		Ave.=0.53

Table 5. *Cost and Effect of Precision Improvement Features*

Program	Simplified Analysis Time	Full-fledged Analysis Time	Time Ratio	Precision Ratio
browse	100.00	110.00	0.90	0.74
cs_r	589.00	661.00	0.89	0.77
disj_r	151.00	171.00	0.88	0.73
dnf	190.00	200.00	0.95	0.00
kalah	420.00	590.00	0.71	0.49
life	89.00	89.00	1.00	0.35
meta	40.00	50.00	0.80	0.05
neural	200.00	250.00	0.80	0.49
nbody	231.00	281.00	0.82	0.23
press	159.00	161.00	0.98	0.95
serialize	80.00	80.00	1.00	0.83
zebra	40.00	40.00	1.00	0.35
			Ave. =0.85	Ave. =0.54

The least upper bound operation on abstract substitutions is the point-wise extension of the least upper bound operation on types.

Table 5 compares two type analyses. The two analyses are performed on each test case with the same input type information. The input abstract substitution for the full-fledged analysis is a singleton set of a variable typing. The corresponding abstract substitution for the simplified analysis is the variable typing. For each test case, the table gives analysis times by the two analyzers and their ratio. The relative performance of the two analyzers varies with the test case. On average, the simplified analysis takes 85 percent of the analysis time of the full-fledged type analysis. This illustrates that the precision improvement features does not substantially increase analysis time.

The fifth column in Table 5 gives information about the effect of the precision improvement features. For each program, it lists the ratio of the number of the pro-

gram points at which the full-fledged analysis derives more precise type information than the simplified analysis over the number of all program points. Whether or not these features improve analysis precision depends on the program that is analysed. For some programs like dnf and meta, there is little or no improvement. For some other programs like press and serialize, there is a substantial improvement. On average, the full-fledged analysis derives more precise type information at 54% of the program points in a program. This indicates that the precision improvement features is cost effective.

7.7 Termination

The abstract domain of our type analysis contains chains of infinite length, which may lead to non-termination of the analysis of a program.

Example 7.8
Let the program consist of a single clause `p(x) :-` $\circledast$ `p([x])` where $\circledast$ is a label of a program point. Let the query be of the form `:- p(u)` with u being of type nat. Then x is a term of type $list^i(nat)$ at the i^{th} time the execution reaches the program point $\circledast$. Thus, the chain of the abstract substitutions at the program point $\circledast$ is

$$\begin{array}{l}\{\{x \mapsto list(nat)\}\}\\ \{\{x \mapsto list(nat)\}, \{x \mapsto list(list(nat))\}\}\\ \vdots\\ \{\{x \mapsto list^j(nat)\} \mid 0 < j \leq k\}\\ \vdots\end{array}$$

which is infinite. The program is an instance of polymorphic recursion (Kahrs 1996) which is prohibited in ML. ∎

The analyzer uses a canonical representation of types and a depth abstraction to ensure termination. A conjunctive type is compact if it contains no duplicated type atoms. A type in disjunctive normal form is compact if it contains no duplicated conjuncts and all of its conjuncts are compact. A type is canonical if it is in disjunctive normal form, it is compact and all arguments of its type atoms are canonical. For every type R, a canonical equivalent of R – a canonical type R_c such that $R_c \equiv R$ – can be obtained as follows. A disjunctive normal form R' of R is first computed. Each argument of each type atom in R' is then replaced with its canonical equivalent, resulting in a type R''. Finally, R_c is obtained by deleting duplicate type atoms in each conjunct of R'' and then deleting duplicate conjuncts. Let $cn(R)$ denote the canonical equivalent of R obtained by the above procedure. For instance, $cn(tree(tree(list(\mathbf{1}) \text{ or } list(\mathbf{1})))) = tree(tree(list(\mathbf{1})))$.

Let R be a type. An atomic sub-term A of R is both a sub-term of R and a type atom. The depth of A in R is the number of the occurrences of type constructors in Cons on the path from the root of R to but excluding the root of A. Thus, the depth of the only occurrence of $list(nat)$ in $tree(tree(list(even) \text{ or } list(list(nat))))$ is 3 and the depth of the only occurrence of $list(even)$ in the same type is 2. Note

that type constructors and and or are ignored in determining the depth of A in R. If the depth of A in R is k then A is called an atomic sub-term of R at depth k. The depth of R is defined as the maximum of the depths of all its atomic sub-terms.

Definition 7.9
Let R be a type and k a positive integer. The depth k abstraction of R, denoted as $d_k(R)$, is the result of replacing each argument of any atomic sub-term of R at depth k by $\mathbf{1}$. ∎

For instance,

$$\begin{aligned} & d_2(\mathit{tree}(\mathit{tree}(\mathit{list}(\mathit{even})\ \mathsf{or}\ \mathit{list}(\mathit{list}(\mathit{nat})))))\\ &= \mathit{tree}(\mathit{tree}(\mathit{list}(\mathbf{1})\ \mathsf{or}\ \mathit{list}(\mathbf{1}))) \end{aligned}$$

During analysis, the abstract substitution for a program point is initialized to the empty set of variable typings. It is updated by adding new variable typings and removing redundant ones. The analyzer ensures termination as follows. For each program point, the analyzer determines a depth k the first time a non-empty set of variable typings $\mathcal{S}_0$ is added. The depth k is the maximum of the depths of the types occurring in $\mathcal{S}_0$ plus some fixed constant k_0 with $k_0 \geq 0$. After that, each time a set of variable typings $\mathcal{S}$ is added, each type R occurring in $\mathcal{S}$ is replaced by $cn(d_k(cn(R)))$. The above abstraction preserves analysis correctness because $R' \sqsubseteq d_k(R')$ and $cn(R) \equiv R$. The number of depth k abstractions of the canonical types occurring in the abstract substitution is bounded and so is the number of variable typings in the abstract substitution. This ensures termination.

Example 7.10
Continue Example 7.8 and let $k_0 = 1$. We have $\mathcal{S}_0 = \{\{x \mapsto \mathit{list}(\mathit{nat})\}\}$ and hence $k = 2$ since the depth of the only type $\mathit{list}(\mathit{nat})$ in $\mathcal{S}_0$ is 1. The chain of the abstract substitutions for the program point $\circledast$ is

$$\begin{aligned} &\{\{x \mapsto \mathit{list}(\mathit{nat})\}\}\\ &\{\{x \mapsto \mathit{list}(\mathit{nat})\}, \{x \mapsto \mathit{list}(\mathit{list}(\mathit{nat}))\}\}\\ &\{\{x \mapsto \mathit{list}(\mathit{nat})\}, \{x \mapsto \mathit{list}(\mathit{list}(\mathit{nat}))\}, \{x \mapsto \mathit{list}(\mathit{list}(\mathit{list}(\mathbf{1})))\}\} \end{aligned}$$

The last in the chain is the final abstract substitution for the program point $\circledast$. ∎

8 Related Work

There is a rich literature on type inference analysis for logic programs. Type analyses in (Frühwirth et al. 1991; Gallagher and de Waal 1994; Gallagher and Puebla 2002; Mishra 1984; Zobel 1987) are performed without *a priori* type definitions. They generate regular tree grammars, or type graphs (Van Hentenryck et al. 1995; Janssens and Bruynooghe 1992) or set constraints (Heintze and Jaffar 1990; Heintze and Jaffar 1992) as type definitions. These different formalisms for expressing type definitions are equivalent. A type graph is equivalent to a regular tree grammar such that a production rule in the grammar corresponds to a subgraph that is composed of a node and its successors in the graph. For a system of set constraints,

there is a regular tree grammar that generates the least solution to the system of set constraints, and vice versa (Cousot and Cousot 1995). The production rules in a regular tree grammar are similar to type rules used in our analysis but are not parameterized. This kind of analysis is useful for compiler-time optimizations and transformations but inferred type definitions can be difficult for the programmer to interpret. Like those in (Horiuchi and Kanamori 1988; Kanamori and Horiuchi 1985; Barbuti and Giacobazzi 1992; Kanamori and Kawamura 1993; Codish and Demoen 1994; Lu 1995; Saglam and Gallagher 1995; Codish and Lagoon 2000; Lu 1998; Hill and Spoto 2002), our type analysis is performed with *a priori* type definitions. The type expressions it infers are formed of given type constructors. Since the meaning of a type constructor is given by *a priori* type definitions that are well understood to the programmer, the inferred types are easier for the programmer to interpret and thus they are more useful in an interactive programming environment.

The type analyses with *a priori* type definitions in (Horiuchi and Kanamori 1988; Kanamori and Horiuchi 1985; Kanamori and Kawamura 1993; Lu 1995) are based on top-down abstract interpretation frameworks. They are performed with a type description of possible queries as an input and are thus goal-dependent. They infer for each program point a type description of all the program states that might be obtained when the execution of the program reaches that program point. These are also characteristics of our analysis. However, these analyses do not support non-deterministic type definitions or non-discriminative union at the levels of types and abstract substitutions. The analysis in (Lu 1998) traces non-discriminative union at the level of abstract substitutions but not at the level of types. In addition, it does not allow non-determinism in type definitions. The above mentioned top-down type analyses with *a priori* type definitions approximate non-discriminative union of two types by their least upper bound. The least upper bound may have a strictly larger denotation than the set union of the denotations of the two types since set union is not a type constructor. Thus, our type analysis is strictly more precise than (Horiuchi and Kanamori 1988; Kanamori and Horiuchi 1985; Kanamori and Kawamura 1993; Lu 1995; Lu 1998).

The type analyses with *a priori* type definitions in (Barbuti and Giacobazzi 1992; Codish and Demoen 1994; Saglam and Gallagher 1995; Codish and Lagoon 2000; Hill and Spoto 2002) are based on bottom-up abstract interpretaion frameworks. They infer a type description of the success set of the program. The inferred type description is a set of type atoms each of which is a predicate symbol applied to a tuple of types. Some general remarks can be made about the differences between our analysis and these analyses. Firstly, our analysis is goal-dependent while these analyses are goal-independent. Secondly, our analysis allows non-deterministic type definitions that are disallowed by these analyses. Consequently, more natual typings are allowed by our type analysis than by these analyses. However, non-deterministic type definitions also make abstract operations in our analysis more complex than in these analyses. Thirdly, like our analysis, these analyses can express non-discriminative union at the level of predicates. For example, the two type atoms $p(list(integer))$ and $p(tree(integer))$ express the same information as $\langle p(x), x \in (list(integer) \text{ or } tree(integer))\rangle$ in our type analysis. How-

ever, these analyses except an informal proposal in (Barbuti and Giacobazzi 1992) cannot trace non-discriminative union at the level of arguments, which leads to imprecise analysis results. For instance, the inferred type for the concrete atom $p([1,[1]])$ is $p(list(\mathbf{1}))$ according to (Codish and Demoen 1994; Saglam and Gallagher 1995; Codish and Lagoon 2000; Hill and Spoto 2002) and the main proposal in (Barbuti and Giacobazzi 1992). The inferred type $p(list(\mathbf{1}))$ is less precise than $\langle p(x), x \in list(integer \text{ or } list(integer))\rangle$ which is inferred by our type analysis. Lastly, as a minor note, set intersection is not used as a type constructor in these type analyses except (Hill and Spoto 2002). The two type clauses $x(list(\beta)) \leftarrow$ and $x(tree(\beta)) \leftarrow$ in an abstract substitution of (Hill and Spoto 2002) indicates that x is both a list and a tree. Some comparisons on other aspects between our type analysis and these bottom-up analyses are in order.

Barbuti and Giacobazzi (1992) infer polymorphic types of Horn clause logic programs using a bottom-up abstract interpretation framework (Barbuti et al. 1993). The type description of the success set of a Horn logic program is computed as the least fixed-point of an abstract immediate consequence operator associated with the program. The abstract immediate consequence operator is defined in terms of abstract unification and abstract application. Abstract unification computes an abstract substitution given a term and a type. Abstract application computes a type given an abstract substitution and a term. Both computations are derivations of a Prolog program that is derived from *a priori* type definitions. The inferred type description describes only part of the success set of the program though abstract operations can be modified so that the type description approximates the whole success set. Ill-typed atoms are not described by the type description. Nor are those well-typed atoms that possess only ill-typed SLD resolutions. An SLD resolution is ill-typed if any of its selected atoms is ill-typed. Their type definitions are slightly different form ours. For instance, they define the type of the empty list [ ] as $[\,] \rightarrow list(\bot)$ which is equivalent to $list(\mathbf{0})\rightarrowtriangle[\,]$ in our notation whilst the empty list [ ] is typed by $list(\beta)\rightarrowtriangle[\,]$ in our analysis. Barbuti and Giacobazzi also informally introduced and exemplified an associative, commutative and idempotent operator $\cup$ that expresses non-deterministic union at the level of types. However, abstract unification and abstract application operations for this modified domain of types are not given. In addition, it requires changing type definitions, for instance, from $cons(\beta, list(\beta)) \rightarrow list(\beta)$ to $cons(\alpha, list(\beta)) \rightarrow list(\alpha \cup \beta)$. Barbuti and Giacobazzi's analysis captures more type dependency than ours. This is achieved through type parameters. For instance, the type description for the program $\{p(X,[X]) \leftarrow\}$ is $\{p(\alpha, list(\alpha))\}$. Abstract unification of the query $p(X,Y)$ with the only type atom in the type description yields the abstract substitution $\{X \mapsto \alpha, Y \mapsto list(\alpha)\}$, This kind of type dependency will be lost in our analysis. The use of type parameters and the use of non-discriminative union are orthogonal to each other and it is an interesting topic for future research to combine them for more analysis precision.

Codish and Demoen (1994) apply abstract compilation (Hermenegildo et al. 1992) to infer type dependencies by associating each type with an incarnation of the abstract domain Prop (Marriott and Søndergaard 1989). The incarnations of Prop define meanings of types and capture interactions between types. The type depen-

dencies of a logic program is similar to the type description of the program inferred by the type analysis of Barbuti and Giacobazzi (1992) except that the type dependencies describe the whole success set of the program. Codish and Lagoon (2000) improve (Codish and Demoen 1994) 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 sub-terms. It has the flavor of set union. Nevertheless, it does not denote the set union. For an example, the term $[1,[1]]$ has type $\mathit{list}(\mathit{integer}) \oplus \mathit{list}(\mathit{list}(\mathit{integer}))$ according to (Codish and Lagoon 2000) while it has type $\mathit{list}(\mathit{integer}$ or $\mathit{list}(\mathit{integer}))$ in our type analysis. Like (Barbuti and Giacobazzi 1992), type analyses in (Codish and Demoen 1994; Codish and Lagoon 2000) capture type dependency via type parameters. In addition, they have the desired property of condensing which our analysis does not have.

Hill and Spoto (2002) provide a method that enriches an abstract domain with type dependency information. The enriched domain contain elements like $(x \in \mathit{nat}) \rightarrow (y \in \mathit{list}(\mathit{nat}))$ meaning if x has type nat then y has type $\mathit{list}(\mathit{nat})$. Each element in the enriched domain is represented as a logic program. Type analysis is performed by abstract compilation. Their approach to improving precision of type analysis is different from ours. Their domain can express type dependencies that ours cannot whilst our domain can express non-discriminative union at the level of types but theirs cannot. Hill and Spoto do not take subtyping into account in their design of abstract operations possibly because subtyping is outside the focus of their work.

Gallagher and de Waal (1994) approximates the success set of the program by a unary regular logic program (Yardeni and Shapiro 1991). This analysis infers both type definitions and types and is incorporated into the Ciao System (Hermenegildo et al. 1999). Saglam and Gallagher (1995) extend (Gallagher and de Waal 1994) by allowing the programmer to supply deterministic type definitions for some function symbols. The supplied type definitions are used to transform the program and the transformed program is analyzed as in (Gallagher and de Waal 1994). An interesting topic for further study is to integrate non-deterministic type definitions and non-discriminative union into (Saglam and Gallagher 1995) and evaluate their impact on analysis precision and analysis cost.

Finally, it is also worthy mentioning work on directional types (Bronsard et al. 1992; Aiken and Lakshman 1994; Boye and Malúszynski 1996; Charatonik and Podelski 1998; Rychlikowski and Truderung 2001). Aiken and Lakshman (1994) present an algorithm for automatic checking directional types of logic programs. Directional types describe both the structure of terms and the directionality of predicates. A directional type for a predicate p/n is of the form $\tau_I \rightarrow \tau_O$. Type τ_I is called an input type and type τ_O an output type. They are type tuples of dimension n. The directional type expresses two requirements. Firstly, if p/n is called with an argument of type τ_I then the argument has type τ_O upon its success. Secondly, each predicate q/m invoked by p is called with an argument that has the input type of a directional type for q/m. A program is well-typed with respect to a collection of directional types if each directional type in the collection is verified. The type checking problem is reduced to a decision problem on systems of inclusion

constraints over set expressions. The algorithm is sound and complete for discriminative directional types. Charatonik and Podelski (1998) provide an algorithm for inferring directional types with respect to which the program is well-typed.

9 Conclusion

We have presented a type analysis. The type analysis supports non-deterministic type definitions, allows set operators in type expressions, and uses a set of variable typings to describe type information in a set of substitutions. The analysis is presented as an abstract domain and four abstract operations for Nilsson's abstract semantics (Nilsson 1988) extended to deal with negation and built-in predicates. These operations are defined in detail and their local correctness proved. The abstract unification involves propagation of type information downwards and upwards the structure of a term. Given a set of equations in solved form and an abstract substitution, abstract unification is accomplished in two steps. In the first step, more type information for variables occurring on the right-hand side of each equation is derived from type information for the variable on the left-hand side. The second step derives more type information for the variable on the left-hand side of each equation from type information for the variables on the right-hand side. The abstract built-in execution operation approximates the execution of built-in predicates. Each built-in is modeled as a function of abstract substitutions.

Detection of the least fixpoint and elimination of redundancy in a set of variable typings are both reduced to checking the emptiness of types. Though types denote sets of possibly non-ground terms and are not closed under set complement, checking the emptiness of types can be done by using an algorithm that checks for the emptiness of the types that denote sets of ground terms. An experimental study shows that due to a large repetition of emptiness checks, with tabling, the precision improvement measures incurs only a small increase in analysis time.

Acknowledgments: The work is supported in part by the National Science Foundation under grants CCR-0131862 and INT-0327760. A preliminary version of this article appeared, under the title "A Precise Type Analysis of Logic Programs", in *Proceedings of the Second International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Montreal, Canada, 2000. We would like to thank anonymous reviewers for insightful comments on previous drafts of this report.

Appendix A Proofs

Let N denote the set of natural numbers. Define $h : \mathsf{Term}(\Sigma, \mathsf{Var}) \mapsto \mathsf{N}$ as follows. $h(x) = 0$ for all $x \in \mathsf{Var}$ and $h(f(t_1, \cdots, t_n)) = 1 + max\{h(t_i) \mid 1 \leq i \leq n\}$. Define $h : \mathsf{Type} \mapsto \mathsf{N}$ in the same way. Note $h(R) \geq 1$ for any $R \in \mathsf{Type}$. Let $\langle x^1, y^1\rangle \prec \langle x^2, y^2\rangle = (x^1 < x^2) \vee ((x^1 = x^2) \wedge (y^1 < y^2))$.

Lemma 4.4. Let $R \in \mathsf{Type}$ and $t \in \mathsf{Term}$. If $t \in [\![R]\!]_\Delta$ then $\sigma(t) \in [\![R]\!]_\Delta$ for any $\sigma \in Sub$.

Proof
The proof is done by induction on $\langle h(t), h(R)\rangle$. Let $\sigma \in Sub$ be an arbitrary substitution.

Basis. We have that $h(t) = 0$ and that $h(R) = 1$. So, $t \in \mathsf{Var}$ which implies $R = \mathbf{1}$ since $t \in [\![R]\!]_\Delta$ and $h(R) = 1$. Thus, $[\![R]\!]_\Delta = \mathsf{Term}$ and $\sigma(t) \in [\![R]\!]_\Delta$.

Induction. Either that $h(t) = 0$ or that $h(t) > 0$. Consider the case where $h(t) = 0$ first. Then $t \in \mathsf{Var}$. Either (i) $R = \mathbf{1}$; (ii) $R = R_1$ or R_2; or (iii) $R = R_1$ and R_2. The case (i) is a special case of the base case. Consider the case (ii). We have either that $t \in [\![R_1]\!]_\Delta$ or that $t \in [\![R_2]\!]_\Delta$. If $t \in [\![R_j]\!]_\Delta$ then, by induction hypothesis, $\sigma(t) \in [\![R_j]\!]_\Delta$ for $j = 1, 2$ since $h(R_j) < h(R)$. So, $\sigma(t) \in [\![R]\!]_\Delta$ by the definition of $[\![\cdot]\!]_\Delta$. The case (iii) is symmetric to the case (ii). Thus, $\sigma(t) \in [\![R]\!]_\Delta$.

Now consider the case where $h(t) > 0$. Then $t = f(t_1, \cdots, t_n)$. Either (i) $R = \mathbf{1}$; (ii) $R = R_1$ or R_2; (iii) $R = R_1$ and R_2; or (iv) $R = c(R_1, \cdots, R_m)$. The proof for that $\sigma(t) \in [\![R]\!]_\Delta$ in the cases (i), (ii) and (iii) is the same as in the previous paragraph. Consider the case (iv). Since $t \in [\![R]\!]_\Delta$, there is a type rule $c(\beta_1, \cdots, \beta_m) \rightarrowtriangle f(\tau_1, \cdots, \tau_n)$ such that $t_j \in [\![\Bbbk(\tau_j)]\!]_\Delta$ where $\Bbbk = \{\beta_1 \mapsto R_1, \cdots, \beta_m \mapsto R_m\}$. We have that $h(\Bbbk(\tau_j)) \leq h(R)$ and that $h(t_j) < h(t)$. By the induction hypothesis, $\sigma(t_j) \in [\![\Bbbk(\tau_j)]\!]_\Delta$, which together with the definition for $[\![\cdot]\!]_\Delta$, implies that $\sigma(t) \in [\![R]\!]_\Delta$. □

Lemma 5.2. $\gamma(ASub^\flat)$ is a Moore family.

Proof
Since, $\gamma([\{x \mapsto \mathbf{1} \mid x \in V_P\}]_\approx) = Sub$ and Sub is the supremum on $\wp(Sub)$, $\gamma(ASub^\flat)$ contains the supremum on $\wp(Sub)$. Let $[\mathcal{S}_1]_\approx, [\mathcal{S}_2]_\approx \in ASub^\flat$. Then $[\mathcal{S}_1]_\approx \sqcap^\flat [\mathcal{S}_2]_\approx \in ASub^\flat$. Furthermore,

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

Thus, $\gamma(ASub^\flat)$ is closed under $\cap$ – the meet on $\wp(Sub)$. So, $\gamma(ASub^\flat)$ is a Moore family. □

Lemma 5.3. $\gamma([\mathcal{S}_1 \otimes \mathcal{S}_2]_\approx) = \gamma([\mathcal{S}_1]_\approx \sqcap^\flat [\mathcal{S}_2]_\approx)$.

Proof
We first prove that $\gamma([\mathcal{S}_1 \otimes \mathcal{S}_2]_\approx) \subseteq \gamma([\mathcal{S}_1]_\approx \sqcap^\flat [\mathcal{S}_2]_\approx)$. Let $\theta \in \gamma([\mathcal{S}_1 \otimes \mathcal{S}_2]_\approx)$. Then there is ρ in $(\mathcal{S}_1 \otimes \mathcal{S}_2)$ such that $\theta \in \gamma_{\mathsf{VT}}(\rho)$. This implies that there are μ in $\mathcal{S}_1$ and $\nu \in \mathcal{S}_2$ such that $\rho = \lambda x \in V_P.(\mu(x)$ and $\nu(x))$. 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, $\rho \in (\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow})$. Since $\theta \in \gamma_{\mathsf{VT}}(\rho)$, we have that $\theta \in \gamma([\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow}]_{\approx})$ and $\theta \in \gamma([\mathcal{S}_1]_{\approx} \sqcap^{\flat} [\mathcal{S}_2]_{\approx})$.

We now prove that $\gamma([\mathcal{S}_1 \otimes \mathcal{S}_2]_{\approx}) \supseteq \gamma([\mathcal{S}_1]_{\approx} \sqcap^{\flat} [\mathcal{S}_2]_{\approx})$. Let $\theta \in \gamma([\mathcal{S}_1]_{\approx} \sqcap^{\flat} [\mathcal{S}_2]_{\approx})$. Then $\theta \in \gamma_{\mathsf{VT}}(\rho)$ for some $\rho \in (\mathcal{S}_1^{\downarrow} \cap \mathcal{S}_2^{\downarrow})$ by the definition of $\sqcap^{\flat}$. There are $\mu \in \mathcal{S}_1$ and $\nu \in \mathcal{S}_2$ such that $\gamma_{\mathsf{VT}}(\rho) \subseteq \gamma_{\mathsf{VT}}(\mu)$ and $\gamma_{\mathsf{VT}}(\rho) \subseteq \gamma_{\mathsf{VT}}(\nu)$, implying $\forall x \in V_P.(\theta(x) \in [\![\mu(x) \ \mathsf{and} \ \nu(x)]\!]_{\Delta})$. Thus, $\theta \in \gamma([\mathcal{S}_1 \otimes \mathcal{S}_2]_{\approx})$ by the definition of $\otimes$. $\square$

Lemma 6.1. For any $R \in \mathsf{Type}$ and $t \in \mathsf{Term}(\Sigma, V'_P)$, $\{\theta \mid \theta(t) \in [\![R]\!]_{\Delta}\} \subseteq \gamma([vts(R,t)]_{\approx})$.

Proof
The proof is done by induction on $\langle h(t), h(R)\rangle$.

Basis. $\langle h(t), h(R)\rangle = \langle 0, 1\rangle$. Then $t \in V'_P$ and

$$vts(R,t) = \{\lambda x \in V'_P.(\text{if } x = t \text{ then } R \text{ else } \mathbf{1})\}$$

The lemma holds since $\{\theta \mid \theta(t) \in [\![R]\!]_{\Delta}\} = \gamma([vts(R,t)]_{\approx})$.

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

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

$$\theta \in \gamma([vts(\Bbbk(\tau_i), t_i)]_{\approx})$$

for all $1 \leq i \leq n$. By Lemmas 5.2 and 5.3,

$$\theta \in \gamma(\left[\bigotimes_{1 \leq i \leq n} vts(\Bbbk(\tau_i), t_i)\right]_{\approx})$$

By the definition of vts, we have

$$\theta \in \gamma([vts(c(R_1, \cdots, R_m), f(t_1, \cdots, t_n))]_{\approx})$$

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

Now consider the case (2). We have that $t = f(t_1, \cdots, t_n)$. The proof is the same as that for the case (1).(iii). $\square$

Lemma 6.3. Let $\mathcal{S}' = down(E, \mathcal{S})$. Then $mgu(\theta(E)) \circ \theta \in \gamma([\mathcal{S}']_{\approx})$ for all $\theta \in \gamma([\mathcal{S}]_{\approx})$.

Proof
Let $\mathcal{S}_{\mu} = \{\mu\} \otimes \bigotimes_{(x=t) \in E} vts(\mu(x), t)$. It suffices to prove that $mgu(\sigma(E)) \circ \sigma \in \gamma([\mathcal{S}_{\mu}]_{\approx})$ 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 Type is closed under substitution. By Lemma 6.1, we have $mgu(\sigma(E)) \circ \sigma \in \gamma([vts(\mu(x), t)]_{\approx})$ for any $(x = t)$ in E. So, $mgu(\sigma(E)) \circ \sigma \in \gamma([\mathcal{S}_{\mu}]_{\approx})$. $\square$

Lemma 6.5. For any $\tau \in \mathsf{Schm}$ and any $\Bbbk_1, \Bbbk_2 \in \mathsf{TSub}$,

(a) $(\Bbbk_1(\tau)\ \mathsf{or}\ \Bbbk_2(\tau)) \sqsubseteq (\Bbbk_1 \curlyvee \Bbbk_2)(\tau)$; and
(b) $(\Bbbk_1(\tau)\ \mathsf{and}\ \Bbbk_2(\tau)) \equiv (\Bbbk_1 \curlywedge \Bbbk_2)(\tau)$.

Proof
We prove only (a) since the proof for (b) is similar to that for (a). Let $t \in [\![\Bbbk_1(\tau)\ \mathsf{or}\ \Bbbk_2(\tau)]\!]_\Delta$. Either (1) $t \in [\![\Bbbk_1(\tau)]\!]_\Delta$ or (2) $t \in [\![\Bbbk_2(\tau)]\!]_\Delta$. Without loss of generality, we assume (1). We prove $t \in [\![(\Bbbk_1 \curlyvee \Bbbk_2)(\tau)]\!]_\Delta$ by induction on $\langle h(\tau), h(t)\rangle$.

Basis. $h(\tau) = 0$. Then $\tau \in \mathsf{Para}$ and (a) holds since $(\Bbbk_1(\tau)\ \mathsf{or}\ \Bbbk_2(\tau)) \equiv (\Bbbk_1 \curlyvee \Bbbk_2)(\tau)$ by definition of $\curlyvee$.

Induction. $h(\tau) \neq 0$ implies that $\tau = c(\tau_1, \cdots, \tau_m)$. If $t \in \mathsf{Var}$ then $\Bbbk_1(\tau) \equiv \mathbf{1}$ and hence $(\Bbbk_1 \curlyvee \Bbbk_2)(\tau) \equiv \mathbf{1}$ and $t \in [\![(\Bbbk_1 \curlyvee \Bbbk_2)(\tau)]\!]_\Delta$. Otherwise, $t = f(t_1, \cdots, t_n)$. Since $t \in [\![\Bbbk_1(\tau)]\!]_\Delta$, there is a type rule $\tau \twoheadrightarrow f(\tau_1, \cdots, \tau_n)$ such that $t_i \in [\![\Bbbk_1(\tau_i)]\!]_\Delta$ for $1 \leq i \leq n$. We have $h(\tau_i) \leq h(\tau)$ and $h(t_i) < h(t)$. Thus, $t_i \in [\![(\Bbbk_1 \curlyvee \Bbbk_2)(\tau_i)]\!]_\Delta$ by the induction hypothesis and hence $t \in [\![(\Bbbk_1 \curlyvee \Bbbk_2)(\tau)]\!]_\Delta$ by the definition $[\![\cdot]\!]_\Delta$. □

Lemma 6.7. Let $\mathcal{K}_1, \mathcal{K}_2 \in \wp(\mathsf{TSub})$, $R \in \mathsf{Type}$, $\tau \in \mathsf{Schm}$, $\vec{R} \in \mathsf{Type}^*$ and $\vec{\tau} \in \mathsf{Schm}^*$ such that $\|\vec{R}\| = \|\vec{\tau}\|$. If $R \sqsubseteq \mathsf{or}_{\Bbbk_1 \in \mathcal{K}_1} \Bbbk_1(\tau)$ and $\vec{R} \sqsubseteq \mathsf{or}_{\Bbbk_2 \in \mathcal{K}_2} \Bbbk_2(\vec{\tau})$ then $R \bullet \vec{R} \sqsubseteq \mathsf{or}_{\Bbbk \in (\mathcal{K}_1 \curlyvee \mathcal{K}_2)} \Bbbk(\tau \bullet \vec{\tau})$.

Proof
Let $t \bullet \vec{t} \in [\![R \bullet \vec{R}]\!]_\Delta$. Then $t \in [\![R]\!]_\Delta$ and $\vec{t} \in [\![\vec{R}]\!]_\Delta$. By assumption, there are $\Bbbk_1 \in \mathcal{K}_1$ such that $t \in [\![\Bbbk_1(\tau)]\!]_\Delta$ and $\Bbbk_2 \in \mathcal{K}_2$ such that $\vec{t} \in [\![\Bbbk_2(\vec{\tau})]\!]_\Delta$. Let $\Bbbk = \Bbbk_1 \curlyvee \Bbbk_2$. We have $\Bbbk \in (\mathcal{K}_1 \curlyvee \mathcal{K}_2)$ by the definition of $\curlyvee$ and $t \in [\![\Bbbk(\tau)]\!]_\Delta$ and $\vec{t} \in [\![\Bbbk(\vec{\tau})]\!]_\Delta$ by Lemma 6.5. Thus, $t \bullet \vec{t} \in [\![\Bbbk(\tau \bullet \vec{\tau})]\!]_\Delta$ by the definition of $[\![\cdot]\!]_\Delta$. □

Lemma 6.9. Let $\tau \in \mathsf{Schm}$, $R \in \mathsf{Type}$ and $\mathcal{K} = \mathit{cover}(R, \tau)$. Then $R \sqsubseteq \mathsf{or}_{\Bbbk \in \mathcal{K}} \Bbbk(\tau)$.

Proof
The proof is done by induction on the structure of R.

Basis. R is atomic. $R = \mathbf{1}$ or $R = \mathbf{0}$ or $R = c(R_1, \cdots, R_m)$ for some $c/m \in \mathsf{Cons}$ and $R_1, \cdots, R_m \in \mathsf{Type}$. If $R = \mathbf{1}$ or $R = \mathbf{0}$ then the lemma holds by the definitions of $\top$, $\bot$ and $[\![\cdot]\!]_\Delta$. Let $R = c(R_1, \cdots, R_m)$. 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 Para. In the case (a), we have $\mathcal{K} = \{\Bbbk\}$ with $\Bbbk = \{\tau \mapsto R\}$. The lemma holds because $\Bbbk(\tau) = R$. Consider the case (b), if $c/m = d/k$ then $\mathcal{K} = \{\Bbbk\}$ with $\Bbbk = \{\beta_j \mapsto R_j \mid 1 \leq j \leq m\}$ by the definition of *cover* and we have $\Bbbk(\tau) = R$. Otherwise, $\mathcal{K} = \{\Bbbk\}$ with $\Bbbk = \top$ by the definition of *cover* and $\Bbbk(\tau) = \mathbf{1}$ by the definition of $\top$. So, the lemma holds in the case (b).

Induction. Either (1) $R = R_1\ \mathsf{or}\ R_2$ or (2) $R = R_1\ \mathsf{and}\ R_2$. In the case (1), let $\mathcal{K}_1 = \mathit{cover}(R_1, \tau)$ and $\mathcal{K}_2 = \mathit{cover}(R_2, \tau)$. We have $[\![R_i]\!]_\Delta \subseteq \bigcup_{\Bbbk \in \mathcal{K}_i} [\![\Bbbk(\tau)]\!]_\Delta$ for $1 \leq i \leq 2$ by the induction hypothesis. Therefore,

$$\begin{aligned} [\![R_1\ \mathsf{or}\ R_2]\!]_\Delta &= [\![R_1]\!]_\Delta \cup [\![R_2]\!]_\Delta \\ &\subseteq \bigcup_{\Bbbk \in \mathcal{K}_1} [\![\Bbbk(\tau)]\!]_\Delta \cup \bigcup_{\Bbbk \in \mathcal{K}_2} [\![\Bbbk(\tau)]\!]_\Delta \end{aligned}$$

$$= \bigcup_{\Bbbk \in (\mathcal{K}_1 \cup \mathcal{K}_2)} [\![\Bbbk(\tau)]\!]_\Delta$$
$$= \bigcup_{\Bbbk \in cover(R,\tau)} [\![\Bbbk(\tau)]\!]_\Delta$$

So the lemma holds for the case (1). Consider the case (2). Let $\mathcal{K}_1 = cover(R_1, \tau)$ and $\mathcal{K}_2 = cover(R_2, \tau)$. We have $[\![R_i]\!]_\Delta \subseteq \bigcup_{\Bbbk \in \mathcal{K}_i} [\![\Bbbk(\tau)]\!]_\Delta$ for $1 \leq i \leq 2$ by the induction hypothesis. So,

$$\begin{aligned}
[\![R_1 \text{ and } R_2]\!]_\Delta &= [\![R_1]\!]_\Delta \cap [\![R_2]\!]_\Delta \\
&\subseteq \bigcup_{\Bbbk \in \mathcal{K}_1} [\![\Bbbk(\tau)]\!]_\Delta \cap \bigcup_{\Bbbk \in \mathcal{K}_2} [\![\Bbbk(\tau)]\!]_\Delta \\
&= \bigcup_{\Bbbk \in (\mathcal{K}_1 \curlywedge \mathcal{K}_2)} [\![\Bbbk(\tau)]\!]_\Delta \text{ by Lemma 6.5.(b)} \\
&= \bigcup_{\Bbbk \in cover(R,\tau)} [\![\Bbbk(\tau)]\!]_\Delta
\end{aligned}$$

Thus, the lemma holds for the case (2). □

Lemma 6.11. Let $t \in \mathsf{Term}(\Sigma, V'_P)$ and $\mu \in (V'_P \mapsto \mathsf{Type})$. Then $\theta(t) \in [\![type(t,\mu)]\!]_\Delta$ for all $\theta \in \gamma_{\mathsf{VT}}(\mu)$.

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

Basis. $h(t) = 0$. Then $t \in V'_P$ and $type(t,\mu) = \mu(t)$. The lemma holds.

Induction. $h(t) > 0$. Let $t = f(t_1, \cdots, t_n)$ and $R_i = type(t_i, \mu)$ for $i \in \{1, \cdots, n\}$ and $\theta \in \gamma_{\mathsf{VT}}(\mu)$. By the induction hypothesis, we have $\theta(t_i) \in [\![R_i]\!]_\Delta$ for all $1 \leq i \leq n$. Let $\tau \rightarrowtriangle f(\tau_1, \cdots, \tau_i)$ be a type rule in Δ and $\mathcal{K}_i = cover(R_i, \tau_i)$. By Lemma 6.9, $[\![R_i]\!]_\Delta \subseteq \bigcup_{\Bbbk \in \mathcal{K}_i} [\![\Bbbk(\tau_i)]\!]_\Delta$. Thus, $[\![\langle R_1, \cdots, R_n\rangle]\!]_\Delta \subseteq \bigcup_{\Bbbk \in (\curlyvee_{1 \leq i \leq n} \mathcal{K}_i)} [\![\Bbbk(\langle \tau_1, \cdots, \tau_n\rangle)]\!]_\Delta$ by Lemma 6.7, which implies $\theta(t) \in \bigcup_{\Bbbk \in (\curlyvee_{1 \leq i \leq n} \mathcal{K}_i)} [\![\Bbbk(\tau)]\!]_\Delta$ by the definition of $[\![\cdot]\!]_\Delta$. This is true of each type rule for f/n. Therefore, $\theta(t) \in [\![type(t,\mu)]\!]_\Delta$. □

Lemma 6.13. Let $\mathcal{S} \in \wp(V'_P \mapsto \mathsf{Type})$ and $E \in \wp(\mathsf{Eqn})$. Then $mgu(\theta(E)) \circ \theta \in \gamma([up(E, \mathcal{S})]_{\approx})$ for all $\theta \in \gamma([\mathcal{S}]_{\approx})$.

Proof
Let

$$\mu' = \lambda x \in V'_P. \left(\begin{array}{l} if\ \exists t.(x = t) \in E \\ then\ \mu(x)\ \mathsf{and}\ type(t,\mu) \\ else\ \mu(x) \end{array} \right)$$

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 6.11, $\sigma(t) \in [\![type(t,\mu)]\!]_\Delta$. 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, $(mgu(\sigma(E)) \circ \sigma) \in \gamma_{\mathsf{VT}}(\mu')$. □

Theorem 6.16. For any $[\mathcal{S}_1]_{\approx}, [\mathcal{S}_2]_{\approx} \in ASub^\flat$ and any $a_1, a_2 \in \mathsf{Atom}_P$,

$$Uf(a_1, \gamma([\mathcal{S}_1]_{\approx}), a_2, \gamma([\mathcal{S}_2]_{\approx})) \quad \subseteq \quad \gamma(\,Uf^\flat(a_1, [\mathcal{S}_1]_{\approx}, a_2, [\mathcal{S}_2]_{\approx}))$$

Proof
We first prove a preliminary result on substitution and unification. Let $\eta, \theta \in Sub$ and $E, E_1, E_2 \in \wp(\mathsf{Eqn})$ and assume that $\theta = mgu(\eta(E)) \circ \eta \neq fail$. Recall that $mgu(E_1 \cup E_2) = mgu(E_1 \cup eq(mgu(E_2)))$ and $mgu(\eta(E)) = mgu(eq(\eta) \cup E)$ (Eder 1985). Then

$$\begin{aligned}
mgu(\theta(E)) \circ \theta &= mgu(mgu(\eta(E)) \circ \eta(E)) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(mgu(\eta(E))(\eta(E))) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(eq(mgu(\eta(E))) \cup eq(\eta) \cup E) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(eq(mgu(eq(\eta) \cup E)) \cup eq(\eta) \cup E) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(eq(\eta) \cup E \cup eq(\eta) \cup E) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(eq(\eta) \cup E) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(\eta(E)) \circ mgu(\eta(E)) \circ \eta \\
&= mgu(\eta(E)) \circ \eta \\
&= \theta
\end{aligned}$$

We are now ready to prove the theorem. Let $\theta_1 \in \gamma([\mathcal{S}_1]_{\approx})$, $\theta_2 \in \gamma([\mathcal{S}_2]_{\approx})$ and $E_0 = eq \circ mgu(\Psi(a_1), a_2)$. Assume that $uf(a_1, \theta_1, a_2, \theta_2) \neq fail$. It is equivalent to prove $uf(a_1, \theta_1, a_2, \theta_2) \in \gamma(Uf^\flat(a_1, [\mathcal{S}_1]_{\approx}, a_2, [\mathcal{S}_2]_{\approx}))$. By the definition of γ and *rest*, if $\zeta \in \gamma([\mathcal{S}]_{\approx})$ then $\zeta \in \gamma([rest(\mathcal{S})]_{\approx})$ for any substitution ζ and any set of variable typings over V'_P. Thus, it suffices to prove that $uf(a_1, \theta_1, a_2, \theta_2) \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx})$ by the definitions for $Uf^\flat$ and *solve*. Without loss of generality, assume that Ψ renames $\theta_1(a_1)$ apart from $\theta_2(a_2)$. Let $\eta = \theta_2 \cup \Psi(\theta_1)$ and $\theta = mgu(\eta(E_0)) \circ \eta$. Then

$$\begin{aligned}
&uf(a_1, \theta_1, a_2, \theta_2) \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx}) \\
&\quad \leftrightarrow \quad mgu((\Psi(\theta_1))(\Psi(a_1)), \theta_2(a_2)) \circ \theta_2 \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx}) \\
&\quad \leftrightarrow \quad mgu(\eta(E_0)) \circ \eta \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx}) \\
&\quad \leftrightarrow \quad \theta \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx})
\end{aligned}$$

Thus, it remains to prove $\theta \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx})$. Since $\eta \in \gamma(\Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2)$ and $\theta = mgu(\eta(E_0)) \circ \eta$, it holds that $\theta \in \gamma([down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2)]_{\approx})$ according to Lemma 6.3. According to Lemma 6.13, we have $mgu(\theta(E)) \circ \theta \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx})$. Note that $mgu(\theta(E)) \circ \theta = \theta$. Thus, $\theta \in \gamma([up(E_0, down(E_0, \Psi(\mathcal{S}_1) \biguplus \mathcal{S}_2))]_{\approx})$. □

Theorem 7.4. For any term t in $\mathsf{Term}(\Sigma, \mathsf{Var})$ and any type R in Type, $t \in [\![R]\!]_\Delta$ iff $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$.

Proof
We first prove necessity. Assume that $t \in [\![R]\!]_\Delta$. We prove that $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$ by induction on $\langle h(t), h(R) \rangle$.

Basis. $h(t) = 0$ and $h(R) = 1$. We have that $t \in \mathsf{Var}$ and that $R = \mathbf{1}$ by the definition of $[\![\cdot]\!]_\Delta$. Thus, $\chi(t) = \varrho \in \langle\!\langle R \rangle\!\rangle_\Delta$.

Induction. Either $h(t) = 0$ and $h(R) > 1$ or $h(t) > 0$ and $h(R) \geq 1$. Consider first the case where $h(t) = 0$ and $h(R) > 1$. Then $t \in \mathsf{Var}$ and either (i) $R = (R_1 \text{ or } R_2)$; or (ii) $R = (R_1 \text{ and } R_2)$. We only prove the case (i) since the case (ii) is dual to the case (i). Since $t \in [\![R]\!]_\Delta$, either $t \in [\![R_1]\!]_\Delta$ or $t \in [\![R_2]\!]_\Delta$. So, we have either $\chi(t) \in \langle\!\langle R_1 \rangle\!\rangle_\Delta$ or $\chi(t) \in \langle\!\langle R_2 \rangle\!\rangle_\Delta$ by the induction hypothesis. Thus, $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$.

Now consider the case $h(t) > 0$ and $h(R) \geq 1$. Then $t = f(t_1, \cdots, t_n)$. Either (i) $R = (R_1 \text{ or } R_2)$; (ii) $R = (R_1 \text{ and } R_2)$; (iii) $R = \mathbf{1}$ or (iv) $R = c(R_1, \cdots, R_m)$. The proof for that $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$ in cases (i) and (ii) are the same as in the previous paragraph. The case (iii) is vacuous. Consider the case (iv). Since $t \in [\![R]\!]_\Delta$, by the definition of $[\![\cdot]\!]_\Delta$, there is a type rule $c(\beta_1, \cdots, \beta_m) \rightarrowtriangle f(\tau_1, \cdots, \tau_n)$ in Δ such that $t_i \in [\![\Bbbk(\tau_i)]\!]_\Delta$ for all $1 \leq i \leq n$ where $\Bbbk = \{\beta_1 \mapsto R_1, \cdots, \beta_m \mapsto R_m\}$. Observe that $h(t_i) < h(t)$ and $h(\Bbbk(\tau_i)) \leq h(R)$. By induction hypothesis, $\chi(t_i) \in \langle\!\langle \Bbbk(\tau_i) \rangle\!\rangle_\Delta$. By the definition of $\langle\!\langle \cdot \rangle\!\rangle_\Delta$, $t \in \langle\!\langle R \rangle\!\rangle_\Delta$.

We now prove sufficiency. Assume that $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$. We prove that $t \in [\![R]\!]_\Delta$ by induction on $\langle h(t), h(R) \rangle$.

Basis. $h(t) = 0$ and $h(R) = 1$. Then $t \in \mathsf{Var}$ and $\chi(t) = \varrho$. We have that $R = \mathbf{1}$ by the definition of $\langle\!\langle \cdot \rangle\!\rangle_\Delta$. Thus, $t \in [\![R]\!]_\Delta$.

Induction. Either $h(t) = 0$ and $h(R) > 1$ or $h(t) > 0$ and $h(R) \geq 1$. Consider first the case where $h(t) = 0$ and $h(R) > 1$. Then $t \in \mathsf{Var}$ and either (i) $R = (R_1 \text{ or } R_2)$; or (ii) $R = (R_1 \text{ and } R_2)$. We only prove the case (i) since the case (ii) is dual to the case (i). Since $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$, either $\chi(t) \in \langle\!\langle R_1 \rangle\!\rangle_\Delta$ or $\chi(t) \in \langle\!\langle R_2 \rangle\!\rangle_\Delta$. So, we have either $t \in [\![R_1]\!]_\Delta$ or $t \in [\![R_2]\!]_\Delta$ by the induction hypothesis. Thus, $t \in [\![R]\!]_\Delta$.

Now consider the case $h(t) > 0$ and $h(R) \geq 1$. Then $t = f(t_1, \cdots, t_n)$. Either (i) $R = (R_1 \text{ or } R_2)$; (ii) $R = (R_1 \text{ and } R_2)$; (iii) $R = \mathbf{1}$ or (iv) $R = c(R_1, \cdots, R_m)$. The proof for that $t \in [\![R]\!]_\Delta$ in cases (i) and (ii) are the same as in the previous paragraph. The case (iii) is vacuous. Consider the case (iv). Since $\chi(t) \in \langle\!\langle R \rangle\!\rangle_\Delta$, by the definition of $\langle\!\langle \cdot \rangle\!\rangle_\Delta$, there is a type rule $c(\beta_1, \cdots, \beta_m) \rightarrowtriangle f(\tau_1, \cdots, \tau_n)$ in Δ such that $\chi(t_i) \in \langle\!\langle \Bbbk(\tau_i) \rangle\!\rangle_\Delta$ for all $1 \leq i \leq n$ where $\Bbbk = \{\beta_1 \mapsto R_1, \cdots, \beta_m \mapsto R_m\}$. Observe that $h(t_i) < h(t)$ and $h(\Bbbk(\tau_i)) \leq h(R)$. By induction hypothesis, $t_i \in [\![\Bbbk(\tau_i)]\!]_\Delta$. By the definition of $[\![\cdot]\!]_\Delta$, $\chi(t) \in [\![R]\!]_\Delta$. □

Corollary 7.5. For any $R_1, R_2 \in \mathsf{Type}$, $[\![R_1]\!]_\Delta \subseteq [\![R_2]\!]_\Delta$ iff $\langle\!\langle R_1 \rangle\!\rangle_\Delta \subseteq \langle\!\langle R_2 \rangle\!\rangle_\Delta$.

Proof

Both sufficiency and necessity are proved by contradiction. We first consider sufficiency. Assume that $\langle\!\langle R_1 \rangle\!\rangle_\Delta \subseteq \langle\!\langle R_2 \rangle\!\rangle_\Delta$ but $[\![R_1]\!]_\Delta \not\subseteq [\![R_2]\!]_\Delta$. Then there is a term t such that $t \in [\![R_1]\!]_\Delta$ and $t \notin [\![R_2]\!]_\Delta$. By Theorem 7.4, we have that $\chi(t) \in \langle\!\langle R_1 \rangle\!\rangle_\Delta$. By the assumption, $\chi(t) \in \langle\!\langle R_2 \rangle\!\rangle_\Delta$. By Theorem 7.4, $t \in [\![R_2]\!]_\Delta$ that contradicts with that $t \notin [\![R_2]\!]_\Delta$.

We now prove necessity. Assume that $[\![R_1]\!]_\Delta \subseteq [\![R_2]\!]_\Delta$ but $\langle\!\langle R_1 \rangle\!\rangle_\Delta \not\subseteq \langle\!\langle R_2 \rangle\!\rangle_\Delta$. Then there is a term t such that $t \in \langle\!\langle R_1 \rangle\!\rangle_\Delta$ and $t \notin \langle\!\langle R_2 \rangle\!\rangle_\Delta$. By Theorem 7.4, there is a term t' such that $\chi(t') = t$ and $t' \in [\![R_1]\!]_\Delta$. By the assumption, $t' \in [\![R_2]\!]_\Delta$. By Theorem 7.4, $t \in \langle\!\langle R_2 \rangle\!\rangle_\Delta$ that contradicts with that $t \notin \langle\!\langle R_2 \rangle\!\rangle_\Delta$. □

References

Aiken, A. and Lakshman, T. 1994. Directional type checking of logic programs. In *Proceedings of the First International Static Analysis Symposium*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer, 43–60.

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

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

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

Bronsard, F., Lakshman, T. K., and Reddy, U. S. 1992. A framework of directionality for proving termination of logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, K. Apt, Ed. The MIT Press, 321–335.

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

Charatonik, W. and Podelski, A. 1998. Directional type inference for logic programs. In *Proceedings of the Fifth International Symposium on Static Analysis*, G. Levi, Ed. Lecture Notes in Computer Science, vol. 1503. Springer, 278–294.

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

Codish, M. and Lagoon, V. 2000. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science 238,* 1–2, 131–159.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. 2002. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*. The ACM Press, 238–252.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *Journal of Logic Programming 13,* 1–4, 103–179.

Cousot, P. and Cousot, R. 1995. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*. The ACM Press, 170–181.

Dart, P. and Zobel, J. 1992a. Efficient runtime type checking of typed logic programs. *Journal of Logic Programming 14,* 1-2, 31–69.

Dart, P. and Zobel, J. 1992b. A regular type language for logic programs. In *Types in Logic Programming*, F. Pfenning, Ed. The MIT Press, 157–189.

Eder, E. 1985. Properties of substitutions and unifications. *Journal of Symbolic Computation 1,* 1, 31–46.

Fages, F. and Coquery, E. 2001. Typing constraint logic programs. *Theory and Practice of Logic Programming 1,* 6, 751–777.

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

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

Gallagher, J. P., Boulanger, D., and Saglam, H. 1995. Practical model-based static analysis for definite logic programs. In *Proceedings of the Fifteenth International Symposium on Logic Programming*, J. W. Lloyd, Ed. The MIT Press, 351–368.

Gallagher, J. P. and Puebla, G. 2002. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *Proceedings of the Fourth International Symposium on Practical Aspects of Declarative Languages*, S. Krishnamurthi and C. R. Ramakrishnan, Eds. Lecture Notes in Computer Science, vol. 2257. Springer, 243–261.

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

Heintze, N. and Jaffar, J. 1990. A finite presentation theorem for approximating logic programs. In *Principles of Programming Languages.* The ACM Press, 197–209.

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

Hermenegildo, M., Warren, R., and Debray, S. 1992. Global flow analysis as a practical compilation tool. *Journal of Logic Programming 13,* 1–4, 349–366.

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

Hill, P. and Lloyd, J. 1994. *The Gödel Programming Language.* The MIT Press.

Hill, P. M. and Spoto, F. 2002. Generalising *Def* and *Pos* to type analysis. *Journal of Logic and Computation 12,* 3, 497–542.

Horiuchi, K. and Kanamori, T. 1988. Polymorphic type inference in Prolog by abstract interpretation. In *Proceedings of the Sixth Conference on Logic Programming*, K. Furukawa, H. Tanaka, and T. Fujisaki, Eds. Lecture Notes in Computer Science, vol. 315. Springer, 195–214.

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

Kahrs, S. 1996. Limits of ML-definability. In *Proceedings of the Eighth International Symposium on Programming Languages: Implementation, Logic and Programs*, H. Kuchen and S. D. Swierstra, Eds. Lecture Notes in Computer Science, vol. 1140. Springer, 17–31.

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

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

Lagoon, V. and Stuckey, P. J. 2001. A framework for analysis of typed logic programs. In *Proceedings of the Fifth International Symposium on Functional and Logic Programming*, H. Kuchen and K. Ueda, Eds. Lecture Notes in Computer Science, vol. 2024. Springer, 296–310.

Lloyd, J. 1987. *Foundations of Logic Programming.* Springer-Verlag.

Lu, L. 1995. 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.* The ACM Press, 241–252.

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

Lu, L. 2003. Path dependent analysis of logic programs. *Higher-Order and Symbolic Computation 16*, 341–377.

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

Marriott, K. and Søndergaard, H. 1989. Semantics-based dataflow analysis of logic programs. In *Information Processing 89, Proceedings of the Eleventh IFIP World Computer Congress*, G. Ritter, Ed. North-Holland, 601–606.

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

Mycroft, A. and O'Keefe, R. 1984. A polymorphic type system for Prolog. *Artificial Intelligence 23,* 3, 295–307.

Nilsson, U. 1988. Towards a framework for abstract interpretation of logic programs. In *Proceedings of the First International Workshop on Programming Language Implementation and Logic Programming*, P. Deransart, B. Lorho, and J. Małuszynski, Eds. Lecture Notes in Computer Science, vol. 348. Springer, 68–82.

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

Rychlikowski, P. and Truderung, T. 2001. Polymorphic directional types for logic programming. In *Proceedings of the Third ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. The ACM Press, 61–72.

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

Smaus, J.-G. 2001. Analysis of polymorphically typed logic programs using ACI-unification. In *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Lecture Notes in Artificial Intelligence, vol. 2250. Springer, 282–298.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1–3, 19–64.

Van Hentenryck, P., Cortesi, A., and Le Charlier, B. 1995. Type analysis of Prolog using type graphs. *Journal of Logic Programming 22,* 3, 179–208.

Warren, D. S. 1992. Memoing for logic programs. *Communications of the ACM 35,* 3, 93–111.

Yardeni, E., Frühwirth, T., and Shapiro, E. 1991. Polymorphically typed logic programs. In *Proceedings of the Eighth International Conference on Logic Programming*, K. Furukawa, Ed. The MIT Press, 379–93.

Yardeni, E. and Shapiro, E. 1991. A type system for logic programs. *Journal of Logic Programming 10,* 2, 125–153.

Zobel, J. 1987. Derivation of polymorphic types for Prolog programs. In *Proceedings of the Fourth International Conference on Logic Programming*, J. Lassez, Ed. The MIT Press, 817–838.