POLYMORPHIC TYPE ANALYSIS IN LOGIC PROGRAMS BY ABSTRACT INTERPRETATION

LUNJIN LU
LIX
Ecole Polytechnique
91128 Palaiseau Cedex
France

Abstract

In this paper, we first introduce a notion of polymorphic abstract interpretation that formalises a polymorphic analysis as a generalisation of possibly infinitely many monomorphic analyses in the sense that the results of the monomorphic analyses can be obtained as instances of that of the polymorphic analysis. We then present a polymorphic type analysis of logic programs in terms of an abstract domain for polymorphic descriptions of type information and two operators on the abstract domain, namely the least upper bound operator and the abstract unification operator. The abstract domain captures type information more precisely than other abstract domains for similar purposes. The abstract unification operator for the polymorphic type analysis is designed by lifting the abstract unification operator for a monomorphic type analysis in logic programs, which simplifies the proof of the safeness of the polymorphic type analysis. Some experimental results with a prototype implementation of the polymorphic type analysis are also presented.

1 INTRODUCTION

A program analysis is to infer information from programs. Let P be a program, In express input information before analysis, and Out express output information inferred from P and In. $[\mathsf{In}], \mathsf{P} \Longrightarrow [\mathsf{Out}]$ denotes the analysis that infers Out from P and In. Cousot and Cousot [13] introduce a methodology for static analysis called abstract interpretation whereby a program analysis is viewed as the execution of the program over a non-standard data domain. A typical analysis by abstract interpretation is monomorphic whereby input information about the program is not parameterised and the program has to be analysed separately for different input information. Note that program variables are not parameters for input information, though input information can be thought of as predicates over program variables. Take the generic sorting program $sort(x, y)$ for instance, letting nat denote the set of natural numbers, int the set of intergers, and $list(\alpha)$ the set of lists of elements from α, program analyses

$[x \in \mathit{list}(nat)], sort(x,y) \Longrightarrow [y \in \mathit{list}(nat)]$ and $[x \in \mathit{list}(int)], sort(x,y) \Longrightarrow [y \in \mathit{list}(int)]$ are accomplished separately even if they are two instances of a polymorphic analysis $\forall \alpha.([x \in \mathit{list}(\alpha)], sort(x,y) \Longrightarrow [y \in \mathit{list}(\alpha)])$ where both input information and output information are parameterised. By assigning different values to α which serves as a place holder for information to be filled in after analysis, $\forall \alpha.([x \in \mathit{list}(\alpha)], sort(x,y) \Longrightarrow [y \in \mathit{list}(\alpha)])$ can be instantiated into many different monomorphic analyses such as $[x \in \mathit{list}(nat)], sort(x,y) \Longrightarrow [y \in \mathit{list}(nat)]$ and $[x \in \mathit{list}(int)], sort(x,y) \Longrightarrow [y \in \mathit{list}(int)]$. Polymorphic program analyses have advantages over monomorphic program analyses since more general result can be obtained from a polymorphic analysis. Firstly, a sub-program or a library program that may be used in different places need not to be analysed separatly for its different uses. In that sense, the result of a polymorphic analysis is re-usable. This has positive bearing on efficiency of analysis because output information for different uses of the same sub-program can be obtained by instantiation rather than by re-analysis. Secondly, polymorphic program analyses are ameable to program modifications since changes to the program does not necessite re-analyses of the sub-program so long as the sub-program itself is not changed. This paper formalises the notion of a polymorphic abstract interpretation which facilitates the design of polymorphic program analyses $\forall \alpha_1, \cdots, \alpha_n.([\mathsf{In}(\alpha_1, \cdots, \alpha_n)], \mathsf{P} \Longrightarrow [\mathsf{Out}(\alpha_1, \cdots, \alpha_n)])$, and devises a polymorphic type analysis of logic programs.

Prolog is a type-free language. The programmer does not have to specify the types of variables, functions and predicates. This may make it simple to write simple programs. However, it also makes it difficult to debug programs because type errors cannot be detected by Prolog systems. A type error will manifest itself in the form of a wrong result or a missing result rather than an indication of a type violation.

There have been many efforts to augment Prolog with type systems in the forms of type checking systems [4, 18, 39] and type analysis systems. Type analysis systems infer types from the text of a program. Some type analysis systems infer a type for the program and the type is meant to denote a superset of the success set of the program [1, 2, 10, 16, 19, 20, 37, 42, 48, 49, 50]. Other type analysis systems infer types for procedure calls and successes [6, 22, 24, 27, 28].

There are three main approaches to inferring types [20]. The first approach obtains from a program P a system of formulae which describes set-theoretic relationships between the atoms appearing in P. A type is then defined to be a distinguished model of the system of formulae. Examples of this approach are to be found in [1, 16, 37, 42, 48, 49, 50]. The second approach uses an abstract immediate consequence operator to approximate the immediate consequence operator T_P associated with P [46]. A type is defined as a fixed-point of the abstract immediate consequence operator. An example of this approach appears in [49]. Heintze and Jaffar [19, 20] show that the first and the second approaches are equivalent in the sense that a type in one approach has its counterpart in the other. The third approach involves specialising an abstract interpretation framework by developing abstract domains and their associated abstract operators. Some examples are to be found in [6, 22, 24, 27, 28].

Most type analysis systems infer descriptive types. A descriptive type is a description of a set of terms by an expression in a type language. Different type analysis systems may use different type languages. Most of them use regular types or deterministic regular types [16] in one form or another. A descriptive type is not intuitive. Although the sets of terms described by descriptive types are well-defined, it is difficult for the programmer to understand descriptive types. Horiuchi and Kanamori and Kawamura [22, 27, 28] and Lu [31] infer prescriptive types where the types of function symbols are provided by the programmer. Although function symbols are given polymorphic types, only monomorphic types are inferred.

The type inference system proposed by Barbuti and Giacobazzi [2] infers polymorphic types of Horn clause logic programs using a bottom-up abstract interpretation technique. 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 associated with the program is derived from the non-ground concrete immediate consequence operator associated with the program defined in [17]. The type description only describes part of the success set of the program. The atoms that are not well-typed are not described by the type description. Neither are those atoms whose successful SLD resolutions with the program all select at least one atom that is not well-typed.

Codish and Demoen [10] apply abstract compilation to infer type dependencies of logic programs by associating each type with an incarnation of the abstract domain Prop [32]. Abstract compilation [21] is an alternative to abstract interpretation [13]. In abstract compilation, a program is analysed by applying a concrete semantics to an abstraction of the program. The type dependencies of a logic program are obtained by first abstracting the program and then computing the minimal model of the abstracted program together with incarnations of Prop. The incarnations of Prop define meanings of types and capture interactions between types. The type dependencies of a logic program is similar to the type description of the program inferred by the type inference system of Barbuti and Giacobazzi [2] except that the type dependencies describes the whole success set of the program although Codish and Demoen do not make this explicit. The computation of the minimal model is made efficient in both time and space by using the bottom-up semantics due to Bossi, Gabbrielli, Levi and Meo [3].

This paper presents a polymorphic type analysis of logic programs by means of abstract interpretation. For this purpose, we first formalise the notion of polymorphic abstract interpretation that is of generality in that it is language-independent and applicable to other analyses as well as type analysis. Much work has been done on abstract interpretation of logic programs [14]. A number of frameworks have been brought about for abstract interpretation of logic programs [5, 26, 33, 36]. An abstract interpretation framework is a generic abstract semantics that has as a parameter a domain, called an abstract domain, and a fixed number of operators, called abstract operators, associated with the abstract domain. A particular analysis corresponds to a particular abstract domain and its associated abstract operators. Usually, specialising a framework for a particular analysis involves devising an abstract domain for descriptions of sets of substitutions, called abstract substitutions, and corresponding abstract operators on abstract substitutions. One important abstract operator commonly required is abstract unification [9, 25] which mainly abstracts the normal unification. Another important abstract operator commonly required is an operator that computes (an approximation of) the least upper bound of abstract substitutions. In fact, these two abstract operators are the only abstract operators required by the frameworks proposed in [9, 23, 30, 40]. The polymorphic type analysis will be presented as an abstract domain for polymorphic type descriptions of sets of substitutions together with its associated abstract unification operator and least upper bound operator as required by the framework in [30]. The adaptation to the frameworks in

[9, 23, 35, 40] needs only minor technical adjustments since the functionalities required of the above two abstract operators by these frameworks and that in [30] are almost the same. The adaptation to the frameworks in [5, 26, 36] needs more technical work but should not be difficult because most functionalities of the abstract operators in these frameworks are covered by the functionalities of the abstract unification operator and the least upper bound operator in [9, 23, 30, 40].

This paper makes two contributions. Firstly, it formalises the notion of a polymorphic analysis as a representation of a possibly infinite number of monomorphic analyses. Specifically, the result of a polymorphic analysis subsumes those of many monomorphic analyses in the sense that the results of the monomorphic analyses may be obtained as instances of that of the polymorphic analysis. A polymorphic analysis is distinguished from a monomorphic analysis in that the former is parameterised by a number of parameters while the latter is not. A parameter serves as a place holder for information that is provided after analysis. The conception of polymorphic analyses is general and hence not limited to type analysis or logic programs. Secondly, it presents a polymorphic type analysis by desiging a novel abstract domain of polymorphic abstract substitutions and an abstract unification operator on the abstract domain of polymorphic abstract substitutions.

The remainder of this paper is organised as follows. Section 2 briefly recalls the notion of abstract interpretation and the concrete interpretation in the abstract interpretation framework for normal logic programs [30]. In section 3, we present the notion of polymorphic abstract interpretation whereby a polymorphic analysis is considered as a representation of possibly infinitely many monomorphic analyses. Section 4 is devoted to types, their definitions by type clauses and meanings of mono-types as sets of terms. The meaning of a mono-type is given by a meaning function which is derived from type definitions. In section 5, we present a monomorphic type analysis as an abstract interpretation relative to the concrete interpretation presented in section 2. The monomorphic type analysis was first proposed in [22] and later developed in [30]. We include it in this paper in order to facilitate the presentation of the polymorphic type analysis. In section 6, we design an abstract domain for the polymorphic type analysis and its associated abstract unification operator. An abstract substitution in the abstract domain is a set of pairs consisting of an assignment of poly-types to variables and a condition, called constrained variable poly-typings. The meaning of a set of constrained variable poly-typings is dependent on type parameters and is the union of the sets of substitutions described by its members for a given assignment of mono-types to type parameters. When type parameters are assigned mono-types as their values, a constrained variable poly-typing either describes the empty set of substitution if its condition part is *false* or the set of substitutions determined by its assignment part if its condition is *true*. The abstract unification operator for the polymorphic type analysis is defined by lifting the corresponding abstract unification operator for monomorphic type analysis. Section 7 describes our method for removing redundant elements in an abstract substitution. Section 8 briefly describes an prototype implementation of the polymorphic type analysis and section 9 concludes the paper.

$$
\begin{tikzcd}
\mathcal{D}^\dagger \arrow[r, "\mathcal{O}^\dagger"] \arrow[d, "\gamma^\dagger"] & \mathcal{D}^\dagger \arrow[d, "\gamma^\dagger"] \\
\sqsubseteq\ \mathcal{D} \arrow[r, "\mathcal{O}"] & \sqsubseteq\ \mathcal{D}
\end{tikzcd}
$$

Figure 1: Abstract Interpretation

2 PRELIMINARIES

This section recalls the concept of abstract interpretation and a concrete interpretation based on which we will present our polymorphic type analysis. The reader is assumed to be familiar with terminology of abstract interpretation [13] and that of logic programming [29].

2.1 Abstract Interpretation

Let $\mathcal{D}$ be a set. An n-ary operator on $\mathcal{D}$ is a function from $\mathcal{D}^n$ to $\mathcal{D}$. An interpretation I is a tuple $\langle(\mathcal{D}, \sqsubseteq), (\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_k)\rangle$ where $(\mathcal{D}, \sqsubseteq)$ is a complete lattice and $\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_k$ are operators of fixed arities.

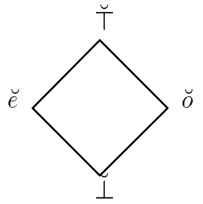
Definition 2.1 *Let $I = \langle(\mathcal{D}, \sqsubseteq), (\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_k)\rangle$ and $I^\dagger = \langle(\mathcal{D}^\dagger, \sqsubseteq^\dagger), (\mathcal{O}_1^\dagger, \mathcal{O}_2^\dagger, \cdots, \mathcal{O}_k^\dagger)\rangle$ be two interpretations such that $\mathcal{O}_i$ and $\mathcal{O}_i^\dagger$ are of same arity n_i for each $1 \leq i \leq k$, and $\gamma^\dagger$ be a monotonic function from $\mathcal{D}^\dagger$ to $\mathcal{D}$. $I^\dagger$ is called a $\gamma^\dagger$-abstraction of I if, for each $1 \leq i \leq k$,*

- *for all $d_1, \cdots, d_{n_i} \in \mathcal{D}$ and $d_1^\dagger, \cdots, d_{n_i}^\dagger \in \mathcal{D}^\dagger$,*

$$d_1 \sqsubseteq \gamma^\dagger(d_1^\dagger) \wedge \cdots \wedge d_{n_i} \sqsubseteq \gamma^\dagger(d_{n_i}^\dagger) \rightarrow \mathcal{O}_i(d_1, \cdots, d_{n_i}) \sqsubseteq \gamma^\dagger \circ \mathcal{O}_i^\dagger(d_1^\dagger, \cdots, d_{n_i}^\dagger)$$

$I^\dagger$ is called an abstract interpretation and I a concrete interpretation. An object in $I^\dagger$, say $\mathcal{D}^\dagger$, is called abstract while an object in I, say $\mathcal{D}$, is called concrete. With respect to a given $\gamma^\dagger$, if $d \sqsubseteq \gamma^\dagger(d^\dagger)$ then d is said to be described by $d^\dagger$ or $d^\dagger$ is said to be a description of d. The condition in the above definition is read as that $\mathcal{O}_i^\dagger$ is a $\gamma^\dagger$-abstraction of $\mathcal{O}_i$. Therefore, $I^\dagger$ is a $\gamma^\dagger$-abstraction of I iff each operator $\mathcal{O}^\dagger$ in $I^\dagger$ is a $\gamma^\dagger$-abstraction of its corresponding operator $\mathcal{O}$ in I. Figure 1 is an illustration that $\mathcal{O}^\dagger$ is a $\gamma^\dagger$-abstraction of $\mathcal{O}$.

Example 2.2 *This example illustrates the notion of abstract interpretation by means of a parity analysis [34]. Consider natural number expressions, say $X + (Y \times Z)$, formed of constant natural numbers, variables, addition operator $+$ and multiplication operator $\times$. The usual approach to analyse such expressions is to form a collecting semantics which lifts operators on data to sets of data and then computes an abstraction of the collecting semantics. The collecting semantics of $X + (Y \times Z)$ is $\mathbf{x} \oplus (\mathbf{y} \otimes \mathbf{z})$ where $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ are respectively the sets*



(a) Definition of $(\aleph, \breve{\sqsubseteq}, \breve{\sqcup})$

$$\breve{\gamma}(\breve{\mathbf{p}}) \stackrel{\text{def}}{=} \begin{cases} \emptyset & if\ \breve{\mathbf{p}} = \breve{\bot} \\ \{2n \mid n \in nat\} & if\ \breve{\mathbf{p}} = \breve{e} \\ \{2n+1 \mid n \in nat\} & if\ \breve{\mathbf{p}} = \breve{o} \\ nat & if\ \breve{\mathbf{p}} = \breve{\top} \end{cases}$$

(b) Definition of $\breve{\gamma}$

$\breve{\oplus}$	$\breve{\bot}$	$\breve{e}$	$\breve{o}$	$\breve{\top}$
$\breve{\bot}$	$\breve{\bot}$	$\breve{\bot}$	$\breve{\bot}$	$\breve{\bot}$
$\breve{e}$	$\breve{\bot}$	$\breve{e}$	$\breve{o}$	$\breve{\top}$
$\breve{o}$	$\breve{\bot}$	$\breve{o}$	$\breve{e}$	$\breve{\top}$
$\breve{\top}$	$\breve{\bot}$	$\breve{\top}$	$\breve{\top}$	$\breve{\top}$

(c) Definition of $\breve{\oplus}$

$\breve{\otimes}$	$\breve{\bot}$	$\breve{e}$	$\breve{o}$	$\breve{\top}$
$\breve{\bot}$	$\breve{\bot}$	$\breve{\bot}$	$\breve{\bot}$	$\breve{\bot}$
$\breve{e}$	$\breve{\bot}$	$\breve{e}$	$\breve{e}$	$\breve{e}$
$\breve{o}$	$\breve{\bot}$	$\breve{e}$	$\breve{o}$	$\breve{\top}$
$\breve{\top}$	$\breve{\bot}$	$\breve{e}$	$\breve{\top}$	$\breve{\top}$

(c) Definition of $\breve{\otimes}$

Figure 2: Parity Analysis

of natural numbers that X, Y *and* Z *may take as their values, and* $\oplus, \otimes : \wp(\mathrm{nat}) \times \wp(\mathrm{nat}) \mapsto \wp(\mathrm{nat})$ *are defined as follows.*

$$\begin{aligned} \mathbf{x} \oplus \mathbf{y} &\stackrel{\text{def}}{=} \{X + Y \mid X \in \mathbf{x} \wedge Y \in \mathbf{y}\} \\ \mathbf{x} \otimes \mathbf{y} &\stackrel{\text{def}}{=} \{X \times Y \mid X \in \mathbf{x} \wedge Y \in \mathbf{y}\} \end{aligned}$$

Therefore, the concrete interpretation is $I_{parity} = \langle(\wp(\mathrm{nat}), \subseteq), (\oplus, \otimes)\rangle$.

The domain $(\aleph, \breve{\sqsubseteq})$ *of parity descriptions and the concretisation function* $\breve{\gamma} : (\aleph, \breve{\sqsubseteq}) \mapsto (\wp(\mathrm{nat}), \subseteq)$ *are shown in figure 2.2. It is obvious that* $(\aleph, \breve{\sqsubseteq})$ *is a complete lattice and* $\breve{\gamma}$ *is monotonic.*

Figure 2.2 also defines $\breve{\oplus}, \breve{\otimes} : \aleph \times \aleph \mapsto \aleph$ *that are respectively* $\breve{\gamma}$*-abstractions of* $\oplus$ *and* $\otimes$*. So,* $\breve{I}_{parity} = \langle(\aleph, \breve{\sqsubseteq}), (\breve{\oplus}, \breve{\otimes})\rangle$ *is a* $\breve{\gamma}$*-abstraction of* $I_{parity} = \langle(\wp(\mathrm{nat}), \subseteq), (\oplus, \otimes)\rangle$.

The $\breve{\gamma}$*-abstraction of* $\mathbf{x} \oplus (\mathbf{y} \otimes \mathbf{z})$ *is* $\breve{\mathbf{x}} \breve{\oplus} (\breve{\mathbf{y}} \breve{\otimes} \breve{\mathbf{z}})$ *where* $\breve{\mathbf{x}}, \breve{\mathbf{y}}$ *and* $\breve{\mathbf{z}}$ *are respectively parity descriptions of* $\mathbf{x}, \mathbf{y}$ *and* $\mathbf{z}$.

Let $\breve{\mathbf{x}} = \breve{o}$, $\breve{\mathbf{y}} = \breve{e}$ *and* $\breve{\mathbf{z}} = \breve{\top}$*. We have* $\breve{\mathbf{x}} \breve{\oplus} (\breve{\mathbf{y}} \breve{\otimes} \breve{\mathbf{z}}) = \breve{o} \breve{\oplus} (\breve{e} \breve{\otimes} \breve{\top}) = \breve{o} \breve{\oplus} \breve{e} = \breve{o}$*, which states that if* X *is an odd natural number and* Y *is an even natural number then* $X + (Y \times Z)$ *is an odd natural number.*

2.2 Concrete Interpretation

Let Func be a set of *function symbols*, Pred be a set of *predicate symbols*, Vars be a denumerable set of variables. Term denotes the set of *terms* that can be constructed from Func and Vars.

Atom denotes the set of *atoms* constructible from Pred, Func and Vars. Let θ be a *substitution* and $\mathsf{V} \subseteq \mathsf{Vars}$. $dom(\theta)$ denotes the domain of θ. $\theta\restriction\mathsf{V}$ denotes the restriction of θ to V. As a convention, $\circ$ binds stronger than $\restriction$. For instance, $\eta\circ\sigma\restriction\mathsf{V}$ is equal to $(\eta\circ\sigma)\restriction\mathsf{V}$. An *expression* E is a term, an atom, a literal, a clause, a goal etc. $vars(E)$ denotes the set of variables in E.

An *equation* is a formula of the form $l = r$ where either $l, r \in \mathsf{Term}$ or $l, r \in \mathsf{Atom}$. The set of all equations is denoted as Eqn. Let $E \in \wp(\mathsf{Eqn})$. E is in *solved form* if, for each equation $l = r$ in E, l is a variable that does not occur on the right side of any equation in E. For a set of equations $E \in \wp(\mathsf{Eqn})$, $mgu : \wp(\mathsf{Eqn}) \mapsto Sub \cup \{fail\}$ returns either a most general unifier for E if E is unifiable or $fail$ otherwise, where Sub is the set of substitutions. $mgu(\{l = r\})$ is sometimes written as $mgu(l, r)$. Let $\theta \circ fail \stackrel{def}{=} fail$ and $fail \circ \theta \stackrel{def}{=} fail$ for any $\theta \in Sub \cup \{fail\}$ where $\circ$ is the function composition operator. There is a natural bijection between substitutions and the sets of equations in solved form. $eq(\theta)$ denotes the set of equations in solved form corresponding to a substitution θ. $eq(fail) \stackrel{def}{=} fail$.

The abstract interpretation framework in [30] is based on a collecting semantics of normal logic programs which associates each textual program point with a set of substitutions. The set is a superset of the set of substitutions that may be obtained when the execution of the program reaches that program point. The collecting semantics is defined in terms of two operators on $\langle\wp(Sub), \subseteq\rangle$. One operator is the set union $\cup$ - the least upper bound operator on $\langle\wp(Sub), \subseteq\rangle$ and the another is $UNIFY$ which is defined as follows. Let $A, B \in \mathsf{Atom}$ and $\Theta_A, \Theta_B \in \wp(Sub)$.

$$UNIFY(A, \Theta_A, B, \Theta_B) = \{unify(A, \theta_A, B, \theta_B) \neq fail \mid \theta_A \in \Theta_A \wedge \theta_B \in \Theta_B\}$$

where

$$\begin{array}{rcl} unify(A, \theta_A, B, \theta_B) & \stackrel{def}{=} & mgu(\rho\circ\theta_A(A), \theta_B(B)) \circ \theta_B \text{ where } \rho \text{ is a renaming} \\ & & \text{substitution such that } vars(\rho\circ\theta_A(A)) \cap vars(\theta_B(B)) = \emptyset \end{array}$$

In summary, the collecting semantics corresponds to the following concrete interpretation.

$$I = \langle(\wp(Sub), \subseteq), (\cup, UNIFY)\rangle$$

Specialising the framework for a particular program analysis consists in designing an abstract domain $(ASub, \sqsubseteq)$, a concretisation function $\gamma : ASub \mapsto \wp(Sub)$, and two abstract operators $\sqcup$ and $AUNIFY$ such that $\langle(ASub, \sqsubseteq), (\sqcup, AUNIFY)\rangle$ is a γ-abstraction of $I = \langle(\wp(Sub), \subseteq), (\cup, UNIFY)\rangle$. Once $(ASub, \sqsubseteq)$ and γ are designed, it remains to design $AUNIFY$ such that $AUNIFY$ is a γ-abstraction of $UNIFY$ since $\sqcup$ is a γ-abstraction of $\cup$. $AUNIFY$ is called an abstract unification operator since its main work is to abstract unification. We note that the *procedure-entry* and *procedure-exit* operators in [5], the *type substitution propagation* operator in [26, 28] and the corresponding operators in [38] and [40] can be thought of as variants of $AUNIFY$.

3 POLYMORPHIC ABSTRACT INTERPRETATION

This section conceptualises polymorphic abstract interpretation and exemplifies the notion of polymorphic abstract interpretation by designing a polymorphic abstract interpretation for

$$
\begin{array}{lcl}
append([], L, L) & & \\
append([H|L_1], L_2, [H|L_3]) & \leftarrow & append(L_1, L_2, L_3)
\end{array}
$$

Figure 3: Logic program *append*

parity analysis.

An analysis corresponds to an abstract interpretation. For a monomorphic analysis, data descriptions are monomorphic as they do not contain parameters. For a polymorphic analysis, data descriptions contain parameters. Take the logic program in figure 3 as an example. By a monomorphic type analysis [22, 31], one would infer that

> *if L_1 and L_2 are lists of natural numbers before $append(L_1, L_2, L_3)$ is executed then L_3 is a list of natural numbers after $append(L_1, L_2, L_3)$ is successfully executed*

and that

> *if L_1 and L_2 are lists of real numbers before $append(L_1, L_2, L_3)$ is executed then L_3 is a list of real numbers after $append(L_1, L_2, L_3)$ is successfully executed.*

It is desirable to design a polymorphic type analysis [10] which would infer that

> *if L_1 and L_2 are lists of elements of type α before $append(L_1, L_2, L_3)$ is executed then L_3 is a list of elements of type α after $append(L_1, L_2, L_3)$ is successfully executed.*

The two statements inferred by the monomorphic type analysis are two of an infinite number of instances of the statement inferred by the polymorphic type analysis. A polymorphic analysis is a representation of a possibly infinite number of monomorphic analyses. In other words, the result of a polymorphic analysis subsumes the results of many monomorphic analyses in the sense that the results of the monomorphic analyses may be obtained as instances of the result of the polymorphic analysis. Let $I^\bullet = \langle(\mathcal{D}^\bullet, \sqsubseteq^\bullet), (\mathcal{O}_1^\bullet, \mathcal{O}_2^\bullet, \cdots, \mathcal{O}_k^\bullet)\rangle$ be the abstract interpretation that a polymorphic analysis is based on. The elements in $\mathcal{D}^\bullet$ by necessity contain parameters because the result of the polymorphic analysis contains parameters. Since parameters may take as value any element from an underlying domain, say the domain of monomorphic types, it is necessary to take into account all possible assignments of values to parameters in order to formalise a polymorphic abstraction. In the sequel, an assignment will always mean an assignment of values to the paramenters that serve as placeholders for information to be provided after analysis.

Definition 3.1 *Let $\mathcal{A}$ be the set of assignments, $I = \langle(\mathcal{D}, \sqsubseteq), (\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_k)\rangle$ and $I^\bullet = \langle(\mathcal{D}^\bullet, \sqsubseteq^\bullet), (\mathcal{O}_1^\bullet, \mathcal{O}_2^\bullet, \cdots, \mathcal{O}_k^\bullet)\rangle$ be two interpretations such that $\mathcal{O}_i$ and $\mathcal{O}_i^\bullet$ are of same arity n_i for each $1 \leq i \leq k$, and $\gamma^\bullet : \mathcal{D}^\bullet \times \mathcal{A} \mapsto \mathcal{D}$ be monotonic in its first argument. $I^\bullet$ is called a polymorphic $\gamma^\bullet$-abstraction of I with respect to $\mathcal{A}$ if, for each $1 \leq i \leq k$,*

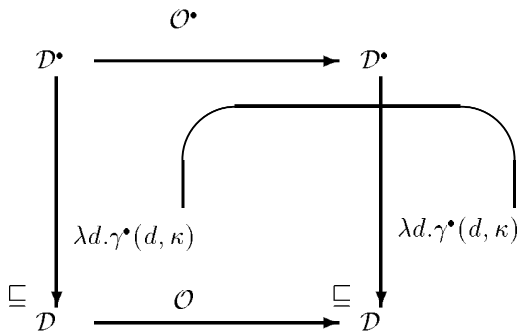


Figure 4: Polymorphic Analysis

- *for all* $\kappa \in \mathcal{A}$, *all* $d_1, \cdots, d_{n_i} \in \mathcal{D}$ *and all* $d_1^\bullet, \cdots, d_{n_i}^\bullet \in \mathcal{D}^\bullet$,

$$d_1 \sqsubseteq \gamma^\bullet(d_1^\bullet, \kappa) \wedge \cdots \wedge d_{n_i} \sqsubseteq \gamma^\bullet(d_{n_i}^\bullet, \kappa) \rightarrow \mathcal{O}_i(d_1, \cdots, d_{n_i}) \sqsubseteq \gamma^\bullet(\mathcal{O}_i^\bullet(d_1^\bullet, \cdots, d_{n_i}^\bullet), \kappa)$$

The above condition is read as that $\mathcal{O}_i^\bullet$ is a polymorphic $\gamma^\bullet$-abstraction of $\mathcal{O}_i$ with respect to $\mathcal{A}$. Therefore, $I^\bullet$ is called a polymorphic $\gamma^\bullet$-abstraction of I with respect to $\mathcal{A}$ iff each operator $\mathcal{O}^\bullet$ in $I^\bullet$ is a polymorphic $\gamma^\bullet$-abstraction of its corresponding operator $\mathcal{O}$ in I with respect to $\mathcal{A}$. Figure 4 is an illustration that $\mathcal{O}^\bullet$ is a polymorphic $\gamma^\bullet$-abstraction of $\mathcal{O}$ with respect to $\mathcal{A}$. With $\mathcal{A}$, $\gamma^\bullet$ and I being understood, $I^\bullet$ is called a polymorphic abstract interpretation. The notion of polymorphic abstract interpretation provides us better understanding of polymorphic analyses by abstract interpretation and simplifies the design and the proof of polymorphic analyses.

The following lemma states that many monomorphic abstract interpretations can be constructed from one polymorphic abstract interpretation. In other words, a polymorphic abstract interpretation can be instantiated into many monomorphic abstract interpretations. It also shows that the result of a polymorphic analysis subsumes those of many monomorphic analyses for they can be obtained by instantiating that of the polymorphic analysis.

Lemma 3.2 $I = \langle(\mathcal{D}, \sqsubseteq), (\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_k)\rangle$ *and* $I^\bullet = \langle(\mathcal{D}^\bullet, \sqsubseteq^\bullet), (\mathcal{O}_1^\bullet, \mathcal{O}_2^\bullet, \cdots, \mathcal{O}_k^\bullet)\rangle$ *be two interpretations such that* $I^\bullet$ *is a polymorphic* $\gamma^\bullet$*-abstraction of* I *with respect to* $\mathcal{A}$*. For each* $\kappa \in \mathcal{A}$*, define* $I_\kappa^\dagger \stackrel{\text{def}}{=} \langle(\mathcal{D}^\dagger, \sqsubseteq^\dagger), (\mathcal{O}_1^\dagger, \mathcal{O}_2^\dagger, \cdots, \mathcal{O}_k^\dagger)\rangle$ *where*

$$\begin{aligned}
\mathcal{D}^\dagger &\stackrel{\text{def}}{=} \mathcal{D}^\bullet \times \{\kappa\} \\
(d_1^\bullet, \kappa) \sqsubseteq^\dagger (d_2^\bullet, \kappa) &\stackrel{\text{def}}{=} d_1^\bullet \sqsubseteq^\bullet d_2^\bullet \\
\gamma^\dagger((d^\bullet, \kappa)) &\stackrel{\text{def}}{=} \gamma^\bullet(d^\bullet, \kappa) \\
\mathcal{O}_i^\dagger((d_1^\bullet, \kappa), \cdots, (d_{n_i}^\bullet, \kappa)) &\stackrel{\text{def}}{=} (\mathcal{O}_i^\bullet(d_1^\bullet, \cdots, d_{n_i}^\bullet), \kappa)
\end{aligned}$$

Then $I^\dagger_\kappa$ is an $\gamma^\dagger$-abstraction of I.

PROOF: The lemma follows immediately from definitions 2.1 and 3.1. ∎

Example 3.3 *In this example, we present a polymorphic abstract interpretation for parity analysis. The polymorphic abstract interpretation allows us to perform a polymorphic parity analysis whose effect corresponds to many monomorphic parity analyses based on the monomorphic abstract interpretation presented in example 2.2. This example is intended to show that the notion of polymorphic abstract interpretations is of generality and not confined to polymorphic type analysis or logic programs.*

Let $\mathcal{P}_p$ be set of parity parameters that are input to a polymorphic parity analysis and whose values are given when the result of the polymorphic parity analysis is instantiated. The set of assignments is $\mathcal{A}_p = \mathcal{P}_p \mapsto \aleph$.

Let $\mathcal{B}_p$ be boolean expressions formed of truth values true *and* false*, logical connectives $\wedge$ and $\vee$, and equations whose left hand sides are from $\mathcal{P}_p \cup \aleph$ and whose right hand sides are from $\aleph$. Define $\precsim \subseteq \wp(\aleph \times \mathcal{B}_p) \times \wp(\aleph \times \mathcal{B}_p)$ as follows.* [1]

$$\ddot{\mathbf{p}}_1 \precsim \ddot{\mathbf{p}}_2 \stackrel{\text{def}}{=} \forall \kappa \in \mathcal{A}_p.(\breve{\bigsqcup}\{\breve{\mathbf{p}}_1 \mid (\breve{\mathbf{p}}_1, \mathbf{b}_1) \in \ddot{\mathbf{p}}_1 \wedge \kappa(\mathbf{b}_1)\} \breve{\sqsubseteq} \breve{\bigsqcup}\{\breve{\mathbf{p}}_2 \mid (\breve{\mathbf{p}}_2, \mathbf{b}_2) \in \ddot{\mathbf{p}}_2 \wedge \kappa(\mathbf{b}_2)\})$$

$\precsim$ is reflexive and transitive but not antisymmetric since both $\{(\breve{e}, \mathsf{true}), (\breve{o}, \mathsf{true})\} \precsim \{(\breve{\top}, \mathsf{true})\}$ and $\{(\breve{\top}, \mathsf{true})\} \precsim \{(\breve{e}, true), (\breve{o}, true)\}$. Let $\simeq \subseteq \wp(\aleph \times \mathcal{B}_p) \times \wp(\aleph \times \mathcal{B}_p)$ be defined as in the following.

$$\ddot{\mathbf{p}}_1 \simeq \ddot{\mathbf{p}}_2 \stackrel{\text{def}}{=} \ddot{\mathbf{p}}_1 \precsim \ddot{\mathbf{p}}_2 \wedge \ddot{\mathbf{p}}_2 \precsim \ddot{\mathbf{p}}_1$$

$\simeq$ is an equivalence relation on $\wp(\aleph \times \mathcal{B}_p)$. Let

$$\begin{aligned} \Im &\stackrel{\text{def}}{=} \wp(\aleph \times \mathcal{B}_p)_{/\simeq} \\ \leqslant &\stackrel{\text{def}}{=} \precsim_{/\simeq} \end{aligned}$$

$\leqslant$ is a partial order on $\Im$. $(\Im, \leqslant)$ is a complete lattice since, for any $[\ddot{\mathbf{p}}_1]_\simeq, [\ddot{\mathbf{p}}_2]_\simeq \in \Im$, $[\ddot{\mathbf{p}}_1 \cup \ddot{\mathbf{p}}_2]_\simeq$ is the least upper bound of $\{[\ddot{\mathbf{p}}_1]_\simeq, [\ddot{\mathbf{p}}_2]_\simeq\}$ and $[\{\ddot{\mathbf{x}} \mid \ddot{\mathbf{x}} \precsim \ddot{\mathbf{p}}_1\} \cap \{\ddot{\mathbf{y}} \mid \ddot{\mathbf{y}} \precsim \ddot{\mathbf{p}}_2\}]_\simeq$ is the greatest lower bound of $\{[\ddot{\mathbf{p}}_1]_\simeq, [\ddot{\mathbf{p}}_2]_\simeq\}$. Note that elements of $\Im$ contain parameters since those of $\mathcal{B}_p$ contain parameters.

$\Im$ is the domain of polymorphic parity descriptions. Each polymorphic parity descripition is an equivalence class, with respect to $\simeq$, of a set of pairs consisting of a parity and a boolean expression. For a given assignment of parities to parameters, a polymorphic parity description denotes the least upper bound of the parities in all the pairs whose boolean expressions evaluate to true, which is modelled by the following concretisation function $\ddot{\gamma} : \Im \times \mathcal{A}_p \mapsto \wp(\mathsf{nat})$.

$$\ddot{\gamma}([\ddot{\mathbf{p}}]_\simeq, \kappa) \stackrel{\text{def}}{=} \breve{\gamma}(\breve{\bigsqcup}\{\breve{\mathbf{p}} \mid (\breve{\mathbf{p}}, \mathbf{b}) \in \ddot{\mathbf{p}} \wedge \kappa(\mathbf{b})\})$$

$\ddot{\gamma}$ is monotonic in its first argument since $\breve{\gamma}$ is monotonic.

---

[1] Let P be a proposition. $P \leftrightarrow \mathit{true}$ is written as P and $P \leftrightarrow \mathit{false}$ as $\neg P$ when there is no ambiguity.

For simplicity, we have restricted occurrences of parameters to boolean expressions. Because of this restriction, a parity parameter β will be expressed as

$$[\{(\breve{\bot},\beta=\breve{\bot}),(\breve{e},\beta=\breve{e}),(\breve{o},\beta=\breve{o}),(\breve{\top},\beta=\breve{\top})\}]_{\backsimeq}$$

and a parity $\breve{\mathbf{p}}$ is represented as $[\{(\breve{\mathbf{p}},\mathrm{true})\}]_{\backsimeq}$

The abstract operators corresponding to $\oplus$ and $\otimes$ are respectively $\ddot{\oplus},\ddot{\otimes}:\Im\times\Im\mapsto\Im$ defined as follows.

$$[\ddot{\mathbf{p}}_1]_{\backsimeq}\ddot{\oplus}[\ddot{\mathbf{p}}_2]_{\backsimeq}\stackrel{\mathrm{def}}{=}[\{(\breve{\mathbf{p}}_1\breve{\oplus}\breve{\mathbf{p}}_2,\mathbf{b}_1\wedge\mathbf{b}_2)\mid(\breve{\mathbf{p}}_1,\mathbf{b}_1)\in\ddot{\mathbf{p}}_1\wedge(\breve{\mathbf{p}}_2,\mathbf{b}_2)\in\ddot{\mathbf{p}}_2\}]_{\backsimeq}$$

$$[\ddot{\mathbf{p}}_1]_{\backsimeq}\ddot{\otimes}[\ddot{\mathbf{p}}_2]_{\backsimeq}\stackrel{\mathrm{def}}{=}[\{(\breve{\mathbf{p}}_1\breve{\otimes}\breve{\mathbf{p}}_2,\mathbf{b}_1\wedge\mathbf{b}_2)\mid(\breve{\mathbf{p}}_1,\mathbf{b}_1)\in\ddot{\mathbf{p}}_1\wedge(\breve{\mathbf{p}}_2,\mathbf{b}_2)\in\ddot{\mathbf{p}}_2\}]_{\backsimeq}$$

What remains to complete the design of the polymorphic abstract interpretation for parity analysis is to prove that $\ddot{\oplus}$ and $\ddot{\otimes}$ are respectively polymorphic $\ddot{\gamma}$-abstractions of $\oplus$ and $\otimes$. Let $\kappa\in\mathcal{A}_p$, $\ddot{\mathbf{x}},\ddot{\mathbf{y}}\in\Im$. Suppose $\mathbf{x}\subseteq\ddot{\gamma}(\ddot{\mathbf{x}},\kappa)$ and $\mathbf{y}\subseteq\ddot{\gamma}(\ddot{\mathbf{y}},\kappa)$. By the definition of $\ddot{\gamma}$,

$$\begin{array}{rcl}\mathbf{x}&\subseteq&\breve{\gamma}(\breve{\bigsqcup}\{\breve{\mathbf{x}}\mid(\breve{\mathbf{x}},\mathbf{b}_{\breve{\mathbf{x}}})\in\ddot{\mathbf{x}}\wedge\kappa(\mathbf{b}_{\breve{\mathbf{x}}})\})\\ \mathbf{y}&\subseteq&\breve{\gamma}(\breve{\bigsqcup}\{\breve{\mathbf{y}}\mid(\breve{\mathbf{y}},\mathbf{b}_{\breve{\mathbf{y}}})\in\ddot{\mathbf{y}}\wedge\kappa(\mathbf{b}_{\breve{\mathbf{y}}})\})\end{array}$$

Since $\breve{\oplus}$ is a $\breve{\gamma}$-abstraction of $\oplus$, we have that

$$\begin{array}{rcl}\mathbf{x}\oplus\mathbf{y}&\subseteq&\breve{\gamma}(\breve{\bigsqcup}\{\breve{\mathbf{x}}\mid(\breve{\mathbf{x}},\mathbf{b}_{\breve{\mathbf{x}}})\in\ddot{\mathbf{x}}\wedge\kappa(\mathbf{b}_{\breve{\mathbf{x}}})\}\ \breve{\oplus}\ \breve{\bigsqcup}\{\breve{\mathbf{y}}\mid(\breve{\mathbf{y}},\mathbf{b}_{\breve{\mathbf{y}}})\in\ddot{\mathbf{y}}\wedge\kappa(\mathbf{b}_{\breve{\mathbf{y}}})\})\\ &&\textit{by the definition of } \breve{\oplus}\\ &=&\breve{\gamma}(\breve{\bigsqcup}\{\breve{\mathbf{x}}\breve{\oplus}\breve{\mathbf{y}}\mid(\breve{\mathbf{x}},\mathbf{b}_{\breve{\mathbf{x}}})\in\ddot{\mathbf{x}}\wedge\kappa(\mathbf{b}_{\breve{\mathbf{x}}})\wedge(\breve{\mathbf{y}},\mathbf{b}_{\breve{\mathbf{y}}})\in\ddot{\mathbf{y}}\wedge\kappa(\mathbf{b}_{\breve{\mathbf{y}}})\})\\ &&\textit{by the definition of } \ddot{\gamma}\\ &=&\ddot{\gamma}(\{(\breve{\mathbf{x}}\breve{\oplus}\breve{\mathbf{y}},\mathbf{b}_{\breve{\mathbf{x}}}\wedge\mathbf{b}_{\breve{\mathbf{y}}})\mid(\breve{\mathbf{x}},\mathbf{b}_{\breve{\mathbf{x}}})\in\ddot{\mathbf{x}}\wedge(\breve{\mathbf{y}},\mathbf{b}_{\breve{\mathbf{y}}})\in\ddot{\mathbf{y}}\},\kappa)\\ &&\textit{by the definition of } \ddot{\oplus}\\ &=&\ddot{\gamma}(\ddot{\mathbf{x}}\ddot{\oplus}\ddot{\mathbf{y}},\kappa)\end{array}$$

Therefore, $\ddot{\oplus}$ is a polymorphic $\ddot{\gamma}$-abstraction of $\oplus$ with respect to $\mathcal{A}_p$. It can be similarly proved that $\ddot{\otimes}$ is a polymorphic $\ddot{\gamma}$-abstraction of $\otimes$ with respect to $\mathcal{A}_p$. So, $\ddot{I}_{parity}=\langle(\Im,\leqslant),(\ddot{\oplus},\ddot{\otimes})\rangle$ is a polymorphic $\ddot{\gamma}$-abstraction of $I_{parity}=\langle(\wp(\mathrm{nat}),\subseteq),(\oplus,\otimes)\rangle$ with respect to $\mathcal{A}_p$.

We conclude this example by calculating a polymorphic parity description of an expression $(X\times X)+(X\times Y)$ with Y being an even natural number. The expression to be abstracted is thus $(\mathbf{x}\otimes\mathbf{x})\oplus(\mathbf{x}\otimes\mathbf{y})$ with $\mathbf{x}$ being unknown and $\mathbf{y}=\{2n\mid n\in\mathrm{nat}\}$. The polymorphic parity description of $\mathbf{y}$ is $\ddot{\mathbf{y}}=[\{(\breve{e},\mathrm{true})\}]_{\backsimeq}$ and that of $\mathbf{x}$ is

$$\ddot{\mathbf{x}}=[\{(\breve{\bot},\beta=\breve{\bot}),(\breve{e},\beta=\breve{e}),(\breve{o},\beta=\breve{o}),(\breve{\top},\beta=\breve{\top})\}]_{\backsimeq}$$

with β being a parity parameter. The polymorphic $\ddot{\gamma}$-abstraction of $(\mathbf{x} \otimes \mathbf{x}) \oplus (\mathbf{x} \otimes \mathbf{y})$ with respect to $\mathcal{A}_p$ is $(\ddot{\mathbf{x}} \ddot{\otimes} \ddot{\mathbf{x}}) \ddot{\oplus} (\ddot{\mathbf{x}} \ddot{\otimes} \ddot{\mathbf{y}})$. We have

$$
\begin{array}{rcl}
(\ddot{\mathbf{x}} \ddot{\otimes} \ddot{\mathbf{x}}) \ddot{\oplus} (\ddot{\mathbf{x}} \ddot{\otimes} \ddot{\mathbf{y}}) & = & [\{(\check{\bot}, \beta = \check{\bot}), (\check{e}, \beta = \check{e}), (\check{o}, \beta = \check{o}), (\check{\top}, \beta = \check{\top})\}]_{\simeq} \\
 & & \ddot{\oplus} \\
 & & [\{(\check{\bot}, \beta = \check{\bot}), (\check{e}, \beta = \check{e} \vee \beta = \check{o} \vee \beta = \check{\top})\}]_{\simeq} \\
 & = & [\{(\check{\bot}, \beta = \check{\bot}), (\check{e}, \beta = \check{e}), (\check{o}, \beta = \check{o}), (\check{\top}, \beta = \check{\top})\}]_{\simeq} \\
 & = & \ddot{\mathbf{x}}
\end{array}
$$

This indicates that if Y is an even natural number then $(X \times X) + (X \times Y)$ has the same parity as X. If $(X \times X) + (X \times Y)$ occurs in several places where Y has the same parity $\check{e}$ while X has different parities, then the parity of $(X \times X) + (X \times Y)$ in each of these places can be retrieved from the polymorphic parity description $[\{(\check{\bot}, \beta = \check{\bot}), (\check{e}, \beta = \check{e}), (\check{o}, \beta = \check{o}), (\check{\top}, \beta = \check{\top})\}]_{\simeq}$ by substituting an appropriate parity for β and taking the least upper bound of the parities paired with boolean expressions evaluating to true, in this case, choosing the parity paired with the only boolean expression evaluating to true.

4 TYPES AND THEIR MEANINGS

A *type* is a finite expression denoting a possibly infinite set of values. Values in logic programs are terms. A type denotes a set of terms. Syntactically, a type is a term constructible from a set Cons of *type constructors* and a denumerable set Para of *type parameters*. The set of types is denoted as Type. If a type constructor has an arity of zero then it is called a *base type*. There are two special base types $\mathbf{1}$ and $\mathbf{0}$, denoting Term and $\emptyset$ respectively. A type is a mono-type if it does not contain any type parameter. The set of mono-types is denoted by Mono. Mono-types in Mono can be compared according to a partial order that is inductively defined as follows.

Definition 4.1 *Let $M, M_1, \cdots, M_m, N_1, \cdots, N_m \in \mathsf{Mono}$ and $c/m \in \mathsf{Cons}$.*

$$
\begin{array}{rcl}
\mathbf{0} & \precsim & M \\
M & \precsim & \mathbf{1} \\
c(M_1, \cdots, M_m) & \precsim & c(N_1, \cdots, N_m) \stackrel{\text{def}}{=} \forall 1 \leq j \leq m.(M_j \precsim N_j)
\end{array}
$$

Let $\mathsf{Cons} = \{nat/0, \mathbf{1}/0, \mathbf{0}/0, list/1\}$. Then $\mathbf{0} \precsim nat$, $list(nat) \precsim \mathbf{1}$, $list(\mathbf{0}) \precsim list(nat)$ and $nat \not\precsim list(nat)$.

$\langle \mathsf{Mono}, \precsim, \mathbf{0}, \mathbf{1}, \barwedge, \veebar \rangle$ is a complete lattice where, letting $M = c(M_1, \cdots, M_m) \in \mathsf{Mono}$ and $N = d(N_1, \cdots, N_k) \in \mathsf{Mono}$, the least upper bound operator $\veebar$ is

$$
\begin{array}{lllll}
\mathbf{0} \veebar M & = & M \veebar \mathbf{0} & = & M \\
\mathbf{1} \veebar M & = & M \veebar \mathbf{1} & = & \mathbf{1} \\
M \veebar N & = & N \veebar M & = & \left\{ \begin{array}{ll} c(M_1 \veebar N_1, \cdots, M_m \veebar N_m) & \text{if } c/m = d/k \\ \mathbf{1} & \text{otherwise} \end{array} \right.
\end{array}
$$

and the greatest lower bound operator $\curlywedge$ is

$$\begin{array}{rclcl}
\mathbf{0} \curlywedge M & = & M \curlywedge \mathbf{0} & = & \mathbf{0} \\
\mathbf{1} \curlywedge M & = & M \curlywedge \mathbf{1} & = & M \\
M \curlywedge N & = & N \curlywedge M & = & \left\{ \begin{array}{ll} c(M_1 \curlywedge N_1, \cdots, M_m \curlywedge N_m) & \mathit{if}\ c/m = d/k \\ \mathbf{0} & \mathit{otherwise} \end{array} \right.
\end{array}$$

A type is a general type if it is either a type parameter or a base type except $\mathbf{0}$ or of the form $c(\beta_1, \cdots, \beta_m)$ where $c/m \in \mathsf{Cons}$ and $\beta_1, \cdots, \beta_m \in \mathsf{Para}$ are different type parameters. For instance, $\mathit{list}(\beta)$ is a general type while $\mathit{list}(nat)$ and $\mathit{list}(\mathit{list}(\beta))$ are not. The set of general types is denoted as Gen. The types of terms and the sets of terms denoted by types are both decided by type definitions for function symbols in Func.

Definition 4.2 *A type clause is a clause for predicate* $:/2$ *which is of the form*

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

where $m, n \geq 0$*,* f/n *is a function symbol,* c/m *is a type constructor,* $\beta_1, \cdots, \beta_m$ *are different type parameters,* $X_1, \cdots, X_n$ *are different program variables, and, for* $1 \leq \imath \leq n$*,* $T_\imath$ *is a general type.*

In addition to the above restriction on the form of a type clause, we also require that (i) each function symbol f/n *is typed by exactly one type clause; and (ii)* $pars(T_\jmath) \subseteq \{\beta_1, \cdots, \beta_m\}$ *for* $1 \leq \jmath \leq n$ *where* $pars(T)$ *is the set of the parameters occurring in* T*. (i) disallows overloading [7] and (ii) is often referred to as type preserving [45].*

Example 4.3 *This example illustrates some type definitions.*

$$\begin{array}{rcll}
0 : nat & \leftarrow & & \%\ D1 \\
s(X) : nat & \leftarrow & X : nat & \%\ D2 \\
[\,] : \mathit{list}(\beta') & \leftarrow & & \%\ D3 \\
[H|L] : \mathit{list}(\beta'') & \leftarrow & H : \beta'', L : \mathit{list}(\beta'') & \%\ D4
\end{array}$$

D1 says that 0 *is a member of type* nat*. D2 means that if* X *has type* nat *then* $s(X)$ *has type* nat*. D1 and D2 also gives the set of terms denoted by type* nat*. D3 states that* $[\,]$ *is a member of type* $\mathit{list}(\beta')$ *for any type* β' *while D4 reads that, for any type* β''*, if* H *is a member of type* β'' *and* L *a member of type* $\mathit{list}(\beta'')$ *then* $[H|L]$ *is a member of type* $\mathit{list}(\beta'')$*.*

A type substitution $\Bbbk$ is a set of formulae $\beta \mapsto T$ where β is a type parameter and T is a type. If $\Bbbk = \{\beta_1 \mapsto T_1, \cdots, \beta_n \mapsto T_n\}$ then $dom(\Bbbk) = \{\beta_1, \cdots, \beta_n\}$ is the domain of $\Bbbk$. $\Bbbk$ is a mono-type substitution if $T_1, \cdots, T_n$ are mono-types. The application of $\Bbbk$ to a type is to replace simultaneously each occurrence of β_i in that type with T_i. For instance, $\{\beta_1 \mapsto \mathit{list}(nat), \beta_2 \mapsto nat\}(\mathit{list}(\beta_1)) = \mathit{list}(\mathit{list}(nat))$. $\Bbbk(\beta) \stackrel{def}{=} \mathbf{0}$ for any $\beta \notin dom(\Bbbk)$. The choice of $\mathbf{0}$ as the default value for parameters outside the domain of a type substitution is motivated by computational consideration only. Both polymorphic and monomorphic type analyses frequently compute the least upper bounds of type substitutions, this choice simplifies these computations. To facilitate presentation, we introduce a special type substitution $\bowtie$

and define $\bowtie(T) \stackrel{def}{=} \mathbf{1}$ for any type T. The type of a term $f(t_1, \cdots, t_n)$ will be computed from types of terms $t_1, \cdots, t_n$ by applying the type clause for f/n. The role of $\bowtie$ is to indicate that types of $t_1, \cdots, t_n$ do not satisfy the type conditions in the type clause, and the least type of $f(t_1, \cdots, t_n)$ is $\mathbf{1}$ in this case.

Definition 4.4 *The meaning of a mono-type is defined as follows. Let $M \in$ Mono.*

$$\gamma(M) \stackrel{\text{def}}{=} \begin{cases} \emptyset & if\ M = \mathbf{0} \\ \mathsf{Term} & if\ M = \mathbf{1} \\ \left\{ f(t_1, \cdots, t_n) \,\middle|\, \begin{array}{c} f(X_1,\cdots,X_n):c(\beta_1,\cdots,\beta_m) \leftarrow X_1:T_1,\cdots,X_n:T_n \\ t_i \in \gamma(\{\beta_j \mapsto M_j | 1 \le j \le m\}(T_i)) \end{array} \right\} & \\ & if\ M = c(M_1, \cdots, M_m) \end{cases}$$

In the above equation, $\{\beta_j \mapsto M_j | 1 \le j \le m\}$ is a type substitution. Since M_j is a mono-type and $\{\beta_1, \cdots, \beta_m\}$ contains all the type parameters in T_i, $\{\beta_j \mapsto M_j | 1 \le j \le m\}(T_i)$ is a mono-type. γ is monotonic and $\gamma(M)$ is closed under instantiation [11] for any $M \in$ Mono, for which proofs can be found in [31].

Example 4.5 *Let* $\mathsf{Cons} = \{nat/0, \mathbf{1}/0, \mathbf{0}/0, list/1\}$, $\mathsf{Func} = \{0/0, s/1, [\,]/0, ./2\}$, $\mathsf{Vars} = \{X, Y, \cdots\}$, *and the type definitions for function symbols in* Func *be those in example 4.3.*

$$\begin{aligned} \gamma(list(\mathbf{0})) &= \{[\,]\} \\ \gamma(list(nat)) &= \{[\,], [0], \cdots, [0, s(0)], \cdots\} \\ \gamma(list(\mathbf{1})) &= \{[\,], \cdots, [X], \cdots, [Y, s(0)], \cdots\} \end{aligned}$$

5 MONOMORPHIC TYPE ANALYSIS

This section presents the monomorphic type analysis proposed in [22, 27, 28] and later developed in [31]. This section is based on [31] where monomorphic type information is inferred together with sharing and alising and the soundness of the monomorphic type analysis is proved.

5.1 Abstract domain

This subsection constructs the abstract domain for the monomorphic type analysis and two other abstract domains on which the ancillary operators that we use to define the abstract unification operator are defined. The first abstract domain is the complete lattice $(\mathsf{Mono}, \preceq)$ of mono-types.

The second abstract domain is the complete lattice $(\mathsf{MTS}, \sqsubseteq)$ of mono-type substitutions that is constructed as follows. A mono-type substitution $\Bbbk$ as defined in section 4 is either a partial function from Para to Mono or $\bowtie$. Let $\Bbbk \in \mathsf{Para} \rightarrowtail \mathsf{Mono} \cup \{\bowtie\}$ and $\Bbbk_1, \Bbbk_2 \in$

$\mathsf{Para}\rightarrowtail\!\!\!\rightarrow\mathsf{Mono}$ and define $\preceq$ as follows.

$$\begin{array}{lcl} \mathbb{k}\preceq\bowtie & & \\ \mathbb{k}_1\preceq\mathbb{k}_2 & \stackrel{def}{=} & \forall\beta\in\mathsf{Para}.(\mathbb{k}_1(\beta)\precsim\mathbb{k}_2(\beta)) \end{array}$$

$\preceq$ is a preorder on $(\mathsf{Para}\rightarrowtail\!\!\!\rightarrow\mathsf{Mono}\cup\{\bowtie\})$. It is reflexive and transitive but not antisymmetric since both $\{\beta\mapsto\mathbf{0}\}\preceq\emptyset$ and $\emptyset\preceq\{\beta\mapsto\mathbf{0}\}$. Let $\mathbb{k}\approx\mathbb{k}'\stackrel{def}{=}(\mathbb{k}\preceq\mathbb{k}')\wedge(\mathbb{k}'\preceq\mathbb{k})$. $\approx$ is an equivalence relation on $(\mathsf{Para}\rightarrowtail\!\!\!\rightarrow\mathsf{Mono}\cup\{\bowtie\})$. The domain of mono-type substitutions is the set of equivalence classes of $\approx$ ordered by $\preceq_{/\approx}$. Define

$$\begin{array}{rcll} \mathsf{MTS} & \stackrel{def}{=} & (\mathsf{Para}\rightarrowtail\!\!\!\rightarrow\mathsf{Mono}\cup\{\bowtie\})_{\approx} & \\ \sqsubseteq & \stackrel{def}{=} & \preceq_{/\approx} & \\ [\mathbb{k}_1]_{\approx}\sqcup[\mathbb{k}_2]_{\approx} & \stackrel{def}{=} & \left\{\begin{array}{ll} [\bowtie]_{\approx} & \text{if } \mathbb{k}_1=\bowtie\vee\mathbb{k}_2=\bowtie \\ [\{\beta\mapsto\mathbb{k}_1(\beta)\veebar\mathbb{k}_2(\beta)\mid\beta\in dom(\mathbb{k}_1)\cup dom(\mathbb{k}_2)\}]_{\approx} & \text{otherwise} \end{array}\right. & \\ [\mathbb{k}_1]_{\approx}\sqcap[\mathbb{k}_2]_{\approx} & \stackrel{def}{=} & \left\{\begin{array}{ll} [\mathbb{k}_2]_{\approx} & \text{if } \mathbb{k}_1=\bowtie \\ [\mathbb{k}_1]_{\approx} & \text{if } \mathbb{k}_2=\bowtie \\ [\{\beta\mapsto\mathbb{k}_1(\beta)\barwedge\mathbb{k}_2(\beta)\mid\beta\in dom(\mathbb{k}_1)\cap dom(\mathbb{k}_2)\}]_{\approx} & \text{if } \mathbb{k}_1\neq\bowtie\wedge\mathbb{k}_2\neq\bowtie \end{array}\right. & \\ \top & \stackrel{def}{=} & [\bowtie]_{\approx} & \\ \bot & \stackrel{def}{=} & [\emptyset]_{\approx} & \end{array}$$

Lemma 5.1 $\langle\mathsf{MTS},\sqsubseteq,\bot,\top,\sqcup,\sqcap\rangle$ *is a complete lattice.*

PROOF: It is easy to see that $\bot\sqsubseteq[\mathbb{k}]_{\approx}$ and $[\mathbb{k}]_{\approx}\sqsubseteq\top$ for any $[\mathbb{k}]_{\approx}\in\mathsf{MTS}$. We now prove that $[\mathbb{k}_1]_{\approx}\sqcup[\mathbb{k}_2]_{\approx}$ is the least upper bound of $[\mathbb{k}_1]_{\approx}$ and $[\mathbb{k}_2]_{\approx}$. Assume $\mathbb{k}_1\neq\bowtie$ and $\mathbb{k}_2\neq\bowtie$ for otherwise $[\mathbb{k}_1]_{\approx}\sqcup[\mathbb{k}_2]_{\approx}$ is obviously the least upper bound of $[\mathbb{k}_1]_{\approx}$ and $[\mathbb{k}_2]_{\approx}$. Let $\mathbb{k}=\{\beta\mapsto\mathbb{k}_1(\beta)\veebar\mathbb{k}_2(\beta)\mid\beta\in dom(\mathbb{k}_1)\cup dom(\mathbb{k}_2)\}$. We have $[\mathbb{k}_1]_{\approx}\sqsubseteq[\mathbb{k}]_{\approx}$, $[\mathbb{k}_2]_{\approx}\sqsubseteq[\mathbb{k}]_{\approx}$ and $[\mathbb{k}_1]_{\approx}\sqcup[\mathbb{k}_2]_{\approx}=[\mathbb{k}]_{\approx}$ proving that $[\mathbb{k}_1]_{\approx}\sqcup[\mathbb{k}_2]_{\approx}$ is an upper bound of $[\mathbb{k}_1]_{\approx}$ and $[\mathbb{k}_2]_{\approx}$. Now suppose $[\mathbb{k}']_{\approx}\in\mathsf{MTS}$ and $[\mathbb{k}_1]_{\approx}\sqsubseteq[\mathbb{k}']_{\approx}$ and $[\mathbb{k}_2]_{\approx}\sqsubseteq[\mathbb{k}']_{\approx}$. If $\mathbb{k}'=\bowtie$ then $\mathbb{k}\preceq\mathbb{k}'$. Let $\mathbb{k}'\neq\bowtie$. By the definition of $\sqsubseteq$, $\forall\beta\in dom(\mathbb{k}_1).\mathbb{k}_1(\beta)\preceq\mathbb{k}'(\beta)$ and $\forall\beta\in dom(\mathbb{k}_2).\mathbb{k}_2(\beta)\preceq\mathbb{k}'(\beta)$. Hence, $\forall\beta\in dom(\mathbb{k}_1)\cup dom(\mathbb{k}_2).(\mathbb{k}_1(\beta)\veebar\mathbb{k}_2(\beta)\preceq\mathbb{k}'(\beta))$. So, $\mathbb{k}\preceq\mathbb{k}'$ when $\mathbb{k}'\neq\bowtie$. Therefore, $[\mathbb{k}]_{\approx}\sqsubseteq[\mathbb{k}']_{\approx}$ proving that $[\mathbb{k}_1]_{\approx}\sqcup[\mathbb{k}_2]_{\approx}$ is the least upper bound of $[\mathbb{k}_1]_{\approx}$ and $[\mathbb{k}_2]_{\approx}$. It can be similarly proved that $[\mathbb{k}_1]_{\approx}\sqcap[\mathbb{k}_2]_{\approx}$ is the greatest lower bound of $[\mathbb{k}_1]_{\approx}$ and $[\mathbb{k}_2]_{\approx}$. ∎

It follows that the application of any equivalent (with respect to $\approx$) mono-type substitutions to any type will result in the same mono-type. Therefore, an equivalence class of $\approx$ will be denoted by any mono-type substitution in the equivalence class and elements of MTS will be called mono-type substitutions.

The third abstract domain is the complete lattice of abstract substitutions for the monomorphic type analysis. For the monomorphic type analysis, a set of substitutions is described by associating mono-types with program variables of interest. A *variable typing* μ is a finite set of formulae $X\mapsto T$ where $X\in\mathsf{Vars}$ and $T\in\mathsf{Type}$. Let $\mu=\{X_1\mapsto T_1,\cdots,X_n\mapsto T_n\}$. Then $\{X_1,\cdots,X_n\}$ is the domain of μ, denoted by $dom(\mu)$. If $(X\mapsto T)\in\mu$ then $\mu(X)\stackrel{def}{=}T$.

Otherwise, $\mu(X) \stackrel{def}{=} \mathbf{1}$ indicating that no information is known about X. μ is a *variable monotyping* if $T_i \in \mathsf{Mono}$ for $1 \leq i \leq n$. Let $\mathsf{Vars}{\rightarrowtail}\mathsf{Mono}$ be the set of finite partial functions from Vars to Mono and define

$$\mu \preceq^{\circ} \nu \stackrel{def}{=} \exists V \in dom(\mu).(\mu(V) = \mathbf{0}) \vee \forall V \in dom(\nu).(\mu(V) \preceqq \nu(V))$$

$\preceq^{\circ}$ is a preorder on $\mathsf{Vars}{\rightarrowtail}\mathsf{Mono}$. $\preceq^{\circ}$ is reflexive and transitive but not antisymmetric since both $\{X \mapsto \mathbf{1}, Y \mapsto nat\} \preceq^{\circ} \{Y \mapsto nat, Z \mapsto \mathbf{1}\}$ and $\{Y \mapsto nat, Z \mapsto \mathbf{1}\} \preceq^{\circ} \{X \mapsto \mathbf{1}, Y \mapsto nat\}$. Let $\mu \approx^{\circ} \nu \stackrel{def}{=} (\mu \preceq^{\circ} \nu) \wedge (\nu \preceq^{\circ} \mu)$. $\approx^{\circ}$ is an equivalence relation on $\mathsf{Vars}{\rightarrowtail}\mathsf{Mono}$. For instance, $\{X \mapsto \mathbf{1}, Y \mapsto nat\} \approx^{\circ} \{Y \mapsto nat, Z \mapsto \mathbf{1}\}$. The abstract domain for the monomorphic type analysis is the set of equivalence classes of $\approx^{\circ}$ ordered by $\preceq^{\circ}_{/\approx^{\circ}}$.

$$\begin{array}{rcl}
\mathsf{VMT} & \stackrel{def}{=} & (\mathsf{Vars}{\rightarrowtail}\mathsf{Mono})_{/\approx^{\circ}} \\
\sqsubseteq^{\circ} & \stackrel{def}{=} & \preceq^{\circ}_{/\approx^{\circ}} \\
[\mu]_{\approx^{\circ}} \sqcup^{\circ} [\nu]_{\approx^{\circ}} & \stackrel{def}{=} & [\{V \mapsto (\mu(V) \veebar \nu(V)) \mid V \in dom(\mu) \cap dom(\nu)\}]_{\approx^{\circ}} \\
[\mu]_{\approx^{\circ}} \sqcap^{\circ} [\nu]_{\approx^{\circ}} & \stackrel{def}{=} & [\{V \mapsto (\mu(V) \barwedge \nu(V)) \mid V \in dom(\mu) \cup dom(\nu)\}]_{\approx^{\circ}} \\
\top^{\circ} & \stackrel{def}{=} & [\emptyset]_{\approx^{\circ}} \\
\bot^{\circ} & \stackrel{def}{=} & [\mu]_{\approx^{\circ}} \\
 & & \text{with } \exists V \in dom(\mu).\mu(V) = \mathbf{0}
\end{array}$$

Note that the domain of $[\mu]_{\approx^{\circ}} \sqcup^{\circ} [\nu]_{\approx^{\circ}}$ is $dom(\mu) \cap dom(\nu)$ and that of $[\mu]_{\approx^{\circ}} \sqcap^{\circ} [\nu]_{\approx^{\circ}}$ is $dom(\mu) \cup dom(\nu)$. This is the consequence of choosing $\mathbf{1}$ as the default value for variables.

Lemma 5.2 $\langle \mathsf{VMT}, \sqsubseteq^{\circ}, \bot^{\circ}, \top^{\circ}, \sqcap^{\circ}, \sqcup^{\circ} \rangle$ *is a complete lattice.*

Proof: It is immediate that $\bot^{\circ} \sqsubseteq^{\circ} [\mu]_{\approx^{\circ}}$ and $[\mu]_{\approx^{\circ}} \sqsubseteq^{\circ} \top^{\circ}$ for any $[\mu]_{\approx^{\circ}} \in \mathsf{VMT}$. We now prove that $[\mu]_{\approx^{\circ}} \sqcap^{\circ} [\nu]_{\approx^{\circ}}$ is the greatest lower bound of $[\mu]_{\approx^{\circ}}$ and $[\nu]_{\approx^{\circ}}$. Let $\xi = \{V \mapsto \mu(V) \barwedge \nu(V) \mid V \in dom(\mu) \cup dom(\nu)\}$. We have $[\xi]_{\approx^{\circ}} \sqsubseteq^{\circ} [\mu]_{\approx^{\circ}}$, $[\xi]_{\approx^{\circ}} \sqsubseteq^{\circ} [\nu]_{\approx^{\circ}}$ and $[\mu]_{\approx^{\circ}} \sqcap^{\circ} [\nu]_{\approx^{\circ}} = [\xi]_{\approx^{\circ}}$, proving that $[\mu]_{\approx^{\circ}} \sqcap^{\circ} [\nu]_{\approx^{\circ}}$ is a lower bound of $[\mu]_{\approx^{\circ}}$ and $[\nu]_{\approx^{\circ}}$. Let $[\varphi]_{\approx^{\circ}} \in \mathsf{VMT}$ be another lower bound of $[\mu]_{\approx^{\circ}}$ and $[\nu]_{\approx^{\circ}}$. We have $[\varphi]_{\approx^{\circ}} \sqsubseteq^{\circ} [\mu]_{\approx^{\circ}}$ and $[\varphi]_{\approx^{\circ}} \sqsubseteq^{\circ} [\nu]_{\approx^{\circ}}$. By the definition of $\sqsubseteq^{\circ}$, we have both $\exists V \in dom(\varphi).\varphi(V) = \mathbf{0} \vee \forall V \in dom(\mu).(\varphi(V) \preceqq \mu(V))$ and $\exists V \in dom(\varphi).\varphi(V) = \mathbf{0} \vee \forall V \in dom(\mu).(\varphi(V) \preceqq \mu(V))$. Therefore, $\exists V \in dom(\varphi).\varphi(V) = \mathbf{0} \vee \forall V \in dom(\mu) \cup dom(\nu).(\varphi(V) \preceqq \mu(V) \barwedge \nu(V))$ that is equivalent to $\exists V \in dom(\varphi).\varphi(V) = \mathbf{0} \vee \forall V \in dom(\xi).(\varphi(V) \preceqq \xi(V))$. So, $[\varphi]_{\approx^{\circ}} \sqsubseteq^{\circ} [\xi]_{\approx^{\circ}}$ proving that $[\mu]_{\approx^{\circ}} \sqcap^{\circ} [\nu]_{\approx^{\circ}} = [\xi]_{\approx^{\circ}}$ is the greatest lower bound of $[\mu]_{\approx^{\circ}}$ and $[\nu]_{\approx^{\circ}}$. It can be similarly proved that $[\mu]_{\approx^{\circ}} \sqcup^{\circ} [\nu]_{\approx^{\circ}}$ is the least upper bound of $[\mu]_{\approx^{\circ}}$ and $[\nu]_{\approx^{\circ}}$. ∎

Define $\gamma^{\circ} : \mathsf{VMT} \mapsto \wp(Sub)$ as follows.

$$\gamma^{\circ}([\mu]_{\approx^{\circ}}) \stackrel{def}{=} \{\theta \mid \forall V \in dom(\mu).(\theta(V) \in \gamma^{\ddagger} \circ \mu(V))\}$$

γ° is monotonic since $\gamma^{\ddagger}$ is monotonic.

5.2 Abstract unification operator

The monomorphic type analysis requires an abstract unification operator $MUNIFY : \mathsf{Atom} \times \mathsf{VMT} \times \mathsf{Atom} \times \mathsf{VMT} \mapsto \mathsf{VMT}$ that is a γ°-abstraction of $UNIFY$, in addition to the least upper bound operator $\sqcup^{\circ} : \mathsf{VMT} \times \mathsf{VMT} \mapsto \mathsf{VMT}$ that is a γ°-abstraction of $\cup$. An equivalence class of $\approx^{\circ}$ will be represented by one of its members for simplicity. Thus, $[\mu]_{\approx^{\circ}}$ will be written as μ. Let $\mu, \nu \in \mathsf{VMT}$ and $A, B \in \mathsf{Atom}$. $MUNIFY(A, \mu, B, \nu)$ is computed by first performing pre-unification [43] and then propagating type information. A renaming substitution Ψ is first applied to A and μ such that $\Psi(vars(A) \cup dom(\mu)) \cap (vars(B) \cup dom(\nu)) = \emptyset$. $E = eq \circ mgu(\{\Psi(A) = B\})$ is then computed. If $E = \mathit{fail}$ then $MUNIFY(A, \mu, B, \nu) = \bot^{\circ}$. If $E \neq \mathit{fail}$, an abstract substitution $\varpi \in \mathsf{VMT}$ is computed such that, for any $\zeta \in \gamma^{\circ}(\Psi(\mu) \cup \nu)$, if $mgu(\zeta(E)) \neq \mathit{fail}$ then $mgu(\zeta(E)) \circ \zeta \in \gamma^{\circ}(\varpi)$. This is accomplished by an operator $solve : \wp(\mathsf{Eqn}) \times \mathsf{VMT} \mapsto \mathsf{VMT}$. Finally, ϖ is restricted to $(vars(B) \cup dom(\nu))$ by $\restriction^{\circ} : \mathsf{VMT} \times \wp(\mathsf{Vars}) \mapsto \mathsf{VMT}$. $MUNIFY$ is defined in the following.

$$MUNIFY(A, \mu, B, \nu) \stackrel{def}{=} \left\{ \begin{array}{l} let \quad \Psi \text{ be a renaming substitution such that} \\ \qquad \Psi(vars(A) \cup dom(\mu)) \cap (vars(B) \cup dom(\nu)) = \emptyset, \\ \qquad E = eq \circ mgu(\Psi(A), B), \\ \qquad \mathsf{V} = (vars(B) \cup dom(\nu)) \\ in \\ \left\{ \begin{array}{ll} solve(E, \Psi(\mu) \cup \nu) \restriction^{\circ} \mathsf{V} & if\ E \neq \mathit{fail} \\ \bot^{\circ} & if\ E = \mathit{fail} \end{array} \right. \end{array} \right.$$

$$\begin{array}{rcl} \mu \restriction^{\circ} \mathsf{V} & \stackrel{def}{=} & \left\{ \begin{array}{ll} \bot^{\circ} & if\ \mu = \bot^{\circ} \\ \mu \restriction \mathsf{V} & otherwise \end{array} \right. \\ solve(E, \mu) & \stackrel{def}{=} & up(E, down(E, \mu)) \\ down(E, \nu) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \bot^{\circ} & if\ \nu = \bot^{\circ} \\ \nu \sqcap^{\circ} \sqcap^{\circ}_{(X=t) \in E} lvt(\nu(X), t) & if\ \nu \neq \bot^{\circ} \end{array} \right. \\ up(E, \nu) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \bot^{\circ} & if\ \nu = \bot^{\circ} \\ \nu \sqcap^{\circ} \{X \mapsto lt(t, \nu) \mid (X = t) \in E\} & if\ \nu \neq \bot^{\circ} \end{array} \right. \end{array}$$

$solve(E, \mu)$ is computed by first propagating (type information in) μ downwards equations in E by an operator $down$ and then propagating the result upwards equations in E by an operator $up : \wp(\mathsf{Eqn}) \times \mathsf{VMT} \mapsto \mathsf{VMT}$. When μ is propagated downwards an equation $X = t$ by $down : \wp(\mathsf{Eqn}) \times \mathsf{VMT} \mapsto \mathsf{VMT}$, a variable mono-typing $lvt(\mu(X), t)$ is computed to describe the set of substitutions θ such that $\theta(t) \in \gamma(\mu(X))$. The type associated with variables Y in t after the downward propagation of μ through $X = t$ is the greatest lower bound of the type assigned to Y by μ and the type assigned to Y by $lvt(\mu(X), t)$. When ν is propagated upwards an equation $X = t$ by up, a mono-type $lt(t, \nu)$ is first computed to describe the set of terms $\sigma(t)$ for all substitutions $\sigma \in \gamma^{\circ}(\nu)$. The type associated with X after the upward propagation of ν through $X = t$ is the greatest lower bound of $\nu(X)$ and $lt(t, \nu)$.

For a mono-type $M \in \mathsf{Mono}$, and a term $t \in \mathsf{Term}$, $lvt(M, t)$ is the *l*east *v*ariable mono-*t*yping that describes all substitutions θ such that $\theta(t) \in \gamma(M)$. Let $X \in \mathsf{Vars}$, $f \in \mathsf{Func}$,

$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$, $M, M_1, \cdots, M_l \in \mathsf{Mono}$ and $d \in \mathsf{Cons}$. $lvt : \mathsf{Mono} \times \mathsf{Term} \mapsto \mathsf{VMT}$ is defined as follows.

$$
\begin{array}{rcl}
lvt(M, X) & \stackrel{def}{=} & \{X \mapsto M\} \\
lvt(\mathbf{1}, t) & \stackrel{def}{=} & \top \\
lvt(d(M_1, \cdots, M_l), f(t_1, \cdots, t_n)) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \bot & if\ d/l \neq c/m \\ \sqcap_{1 \leq i \leq n} lvt(\Bbbk(T_i), t_i) & if\ d/l = c/m \\ where\ \Bbbk = \{\beta_j \mapsto M_j \mid 1 \leq j \leq l\} & \end{array} \right.
\end{array}
$$

$lvt(M, t)$ is computed as follows. If t is a variable then $lvt(M, t) = \{t \mapsto M\}$ by the first equation. If M is $\mathbf{1}$ then $lvt(M, t) = \top$ by the second equation. For $M = d(M_1, \cdots, M_l)$ and $t = f(t_1, \cdots, t_n)$, lvt checks if M conforms to the type definition for f. If so, it computes a mono-type for t_i from M and the type definition for f, then computes the least variable mono-typing for variables in t_i and finally returns the greatest lower bound of these variable mono-typings.

Example 5.3 *This example illustrates how lvt and $\sqcap$ work. Let trees be defined by the following two type clauses. D5 says that void is a term of type $\mathsf{tree}(\beta)$ for any β while D6 says that $tr(X, L, R)$ is a term of type $\mathsf{tree}(\beta)$ if X is a term of type β, and L and R are terms of type $\mathsf{tree}(\beta)$.*

$$
\begin{array}{rcll}
void : \mathsf{tree}(\beta) & \leftarrow & & \%\ D5 \\
tr(X, L, R) : \mathsf{tree}(\beta) & \leftarrow & X : \beta, L : \mathsf{tree}(\beta), R : \mathsf{tree}(\beta) & \%\ D6
\end{array}
$$

We have

$$
\begin{array}{rcl}
lvt(\mathsf{tree}(\mathsf{nat}), tr(U_0, V_0, W_0)) & = & lvt(\Bbbk(\beta), U_0) \sqcap lvt(\Bbbk(\mathsf{tree}(\beta)), V_0) \\
& & \qquad \sqcap\ lvt(\Bbbk(\mathsf{tree}(\beta)), W_0) \\
& & with\ \Bbbk = \{\beta \mapsto \mathsf{nat}\} \\
& = & lvt(\mathsf{nat}, U_0) \sqcap lvt(\mathsf{tree}(\mathsf{nat}), V_0) \\
& & \qquad \sqcap\ lvt(\mathsf{tree}(\mathsf{nat}), W_0) \\
& = & \{U_0 \mapsto \mathsf{nat}\} \sqcap \{V_0 \mapsto \mathsf{tree}(\mathsf{nat})\} \sqcap \{W_0 \mapsto \mathsf{tree}(\mathsf{nat})\} \\
& = & \{U_0 \mapsto \mathsf{nat}, V_0 \mapsto \mathsf{tree}(\mathsf{nat}), W_0 \mapsto \mathsf{tree}(\mathsf{nat})\} \\
lvt(\mathbf{1}, tr(U_0, V_0, X_0)) & = & \top \\
lvt(\mathsf{nat}, T) & = & \{T \mapsto \mathsf{nat}\}
\end{array}
$$

For $t \in \mathsf{Term}$ and $\nu \in \mathsf{VMT}$, $lt(t, \nu)$ is the least mono-type that describes terms $\theta(t)$ for all $\theta \in \gamma(\nu)$. Let $X \in \mathsf{Vars}$, $f \in \mathsf{Func}$, $f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$, $M, M_1, \cdots, M_m \in \mathsf{Mono}$, $c \in \mathsf{Cons}$ and $G \in \mathsf{Gen}$. $lt : \mathsf{Term} \times \mathsf{VMT} \mapsto \mathsf{Mono}$ is defined as follows.

$$
lt(X, \nu) \stackrel{def}{=} \nu(X)
$$

$$lt(f(t_1,\cdots,t_n),\nu) \stackrel{def}{=} \Bbbk(c(\beta_1,\cdots,\beta_m)) \quad where\ \Bbbk = \bigsqcup\!\!\!\ddagger_{1\le i\le n} lts(lt(t_i,\nu),T_i)$$

where lts : $\mathsf{Mono} \times \mathsf{Gen} \mapsto \mathsf{MTS}$ is defined as follows.

$$lts(M,G) \stackrel{def}{=} \begin{cases} \pm & if\ G = \mathbf{1} \vee M = \mathbf{0} \\ \{G \mapsto M\} & if\ G \in \mathsf{Para} \\ \{\beta_j \mapsto M_j \mid 1 \le j \le m\} & if\ M = c(M_1,\cdots,M_m) \\ & \wedge\ G = c(\beta_1,\cdots,\beta_m) \\ \mp & otherwise \end{cases}$$

$lt(t,\nu)$ is computed as follows. If t is a variable then $lt(t,\nu) = \nu(t)$. For $t = f(t_1,\cdots,t_n)$ and $f(X_1,\cdots,X_n) : c(\beta_1,\cdots,\beta_m) \leftarrow X_1 : T_1,\cdots,X_n : T_n$, lt first computes a mono-type $M_i = lt(t_i,\nu)$ for each t_i, then checks if M_i conforms to T_i and, if so, computes the *l*east mono-*t*ype *s*ubstitution $\Bbbk_i = lts(M_i,T_i)$ such that $M_i \precsim \Bbbk_i(T_i)$, $lt(t,\nu)$ is $\Bbbk(c(\beta_1,\cdots,\beta_m))$ where $\Bbbk$ is the least upper bound of the mono-type substitutions $\Bbbk_i$. If M_i does not conform to T_i for some i then $\Bbbk_i = \mp$, which makes $\Bbbk = \mp$ and hence $lt(t,\nu) = \mathbf{1}$.

Example 5.4 *This example illustrates how lt, lts and $\sqcup\!\!\ddagger$ work. Suppose that type clauses be the same as in example 5.3. Let*

$$\pi_1 = \left\{ \begin{array}{c} Y \mapsto nat, U \mapsto list(nat), V \mapsto tree(nat), \\ Y_0 \mapsto nat, U_0 \mapsto nat, V_0 \mapsto tree(nat), W_0 \mapsto tree(\mathbf{0}), X_0 \mapsto tree(nat) \end{array} \right\}$$

We have

$$\begin{array}{lllll} lt(U_0,\pi_1) & = & \pi_1(U_0) & = & nat \\ lt(V_0,\pi_1) & = & \pi_1(V_0) & = & tree(nat) \\ lt(W_0,\pi_1) & = & \pi_1(W_0) & = & tree(\mathbf{0}) \\ lt(X_0,\pi_1) & = & \pi_1(X_0) & = & tree(nat) \end{array}$$

So,

$$\begin{array}{rcl} lt(tr(U_0,V_0,W_0),\pi_1) & = & \Bbbk'(tree(\beta)) \\ & & \begin{array}{lcl} with\ \ \Bbbk' & = & lts(nat,\beta) \sqcup\!\!\ddagger\ lts(tree(nat),tree(\beta)) \\ & & \qquad \sqcup\!\!\ddagger\ lts(tree(\mathbf{0}),tree(\beta)) \\ & = & \{\beta \mapsto nat\} \sqcup\!\!\ddagger \{\beta \mapsto nat\} \sqcup\!\!\ddagger \{\beta \mapsto \mathbf{0}\} \\ & = & \{\beta \mapsto nat\} \end{array} \\ & = & tree(nat) \\ lt(tr(U_0,V_0,X_0),\pi_1) & = & \Bbbk''(tree(\beta)) \\ & & \begin{array}{lcl} with\ \ \Bbbk'' & = & lts(nat,\beta) \sqcup\!\!\ddagger\ lts(tree(nat),tree(\beta)) \\ & & \qquad \sqcup\!\!\ddagger\ lts(tree(nat),tree(\beta)) \\ & = & \{\beta \mapsto nat\} \sqcup\!\!\ddagger \{\beta \mapsto nat\} \sqcup\!\!\ddagger \{\beta \mapsto nat\} \\ & = & \{\beta \mapsto nat\} \end{array} \\ & = & tree(nat) \\ lt(Y,\pi_1) & = & \pi_1(Y) \ = \ nat \end{array}$$

Theorem 5.5
$I^{\circ} = \langle(\mathsf{VMT}, \sqsubseteq^{\circ}), (\sqcup^{\circ}, MUNIFY)\rangle$ is a γ°-abstraction of $I = \langle(\wp(Sub), \subseteq), (\cup, UNIFY)\rangle$.
  PROOF: See [30]. ∎

Example 5.6 *This example illustrates how* $MUNIFY$*, solve, down, up and* $\upharpoonright^{\circ}$ *work. Let*

$$\begin{array}{rcl} A & = & insert(Y, tr(U,V,W), tr(U,V,X)) \\ B & = & insert(Y,V,X) \\ \mu & = & \{Y \mapsto \mathrm{nat}, W \mapsto tree(\mathbf{0}), X \mapsto tree(nat)\} \\ \nu & = & \{Y \mapsto \mathrm{nat}, U \mapsto list(nat), V \mapsto tree(nat)\} \end{array}$$

Suppose that $\Psi(Z) = Z_0$ *for any* $Z \in \mathsf{Vars}$*. We have*

$$\begin{array}{rcl} E & = & eq \circ mgu(\Psi(A), B) \\ & = & \{V \mapsto tr(U_0, V_0, W_0), X \mapsto tr(U_0, V_0, X_0), Y_0 \mapsto Y\} \\ \pi & = & \Psi(\mu) \cup \nu = \left\{ \begin{array}{c} Y \mapsto \mathrm{nat}, U \mapsto list(nat), V \mapsto tree(nat), \\ Y_0 \mapsto \mathrm{nat}, W_0 \mapsto tree(\mathbf{0}), X_0 \mapsto tree(nat) \end{array} \right\} \end{array}$$

So,

$$\begin{array}{rcl} down(E, \pi) & = & \pi \sqcap^{\circ} lvt(\pi(V), tr(U_0, V_0, W_0)) \sqcap^{\circ} lvt(\pi(X), tr(U_0, V_0, X_0)) \\ & & \sqcap^{\circ} lvt(\pi(Y_0), Y) \\ & = & \pi \sqcap^{\circ} lvt(tree(nat), tr(U_0, V_0, W_0)) \sqcap^{\circ} lvt(\mathbf{1}, tr(U_0, V_0, X_0)) \\ & & \sqcap^{\circ} lvt(\mathrm{nat}, Y) \\ & & \textit{by example 5.3} \\ & = & \pi \sqcap^{\circ} \{U_0 \mapsto \mathrm{nat}, V_0 \mapsto tree(nat), W_0 \mapsto tree(nat)\} \sqcap^{\circ} \top^{\circ} \sqcap^{\circ} \{Y \mapsto \mathrm{nat}\} \\ & = & \left\{ \begin{array}{c} Y \mapsto \mathrm{nat}, U \mapsto list(nat), V \mapsto tree(nat), \\ Y_0 \mapsto \mathrm{nat}, U_0 \mapsto \mathrm{nat}, V_0 \mapsto tree(nat), W_0 \mapsto tree(\mathbf{0}), X_0 \mapsto tree(nat) \end{array} \right\} \\ & = & \pi_1 \textit{ (see example 5.4)} \end{array}$$

and

$$\begin{array}{rcl} up(E, \pi_1) & = & \pi_1 \sqcap^{\circ} \left\{ \begin{array}{lcl} V & \mapsto & lt(tr(U_0, V_0, W_0), \pi_1), \\ X & \mapsto & lt(tr(U_0, V_0, X_0), \pi_1), \\ Y_0 & \mapsto & lt(Y, \pi_1) \end{array} \right\} \\ & & \textit{by example 5.4} \\ & = & \pi_1 \sqcap^{\circ} \{V \mapsto tree(nat), X \mapsto tree(nat), Y_0 \mapsto \mathrm{nat}\} \\ & = & \left\{ \begin{array}{c} Y \mapsto \mathrm{nat}, U \mapsto list(nat), V \mapsto tree(nat), X \mapsto tree(nat), \\ Y_0 \mapsto \mathrm{nat}, U_0 \mapsto \mathrm{nat}, V_0 \mapsto tree(nat), W_0 \mapsto tree(\mathbf{0}), X_0 \mapsto tree(nat) \end{array} \right\} \end{array}$$

Therefore,

$$solve(E, \pi) = \left\{ \begin{array}{c} Y \mapsto \mathrm{nat}, U \mapsto list(nat), V \mapsto tree(nat), X \mapsto tree(nat), \\ Y_0 \mapsto \mathrm{nat}, U_0 \mapsto \mathrm{nat}, V_0 \mapsto tree(nat), W_0 \mapsto tree(\mathbf{0}), X_0 \mapsto tree(nat) \end{array} \right\}$$

Finally,

$$\begin{aligned}
MUNIFY(A,\mu,B,\nu) &= solve(E,\pi)\upharpoonright^{\circ}(vars(B)\cup dom(\nu)) \\
&= solve(E,\pi)\upharpoonright^{\circ}\{Y,U,V,W,X\} \\
&= \{Y\mapsto nat, U\mapsto list(nat), V\mapsto tree(nat), X\mapsto tree(nat)\}
\end{aligned}$$

In summary, we have designed the monomorphic type analysis by means of a monomorphic abstract interpretation $I^{\circ} = \langle(\mathsf{VMT},\sqsubseteq^{\circ}),(\sqcup^{\circ}, MUNIFY)\rangle$. I° uses two abstract operators $\sqcup^{\circ} : \mathsf{VMT}\times\mathsf{VMT}\mapsto\mathsf{VMT}$ and $MUNIFY : \mathsf{Atom}\times\mathsf{VMT}\times\mathsf{Atom}\times\mathsf{VMT}\mapsto\mathsf{VMT}$ which is defined in terms of the following abstract operators: $solve, down, up : \wp(\mathsf{Eqn})\times\mathsf{VMT}\mapsto\mathsf{VMT}$, $\sqcap^{\circ} : \mathsf{VMT}\times\mathsf{VMT}\mapsto\mathsf{VMT}$, $\upharpoonright^{\circ} : \mathsf{VMT}\times\wp(\mathsf{Vars})\mapsto\mathsf{VMT}$, $lvt : \mathsf{Mono}\times\mathsf{Term}\mapsto\mathsf{VMT}$, $lt : \mathsf{Term}\times\mathsf{VMT}\mapsto\mathsf{Mono}$, $lts : \mathsf{Mono}\times\mathsf{Gen}\mapsto\mathsf{MTS}$, $\sqcup^{\ddagger} : \mathsf{MTS}\times\mathsf{MTS}\mapsto\mathsf{MTS}$, and $\veebar,\barwedge : \mathsf{Mono}\times\mathsf{Mono}\mapsto\mathsf{Mono}$.

6 POLYMORPHIC TYPE ANALYSIS

This section designs a polymorphic abstract interpretation $I^{\flat}$ for polymorphic type analysis by lifting the monomorphic abstract interpretation I° presented in section 5.

In a polymorphic type analysis, type information available before analysis is given in terms of polymorphic types. More type information is to be inferred from the given type information and the program. The mono-type that a type parameter may take as its value is unknown and inferred type information is expected to be correct with respect to all assignments of mono-types to type parameters. This gives rise to difficulty in lifting the monomorphic abstract interpretation. For instance, the least upper bound of two poly-types, say β and nat can no longer be expressed precisely by a single type since a type parameter may assume any mono-type as its value. We deal with this problem by a case analysis of the structures of the types involved. Take $\beta\veebar nat$ for an example, we consider the following three mutually exclusive cases: (1) $\beta\equiv\mathbf{0}$, (2) $\beta\equiv nat$ and (3) $\beta\not\equiv\mathbf{0}\wedge\beta\not\equiv nat$. $\beta\veebar nat$ is nat in the cases (1) and (2) and it is $\mathbf{1}$ in the case (3). The $\beta\veebar nat$ is approximated by $\{(nat,\beta\equiv\mathbf{0}),(nat,\beta\equiv nat),(\mathbf{1},\beta\not\equiv\mathbf{0}\wedge\beta\not\equiv nat)\}$ which is a set of pairs consisting of a type and a condition. The case conditions are so designed that at least one of them is true for any assignment of mono-types to type parameters. Similar problems arise when we compute a least type substitution $\Bbbk$ such that $\beta\,\bar{\preceq}\,\Bbbk(G)$ where $\beta\in\mathsf{Para}$ and $G\in\mathsf{Gen}$ and when we compute types for subterms t_i of a compound term $f(t_1,\cdots,t_n)$ given a type parameter β as the type for $f(t_1,\cdots,t_n)$. These problems will be also dealt with by case analysis of the structures of the types involved.

Making case analysis of type structures, we need to access and express sub-structures of types. If a mono-type M is represented as an ordered tree in the usual way then a sub-structure of M is identified by the *path* from the root of M to the root of the sub-structure. A path is a possibly empty sequence of positive integers separated from each other by a dot $\cdot$. The empty path is written as ϵ. The set of all possible paths is denoted as Path. For a mono-type $M\in\mathsf{Mono}$ and a path $s\in\mathsf{Path}$, s may or may not be a valid path for M. The sub-structure of $M\in\mathsf{Mono}$ identified by s, denoted by (M,s), is defined as follows. Let i be a positive integer, $c/n \in \mathsf{Cons}$ and $M_1, \cdots, M_n \in \mathsf{Mono}$.

$$
\begin{array}{rcl}
(M, \epsilon) & \stackrel{def}{=} & M \\
(c(M_1, \cdots, M_n), i \cdot s) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \mathit{undefined} & \mathit{if}\ n < i \\ (M_i, s) & \mathit{if}\ n \geq i \end{array} \right. \\
(\mathit{undefined}, s) & \stackrel{def}{=} & \mathit{undefined}
\end{array}
$$

The expression (β, s) is a type parameter whose value depends on the value of β. Let $\mathsf{Origin} \subseteq \mathsf{Para}$ be the set of type parameters used in expressions of type information available before analysis. The expression (β, s) will be used only with type parameters β in Origin. Type parameters in Origin are called *original type parameters* and members of $\mathsf{Derived} \stackrel{def}{=} \mathsf{Origin} \times \mathsf{Path}$, denoted by tilded lower case Greek letters, say, $\tilde{\beta}$, are called *derived type parameters*. The set of assignments for the polymorphic type analysis is then $\mathcal{A}_t = \mathsf{Origin} \mapsto \mathsf{Mono}$.

6.1 Abstract domains

Let Poly be the set of terms that can be constructed from Cons and $\mathsf{Derived}$. It is expected that the result of a polymorphic type analysis is expressed in terms of types in Poly. In this section, a poly-type means a type in Poly. The notion of *poly-type substitutions* is derived from that of mono-type substitutions by replacing mono-types in Mono with poly-types in Poly. The set of poly-type substitutions is denoted by PTS. The notion of *variable poly-typings* is derived from that of variable mono-typings in the same way. The set of variable poly-typings is denoted by VPT. Poly-types, poly-type substitutions and variable poly-typings are elementary polymorphic data descriptions corresponding to mono-types, mono-type substitutions and variable mono-typings respectively.

The case conditions generated by case analysis are conjunctions of primitive case conditions. A primitive case condition is a constraint on the principal constructor of a sub-structure of a mono-type that may be assigned to an original type parameter. Syntactically, a primitive case condition is either *true*, or *false*, or of the form $|\tilde{\beta}, c/m|$ or of the form $\backslash\!\!\backslash \tilde{\beta}, c/m \backslash\!\!\backslash$ where $\tilde{\beta} \in \mathsf{Derived}$ and $c/m \in \mathsf{Cons}$. $|\tilde{\beta}, c/m|$ is read as that $\tilde{\beta}$ is well-defined and its principal constructor is c/m and $\backslash\!\!\backslash \tilde{\beta}, c/m \backslash\!\!\backslash$ is read as that either $\tilde{\beta}$ is not well-defined or its principal constructor is not c/m. The set of all case conditions is denoted by Case.

For the polymorphic type analysis, a polymorphic data description is a set of pairs consisting of an elementary polymorphic data description and a case condition. In correspondence to mono-types, mono-type substitutions, and variable mono-typings, we define

$$
\begin{array}{rcl}
\mathsf{Poly} \diamondsuit \mathsf{Case} & \stackrel{def}{=} & \{(T, \mathsf{C}) \mid T \in \mathsf{Poly} \wedge \mathsf{C} \in \mathsf{Case} \wedge \forall \kappa \in \mathcal{A}_t.(\kappa(\mathsf{C}) \rightarrow (\kappa(T) \in \mathsf{Mono}))\} \\
\mathsf{PTS} \diamondsuit \mathsf{Case} & \stackrel{def}{=} & \{(\mathbb{k}, \mathsf{C}) \mid \mathbb{k} \in \mathsf{PTS} \wedge \mathsf{C} \in \mathsf{Case} \wedge \forall \kappa \in \mathcal{A}_t.(\kappa(\mathsf{C}) \rightarrow (\kappa(\mathbb{k}) \in \mathsf{MTS}))\} \\
\mathsf{VPT} \diamondsuit \mathsf{Case} & \stackrel{def}{=} & \{(\mu, \mathsf{C}) \mid \mu \in \mathsf{VPT} \wedge \mathsf{C} \in \mathsf{Case} \wedge \forall \kappa \in \mathcal{A}_t.(\kappa(\mathsf{C}) \rightarrow (\kappa(\mu) \in \mathsf{VMT}))\}
\end{array}
$$

A pair in $\mathsf{Poly} \diamondsuit \mathsf{Case}$ is called a constrained poly-type. It consists of a poly-type and a case condition such that the instance of the poly-type under any assignment satisfying the case condition is a mono-type. A pair in $\mathsf{PTS}\diamondsuit\mathsf{Case}$ is called a constrained poly-type substitution. It consists of a poly-type substitution and a case condition such that the instance of the poly-type substitution under any assignment satisfying the case condition is a mono-type substitution. A pair in $\mathsf{VPT}\diamondsuit\mathsf{Case}$ is called a constrained variable poly-typing. It consists of a variable poly-typing and a case condition such that the instance of the variable poly-typing under any assignment satisfying the case condition is a variable mono-typing.

Definition 6.1 *Let* $\langle(\mathcal{D}^\dagger, \sqsubseteq^\dagger), \mathcal{E}, \diamondsuit\rangle$ *be* $\langle(\mathsf{Mono}, \preceq), \mathsf{Poly}, \diamondsuit\rangle$ *or* $\langle(\mathsf{MTS}, \sqsubseteq), \mathsf{PTS}, \diamondsuit\rangle$ *or* $\langle(\mathsf{VMT}, \sqsubseteq^{\circ}), \mathsf{VPT}, \diamondsuit\rangle$. *Let* $\kappa \in \mathcal{A}_t$, $d^\dagger \in \mathcal{D}^\dagger$, $(\mathbf{e}, \mathsf{C}) \in \mathcal{E}\diamondsuit\mathsf{Case}$ *and* $d^\bullet \in \wp(\mathcal{E}\diamondsuit\mathsf{Case})$. $d^\dagger$ *is said to be covered by* $(\mathbf{e}, \mathsf{C})$ *under* κ *iff* $\kappa(\mathsf{C})$ *and* $d^\dagger \sqsubseteq^\dagger \kappa(\mathbf{e})$. $d^\dagger$ *is said to be covered by* $d^\bullet$ *under* κ *iff* $d^\dagger$ *is covered by* $(\mathbf{e}', \mathsf{C}')$ *under* κ *for some* $(\mathbf{e}', \mathsf{C}') \in d^\bullet$.

Let $d^\bullet$ be a polymorphic data description. Under an assignment κ, a pair $(\mathbf{e}, \mathsf{C}) \in d^\bullet$ either covers a set of monomorphic data descriptions if $\kappa(\mathsf{C}) \leftrightarrow \mathit{true}$ or covers the empty set of monomorphic data descriptions if $\kappa(\mathsf{C}) \leftrightarrow \mathit{false}$. The set of monomorphic data descriptions covered by a polymorphic data description $d^\bullet$ under κ is the union of the sets of monomorphic data descriptions covered by its memebers under κ. For instance, the set $\{\langle \mathit{list}(\mathit{nat}), |(\alpha, \epsilon), \mathbf{1}/0|\rangle, \langle(\alpha, \epsilon), |(\alpha, \epsilon), \mathit{nat}/0|\rangle, \langle \mathit{list}(\mathbf{0}), \mathit{true}\rangle\}$ of constrained poly-types covers mono-types $\mathit{nat}, \mathit{list}(\mathbf{0})$ and $\mathbf{0}$ under the assignment $\{\alpha \mapsto \mathit{nat}\}$; and it covers mono-types $\mathit{list}(\mathit{nat}), \mathit{list}(\mathbf{0})$ and $\mathbf{0}$ under the assignment $\{\alpha \mapsto \mathbf{1}\}$. Note that the set of monomorphic data descriptions covered by a polymorphic data description under an assignment is downwards closed with respect to the partial order on the complete lattice of monomorphic data descriptions.

Let $(\mu, \mathsf{C}) \in \mathsf{VPT}\diamondsuit\mathsf{Case}$ and $\kappa \in \mathcal{A}_t$. $(\kappa \circ \mu, \kappa(\mathsf{C}))$ describes the same set of substitutions as $\kappa \circ \mu$ if $\kappa(\mathsf{C}) \leftrightarrow \mathit{true}$ and it describes the empty set of substitutions if $\kappa(\mathsf{C}) \leftrightarrow \mathit{false}$. A constrained variable poly-typing (μ_1, C_1) is said to be subsumed by another constrained variable poly-typing (μ_2, C_2), denoted as, $(\mu_1, \mathsf{C}_1) \trianglelefteq^\flat (\mu_2, \mathsf{C}_2)$, if, for any $\kappa \in \mathcal{A}_t$, $(\kappa \circ \mu_1, \kappa(\mathsf{C}_1))$ describes a smaller set of substitutions than $(\kappa \circ \mu_2, \kappa(\mathsf{C}_2))$. [2]

$$(\mu_1, \mathsf{C}_1) \trianglelefteq^\flat (\mu_2, \mathsf{C}_2) \stackrel{def}{=} \begin{array}{c} \forall.(\mathsf{C}_1 \wedge \mathsf{C}_2 \rightarrow \mu_1 \sqsubseteq^{\circ} \mu_2) \\ \wedge \\ \forall.(\mathsf{C}_1 \wedge \neg\mathsf{C}_2 \rightarrow \mu_1 = \top) \end{array} \tag{1}$$

$\trianglelefteq^\flat$ is a preorder on $\mathsf{VPT}\diamondsuit\mathsf{Case}$, which is justified as follows. $\trianglelefteq^\flat$ is obviously reflexive. Let $(\mu_1, \mathsf{C}_1), (\mu_2, \mathsf{C}_2), (\mu_3, \mathsf{C}_3) \in \mathsf{VPT}\diamondsuit\mathsf{Case}$ such that $(\mu_1, \mathsf{C}_1) \trianglelefteq^\flat (\mu_2, \mathsf{C}_2)$ and $(\mu_2, \mathsf{C}_2) \trianglelefteq^\flat (\mu_3, \mathsf{C}_3)$. For any $\kappa \in \mathcal{A}_t$, we have $(\kappa(\mathsf{C}_1) \wedge \kappa(\mathsf{C}_2) \rightarrow \kappa \circ \mu_1 \sqsubseteq^{\circ} \kappa \circ \mu_2)$, $\kappa(\mathsf{C}_1) \wedge \neg\kappa(\mathsf{C}_2) \rightarrow \kappa \circ \mu_1 = \top$, $(\kappa(\mathsf{C}_2) \wedge \kappa(\mathsf{C}_3) \rightarrow \kappa \circ \mu_2 \sqsubseteq^{\circ} \kappa \circ \mu_3)$ and $\kappa(\mathsf{C}_2) \wedge \neg\kappa(\mathsf{C}_3) \rightarrow \kappa \circ \mu_2 = \top$. Assume $\kappa(\mathsf{C}_1) \wedge \kappa(\mathsf{C}_3) \leftrightarrow \mathit{true}$. If $\kappa(\mathsf{C}_2) \leftrightarrow \mathit{true}$ then $\kappa \circ \mu_1 \sqsubseteq^{\circ} \kappa \circ \mu_3$ since $\kappa \circ \mu_1 \sqsubseteq^{\circ} \kappa \circ \mu_2$ and $\kappa \circ \mu_2 \sqsubseteq^{\circ} \kappa \circ \mu_3$. If $\kappa(\mathsf{C}_2) \leftrightarrow \mathit{false}$ then we also have $\kappa \circ \mu_1 \sqsubseteq^{\circ} \kappa \circ \mu_3$ since $\kappa \circ \mu_1 = \top$. Therefore, we have $\forall.(\mathsf{C}_1 \wedge \mathsf{C}_3 \rightarrow \mu_1 \sqsubseteq^{\circ} \mu_3)$. Now assume $\kappa(\mathsf{C}_1) \wedge \neg\kappa(\mathsf{C}_3) \leftrightarrow \mathit{true}$. If $\kappa(\mathsf{C}_2) \leftrightarrow \mathit{true}$ then we have $\kappa \circ \mu_1 = \top$ for $\kappa \circ \mu_2 = \top$ If $\kappa(\mathsf{C}_2) \leftrightarrow \mathit{false}$ then we also have $\kappa \circ \mu_1 = \top$. So, $\forall.(\mathsf{C}_1 \wedge \neg\mathsf{C}_3 \rightarrow \mu_1 = \top)$. Therefore, $(\mu_1, \mathsf{C}_1) \trianglelefteq^\flat (\mu_3, \mathsf{C}_3)$ proving the transitivity of $\trianglelefteq^\flat$. So, $\trianglelefteq^\flat$ is a preorder on $\mathsf{VPT}\diamondsuit\mathsf{Case}$.

[2] Implicit universal quantifiers bound all original type parameters in its scope to mono-types.

Let $\bar{\Phi}_1, \bar{\Phi}_2 \in \wp(\mathsf{VPT} \diamondsuit \mathsf{Case})$ and define

$$\bar{\Phi}_1 \preceq^\flat \bar{\Phi}_2 \stackrel{def}{=} \forall(\mu_1, \mathsf{C}_1) \in \bar{\Phi}_1. \left(\begin{array}{c} \forall.(\mathsf{C}_1 \rightarrow \mu_1 = \underline{\varphi}) \\ \vee \\ \exists(\mu_2, \mathsf{C}_2) \in \bar{\Phi}_2.((\mu_1, \mathsf{C}_1) \trianglelefteq^\flat (\mu_2, \mathsf{C}_2)) \end{array} \right)$$

$\preceq^\flat$ is a preorder on $\wp(\mathsf{VPT} \diamondsuit \mathsf{Case})$ with the transitivity of $\preceq^\flat$ following from that of $\trianglelefteq^\flat$. Define $\bar{\Phi}_1 \approx^\flat \bar{\Phi}_2 \stackrel{def}{=} (\bar{\Phi}_1 \preceq^\flat \bar{\Phi}_2) \wedge (\bar{\Phi}_2 \preceq^\flat \bar{\Phi}_1)$. $\approx^\flat$ is an equivalence relation on $\wp(\mathsf{VPT} \diamondsuit \mathsf{Case})$. The abstract domain for polymorphic analysis is the set of equivalence classes of $\approx^\flat$ ordered by ${\preceq^\flat}_{/\approx^\flat}$.

$$\begin{array}{rcl}
\mathcal{D}^\flat & \stackrel{def}{=} & (\wp(\mathsf{VPT} \diamondsuit \mathsf{Case}))_{/\approx^\flat} \\
\sqsubseteq^\flat & \stackrel{def}{=} & {\preceq^\flat}_{/\approx^\flat} \\
[\bar{\Phi}_1]_{\approx^\flat} \sqcup^\flat [\bar{\Phi}_2]_{\approx^\flat} & \stackrel{def}{=} & [(\bar{\Phi}_1 \cup \bar{\Phi}_2)]_{\approx^\flat} \\
[\bar{\Phi}_1]_{\approx^\flat} \sqcap^\flat [\bar{\Phi}_2]_{\approx^\flat} & \stackrel{def}{=} & \left[\begin{array}{c} \{\bar{\mu} \mid \exists \bar{\mu}_1 \in \bar{\Phi}_1.(\bar{\mu} \trianglelefteq^\flat \bar{\mu}_1)\} \\ \cap \\ \{\bar{\mu} \mid \exists \bar{\mu}_2 \in \bar{\Phi}_2.(\bar{\mu} \trianglelefteq^\flat \bar{\mu}_2)\} \end{array} \right]_{\approx^\flat} \\
\top^\flat & \stackrel{def}{=} & [\{(\emptyset, \mathit{true})\}]_{\approx^\flat} \\
\perp^\flat & \stackrel{def}{=} & [\emptyset]_{\approx^\flat}
\end{array}$$

$[\{\bar{\mu} \mid \exists \bar{\mu}_1 \in \bar{\Phi}_1.(\bar{\mu} \trianglelefteq^\flat \bar{\mu}_1)\}]_{\approx^\flat}$ is equivalent to $[\bar{\Phi}_1]_{\approx^\flat}$ with respect to $\approx^\flat$ and is down-closed with respect to $\preceq^\flat$.

Lemma 6.2 *$\langle \mathcal{D}^\flat, \sqsubseteq^\flat, \perp^\flat, \top^\flat, \sqcap^\flat, \sqcup^\flat \rangle$ is a complete lattice.*

Proof: By the definition of $\preceq^\flat$, we have $\emptyset \preceq^\flat \bar{\Phi}$ and $\bar{\Phi} \preceq^\flat \{(\emptyset, \mathit{true})\}$ for any $[\bar{\Phi}]_{\approx^\flat} \in \mathcal{D}^\flat$. So, $\perp^\flat = [\emptyset]_{\approx^\flat}$ and $\top^\flat = [\{(\emptyset, \mathit{true})\}]_{\approx^\flat}$ are respectively the infimum and the supremum of $\mathcal{D}^\flat$. We now prove that $[\bar{\Phi}_1]_{\approx^\flat} \sqcap^\flat [\bar{\Phi}_2]_{\approx^\flat}$ is the greatest lower bound of $[\bar{\Phi}_1]_{\approx^\flat}$ and $[\bar{\Phi}_2]_{\approx^\flat}$ for any $[\bar{\Phi}_1]_{\approx^\flat}, [\bar{\Phi}_2]_{\approx^\flat} \in \mathcal{D}^\flat$. Let $\bar{\Phi} = \{\bar{\mu} \mid \exists \bar{\mu}_1 \in \bar{\Phi}_1.(\bar{\mu} \trianglelefteq^\flat \bar{\mu}_1)\} \cap \{\bar{\mu} \mid \exists \bar{\mu}_2 \in \bar{\Phi}_2.(\bar{\mu} \trianglelefteq^\flat \bar{\mu}_2)\}$. It follows that $[\bar{\Phi}]_{\approx^\flat} \sqsubseteq^\flat [\bar{\Phi}_1]_{\approx^\flat}$ and $[\bar{\Phi}]_{\approx^\flat} \sqsubseteq^\flat [\bar{\Phi}_2]_{\approx^\flat}$ because for every $\bar{\mu} \in \bar{\Phi}$, there are $\bar{\mu}_1 \in \bar{\Phi}_1$ and $\bar{\mu}_2 \in \bar{\Phi}_2$ such that $\bar{\mu} \trianglelefteq^\flat \bar{\mu}_1$ and $\bar{\mu} \trianglelefteq^\flat \bar{\mu}_2$. Now suppose $[\bar{\Phi}']_{\approx^\flat}$ be another lower bound of $[\bar{\Phi}_1]_{\approx^\flat}$ and $[\bar{\Phi}_2]_{\approx^\flat}$. Let $\bar{\mu}' = (\mu', \mathsf{C}') \in \bar{\Phi}'$. Either $\forall.(\mathsf{C}' \rightarrow \mu' = \underline{\varphi})$ or $\neg\forall.(\mathsf{C}' \rightarrow \mu' = \underline{\varphi})$. If $\neg\forall.(\mathsf{C}' \rightarrow \mu' = \underline{\varphi})$ then $\exists \bar{\mu}_1 \in \bar{\Phi}_1.\bar{\mu}' \trianglelefteq^\flat \bar{\mu}_1$ and $\exists \bar{\mu}_2 \in \bar{\Phi}_2.\bar{\mu}' \trianglelefteq^\flat \bar{\mu}_2$ and hence $\bar{\mu}' \in \bar{\Phi}$. Therefore, $\bar{\Phi}' \preceq^\flat \bar{\Phi}$ and hence $[\bar{\Phi}']_{\approx^\flat} \sqsubseteq^\flat [\bar{\Phi}]_{\approx^\flat}$. So, $[\bar{\Phi}_1]_{\approx^\flat} \sqcap^\flat [\bar{\Phi}_2]_{\approx^\flat} = [\bar{\Phi}]_{\approx^\flat}$ is the greatest lower bound of $[\bar{\Phi}_1]_{\approx^\flat}$ and $[\bar{\Phi}_2]_{\approx^\flat}$. It can be similarly proved that $[\bar{\Phi}_1]_{\approx^\flat} \sqcup^\flat [\bar{\Phi}_2]_{\approx^\flat} = [\bar{\Phi}]_{\approx^\flat}$ is the least upper bound of $[\bar{\Phi}_1]_{\approx^\flat}$ and $[\bar{\Phi}_2]_{\approx^\flat}$ for any $[\bar{\Phi}_1]_{\approx^\flat}, [\bar{\Phi}_2]_{\approx^\flat} \in \mathcal{D}^\flat$. ∎

Let $[\bar{\Phi}]_{\approx^\flat} \in \mathcal{D}^\flat$ and $\kappa \in \mathcal{A}_t$.

$$\gamma^\flat([\bar{\Phi}]_{\approx^\flat}, \kappa) \stackrel{def}{=} \bigcup_{((\mu, \mathsf{C}) \in \bar{\Phi}) \wedge \kappa(\mathsf{C})} \gamma^\wp([\kappa \circ \mu]_{\approx^\wp})$$

$\gamma^\flat$ is well-defined because $\kappa \circ \mu$ is a variable mono-typing whenever $\kappa(\mathsf{C}) \leftrightarrow \mathit{true}$. $\gamma^\flat$ is monotonic in its first argument since $\gamma^\wp$ is monotonic on VMT and function composition $\circ$ is monotonic in both of its arguments. For simplicity, an equivalence class $[\bar{\Phi}]_{\approx^\flat}$ of $\approx^\flat$ will be represented by one of its members and will be written as $\bar{\Phi}$ which is a set of constrained variable poly-typings. For the efficiency of analysis, it is desirable to represent an equivalence class of $\approx^\flat$ by a non-redundant set of constrained variable poly-typings in the sense that no constrained variable poly-typing in the set is subsumed by another and that the set does not contain any constrained variable poly-typing with an unsatisfiable case condition. We will consider this issue later.

6.2 Lifting monomorphic operators

With a polymorphic data description covers a set of monomorphic data descriptions under an assignment, an abstract operator $\mathcal{O}^\bullet$ for the polymorphic type analysis is designed by lifting a corresponding abstract operator $\mathcal{O}^\dagger$ for the monomorphic type analysis. The design strategy is to make sure that, for any assignment κ, any polymorphic data description $d^\bullet$, and any monomorphic data description $d^\dagger$ covered by $d^\bullet$ under κ, the result of applying $\mathcal{O}^\dagger$ to $d^\dagger$ will be covered by the result of applying $\mathcal{O}^\bullet$ to $d^\bullet$ under κ.

Definition 6.3 *Let* $\langle(\mathcal{D}^\dagger_i, \sqsubseteq^\dagger_i), \mathcal{E}_i, \diamondsuit\rangle$ *be* $\langle(\mathsf{Mono}, \precsim), \mathsf{Poly}, \diamondsuit\rangle$ *or* $\langle(\mathsf{MTS}, \sqsubseteq^\ddagger), \mathsf{PTS}, \diamondsuit\rangle$ *or* $\langle(\mathsf{VMT}, \sqsubseteq^\ominus), \mathsf{VPT}, \diamondsuit\rangle$ *for* $0 \le i \le n$, *and* $\mathcal{O}^\dagger : \mathcal{D}^\dagger_1 \times \cdots \times \mathcal{D}^\dagger_n \mapsto \mathcal{D}^\dagger_0$ *and* $\mathcal{O}^\ell : (\mathcal{E}_1 \diamondsuit \mathsf{Case}) \times \cdots \times (\mathcal{E}_n \diamondsuit \mathsf{Case}) \mapsto \wp(\mathcal{E}_0 \diamondsuit \mathsf{Case})$ *be operators.* $\mathcal{O}^\ell$ *is a lifting of* $\mathcal{O}^\dagger$ *iff, for all* $(\mathbf{e}_i, \mathbf{b}_i) \in \mathcal{E}_i \diamondsuit \mathsf{Case}$ *with* $1 \le i \le n$ *and all* $\kappa \in \mathcal{A}_t$ *such that* $\bigwedge_{1 \le i \le n} \kappa(\mathbf{b}_i)$,

$$\exists(\mathbf{e}', \mathbf{b}') \in \mathcal{O}^\ell((\mathbf{e}_1, \mathbf{b}_1), \cdots, (\mathbf{e}_n, \mathbf{b}_n)).(\kappa(\mathbf{b}') \wedge \mathcal{O}^\dagger(\kappa(\mathbf{e}_1), \cdots, \kappa(\mathbf{e}_n)) \sqsubseteq^\dagger \kappa(\mathbf{e}')) \tag{2}$$

With a lifting $\mathcal{O}^\ell$ of an abstract operator $\mathcal{O}^\dagger$ for the monomorphic type analysis, the corresponding abstract operator $\mathcal{O}^\bullet$ for the polymorphic type analysis is constructed as follows. The application of $\mathcal{O}^\bullet$ to a polymorphic data description is obtained as the union of the results of applying $\mathcal{O}^\ell$ to all pairs in the polymorphic data description, with redundancy being removed. In the following sub-section, we will design the abstract unification operator for the polymorphic type analysis by lifting the abstract unifcation operator for the monomorphic type analysis.

6.3 Abstract unification operator

Since $\sqcup^\flat$ is a polymorphic $\gamma^\flat$-abstraction of $\cup$ with respect to $\mathcal{A}_t$, it remains to define a polymorphic $\gamma^\flat$-abstraction *PUNIFY* of *UNIFY* with respect to $\mathcal{A}_t$ in order to complete $I^\flat$. *PUNIFY* is defined by lifting *MUNIFY*. Moreover, *PUNIFY* is defined in terms of operators $\mathcal{O}^\ell$ that lift the operators $\mathcal{O}^\dagger$ in terms of which *MUNIFY* is defined. In this way, the proof that *PUNIFY* a polymorphic $\gamma^\flat$-abstraction of *UNIFY* with respect to $\mathcal{A}_t$ will be broken down into proving that $\mathcal{O}^\ell$ is a lifting of $\mathcal{O}^\dagger$. If $\langle(\mathcal{D}^\dagger_0, \sqsubseteq^\dagger_0), \mathcal{E}_0, \diamondsuit\rangle$ is $\langle(\mathsf{VMT}, \sqsubseteq^\ominus), \mathsf{VPT}, \diamondsuit\rangle$ then the condition for $\mathcal{O}^\ell$ to be a lifting of $\mathcal{O}^\dagger$ can be weakened into the following: for all

$(\mathbf{e}_i, \mathbf{b}_i) \in \mathcal{E}_i \Diamond \mathsf{Case}$ with $1 \leq i \leq n$ and all $\kappa \in \mathcal{A}_t$ such that $\bigwedge_{1 \leq i \leq n} \kappa(\mathbf{b}_i)$,

$$\begin{array}{l} \mathcal{O}^{\dagger}(\kappa(\mathbf{e}_1), \cdots, \kappa(\mathbf{e}_n)) \neq \bot \rightarrow \\ \qquad \exists(\mathbf{e}', \mathbf{b}') \in \mathcal{O}^{\ell}((\mathbf{e}_1, \mathbf{b}_1), \cdots, (\mathbf{e}_n, \mathbf{b}_n)).(\kappa(\mathbf{b}') \wedge \mathcal{O}^{\dagger}(\kappa(\mathbf{e}_1), \cdots, \kappa(\mathbf{e}_n)) \sqsubseteq^{\dagger} \kappa(\mathbf{e}')) \end{array}$$

This is because $\bot$ describes the empty set of substitutions and hence removing any pair (μ, C) satisfying $\forall \kappa \in \mathcal{A}_t.(\kappa(\mathsf{C}) \rightarrow \kappa(\mu) = \bot)$ from a set of constrained variable poly-typings will result in an equivalent set of constrained variable poly-typings with respect to $\approx^{\flat}$.

Lifting $\veebar$ and $\barwedge$

The difficulty in expressing the least upper bound of poly-types is overcome by a case analysis of the structures of poly-types. The result is that the least upper bound is approximated by a set of constrained poly-types. Let $\tilde{\beta} \in \mathsf{Derived}$ and $\tau \in \mathsf{Poly}$. If $\tau = \mathbf{1}$ then $\tilde{\beta} \veebar \tau$ is represented by $\{(\mathbf{1}, \mathit{true})\}$. If $\tau = \mathbf{0}$ then $\tilde{\beta} \veebar \tau$ is represented by $\{(\tilde{\beta}, \mathit{true})\}$. If $\tau = \tilde{\beta}$ then $\tilde{\beta} \veebar \tau$ is represented by $\{(\tilde{\beta}, \mathit{true})\}$. If $\tau = \tilde{\beta}'$ and $\tilde{\beta}' \not\equiv \tilde{\beta}$ then $\tilde{\beta} \veebar \tau$ is approximated by $\{(\tilde{\beta}, \tilde{\beta}' = \mathbf{0}), (\tilde{\beta}', \tilde{\beta} = \mathbf{0}), (\mathbf{1}, \tilde{\beta} \neq \mathbf{0} \wedge \tilde{\beta}' \neq \mathbf{0})\}$. If $\tau = c(\tau_1, \cdots, \tau_m)$ with $m \geq 0$ then the approximation of $\tilde{\beta} \veebar \tau$ is derived by considering three mutually exclusive cases: (1) $\tilde{\beta} = \mathbf{0}$; (2) $|\tilde{\beta}, c/m|$; and (3) $\tilde{\beta} \neq \mathbf{0} \wedge \backslash\!\!\backslash \tilde{\beta}, c/m \backslash\!\!\backslash$. The cases (1) and (3) correspond to $(\tau, \tilde{\beta} = \mathbf{0})$ and $(\mathbf{1}, \tilde{\beta} \neq \mathbf{0} \wedge \backslash\!\!\backslash \tilde{\beta}, c/m \backslash\!\!\backslash)$ respectively. The case (2) involves approximating each $(\tilde{\beta}, i) \veebar \tau_i$ for $1 \leq i \leq m$ by a collection of constrained poly-types and forming constrained poly-types from the constructor c/m, the constrained poly-types approximating $(\tilde{\beta}, i) \veebar \tau_i$ for each $1 \leq i \leq m$ and the case condition $|\tilde{\beta}, c/m|$.

Symmetric operators $\veebar^{\ell}, \barwedge^{\ell} : (\mathsf{Poly} \Diamond \mathsf{Case}) \times (\mathsf{Poly} \Diamond \mathsf{Case}) \mapsto \wp(\mathsf{Poly} \Diamond \mathsf{Case})$ lifting $\veebar$ and $\barwedge$ respectively are defined in figure 5 where the symmetry of both $\veebar^{\ell}$ and $\barwedge^{\ell}$ is removed for simplicity. Note that $\tilde{\beta} = \mathbf{1}$ is logically equivalent to $|\tilde{\beta}, \mathbf{1}/0|$, $\tilde{\beta} \neq \mathbf{1}$ to $\backslash\!\!\backslash \tilde{\beta}, \mathbf{1}/0 \backslash\!\!\backslash$, $\tilde{\beta} = \mathbf{0}$ to $|\tilde{\beta}, \mathbf{0}/0|$ and $\tilde{\beta} \neq \mathbf{0}$ to $\backslash\!\!\backslash \tilde{\beta}, \mathbf{0}/0 \backslash\!\!\backslash$. If we replace every occurrence of $\tilde{\beta} = \mathbf{1}$, $\tilde{\beta} \neq \mathbf{1}$, $\tilde{\beta} = \mathbf{0}$ and $\tilde{\beta} \neq \mathbf{0}$ with its equivalent then the case condition of a constrained poly-type is a conjunction of primitive case conditions. The subformula $\mathsf{C}_1 \wedge \mathsf{C}_2 \wedge |\tilde{\beta}, c/m|$ in the formula $\mathsf{C}_1 \wedge \mathsf{C}_2 \wedge |\tilde{\beta}, c/m| \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i$ is necessary for $m = 0$ but redundent for $m > 0$. The subformula $\mathsf{C}_1 \wedge \mathsf{C}_2$ in the formula $\mathsf{C}_1 \wedge \mathsf{C}_2 \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i$ is necessary for $m = 0$ but redundent for $m > 0$.

Example 6.4 *By the definition of* $\barwedge^{\ell}$,

$$\begin{array}{l} \langle (\alpha, 1), (\alpha, 1) = \mathbf{1} \wedge |(\alpha, \epsilon), \mathit{list}/1| \rangle \barwedge^{\ell} \langle \mathit{nat}, \mathit{true} \rangle \\ \quad = \left\{ \begin{array}{c} \langle \mathit{nat}, |(\alpha, 1), \mathit{nat}/0| \wedge (\alpha, 1) = \mathbf{1} \wedge |(\alpha, \epsilon), \mathit{list}/1| \wedge \mathit{true} \rangle, \\ \langle \mathit{nat}, (\alpha, 1) = \mathbf{1} \wedge (\alpha, 1) = \mathbf{1} \wedge |(\alpha, \epsilon), \mathit{list}/1| \wedge \mathit{true} \rangle, \\ \langle \mathbf{0}, (\alpha, 1) \neq \mathbf{1} \wedge \backslash\!\!\backslash (\alpha, 1), \mathit{nat}/0 \backslash\!\!\backslash \wedge (\alpha, 1) = \mathbf{1} \wedge |(\alpha, \epsilon), \mathit{list}/1| \wedge \mathit{true} \rangle \end{array} \right\} \end{array}$$

Two of the three constrained poly-types have unsatisfiable case conditions which will make case conditions in two constrained variable poly-typings in example 6.6 unsatisfiable.

The following lemma states that $\veebar^{\ell}$ and $\barwedge^{\ell}$ are respectively liftings of $\veebar$ and $\barwedge$.

Lemma 6.5 For any $(\tau, \mathsf{C}_1), (\sigma, \mathsf{C}_2) \in \mathsf{Poly} \Diamond \mathsf{Case}$ and any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C}_1)$ and $\kappa(\mathsf{C}_2)$,

$$\text{Operator } \veebar^{\ell} : (\mathsf{Poly}\Diamond\mathsf{Case}) \times (\mathsf{Poly}\Diamond\mathsf{Case}) \mapsto \wp(\mathsf{Poly}\Diamond\mathsf{Case})$$

$$(\mathbf{0}, \mathsf{C}_1) \veebar^{\ell} (\sigma, \mathsf{C}_2) \;=\; \{(\sigma, \mathsf{C}_1 \wedge \mathsf{C}_2)\}$$

$$(\mathbf{1}, \mathsf{C}_1) \veebar^{\ell} (\sigma, \mathsf{C}_2) \;=\; \{(\mathbf{1}, \mathsf{C}_1 \wedge \mathsf{C}_2)\}$$

$$(\tilde{\beta}, \mathsf{C}_1) \veebar^{\ell} (\tilde{\beta}', \mathsf{C}_2) \;=\; \begin{cases} \{(\tilde{\beta}, \mathsf{C}_1 \wedge \mathsf{C}_2)\} & if\ \tilde{\beta}' \equiv \tilde{\beta} \\ \{(\tilde{\beta}, \tilde{\beta}' = \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2), (\tilde{\beta}', \tilde{\beta} = \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2), (\mathbf{1}, \tilde{\beta} \neq \mathbf{0} \wedge \tilde{\beta}' \neq \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2)\} & if\ \tilde{\beta}' \not\equiv \tilde{\beta} \end{cases}$$

$$(\tilde{\beta}, \mathsf{C}_1) \veebar^{\ell} (c(\sigma_1, \cdots, \sigma_m), \mathsf{C}_2) \;=\;$$
$$\begin{cases} \{(c(\eta_1, \cdots, \eta_m), \mathsf{C}_1 \wedge \mathsf{C}_2 \wedge |\tilde{\beta}, c/m| \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i) | (\eta_i, \mathsf{C}_i) \in ((\tilde{\beta}, i), \mathsf{C}_1 \wedge |\tilde{\beta}, c/m|) \veebar^{\ell} (\sigma_i, \mathsf{C}_2)\} \\ \cup \\ \{(c(\sigma_1, \cdots, \sigma_m), \tilde{\beta} = \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2), (\mathbf{1}, \tilde{\beta} \neq \mathbf{0} \wedge \backslash\tilde{\beta}, c/m\backslash \wedge \mathsf{C}_1 \wedge \mathsf{C}_2)\} \end{cases}$$

$$(b(\tau_1, \cdots, \tau_l), \mathsf{C}_1) \veebar^{\ell} (c(\sigma_1, \cdots, \sigma_m), \mathsf{C}_2) \;=\; \begin{cases} \{(c(\eta_1, \cdots, \eta_m), \mathsf{C}_1 \wedge \mathsf{C}_2 \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i) | (\eta_i, \mathsf{C}_i) \in (\tau_i, \mathsf{C}_1) \veebar^{\ell} (\sigma_i, \mathsf{C}_2)\} \\ \qquad if\ c/m = b/l \\ \{(\mathbf{1}, \mathsf{C}_1 \wedge \mathsf{C}_2)\} \qquad if\ c/m \neq b/l \end{cases}$$

$$\text{Operator } \barwedge^{\ell} : (\mathsf{Poly}\Diamond\mathsf{Case}) \times (\mathsf{Poly}\Diamond\mathsf{Case}) \mapsto \wp(\mathsf{Poly}\Diamond\mathsf{Case})$$

$$(\mathbf{1}, \mathsf{C}_1) \barwedge^{\ell} (\sigma, \mathsf{C}_2) \;=\; \{(\sigma, \mathsf{C}_1 \wedge \mathsf{C}_2)\}$$

$$(\mathbf{0}, \mathsf{C}_1) \barwedge^{\ell} (\sigma, \mathsf{C}_2) \;=\; \{(\mathbf{0}, \mathsf{C}_1 \wedge \mathsf{C}_2)\}$$

$$(\tilde{\beta}, \mathsf{C}_1) \barwedge^{\ell} (\tilde{\beta}', \mathsf{C}_2) \;=\; \begin{cases} \{(\tilde{\beta}, \mathsf{C}_1 \wedge \mathsf{C}_2)\} & if\ \tilde{\beta}' \equiv \tilde{\beta} \\ \{(\tilde{\beta}, \tilde{\beta}' = \mathbf{1} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2), (\tilde{\beta}', \tilde{\beta} = \mathbf{1} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2), (\tilde{\beta}, \tilde{\beta} \neq \mathbf{1} \wedge \tilde{\beta}' \neq \mathbf{1} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2)\} & if\ \tilde{\beta}' \not\equiv \tilde{\beta} \end{cases}$$

$$(\tilde{\beta}, \mathsf{C}_1) \barwedge^{\ell} (c(\sigma_1, \cdots, \sigma_m), \mathsf{C}_2) \;=\;$$
$$\begin{cases} \{(c(\eta_1, \cdots, \eta_m), \mathsf{C}_1 \wedge \mathsf{C}_2 \wedge |\tilde{\beta}, c/m| \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i) | (\eta_i, \mathsf{C}_i) \in ((\tilde{\beta}, i), \mathsf{C}_1 \wedge |\tilde{\beta}, c/m|) \barwedge^{\ell} (\sigma_i, \mathsf{C}_2)\} \\ \cup \\ \{(c(\sigma_1, \cdots, \sigma_m), \tilde{\beta} = \mathbf{1} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2), (\mathbf{0}, \tilde{\beta} \neq \mathbf{1} \wedge \backslash\tilde{\beta}, c/m\backslash \wedge \mathsf{C}_1 \wedge \mathsf{C}_2)\} \end{cases}$$

$$(b(\tau_1, \cdots, \tau_l), \mathsf{C}_1) \barwedge^{\ell} (c(\sigma_1, \cdots, \sigma_m), \mathsf{C}_2) \;=\; \begin{cases} \{(c(\eta_1, \cdots, \eta_m), \mathsf{C}_1 \wedge \mathsf{C}_2 \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i) | (\eta_i, \mathsf{C}_i) \in (\tau_i, \mathsf{C}_1) \barwedge^{\ell} (\sigma_i, \mathsf{C}_2)\} \\ \qquad if\ c/m = b/l \\ \{(\mathbf{0}, \mathsf{C}_1 \wedge \mathsf{C}_2)\} \qquad if\ c/m \neq b/l \end{cases}$$

Figure 5: Approximate least upper bound and greatest lower bound operators

(a) there is an $(\eta, \mathsf{C}) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$ such that $\kappa(\mathsf{C})$ and $\kappa(\tau)\veebar\kappa(\sigma)\preceqq\kappa(\eta)$; and

(b) there is an $(\eta, \mathsf{C}) \in (\tau, \mathsf{C}_1)\barwedge^{\ell}(\sigma, \mathsf{C}_2)$ such that $\kappa(\mathsf{C})$ and $\kappa(\tau)\barwedge\kappa(\sigma)\preceqq\kappa(\eta)$.

Proof: It suffices to prove (a) since (b) is similar to (a). Suppose that $\tau, \sigma \notin \{\mathbf{0}, \mathbf{1}\}$ for otherwise the proof follows vacuously from figure 5. Let $h : \mathsf{Poly} \mapsto \mathsf{N}$ be defined as

$$\begin{aligned} h(\tilde{\beta}) &\stackrel{def}{=} 0 \\ h(c(\tau_1, \cdots, \tau_m)) &\stackrel{def}{=} 1 + \max_{1 \le i \le m} h(\tau_i) \end{aligned}$$

where $\tilde{\beta} \in \mathsf{Derived}$, $c \in \mathsf{Cons}$, and $\tau_i \in \mathsf{Poly}$ for $1 \le i \le m$ and $\le_2 \subseteq (\mathsf{N} \times \mathsf{N}) \times (\mathsf{N} \times \mathsf{N})$ be defined as

$$(i_1, j_1) \le_2 (i_2, j_2) \stackrel{def}{=} (i_1 \le i_2) \vee (i_1 = i_2) \wedge (j_1 \le j_2)$$

$\le_2$ is a total order. The remainder of the proof proceeds by induction on $(h(\tau), h(\sigma))$.

Basis. $(h(\tau), h(\sigma)) = (0, 0)$. We have that $\tau, \sigma \in \mathsf{Derived}$. If $\tau \equiv \sigma$ then $\kappa(\tau)\veebar\kappa(\sigma) = \kappa(\tau)$ by the definition of $\veebar$, and $(\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2) = \{(\tau, \mathsf{C}_1 \wedge \mathsf{C}_2)\}$ by figure 5. So, the lemma holds. Now suppose that $\tau \not\equiv \sigma$. We have that either $\kappa(\tau) = \mathbf{0}$ or $\kappa(\sigma) = \mathbf{0}$ or $\kappa(\tau) \neq \mathbf{0} \wedge \kappa(\sigma) \neq \mathbf{0}$. If $\kappa(\tau) = \mathbf{0}$ then $\kappa(\tau)\veebar\kappa(\sigma) = \kappa(\sigma)$ and hence the lemma holds since $(\sigma, \tau = \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$. If $\kappa(\sigma) = \mathbf{0}$ then $\kappa(\tau)\veebar\kappa(\sigma) = \kappa(\tau)$ and hence the lemma holds since $(\tau, \sigma = \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$. If $\kappa(\tau) \neq \mathbf{0} \wedge \kappa(\sigma) \neq \mathbf{0}$ then the lemma holds since $(\mathbf{1}, \tau \neq \mathbf{0} \wedge \sigma \neq \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$. So, the lemma holds when $(h(\tau), h(\sigma)) = (0, 0)$.

Induction. Now let $(h(\tau), h(\sigma)) \neq (0, 0)$. It suffices to consider the following two cases because $\veebar^{\ell}$ is symmetric and $\tau, \sigma \notin \{\mathbf{0}, \mathbf{1}\}$. (1) $\tau \in \mathsf{Derived}$ and $\sigma = c(\sigma_1, \cdots, \sigma_m)$. (2) $\tau = b(\tau_1, \cdots, \tau_l)$ and $\sigma = c(\sigma_1, \cdots, \sigma_m)$.

We first consider the case (1) . If $|\kappa(\tau), c/m|$ then $\kappa((\tau, i)), \kappa(\sigma_i) \in \mathsf{Mono}$ for $1 \le i \le m$ since $\kappa(\tau), \kappa(\sigma) \in \mathsf{Mono}$. By the induction hypothesis, there is an $(\eta_i, \mathsf{C}_i) \in ((\tau, i), \mathsf{C}_1 \wedge |\kappa(\tau), c/m|)\veebar^{\ell}(\sigma_i, \mathsf{C}_2)$ such that $\kappa(\mathsf{C}_i) \leftrightarrow \mathit{true}$, $\kappa(\eta_i) \in \mathsf{Mono}$ and $\kappa((\tau, i))\veebar\kappa(\sigma_i)\preceqq\kappa(\eta_i)$. Therefore, $\mathsf{C}_1 \wedge \mathsf{C}_2 \wedge |\kappa(\tau), c/m| \wedge \bigwedge_{i=1}^{i=m} \kappa(\mathsf{C}_i) \leftrightarrow \mathit{true}$, $\kappa(c(\eta_1, \cdots, \eta_m)) \in \mathsf{Mono}$ and , by the definition of $\veebar$, $\kappa(\tau)\veebar\kappa(\sigma)\preceqq c(\eta_1, \cdots, \eta_m)$. So, if $|\kappa(\tau), c/m|$ then the lemma holds. Now suppose that $\backslash\kappa(\tau), c/m\backslash$. If $\kappa(\tau) = \mathbf{0}$ then the lemma holds since $(\sigma, \tau = \mathbf{0} \wedge \mathsf{C}_1 \wedge \mathsf{C}_2) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$. Otherwise, the lemma holds since $(\mathbf{1}, \tau \neq \mathbf{0} \wedge \backslash\tau, c/m\backslash \wedge \mathsf{C}_1 \wedge \mathsf{C}_2) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$. This concludes the proof of the case (1).

Now consider the case (2). If $b/l \neq c/m$ then the lemma follows vacuously from figure 5. Let $b/l = c/m$. $\kappa(\tau_i), \kappa(\sigma_i) \in \mathsf{Mono}$ for $1 \le i \le m$ since $\kappa(\tau), \kappa(\sigma) \in \mathsf{Mono}$. By the induction hypothesis, there is an $(\eta_i, \mathsf{C}_i) \in (\tau_i, \mathsf{C}_1)\veebar^{\ell}(\sigma_i, \mathsf{C}_2)$ such that $\kappa(\mathsf{C}_i) \leftrightarrow \mathit{true}$, $\kappa(\eta_i) \in \mathsf{Mono}$ and $\kappa(\tau_i)\veebar\kappa(\sigma_i)\preceqq\kappa(\eta_i)$. By figure 5, $(c(\eta_1, \cdots, \eta_m), \mathsf{C}_1 \wedge \mathsf{C}_2 \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i) \in (\tau, \mathsf{C}_1)\veebar^{\ell}(\sigma, \mathsf{C}_2)$. $\mathsf{C}_1 \wedge \mathsf{C}_2 \wedge \bigwedge_{i=1}^{i=m} \mathsf{C}_i \leftrightarrow \mathit{true}$. $\kappa(c(\eta_1, \cdots, \eta_m)) \in \mathsf{Mono}$. $\kappa(\tau)\veebar\kappa(\sigma)\preceqq\kappa(c(\eta_1, \cdots, \eta_m))$ by the definition of $\veebar$. So, the lemma holds for the case (2). This completes the induction and the proof of the lemma. ∎

Lifting *lvt* and $\mathring{\sqcap}$

For a mono-type $M \in \mathsf{Mono}$ and a term $t \in \mathsf{Term}$, the abstract unification operator $MUNIFY$ for monomorphic type analysis needs to compute $lvt(M,t)$ - the least variable mono-typing to describe all substitutions θ such that $\theta(t) \in \gamma(M)$. We now design a lifting $lvt^{\ell} : (\mathsf{Poly}\diamond\mathsf{Case}) \times \mathsf{Term} \mapsto \wp(\mathsf{VPT}\diamond\mathsf{Case})$ of lvt. Since lvt makes use of $\sqcap$ which computes the greatest lower bound of two variable mono-typings, we first define $\sqcap^{\ell} : (\mathsf{VPT}\diamond\mathsf{Case}) \times (\mathsf{VPT}\diamond\mathsf{Case}) \mapsto \wp(\mathsf{VPT}\diamond\mathsf{Case})$ which approximates the greatest lower bound of two constrained variable poly-typings by a set of constrained variable poly-typings.

Let $\bar{\mu} = (\mu, \mathsf{C}) \in \mathsf{VPT}\diamond\mathsf{Case}$, $\bar{\nu} = (\nu, \mathsf{C}') \in \mathsf{VPT}\diamond\mathsf{Case}$, $dom(\mu) \cup dom(\nu) = \{X_1, \cdots, X_m\}$ and $\bar{\Phi}_1, \bar{\Phi}_2 \in \wp(\mathsf{VPT}\diamond\mathsf{Case})$. Define $\bar{\mu}(T) \stackrel{def}{=} (\mu(T), \mathsf{C})$ for any $T \in \mathsf{Poly}$.

$$\bar{\mu}\sqcap^{\ell}\bar{\nu} \stackrel{def}{=} \bigcup_{\substack{(\tau_1,\mathsf{C}_1) \in \bar{\mu}(X_1)\curlywedge^{\ell}\bar{\nu}(X_1) \\ \vdots \\ (\tau_m,\mathsf{C}_m) \in \bar{\mu}(X_m)\curlywedge^{\ell}\bar{\nu}(X_m)}} \{(\{X_j \mapsto \tau_j | 1 \leq j \leq m\}, \mathsf{C} \wedge \mathsf{C}' \wedge \bigwedge_{1 \leq j \leq m} \mathsf{C}_j)\}$$

$$\bar{\Phi}_1 \sqcap^{\ell\star} \bar{\Phi}_2 \stackrel{def}{=} \bigcup_{\bar{\mu}_1 \in \bar{\Phi}_1 \wedge \bar{\mu}_2 \in \bar{\Phi}_2} \bar{\mu}_1 \sqcap^{\ell} \bar{\mu}_2$$

Example 6.6 *This example illustrates $\sqcap^{\ell}$ and $\sqcap^{\ell\star}$. Let*

$$\begin{aligned}
\bar{\mu}_1 &= \langle \emptyset, (\alpha,\epsilon) = \mathbf{1} \rangle \\
\bar{\mu}_2 &= \langle \{X \mapsto (\alpha,1)\}, (\alpha,1) = \mathbf{1} \wedge |(\alpha,\epsilon), list/1| \rangle \\
\bar{\mu}_3 &= \langle \{Y \mapsto nat, X \mapsto (\alpha,1)\}, |(\alpha,1), nat/0| \wedge |(\alpha,\epsilon), list/1| \rangle \\
\bar{\mu}_4 &= \langle \{X \mapsto nat, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, true \rangle
\end{aligned}$$

By the definition of $\sqcap^{\ell}$,

$$\bar{\mu}_1 \sqcap^{\ell} \bar{\mu}_4 = \langle \{X \mapsto nat, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, (\alpha,\epsilon) = \mathbf{1} \rangle$$

$$\begin{aligned}
&\bar{\mu}_2 \sqcap^{\ell} \bar{\mu}_4 \qquad \textit{by definition of } \sqcap^{\ell} \textit{ and example 6.4} \\
&= \left\{ \begin{array}{l}
\langle \{X \mapsto nat, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, (\alpha,1) = \mathbf{1} \wedge |(\alpha,\epsilon), list/1| \wedge |(\alpha,1), nat/0| \rangle, \\
\langle \{X \mapsto nat, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, (\alpha,1) = \mathbf{1} \wedge |(\alpha,\epsilon), list/1| \wedge (\alpha,1) = \mathbf{1} \rangle, \\
\left\langle \{X \mapsto \mathbf{0}, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, \begin{array}{r} (\alpha,1) = \mathbf{1} \wedge |(\alpha,\epsilon), list/1| \\ \wedge \backslash\!\!|(\alpha,1), nat/0 \backslash\!\!| \wedge (\alpha,1) \neq \mathbf{1} \end{array} \right\rangle,
\end{array} \right\} \\
&\approx^{\flat} \{\langle \{X \mapsto nat, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, (\alpha,1) = \mathbf{1} \wedge |(\alpha,\epsilon), list/1| \rangle\}
\end{aligned}$$

Note that two constrained variable poly-typings with unsatisfiable case conditions have been removed since they describes the empty set of substitutions under any assignement of mono-types to original type parameters. The folowing can be obtained similarly.

$$\bar{\mu}_3 \sqcap^{\ell} \bar{\mu}_4 \approx^{\flat} \{\langle \{Y \mapsto nat, X \mapsto nat, Y_0 \mapsto (\alpha,\epsilon), X_0 \mapsto nat\}, |(\alpha,1), nat/0| \wedge |(\alpha,\epsilon), list/1| \rangle\}$$

Therefore, $\{\bar\mu_1,\bar\mu_2,\bar\mu_3\}\sqcap^{\ell\star}\{\bar\mu_4\} = (\bar\mu_1\sqcap^{\ell}\bar\mu_4)\cup(\bar\mu_2\sqcap^{\ell}\bar\mu_4)\cup(\bar\mu_3\sqcap^{\ell}\bar\mu_4)\approx^{\flat}\{\bar\mu_5,\bar\mu_6,\bar\mu_7\}$ *where*

$$\begin{array}{rcl}
\bar\mu_5 &=& \langle\{X\mapsto nat, Y_0\mapsto(\alpha,\epsilon), X_0\mapsto nat\}, (\alpha,\epsilon)=\mathbf{1}\rangle,\\
\bar\mu_6 &=& \langle\{X\mapsto nat, Y_0\mapsto(\alpha,\epsilon), X_0\mapsto nat\}, (\alpha,1)=\mathbf{1}\wedge|(\alpha,\epsilon), list/1|\rangle,\\
\bar\mu_7 &=& \langle\{Y\mapsto nat, X\mapsto nat, Y_0\mapsto(\alpha,\epsilon), X_0\mapsto nat\}, |(\alpha,1), nat/0|\wedge|(\alpha,\epsilon), list/1|\rangle
\end{array}$$

The following lemma states that $\sqcap^{\ell}$ is a lifting of $\sqcap$ because $\bot$ describes the empty set of substitutions.

Lemma 6.7 For any $(\mu,\mathsf{C}),(\nu,\mathsf{C}')\in\mathsf{VPT}\diamond\mathsf{Case}$ and any $\kappa\in\mathcal{A}_t$ such that $\kappa(\mathsf{C})$ and $\kappa(\mathsf{C}')$, if $\kappa\circ\mu\sqcap\kappa\circ\nu\neq\bot$ then there is a $(\pi,\mathsf{C}'')\in(\mu,\mathsf{C})\sqcap^{\ell}(\nu,\mathsf{C}')$ such that $\kappa(\mathsf{C}'')$ and $(\kappa\circ\mu\sqcap\kappa\circ\nu)\sqsubseteq^{\circ}\kappa\circ\pi$.

PROOF: Let $dom(\mu)\cup dom(\nu)=\{X_1,\cdots,X_m\}$. Since $\kappa(\mathsf{C})$ and $\kappa(\mathsf{C}')$, we have that $\kappa\circ\mu(X_i)\in\mathsf{Mono}$ and $\kappa\circ\nu(X_i)\in\mathsf{Mono}$ for $1\le i\le m$. By hypothesis, $\kappa\circ\mu(X_i)\barwedge\kappa\circ\nu(X_i)\neq\mathbf{0}$ for $1\le i\le m$. By lemma 6.5, there is a $(\tau_i,\mathsf{C}_i)\in(\mu(X_i),\mathsf{C})\barwedge^{\ell}(\nu(X_i),\mathsf{C}')$ such that $\kappa(\mathsf{C}_i)$ and $\kappa\circ\mu(X_i)\barwedge\kappa\circ\nu(X_i)\preceqq\kappa(\tau_i)$. Let $\pi=\{X_1\mapsto\tau_1,\cdots,X_m\mapsto\tau_m\}$ and $\mathsf{C}''=\mathsf{C}\wedge\mathsf{C}'\wedge\bigwedge_{1\le i\le m}\mathsf{C}_i$. $(\pi,\mathsf{C}'')\in(\mu,\mathsf{C})\sqcap^{\ell}(\nu,\mathsf{C}')$ by the definition of $\sqcap^{\ell}$. $\kappa(\mathsf{C}'')$ since $\kappa(\mathsf{C})$, $\kappa(\mathsf{C}')$ and $\kappa(\mathsf{C}_i)$ for $1\le i\le m$. So, the lemma holds since $\kappa\circ\mu(X_i)\barwedge\kappa\circ\nu(X_i)\preceqq\kappa(\tau_i)=\kappa\circ\pi(X_i)$ for $1\le i\le m$. ∎

lvt^{ℓ} propagates a constrained poly-type (τ,C) downwards a term t and obtains a set of constrained variable poly-typings such that, for any $\kappa\in\mathcal{A}_t$, if $\kappa(\mathsf{C})$ then the set of substitutions θ such that $\theta(t)\in\gamma(\kappa(\tau))$ is described by a variable mono-typing that is covered by the set of constrained variable poly-typings under κ. If $\tau=\tilde\beta$ is a derived type parameter and $t=f(t_1,\cdots,t_n)$ is a compound term then it is necessary to make a case analysis of the structure of $\tilde\beta$. Let

$$f(X_1,\cdots,X_n):c(\beta_1,\cdots,\beta_m)\leftarrow X_1:T_1,\cdots,X_n:T_n$$

Constrained variable poly-typings are generated for two mutually exclusive cases $\tilde\beta=\mathbf{1}$ and $\tilde\beta\neq\mathbf{1}\wedge\tilde\beta\neq\mathbf{0}\wedge|\tilde\beta,c/m|$. There is no need to generate constrained variable poly-typings for the case $\tilde\beta=\mathbf{0}$ or the case $\tilde\beta\neq\mathbf{1}\wedge\backslash\!\backslash\tilde\beta,c/m\backslash\!\backslash$ since there is no substitution θ such that $\theta(t)\in\gamma(\kappa(\tilde\beta))$ for any κ satisfying either of them. The case $\tilde\beta=\mathbf{1}$ corresponds to the constrained variable poly-typing $(\emptyset,\mathsf{C}\wedge\tilde\beta=\mathbf{1})$. The case $\tilde\beta=c((\tilde\beta,1),\cdots,(\tilde\beta,m))$ corresponds to a set of constrained variable poly-typings which are obtained by first propagating $(\Bbbk(T_i),\mathsf{C}\wedge\tilde\beta\neq\mathbf{1}\wedge\tilde\beta\neq\mathbf{0}\wedge|\tilde\beta,c/m|)$ downwards t_i where $\Bbbk=\{\beta_j\mapsto(\tilde\beta,j)|1\le j\le m\}$, and then approximating their greatest lower bound.

Let $f\in\mathsf{Func}$ and $f(X_1,\cdots,X_n):c(\beta_1,\cdots,\beta_m)\leftarrow X_1:T_1,\cdots,X_n:T_n$, $(\tau,\mathsf{C})\in\mathsf{Poly}\diamond\mathsf{Case}$, $X\in\mathsf{Vars}$, $t,t_1,\cdots,t_n\in\mathsf{Term}$. Define

$$
\begin{array}{lcl}
lvt^{\ell}((\tau,\mathsf{C}),X) & \stackrel{def}{=} & \{\langle\{X\mapsto\tau\},\mathsf{C}\rangle\} \\
lvt^{\ell}((\mathbf{1},\mathsf{C}),t) & \stackrel{def}{=} & \{\langle\emptyset,\mathsf{C}\rangle\} \\
lvt^{\ell}((\tilde{\beta},\mathsf{C}),f(t_1,\cdots,t_n)) & \stackrel{def}{=} & \{\langle\emptyset,\mathsf{C}\wedge\tilde{\beta}=\mathbf{1}\rangle\} \\
\multicolumn{3}{r}{\cup\left\{\begin{array}{l} \sqcap^{\ell\star}{}_{1\le i\le n} lvt^{\ell}(\langle\Bbbk(T_i),\mathsf{C}\wedge|\tilde{\beta},c/m|\rangle,t_i) \\ \mathit{where}\ \Bbbk=\{\beta_j\mapsto(\tilde{\beta},j)\mid 1\le j\le m\}\end{array}\right.} \\
lvt^{\ell}((b(\tau_1,\cdots,\tau_l),\mathsf{C}),f(t_1,\cdots,t_n)) & \stackrel{def}{=} & \\
\multicolumn{3}{r}{\left\{\begin{array}{ll} \emptyset & \mathit{if}\ b/l\neq c/m \\ \sqcap^{\ell\star}{}_{1\le i\le n} lvt^{\ell}(\langle\Bbbk(T_i),\mathsf{C}\rangle,t_i) & \mathit{if}\ b/l=c/m \\ \mathit{where}\ \Bbbk=\{\beta_j\mapsto\tau_j\mid 1\le j\le m\} & \end{array}\right.}
\end{array}
$$

Example 6.8 *This example illustrates lvt^{ℓ} through computing $lvt^{\ell}(\langle(\alpha,\epsilon),true\rangle,[s(Y),X])$. Let type definitions be given as in example 4.3. By the definition of lvt^{ℓ}, we have*

$$
\begin{array}{l}
lvt^{\ell}(\langle(\alpha,\epsilon),true\rangle,[s(Y),X]) \\
\quad = \quad \{\langle\emptyset,(\alpha,\epsilon)=\mathbf{1}\rangle\}\ \cup \\
\qquad\quad (\ lvt^{\ell}(\langle(\alpha,1),|(\alpha,\epsilon),list/1|\rangle,s(Y))\ \sqcap^{\ell\star}\ lvt^{\ell}(\langle list((\alpha,1)),|(\alpha,\epsilon),list/1|\rangle,[X])\ )
\end{array}
$$

The two invocations to lvt^{ℓ} in the above computation are accomplished as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{lvt^{\ell}(\langle(\alpha,1),|(\alpha,\epsilon),list/1|\rangle,s(Y))} \\
= & \{\langle\emptyset,|(\alpha,\epsilon),list/1|\wedge(\alpha,1)=\mathbf{1}\rangle\}\ \cup\ lvt^{\ell}(\langle nat,|(\alpha,\epsilon),list/1|\wedge|(\alpha,1),nat/0|\rangle,Y) \\
= & \{\langle\emptyset,|(\alpha,\epsilon),list/1|\wedge(\alpha,1)=\mathbf{1}\rangle,\langle\{Y\mapsto nat\},|(\alpha,\epsilon),list/1|\wedge|(\alpha,1),nat/0|\rangle\} \\
\multicolumn{2}{l}{lvt^{\ell}(\langle list((\alpha,1)),|(\alpha,\epsilon),list/1|\rangle,[X])} \\
= & lvt^{\ell}(\langle\Bbbk(\beta''),|(\alpha,\epsilon),list/1|\rangle,X)\ \sqcap^{\ell\star}\ lvt^{\ell}(\langle\Bbbk(list(\beta'')),|(\alpha,\epsilon),list/1|\rangle,[\,]) \\
 & \mathit{with}\ \Bbbk=\{\beta''\mapsto(\alpha,1)\} \\
= & lvt^{\ell}(\langle(\alpha,1),|(\alpha,\epsilon),list/1|\rangle,X)\ \sqcap^{\ell\star}\ lvt^{\ell}(\langle list((\alpha,1)),|(\alpha,\epsilon),list/1|\rangle,[\,]) \\
= & \{\langle\{X\mapsto(\alpha,1)\},|(\alpha,\epsilon),list/1|\rangle)\}\ \sqcap^{\ell\star}\ \{\langle\emptyset,true\rangle\} \\
= & \{\langle\{X\mapsto(\alpha,1)\},|(\alpha,\epsilon),list/1|\rangle\}
\end{array}
$$

Therefore,

$$
\begin{array}{ll}
\multicolumn{2}{l}{lvt^{\ell}(\langle(\alpha,\epsilon),true\rangle,[s(Y),X])} \\
= & \{\langle\emptyset,(\alpha,\epsilon)=\mathbf{1}\rangle\}\ \cup \\
 & \left\{\begin{array}{l}\langle\emptyset,|(\alpha,\epsilon),list/1|\wedge(\alpha,1)=\mathbf{1}\rangle, \\ \langle\{Y\mapsto nat\},|(\alpha,\epsilon),list/1|\wedge|(\alpha,1),nat/0|\rangle\end{array}\right\}\ \sqcap^{\ell\star}\ \{\langle\{X\mapsto(\alpha,1)\},|(\alpha,\epsilon),list/1|\rangle\} \\
= & \left\{\begin{array}{l}\langle\emptyset,(\alpha,\epsilon)=\mathbf{1}\rangle, \\ \langle\{X\mapsto(\alpha,1)\},|(\alpha,\epsilon),list/1|\wedge(\alpha,1)=\mathbf{1}\rangle), \\ \langle\{Y\mapsto nat,X\mapsto(\alpha,1)\},|(\alpha,\epsilon),list/1|\wedge|(\alpha,1),nat/0|\rangle\end{array}\right\} \\
= & \{\bar{\mu}_1,\bar{\mu}_2,\bar{\mu}_3\}\qquad \mathit{with}\ \bar{\mu}_1,\bar{\mu}_2,\bar{\mu}_3\ \mathit{being\ those\ in\ example\ 6.6}
\end{array}
$$

The following lemma states that lvt^{ℓ} is a lifting of lvt since $\mathring{\perp}$ describes the empty set of substitutions.

Lemma 6.9 For any $t \in \mathsf{Term}$, any $(\tau, \mathsf{C}) \in \mathsf{Poly}\Diamond\mathsf{Case}$, and any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C})$, if $lvt(\kappa(\tau), t) \neq \mathring{\perp}$ then there is a $(\mu', \mathsf{C}') \in lvt^{\ell}((\tau, \mathsf{C}), t)$ such that $\kappa(\mathsf{C}')$ and $lvt(\kappa(\tau), t) \stackrel{\circ}{\sqsubseteq} \kappa \circ \mu'$.

PROOF: If $\tau = \mathbf{1}$ then the lemma follows directly from the definitions of lvt and lvt^{ℓ}. Assume $\tau \neq \mathbf{1}$ and define $h : \mathsf{Term} \mapsto \mathsf{N}$ as follows.

$$
\begin{aligned}
h(X) &\stackrel{def}{=} 0 \\
h(f(t_1, \cdots, t_n)) &\stackrel{def}{=} 1 + \max_{1 \leq i \leq n} h(t_i)
\end{aligned}
$$

where $X \in \mathsf{Vars}$, $f \in \mathsf{Func}$ and $t_i \in \mathsf{Term}$ for $1 \leq i \leq n$. The remainder of proof is done by induction on $h(t)$.

Basis. Let $h(t) = 0$. Then t is a variable. By the definitions of lvt and lvt^{ℓ},

$$lvt(\kappa(\tau), t) = \{t \mapsto \kappa(\tau)\}$$

$$lvt^{\ell}((\tau, \mathsf{C}), t) = \{(\{t \mapsto \tau\}, \mathsf{C})\}$$

So, the lemma holds by letting $\mu' = \{t \mapsto \tau\}$ and $\mathsf{C}' = \mathsf{C}$.

Induction. $h(t) \neq 0$. Let $t = f(t_1, \cdots, t_n)$ where $t_i \in \mathsf{Term}$ for $1 \leq i \leq n$ and

$$f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$$

There are two cases to consider. (1) $\tau \in \mathsf{Derived}$; and (2) $\tau = b(\tau_1, \cdots, \tau_l)$. We first consider the case (1). Since $lvt(\kappa(\tau), t) \neq \mathring{\perp}$, either $\kappa(\tau) = \mathbf{1}$ or $|\kappa(\tau), c/m|$. Assume $\kappa(\tau) = \mathbf{1}$ first. The lemma holds because $lvt(\kappa(\tau), t) = \emptyset$ and $(\emptyset, \tau = \mathbf{1} \wedge \mathsf{C}) \in lvt^{\ell}((\tau, \mathsf{C}), t)$ by the definitions of lvt and lvt^{ℓ}. Now assume $|\kappa(\tau), c/m|$. So, $\kappa(\mathsf{C} \wedge |\tau, c/m|)$. Let $\mathbb{k} = \{\beta_j \mapsto (\tau, j) | 1 \leq j \leq m\}$. $\kappa \circ \mathbb{k} = \{\beta_j \mapsto (\kappa(\tau), j) | 1 \leq j \leq m\}$. $lvt(\kappa \circ \mathbb{k}(T_i), t_i) \neq \mathring{\perp}$ since $lvt(\kappa(\tau), t) \neq \mathring{\perp}$. By the induction hypothesis, for any $1 \leq i \leq n$, there is a $(\mu_i', \mathsf{C}_i') \in lvt^{\ell}((\mathbb{k}(T_i), \mathsf{C} \wedge |\tau, c/m|), t_i)$ such that $\kappa(\mathsf{C}_i')$ and $lvt(\kappa \circ \mathbb{k}(T_i), t_i) \stackrel{\circ}{\sqsubseteq} \kappa \circ \mu_i'$. By lemma 6.7 and the definitions of lvt and lvt^{ℓ}, there is $(\mu', \mathsf{C}') \in lvt^{\ell}((\tau, \mathsf{C}), t)$ such that $\kappa(\mathsf{C}')$ and $lvt(\kappa(\tau), t) \stackrel{\circ}{\sqsubseteq} \kappa \circ \mu'$. So, the lemma holds for the case (1). Now consider the case (2). We have that $b/l = c/m$ since $lvt(\kappa(\tau), t) \neq \mathring{\perp}$. The rest of the proof of the case (2) is the same as the case (1) with $|\kappa(\tau), c/m|$. This concludes the induction and hence the proof of the lemma. ∎

Lifting lt, lts, and $\sqsubseteq$

For a $\mu \in (\mathsf{Vars} \rightarrowtail \mathsf{Mono})$ and a $t \in \mathsf{Term}$, the abstract unification operator *MUNIFY* computes $lt(t, \mu)$ - the least mono-type that describes the set of terms $\theta(t)$ for all substitutions $\theta \in \mathring{\gamma}(\mu)$. This section presents a lifting $lt^{\ell} : \mathsf{Term} \times (\mathsf{VPT}\Diamond\mathsf{Case}) \mapsto \wp(\mathsf{Poly}\Diamond\mathsf{Case})$ of lt.

lt makes use of lts to compute, for a mono-type $M \in \mathsf{Mono}$ and a general type $G \in \mathsf{Gen}$, the least mono-type substitution such that $M \stackrel{\circ}{\preceq} \kappa(G)$. For a constrained poly-type $(\tau, \mathsf{C}) \in \mathsf{Poly}\Diamond\mathsf{Case}$, and a general type $G \in \mathsf{Gen}$, $lts^{\ell}((\tau, \mathsf{C}), G)$ is a set of constrained poly-type substitutions that, for any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C})$, contains a constrained variable poly-typing $(\Bbbk', \mathsf{C}')$ such that $\kappa(\mathsf{C}') \leftrightarrow \mathit{true}$ and $\kappa \circ \Bbbk'$ is greater than $lts(\kappa(\tau), t)$ with respect to $\sqsubseteq^{*}$. If $\tau = \tilde{\beta}$ is a derived type parameter and $t = f(t_1, \cdots, t_n)$ with $f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$, then constrained poly-type substitutions are generated for the two exclusive cases $|\tilde{\beta}, c/m|$ and $\backslash\!\!\backslash \tilde{\beta}, c/m \backslash\!\!\backslash$.

Let $\tau, \tau_1, \cdots, \tau_l \in \mathsf{Poly}$, $\mathsf{C} \in \mathsf{Case}$, $b, c \in \mathsf{Cons}$, $\tilde{\beta} \in \mathsf{Derived}$, $\beta, \beta_1, \cdots, \beta_m \in \mathsf{Para}$ and $\bar{\mathcal{T}} \in \wp(\mathsf{Poly} \diamondsuit \mathsf{Case})$ be a set of constrained poly-types. $lts^{\ell} : (\mathsf{Poly} \diamondsuit \mathsf{Case}) \times \mathsf{Gen} \mapsto \wp(\mathsf{PTS} \diamondsuit \mathsf{Case})$ is defined as follows.

$$
\begin{array}{rcl}
lts^{\ell}((\tau, \mathsf{C}), \mathbf{1}) & \stackrel{def}{=} & \{(\emptyset, \mathsf{C})\} \\
lts^{\ell}((\tau, \mathsf{C}), \beta) & \stackrel{def}{=} & \{(\{\beta \mapsto \tau\}, \mathsf{C})\} \\
lts^{\ell}((\mathbf{0}, \mathsf{C}), c(\beta_1, \cdots, \beta_m)) & \stackrel{def}{=} & \{(\emptyset, \mathsf{C})\} \\
lts^{\ell}((\tilde{\beta}, \mathsf{C}), c(\beta_1, \cdots, \beta_m)) & \stackrel{def}{=} & \left\{ \begin{array}{c} (\emptyset, \tilde{\beta} = \mathbf{0} \wedge \mathsf{C}), \\ (\bowtie, \backslash\!\!\backslash \tilde{\beta}, c/m \backslash\!\!\backslash \wedge \mathsf{C}), \\ (\{\beta_j \mapsto (\tilde{\beta}, j) | 1 \leq j \leq m\}, |\tilde{\beta}, c/m| \wedge \mathsf{C}) \end{array} \right\} \\
lts^{\ell}((b(\tau_1, \cdots, \tau_l), \mathsf{C}), c(\beta_1, \cdots, \beta_m)) & \stackrel{def}{=} & \left\{ \begin{array}{ll} \{(\{\beta_j \mapsto \tau_j | 1 \leq j \leq m\}, \mathsf{C})\} & if\ c/m = b/l \\ \{(\bowtie, \mathsf{C})\} & if\ c/m \neq b/l \end{array} \right.
\end{array}
$$

$$
lts^{\ell\star}(\bar{\mathcal{T}}, G) \stackrel{def}{=} \bigcup_{(\tau, \mathsf{C}) \in \bar{\mathcal{T}}} lts^{\ell}((\tau, \mathsf{C}), G)
$$

Example 6.10 *By the definitions of lts^{ℓ} and $lts^{\ell\star}$,*

$$
\begin{array}{rcl}
lts^{\ell\star}(\{\langle nat, (\alpha, \epsilon) = \mathbf{1}\rangle\}, \beta'') & = & lts^{\ell}(\langle nat, (\alpha, \epsilon) = \mathbf{1}\rangle, \beta'') \\
& = & \{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \\
lts^{\ell\star}(\{\langle list(\mathbf{0}), true\rangle\}, list(\beta'')) & = & lts^{\ell}(\langle list(\mathbf{0}), true\rangle, list(\beta''))) \\
& = & \{\langle\emptyset, true\rangle\} \qquad note\ that\ \emptyset(\beta'') = \mathbf{0} \\
lts^{\ell\star}(\{\langle \mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle\}, nat) & = & lts^{\ell}(\langle \mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle, nat) \\
& = & \{\langle\bowtie, (\alpha, \epsilon) = \mathbf{1}\rangle\} \\
lts^{\ell\star}(\{\langle \mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle\}, \beta'') & = & lts^{\ell}(\langle \mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle, \beta'') \\
& = & \{\langle\{\beta'' \mapsto \mathbf{1}\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \\
lts^{\ell\star}(\{\langle list(nat), (\alpha, \epsilon) = \mathbf{1}\rangle\}, list(\beta'')) & = & lts^{\ell}(\langle list(nat), (\alpha, \epsilon) = \mathbf{1}\rangle, list(\beta'')) \\
& = & \{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\}
\end{array}
$$

Lemma 6.11 For any $(\tau, \mathsf{C}) \in \mathsf{Poly} \diamondsuit \mathsf{Case}$, any $G \in \mathsf{Gen}$ and any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C})$, there is a $(\Bbbk', \mathsf{C}') \in lts^{\ell}((\tau, \mathsf{C}), G)$ such that $\kappa(\mathsf{C}')$ and $lts(\kappa(\tau), G) \sqsubseteq^{*} \kappa \circ \Bbbk'$, that is, lts^{ℓ} is a lifting of lts.

PROOF: Assume $\kappa(\tau) \not\equiv \mathbf{0}$ and $G \neq \mathbf{1}$ for otherwise the lemma follows from the definitions of lts and lts^{ℓ} immediately. Either $G \in \mathsf{Para}$ or $G = c(\beta_1, \cdots, \beta_m)$ with $\beta_1, \cdots, \beta_m \in \mathsf{Para}$. Assume $G \in \mathsf{Para}$ first. $lts(\kappa(\tau), G) = \{G \mapsto \kappa(\tau)\}$ and $lts^{\ell}((\tau, \mathsf{C}), G) = \{(\{G \mapsto \tau\}, \mathsf{C})\}$. The lemma holds by letting $\Bbbk' = \{G \mapsto \tau\}$ and $\mathsf{C}' = \mathsf{C}$. Now assume $G = c(\beta_1, \cdots, \beta_m)$ where $\beta_1, \cdots, \beta_m \in \mathsf{Para}$. There are two cases to consider. (1) $\tau \in \mathsf{Derived}$ and (2) $\tau = b(\tau_1, \cdots, \tau_l)$ with $b \in \mathsf{Cons}$ and $\tau_1, \cdots, \tau_l \in \mathsf{Poly}$.

We consider the case (1) first. If $|\kappa(\tau), c/m|$ then $lts(\kappa(\tau), c(\beta_1, \cdots, \beta_m)) = \{\beta_j \mapsto (\kappa(\tau), j) | 1 \leq j \leq m\}$ by the definition of lts. Let $\Bbbk' = \{\beta_j \mapsto (\tau, j) | 1 \leq j \leq m\}$ and $\mathsf{C}' = \nmid\tau, c/m\nmid \wedge \mathsf{C}$. $(\Bbbk', \mathsf{C}') \in lts^{\ell}((\tau, \mathsf{C}), G)$. We have that $\kappa(\mathsf{C}')$ and $lts(\kappa(\tau), c(\beta_1, \cdots, \beta_m)) = \kappa \circ \Bbbk'$. So, the lemma holds. If $\nmid\kappa(\tau), c/m\nmid$ then $lts(\kappa(\tau), c(\beta_1, \cdots, \beta_m)) =\bowtie$. The lemma holds by letting $\Bbbk' =\bowtie$ and $\mathsf{C}' = \nmid\tau, c/m\nmid \wedge \mathsf{C}$.

Now consider the case (2). If $b/l = c/m$ then $lts(\kappa(\tau), c(\beta_1, \cdots, \beta_m)) = \{\beta_j \mapsto \kappa(\tau_j) | 1 \leq j \leq m\}$ by the definition of lts. The lemma holds, letting $\Bbbk' = \{\beta_j \mapsto \tau_j | 1 \leq j \leq m\}$ and $\mathsf{C}' = \mathsf{C}$, since $(\Bbbk', \mathsf{C}') \in lts^{\ell}((\tau, \mathsf{C}), G)$, $\kappa(\mathsf{C}')$ and $lts(\kappa(\tau), c(\beta_1, \cdots, \beta_m)) = \kappa \circ \Bbbk'$. If $b/l \neq c/m$ then $lts(\kappa(\tau), c(\beta_1, \cdots, \beta_m)) =\bowtie$. The lemma holds by letting $\Bbbk' =\bowtie$ and $\mathsf{C}' = \mathsf{C}$. This completes the proof of the lemma. ∎

The operator $\sqcup^{\ell} : (\mathsf{PTS}\diamondsuit\mathsf{Case}) \times (\mathsf{PTS}\diamondsuit\mathsf{Case}) \mapsto \wp(\mathsf{PTS}\diamondsuit\mathsf{Case})$ defined below lifts $\sqcup$. Let $T \in \mathsf{Type}$ and $(\Bbbk, \mathsf{C})$ be a constrained poly-type substitution. $(\Bbbk, \mathsf{C})(T) \stackrel{def}{=} (\Bbbk(T), \mathsf{C})$. Let $\bar{\Bbbk}_i = (\Bbbk_i, \mathsf{C}_i) \in \mathsf{PTS}\diamondsuit\mathsf{Case}$ with $1 \leq i \leq 2$ and $dom(\Bbbk_1) \cup dom(\Bbbk_2) = \{\beta_1, \cdots, \beta_m\}$.

$$\bar{\Bbbk}_1 \sqcup^{\ell} \bar{\Bbbk}_2 \stackrel{def}{=} \begin{cases} \{(\bowtie, \mathsf{C}_1 \wedge \mathsf{C}_2)\} & \text{if } \Bbbk_1 =\bowtie \vee \Bbbk_2 =\bowtie \\ \bigcup_{\substack{(\tau'_1, \mathsf{C}'_1) \in \bar{\Bbbk}_1(\beta_1) \veebar^{\ell} \bar{\Bbbk}_2(\beta_1) \\ \vdots \\ (\tau'_m, \mathsf{C}'_m) \in \bar{\Bbbk}_1(\beta_m) \veebar^{\ell} \bar{\Bbbk}_2(\beta_m)}} \{(\bigcup_{1 \leq j \leq m} \{\beta_j \mapsto \tau'_j\}, \mathsf{C}_1 \wedge \mathsf{C}_2 \wedge \bigwedge_{1 \leq j \leq m} \mathsf{C}'_j)\} & \text{otherwise} \end{cases}$$

$$\bar{\mathcal{K}}_1 \sqcup^{\ell\star} \bar{\mathcal{K}}_2 \stackrel{def}{=} \bigcup_{\bar{\Bbbk}_1 \in \bar{\mathcal{K}}_1 \wedge \bar{\Bbbk}_2 \in \bar{\mathcal{K}}_2} \bar{\Bbbk}_1 \sqcup^{\ell} \bar{\Bbbk}_2 \quad \text{where } \bar{\mathcal{K}}_1, \bar{\mathcal{K}}_2 \in \wp(\mathsf{PTS}\diamondsuit\mathsf{Case})$$

Example 6.12 *By the definitions of* $\sqcup^{\ell\star}$ *and* $\sqcup^{\ell}$,

$$\begin{aligned} &\{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \sqcup^{\ell\star} \{\langle\emptyset, true\rangle\} \\ &\quad = \langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle \sqcup^{\ell} \langle\emptyset, true\rangle = \{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \\ &\{\langle\{\beta'' \mapsto \mathbf{1}\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \sqcup^{\ell\star} \{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \\ &\quad = \langle\{\beta'' \mapsto \mathbf{1}\}, (\alpha, \epsilon) = \mathbf{1}\rangle \sqcup^{\ell} \langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle = \{\langle\{\beta'' \mapsto \mathbf{1}\}, (\alpha, \epsilon) = \mathbf{1}\rangle\} \end{aligned}$$

Lemma 6.13 For any $(\Bbbk_1, \mathsf{C}_1), (\Bbbk_2, \mathsf{C}_2) \in \mathsf{PTS}\diamondsuit\mathsf{Case}$, and any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C}_1) \leftrightarrow true$ and $\kappa(\mathsf{C}_2) \leftrightarrow true$, there is a $(\Bbbk_{12}, \mathsf{C}_{12}) \in (\Bbbk_1, \mathsf{C}_1) \sqcup^{\ell} (\Bbbk_2, \mathsf{C}_2)$ such that $\kappa(\mathsf{C}_{12}) \leftrightarrow true$ and $(\kappa \circ \Bbbk_1 \sqcup \kappa \circ \Bbbk_2) \sqsubseteq \kappa \circ \Bbbk_{12}$. In other words, $\sqcup^{\ell}$ is a lifting of $\sqcup$.

PROOF: Assume $\Bbbk_1 \neq\bowtie \wedge \Bbbk_2 \neq\bowtie$ for otherwise the lemma follows directly from the definitions of $\sqcup$ and $\sqcup^{\ell}$. Let $dom(\Bbbk_1) \cup dom(\Bbbk_2) = \{\beta_1, \cdots, \beta_m\}$. Since $\kappa(\mathsf{C}_1) \leftrightarrow true$ and $\kappa(\mathsf{C}_2) \leftrightarrow$

$true$, $\kappa\circ\mathbb{k}_1(\beta_i) \in \mathsf{Mono}$ and $\kappa\circ\mathbb{k}_2(\beta_i) \in \mathsf{Mono}$ for $1 \leq i \leq m$. By lemma 6.5, there is a $(\tau_i', \mathsf{C}_i') \in (\mathbb{k}_1(\beta_i), \mathsf{C}_1)\veebar^{\ell}(\mathbb{k}_2(\beta_i), \mathsf{C}_2)$ such that $\kappa(\mathsf{C}_i') \leftrightarrow true$ and $\kappa \circ \mathbb{k}_1(\beta_i)\veebar\kappa \circ \mathbb{k}_2(\beta_i)\precsim\kappa(\tau_i')$. Let $\mathbb{k}_{12} = \{\beta_1 \mapsto \tau_1', \cdots, \beta_m \mapsto \tau_m'\}$ and $\mathsf{C}_{12} = \mathsf{C}_1\wedge\mathsf{C}_2\wedge\bigwedge_{1\leq i\leq m} \mathsf{C}_i'$. $(\mathbb{k}_{12}, \mathsf{C}_{12}) \in (\mathbb{k}_1, \mathsf{C}_1)\uplus^{\ell}(\mathbb{k}_2, \mathsf{C}_2)$ by the definition of $\uplus^{\ell}$. $\kappa(\mathsf{C}_{12}) \leftrightarrow true$ since $\kappa(\mathsf{C}_1) \leftrightarrow true$, $\kappa(\mathsf{C}_2) \leftrightarrow true$ and $\kappa(\mathsf{C}_i') \leftrightarrow true$ for $1 \leq i \leq m$. So, the lemma holds since $\kappa \circ \mathbb{k}_1(\beta_i)\veebar\kappa \circ \mathbb{k}_2(\beta_i)\precsim\kappa(\tau_i') = \kappa \circ \mathbb{k}_{12}(\beta_i)$ for $1 \leq i \leq m$. ∎

lt^{ℓ} propagates a constrained variable poly-typing upwards a term and obtains a set of constrained poly-types. Let $t, t_1, \cdots, t_n \in \mathsf{Term}$, $\bar{\mu} \in \mathsf{VPT}\diamondsuit\mathsf{Case}$, $X \in \mathsf{Vars}$ and $f \in \mathsf{Func}$ with $f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$.

$$
\begin{array}{rcl}
lt^{\ell}(X, \bar{\mu}) & \stackrel{def}{=} & \{\bar{\mu}(X)\} \\
lt^{\ell}(f(t_1, \cdots, t_n), \bar{\mu}) & \stackrel{def}{=} & \{\bar{\kappa}(c(\beta_1, \cdots, \beta_m)) | \bar{\kappa} \in {\uplus^{\ell}}^{\star}_{1\leq i\leq n} lts^{\ell^{\star}}(lt^{\ell}(t_i, \bar{\mu}), T_i)\}
\end{array}
$$

Example 6.14 *Let type definitions be given as in example 4.3 and* $\bar{\mu}_5$ *be that in example 6.6. This example show how* lt^{ℓ} *works by computing* $lt^{\ell}([s(Y), X], \bar{\mu}_5)$. *By the defintions of* lt^{ℓ} *and* $lts^{\ell^{\star}}$,

$$lt^{\ell}([\,], \bar{\mu}_5) = \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in \{\langle\emptyset, true\rangle\} = \{\langle list(\mathbf{0}), true\rangle\}$$

$$lt^{\ell}(X, \bar{\mu}_5) = \{\bar{\mu}_5(X)\} = \{\langle nat, (\alpha, \epsilon) = \mathbf{1}\rangle\}$$

Therefore,

$$
\begin{array}{l}
lt^{\ell}([X], \bar{\mu}_5) = \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in lts^{\ell^{\star}}(lt^{\ell}(X, \bar{\mu}_5), \beta'')\uplus^{\ell^{\star}} lts^{\ell^{\star}}(lt^{\ell}([\,], \bar{\mu}_5), list(\beta''))\} \\
\quad = \quad \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in lts^{\ell^{\star}}(\{\langle nat, (\alpha, \epsilon) = \mathbf{1}\rangle\}, \beta'')\uplus^{\ell^{\star}} lts^{\ell^{\star}}(\{\langle list(\mathbf{0}), true\rangle\}, list(\beta''))\} \\
\qquad \textit{by example 6.10} \\
\quad = \quad \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in \{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\}\uplus^{\ell^{\star}}\{\langle\emptyset, true\rangle\}\} \\
\qquad \textit{by example 6.12} \\
\quad = \quad \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in \{\langle\{\beta'' \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1}\rangle\}\} \\
\quad = \quad \{\langle list(nat), (\alpha, \epsilon) = \mathbf{1}\rangle\}
\end{array}
$$

$$
\begin{array}{rcl}
lt^{\ell}(s(Y), \bar{\mu}_5) & = & \{\bar{\mathbb{k}}(nat) \mid \bar{\mathbb{k}} \in lts^{\ell^{\star}}(lt^{\ell}(Y, \bar{\mu}_5), nat)\} \\
& = & \{\bar{\mathbb{k}}(nat) \mid \bar{\mathbb{k}} \in lts^{\ell^{\star}}(\{\langle\mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle\}, nat)\} \\
& & \textit{by example 6.10} \\
& = & \{\langle\mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle\}
\end{array}
$$

So,

$$
\begin{array}{l}
lt^{\ell}([s(Y), X], \bar{\mu}_5) = \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in lts^{\ell^{\star}}(lt^{\ell}(s(Y), \bar{\mu}_5), \beta'')\uplus^{\ell^{\star}} lts^{\ell^{\star}}(lt^{\ell}([X], \bar{\mu}_5), list(\beta''))\} \\
\quad = \quad \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in lts^{\ell^{\star}}(\{\langle\mathbf{1}, (\alpha, \epsilon) = \mathbf{1}\rangle\}, \beta'')\uplus^{\ell^{\star}} lts^{\ell^{\star}}(\{\langle list(nat), (\alpha, \epsilon) = \mathbf{1}\rangle\}, list(\beta''))\}
\end{array}
$$

$$
\begin{array}{rl}
 & \textit{by example 6.10} \\
= & \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in \{\langle\{\beta'' \mapsto \mathbf{1}\}, (\alpha,\epsilon) = \mathbf{1}\rangle\} \sqcup^{\ell\star} \{\langle\{\beta'' \mapsto nat\}, (\alpha,\epsilon) = \mathbf{1}\rangle\}\} \\
 & \textit{by example 6.12} \\
= & \{\bar{\mathbb{k}}(list(\beta'')) \mid \bar{\mathbb{k}} \in \{\langle\{\beta'' \mapsto \mathbf{1}\}, (\alpha,\epsilon) = \mathbf{1}\rangle\}\} \\
= & \{\langle list(\mathbf{1}), (\alpha,\epsilon) = \mathbf{1}\rangle\}
\end{array}
$$

Lemma 6.15 For any $(\mu, \mathsf{C}) \in \mathsf{VPT} \diamond \mathsf{Case}$, any $t \in \mathsf{Term}$ and any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C}) \leftrightarrow$ *true*, there is a $(\tau', \mathsf{C}') \in lt^{\ell}(t, (\mu, \mathsf{C}))$ such that $\kappa(\mathsf{C}')$ and $lt(t, \kappa \circ \mu) \preceq \kappa(\tau')$, that is, lt^{ℓ} is a lifting of lt.

PROOF: Let $h : \mathsf{Term} \mapsto \mathsf{N}$ be defined as in lemma 6.9. The proof is done by induction on $h(t)$.

Basis. $h(t) = 0$. By the definitions of lt and lt^{ℓ}, $lt(t, \kappa \circ \mu) = \kappa \circ \mu(t)$ and $lt^{\ell}(t, (\mu, \mathsf{C})) = \{(\mu(t), \mathsf{C})\}$ since t is a variable. The lemma holds by taking $\tau' = \mu(t)$ and $\mathsf{C}' = \mathsf{C}$.

Induction. $h(t) \neq 0$. Let $t = f(t_1, \cdots, t_n)$ where $t_1, \cdots, t_n \in \mathsf{Term}$ and $f \in \mathsf{Func}$ with $f(X_1, \cdots, X_n) : c(\beta_1, \cdots, \beta_m) \leftarrow X_1 : T_1, \cdots, X_n : T_n$. $h(t_i) < h(t)$ for $1 \leq i \leq n$. By the induction hypothesis, for each $1 \leq i \leq n$, there is a $(\tau_i, \mathsf{C}_i) \in lt^{\ell}(t_i, (\mu, \mathsf{C}))$ such that $\kappa(\mathsf{C}_i)$ and $lt(t_i, \kappa \circ \mu) \preceq \kappa(\tau_i)$. By lemma 6.11 and the monotonicities of lts and lts^{ℓ}, for each $1 \leq i \leq n$, there is a $(\mathbb{k}'_i, \mathsf{C}'_i) \in lts^{\ell}((\tau_i, \mathsf{C}_i), T_i)$ such that $\kappa(\mathsf{C}'_i)$ and $\forall \beta \in \mathsf{Para}.(lts(lt(t_i, \kappa \circ \mu), T_i)(\beta) \preceq \kappa \circ \mathbb{k}'_i(\beta))$. By lemma 6.13 and the monotonicities of $\sqcup$ and $\sqcup^{\ell}$, there is a $(\mathbb{k}'', \mathsf{C}'') \in \sqcup^{\ell}_{1 \leq i \leq n}(\mathbb{k}'_i, \mathsf{C}'_i)$ such that $\kappa(\mathsf{C}'')$ and $\forall \beta \in \mathsf{Para}.((\sqcup_{1 \leq i \leq n} lts(lt(t_i, \kappa \circ \mu), T_i))(\beta) \preceq \kappa \circ \mathbb{k}''(\beta))$. So, the lemma holds by the definitions of lt and lt^{ℓ}. ∎

Lifting *MUNIFY*, *solve*, *up* and *down*

The abstract unification operator *PUNIFY* for polymorphic type analysis is defined by lifting the abstract unification operator *MUNIFY* for monomorphic type analysis. *PUNIFY* performs pre-unification in the same way as *MUNIFY* and then propagates polymorphic type information by an operator $solve^{\ell} : \wp(\mathsf{Eqn}) \times (\mathsf{VPT} \diamond \mathsf{Case}) \mapsto \wp(\mathsf{VPT} \diamond \mathsf{Case})$ which is a lifting of *solve*. The propagation of polymorphic type information downwards and upwards a set of equations in solved form are performed by $down^{\ell} : \wp(\mathsf{Eqn}) \times (\mathsf{VPT} \diamond \mathsf{Case}) \mapsto \wp(\mathsf{VPT} \diamond \mathsf{Case})$ and $up^{\ell} : \wp(\mathsf{Eqn}) \times (\mathsf{VPT} \diamond \mathsf{Case}) \mapsto \wp(\mathsf{VPT} \diamond \mathsf{Case})$ which are liftings of *down* and *up* respectively.

For a $\bar{\mu} = (\mu, \mathsf{C}) \in \mathsf{VPT} \diamond \mathsf{Case}$ and an $X \in \mathsf{Vars}$, $\bar{\mu}(X) \stackrel{def}{=} (\mu(X), \mathsf{C})$. Given a $\bar{\mu} \in \mathsf{VPT} \diamond \mathsf{Case}$ and an $E \in \wp(\mathsf{Eqn})$, $down^{\ell}$ first obtains a set of constrained variable poly-typings for each $(X = t) \in E$ by propagating $\bar{\mu}(X)$ downwards t and then approximates the greatest lower bound of these sets of constrained variable poly-typings plus $\{\bar{\mu}\}$.

$$down^{\ell}(E, \bar{\mu}) \stackrel{def}{=} \{\bar{\mu}\} \sqcap^{\ell\star} \sqcap^{\ell\star}_{(X=t) \in E} lvt^{\ell}(\bar{\mu}(X), t)$$

Example 6.16 *This example illustrates $down^{\ell}$. Let $\bar{\mu}_0 = \langle\{Y_0 \mapsto (\alpha, \epsilon), X_0 \mapsto nat\}, true\rangle$. By the definition of $down^{\ell}$,*

$$down^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_0)$$

$$
\begin{aligned}
&= \{\bar{\mu}_0\} \sqcap^{\ell\star} lvt^{\ell}(\bar{\mu}_0(X_0), X) \sqcap^{\ell\star} lvt^{\ell}(\bar{\mu}_0(Y_0), [s(Y), X]) \\
&= \{\bar{\mu}_0\} \sqcap^{\ell\star} lvt^{\ell}(\langle nat, true\rangle, X) \sqcap^{\ell\star} lvt^{\ell}(\langle(\alpha,\epsilon), true\rangle, [s(Y), X]) \\
&\quad \textit{by example 6.8 and the definition of } lvt^{\ell} \\
&\approx^{\flat} \{\bar{\mu}_0\} \sqcap^{\ell\star} \{\langle\{X \mapsto nat\}, true\rangle\} \sqcap^{\ell\star} \{\bar{\mu}_1, \bar{\mu}_2, \bar{\mu}_3\} \\
&\approx^{\flat} \{\bar{\mu}_4\} \sqcap^{\ell\star} \{\bar{\mu}_1, \bar{\mu}_2, \bar{\mu}_3\} \\
&\quad \textit{by example 6.6} \\
&\approx^{\flat} \{\bar{\mu}_5, \bar{\mu}_6, \bar{\mu}_7\} \qquad \textit{with } \bar{\mu}_1, \cdots, \bar{\mu}_7 \textit{ being those in example 6.6}
\end{aligned}
$$

Given a $\bar{\mu} \in \mathsf{VPT} \diamondsuit \mathsf{Case}$ and an $E \in \wp(\mathsf{Eqn})$, up^{ℓ} first obtains a set of constrained poly-types for each $(X = t) \in E$ by propagating $\bar{\mu}$ upwards t, then forms a set of constrained variable poly-typings from these sets of constrained poly-types and finally approximates the greatest lower bound of the set of constrained variable poly-typings plus $\{\bar{\mu}\}$.

$$
\begin{aligned}
& up^{\ell}(\{X_i = t_i | 1 \le i \le n\}, \bar{\mu}) \\
& \stackrel{def}{=} \{\bar{\mu}\} \sqcap^{\ell\star} \{\langle\{X_i \mapsto T_i | 1 \le i \le n\}, \bigwedge_{1 \le i \le n} \mathsf{C}_i\rangle \mid \forall 1 \le i \le n.((T_i, \mathsf{C}_i) \in lt^{\ell}(t_i, \bar{\mu}))\}
\end{aligned}
$$

Example 6.17 *Let type definitions be given as in example 4.3 and* $\bar{\mu}_5, \bar{\mu}_6, \bar{\mu}_7$ *be those in example 6.6. Then*

$$
\begin{aligned}
& up^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_5) \\
&= \{\bar{\mu}_5\} \sqcap^{\ell\star} \left\{ \langle\{X_0 \mapsto T_1, Y_0 \mapsto T_2\}, \mathsf{C}_1 \wedge \mathsf{C}_2\rangle \;\middle|\; \begin{array}{c} (T_1, \mathsf{C}_1) \in lt^{\ell}(X, \bar{\mu}_5) \\ \wedge \\ (T_2, \mathsf{C}_2) \in lt^{\ell}([s(Y), X], \bar{\mu}_5) \end{array} \right\} \\
& \quad \textit{by example 6.14} \\
&= \{\bar{\mu}_5\} \sqcap^{\ell\star} \left\{ \langle\{X_0 \mapsto T_1, Y_0 \mapsto T_2\}, \mathsf{C}_1 \wedge \mathsf{C}_2\rangle \;\middle|\; \begin{array}{c} (T_1, \mathsf{C}_1) \in \{\langle nat, (\alpha,\epsilon) = \mathbf{1}\rangle\} \\ \wedge \\ (T_2, \mathsf{C}_2) \in \{\langle list(\mathbf{1}), (\alpha,\epsilon) = \mathbf{1}\rangle\} \end{array} \right\} \\
&= \{\bar{\mu}_5\} \sqcap^{\ell\star} \{\langle\{X_0 \mapsto nat, Y_0 \mapsto list(\mathbf{1})\}, (\alpha,\epsilon) = \mathbf{1}\rangle\} \\
&= \{\langle\{X \mapsto nat, Y_0 \mapsto list(\mathbf{1}), X_0 \mapsto nat\}, (\alpha,\epsilon) = \mathbf{1}\rangle\}
\end{aligned}
$$

Similary,

$$
\begin{aligned}
& up^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_6) \\
&\quad = \{\langle\{X \mapsto nat, Y_0 \mapsto list((\alpha, 1)), X_0 \mapsto nat\}, |(\alpha,\epsilon), list/1| \wedge (\alpha, 1) = \mathbf{1}\rangle\} \\
& up^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_7) \\
&\quad = \{\langle\{Y \mapsto nat, X \mapsto nat, Y_0 \mapsto list(nat), X_0 \mapsto nat\}, |(\alpha, 1), nat/0| \wedge |(\alpha,\epsilon), list/1|\rangle\}
\end{aligned}
$$

Given a $\bar{\mu} \in \mathsf{VPT} \diamondsuit \mathsf{Case}$ and an $E \in \wp(\mathsf{Eqn})$, $solve^{\ell}$ first propagates $\bar{\mu}$ downwards E to obtain a set of constrained variable poly-typings and then propagates each of these constrained variable poly-typing upwards E.

$$solve^{\ell}(E, \bar{\mu}) \stackrel{def}{=} \bigcup_{\bar{\nu} \in down^{\ell}(E, \bar{\mu})} up^{\ell}(E, \bar{\nu})$$

Example 6.18 *Let $\bar{\mu}_0, \bar{\mu}_5, \bar{\mu}_6, \bar{\mu}_7$ be those in example 6.16. By example 6.16,*

$$down^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_0) = \{\bar{\mu}_5, \bar{\mu}_6, \bar{\mu}_7\}$$

Therefore,

$$\begin{array}{rl} & solve^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_0) \\ = & up^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_5) \cup up^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_6) \\ & \qquad \cup\, up^{\ell}(\{X_0 = X, Y_0 = [s(Y), X]\}, \bar{\mu}_7) \\ & \textit{by example 6.17} \\ = & \left\{ \begin{array}{l} \langle \{X \mapsto nat, Y_0 \mapsto list(\mathbf{1}), X_0 \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1} \rangle, \\ \langle \{X \mapsto nat, Y_0 \mapsto list((\alpha, 1)), X_0 \mapsto nat\}, |(\alpha, \epsilon), list/1| \wedge (\alpha, 1) = \mathbf{1} \rangle, \\ \langle \{Y \mapsto nat, X \mapsto nat, Y_0 \mapsto list(nat), X_0 \mapsto nat\}, |(\alpha, 1), nat/0| \wedge |(\alpha, \epsilon), list/1| \rangle \end{array} \right\} \end{array}$$

Lemma 6.19 For any $(\mu, \mathsf{C}) \in \mathsf{VPT} \Diamond_{v} \mathsf{Case}$, any $E \in \wp(\mathsf{Eqn})$, and any $\kappa \in \mathcal{A}_t$ such that $\kappa(\mathsf{C})$,

(a) if $down(E, \kappa \circ \mu) \neq \perp$ then there is a $(\mu', \mathsf{C}') \in down^{\ell}(E, (\mu, \mathsf{C}))$ such that $\kappa(\mathsf{C}')$ and $down(E, \kappa \circ \mu) \sqsubseteq^{\circ} \kappa \circ \mu'$, and,

(b) if $up(E, \kappa \circ \mu) \neq \perp$ then there is a $(\mu', \mathsf{C}') \in up^{\ell}(E, (\mu, \mathsf{C}))$ such that $\kappa(\mathsf{C}')$ and $up(E, \kappa \circ \mu) \sqsubseteq^{\circ} \kappa \circ \mu'$, and,

(c) if $solve(E, \kappa \circ \mu) \neq \perp$ then there is a $(\mu', \mathsf{C}') \in solve^{\ell}(E, (\mu, \mathsf{C}))$ such that $\kappa(\mathsf{C}')$ and $solve(E, \kappa \circ \mu) \sqsubseteq^{\circ} \kappa \circ \mu'$.

In other words, $down^{\ell}$, up^{ℓ} and $solve^{\ell}$ are liftings of $down$, up and $solve$ respectively because $\perp$ describes the empty set of substitutions.

PROOF: By lemma 6.9, for each $(X = t) \in E$, there is a $(\mu'', \mathsf{C}'') \in lvt^{\ell}((\mu(X), \mathsf{C}), t)$ such that $\kappa(\mathsf{C}'')$ and $lvt(\kappa \circ \mu(X), t) \sqsubseteq^{\circ} \kappa \circ \mu''$. So, (a) holds by lemma 6.7.

Let $E = \{X_1 \mapsto t_1, \cdots, X_n \mapsto t_n\}$. By lemma 6.15, for each $1 \leq i \leq n$, there is $(\tau_i', \mathsf{C}_i') \in lt^{\ell}(t, (\mu, \mathsf{C}))$ such that $\kappa(\mathsf{C}_i')$ and $lt(t, \kappa \circ \mu) \stackrel{\equiv}{\preceq} \kappa(\tau_i')$. Therefore, (b) holds by lemma 6.7 and the definitions of lt and lt^{ℓ}.

(c) follows from (a) and (b). ∎

Given $A, B \in \mathsf{Atom}$ and $\bar{\Phi}_A, \bar{\Phi}_B \in \wp(\mathsf{VPT} \Diamond_{v} \mathsf{Case})$, *PUNIFY* first performs pre-unification and then propagates each constrained variable poly-typing in a similar way to *MUNIFY*.

$$PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B) \stackrel{def}{=}$$

$$\left\{\begin{array}{l} let \quad \Psi \text{ be a renaming substitution such that} \\ \qquad \Psi(vars(A) \cup dom(\bar{\Phi}_A)) \cap (vars(B) \cup dom(\bar{\Phi}_B)) = \emptyset, \\ \qquad E = eq(mgu(\Psi(A), B)), \\ \qquad \mathsf{V} = vars(B) \cup dom(\bar{\Phi}_B) \\ in \\ \left\{\begin{array}{ll} \bigcup_{\substack{(\mu_A, \mathsf{C}_A) \in \bar{\Phi}_A \\ (\mu_B, \mathsf{C}_B) \in \bar{\Phi}_B}} solve^{\ell}(E, \langle \Psi(\mu_A) \cup \mu_B, \mathsf{C}_A \wedge \mathsf{C}_B \rangle) \upharpoonright^{\flat} \mathsf{V} & if\ E \neq fail \\ \emptyset & if\ E = fail \end{array}\right. \end{array}\right.$$

$$\bar{\Phi} \upharpoonright^{\flat} \mathsf{V} \stackrel{def}{=} \{(\mu \upharpoonright \mathsf{V}, \mathsf{C}) \mid (\mu, \mathsf{C}) \in \bar{\Phi}\}$$

Example 6.20 *Let*

$$\begin{array}{rcl} A & = & foo(X, Y) \\ B & = & foo(X, [s(Y), X]) \\ \bar{\Phi}_A & = & \{\langle \{Y \mapsto (\alpha, \epsilon), X \mapsto nat\}, true \rangle\} \\ \bar{\Phi}_B & = & \{\langle \emptyset, true \rangle\} \end{array}$$

Suppose that $\Psi(Z) = Z_0$ *for all* $Z \in \mathsf{Vars}$. *We have*

$$\begin{array}{rcl} E & = & \{X_0 = X, Y_0 = [s(Y), X]\} \\ \mathsf{V} & = & \{Y, X\} \end{array}$$

So,

$$PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B) = solve^{\ell}(E, \bar{\mu}_0) \upharpoonright^{\flat} \mathsf{V} \qquad \text{with } \bar{\mu}_0 \text{ being that in example 6.16}$$

by example 6.18

$$= \left\{\begin{array}{l} \langle \{X \mapsto nat, Y_0 \mapsto list(\mathbf{1}), X_0 \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1} \rangle, \\ \langle \{X \mapsto nat, Y_0 \mapsto list((\alpha, 1)), X_0 \mapsto nat\}, |(\alpha, \epsilon), list/1| \wedge (\alpha, 1) = \mathbf{1} \rangle, \\ \langle \{Y \mapsto nat, X \mapsto nat, Y_0 \mapsto list(nat), X_0 \mapsto nat\}, |(\alpha, 1), nat/0| \wedge |(\alpha, \epsilon), list/1| \rangle \end{array}\right\} \upharpoonright^{\flat} \mathsf{V}$$

$$= \left\{\begin{array}{l} \langle \{X \mapsto nat\}, (\alpha, \epsilon) = \mathbf{1} \rangle, \\ \langle \{X \mapsto nat\}, |(\alpha, \epsilon), list/1| \wedge (\alpha, 1) = \mathbf{1} \rangle, \\ \langle \{Y \mapsto nat, X \mapsto nat\}, |(\alpha, 1), nat/0| \wedge |(\alpha, \epsilon), list/1| \rangle \end{array}\right\}$$

Theorem 6.21 $I^{\flat} = \langle (\mathcal{D}^{\flat}, \sqsubseteq^{\flat}), (\sqcup^{\flat}, PUNIFY) \rangle$ is a polymorphic $\gamma^{\flat}$-abstraction of $I = \langle (\wp(Sub), \subseteq), (\cup, UNIFY) \rangle$ with respect to $\mathcal{A}_t$.

PROOF: By definition 3.1, we need to prove that for any $\bar{\Phi}_A, \bar{\Phi}_B \in \wp(\mathsf{VPT} \diamond \mathsf{Case})$, any $\kappa \in \mathcal{A}_t$, and any $\Theta_A, \Theta_B \in \wp(Sub)$,

$$\Theta_A \subseteq \gamma^{\flat}(\bar{\Phi}_A, \kappa) \wedge \Theta_B \subseteq \gamma^{\flat}(\bar{\Phi}_B, \kappa) \rightarrow UNIFY(A, \Theta_A, B, \Theta_B) \subseteq \gamma^{\flat}(PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B), \kappa)$$

By the definition of $UNIFY$, it is equivalent to prove that for any $\bar{\Phi}_A, \bar{\Phi}_B \in \wp(\mathsf{VPT} \diamond \mathsf{Case})$, any $\kappa \in \mathcal{A}_t$, and any $\theta_A, \theta_B \in Sub$,

$$\left(\begin{array}{c} \theta_A \in \gamma^\flat(\bar{\Phi}_A, \kappa) \\ \wedge \\ \theta_B \in \gamma^\flat(\bar{\Phi}_B, \kappa) \\ \wedge \\ unify(A, \theta_A, B, \theta_B) \neq fail \end{array}\right) \rightarrow unify(A, \theta_A, B, \theta_B) \in \gamma^\flat(PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B), \kappa)$$

Now suppose that $\theta_A \in \gamma^\flat(\bar{\Phi}_A, \kappa) \wedge \theta_B \in \gamma^\flat(\bar{\Phi}_B, \kappa) \wedge unify(A, \theta_A, B, \theta_B) \neq fail$. By the definition of $\gamma^\flat$, there are a $(\mu_A, \mathsf{C}_A) \in \bar{\Phi}_A$ such that $\kappa(\mathsf{C}_A) \leftrightarrow \mathit{true}$ and $\theta_A \in \gamma^{\wp}(\kappa \circ \mu_A)$ and a $(\mu_B, \mathsf{C}_B) \in \bar{\Phi}_B$ such that $\kappa(\mathsf{C}_B) \leftrightarrow \mathit{true}$ and $\theta_B \in \gamma^{\wp}(\kappa \circ \mu_B)$. We have that $unify(A, \theta_A, B, \theta_B) \in MUNIFY(A, \kappa \circ \mu_A, B, \kappa \circ \mu_B)$ by theorem 5.5. By lemma 6.19, there is a $(\mu_{AB}, \mathsf{C}_{AB}) \in PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B)$ such that $\kappa(\mathsf{C}_{AB}) \leftrightarrow \mathit{true}$ and $MUNIFY(A, \kappa \circ \mu_A, B, \kappa \circ \mu_B) \sqsubseteq^{\wp} \kappa \circ \mu_{AB}$. So, $unify(A, \theta_A, B, \theta_B) \in \gamma^\flat(PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B), \kappa)$. This concludes the proof of the theorem. ∎

7 ELIMINATING REDUNDANCY

We have so far used a set of constrained variable poly-typings $\bar{\Phi}$ as a representative of the set of its equivalents with respect to $\approx^\flat$. Since the time complexity of $PUNIFY(A, \bar{\Phi}_A, B, \bar{\Phi}_B)$ is proportional to the product of the number of constrained variable poly-typings in $\bar{\Phi}_A$ and that of constrained variable poly-typings in $\bar{\Phi}_B$, it is desirable to represent an equivalence class of $\approx^\flat$ by the member of the equivalence class that contains the smallest number of constrained variable poly-typings. Let $\bar{\Phi} \in \wp(\mathsf{VPT} \diamondsuit \mathsf{Case})$ and $\bar{\mu}_1 = (\mu_1, \mathsf{C}_1) \in \bar{\Phi}$. $\bar{\Phi}$ is equivalent to $\bar{\Phi} \setminus \{\bar{\mu}_1\}$ with respect to $\approx^\flat$ if either there is another constrained variable poly-typing $\bar{\mu}_2 = (\mu_2, \mathsf{C}_2)$ such that $\bar{\mu}_1$ is subsumed by $\bar{\mu}_2$, i.e., $\bar{\mu}_1 \trianglelefteq^\flat \bar{\mu}_2$ or $\bar{\mu}_1$ always describes the empty set of substitutions, i.e., $\forall.(\mathsf{C}_1 \rightarrow \mu_1 = \underline{\Omega})$. Therefore, $\bar{\mu}_1$ is redundant in $\bar{\Phi}$ in the sense that its removal from $\bar{\Phi}$ does not reduce type information expressed.

Example 7.1 *Let* $\mathsf{Cons} = \{nat/0, list/1, \mathbf{0}/0, \mathbf{1}/1\}$ *and*

$$\begin{array}{lcl} \bar{\nu}_1 = (\nu_1, \mathsf{C}_{\nu_1}) & = & (\{U \mapsto list(\mathbf{0}), W \mapsto (\beta, \epsilon)\}, |(\beta, \epsilon), \mathbf{0}/0|) \\ \bar{\nu}_2 = (\nu_2, \mathsf{C}_{\nu_2}) & = & (\{T \mapsto (\beta, 1), U \mapsto list(\mathbf{0}), V \mapsto (\beta, \epsilon), W \mapsto (\beta, \epsilon)\}, |(\beta, \epsilon), list/1|) \\ \bar{\nu}_3 = (\nu_3, \mathsf{C}_{\nu_3}) & = & (\{T \mapsto (\beta, 1), U \mapsto list((\beta, 1)), V \mapsto (\beta, \epsilon), W \mapsto list((\beta, 1))\}, |(\beta, \epsilon), list/1|) \end{array}$$

Then $\bar{\nu}_1$ and $\bar{\nu}_2$ in $\{\bar{\nu}_1, \bar{\nu}_2, \bar{\nu}_3\}$ are redundent since $\bar{\nu}_1$ always describes the empty set of substitutions, i.e.,

$$\forall.(\mathsf{C}_{\nu_1} \rightarrow \nu_1 = \underline{\Omega}) \qquad (f1)$$

and $\bar{\nu}_2$ always describes a smaller set of substitutions than $\bar{\nu}_3$ does, i.e.,

$$\bar{\nu}_2 \trianglelefteq^\flat \bar{\nu}_3 \qquad (f2)$$

Eliminating redundant constrained variable poly-typings from a set of constrained variable poly-typings is reduced into solving constraints of special forms. By equation 1, in order to decide whether or not $\bar{\mu}_1$ is subsumed by $\bar{\mu}_2$, it is necessary to decide the truth values of the formula $\forall.(\mathsf{C}_1 \wedge \neg\mathsf{C}_2 \rightarrow \mu_1 = \text{⚲})$ and the formula $\forall.(\mathsf{C}_1 \wedge \mathsf{C}_2 \rightarrow \mu_1 \overset{\circ}{\sqsubseteq} \mu_2)$. If there is a variable $X \in dom(\mu_1)$ such that $\mu_1(X) = \mathbf{0}$ then the formula $\forall.(\mathsf{C}_1 \wedge \neg\mathsf{C}_2 \rightarrow \mu_1 = \text{⚲})$ is trivially true. Suppose that $\mu_1(X) \not\equiv \mathbf{0}$ for all variables $X \in dom(\mu_1)$. Then the formula $\forall.(\mathsf{C}_1 \wedge \neg\mathsf{C}_2 \rightarrow \mu_1 = \text{⚲})$ is transformed into the following form by first removing $X \mapsto \tau$ with $\tau \notin \mathsf{Derived}$ from μ_1 and then writing $X \mapsto \tilde{\beta}$ in μ_1 as $\tilde{\beta} = \mathbf{0}$.

$$\forall.(\mathsf{C}_1 \rightarrow \mathsf{C}_2 \vee \bigvee_{1 \leq i \leq k} \tilde{\beta}_i = \mathbf{0})$$

$X \mapsto \tau$ with $\tau \notin \mathsf{Derived}$ is removed from μ_1 because $\kappa(\tau) \neq \mathbf{0}$ for any $\kappa \in \mathcal{A}_t$ since τ is neither $\mathbf{0}$ nor a derived type parameter. Let $\mathsf{C}_1' = \mathsf{C}_1 \wedge \bigwedge_{1 \leq i \leq k} \backslash\!|\tilde{\beta}_i, \mathbf{0}/0\backslash\!|$. Then the above formula is logically equivalent to the following formula.

(a) $\forall.(\mathsf{C}_1' \rightarrow \mathsf{C}_2)$

The formula $\forall.(\mathsf{C}_1 \wedge \mathsf{C}_2 \rightarrow \mu_1 \overset{\circ}{\sqsubseteq} \mu_2)$ is transformed to the following form by writing $\mu_1(X) \preceq \mu_2(X)$ for each variable X in $dom(\mu_2)$.

(b) $\forall.(\mathsf{C} \rightarrow \mathsf{S})$ where $\mathsf{C} = \mathsf{C}_1 \wedge \mathsf{C}_2$ and $\mathsf{S} = \bigwedge_{1 \leq i \leq k} \tau_i \preceq \tau_i'$.

A constrained variable poly-typing $\bar{\mu}_1 = (\mu_1, \mathsf{C}_1)$ always describes the empty set of substitutions if $\forall.(\mathsf{C}_1 \rightarrow \mu_1 = \text{⚲})$ is true. The formula $\forall.(\mathsf{C}_1 \rightarrow \mu_1 = \text{⚲})$ is a special case of $\forall.(\mathsf{C}_1 \wedge \neg\mathsf{C}_2 \rightarrow \mu_1 = \text{⚲})$ with $\forall.\neg\mathsf{C}_2$. Therefore, removing redundant constrained variable poly-typings reduces to deciding the truth values of the formulas of the forms (a) and (b). Solving a formula of the form (a) is to decide if one case condition C_2 is a logical consequence of another case condition C_1', and, solving a formula of the form (b) is to decide if a case condition C logically implies a subtyping constraint. Sections 7.1 and 7.2 will present decision procedures for these forms of formulas respectively.

Example 7.2 *Formula (f1) in example 7.1 is logically equivalent to*

$$\forall.(|(\beta, \epsilon), \mathbf{0}/0| \wedge \backslash\!|(\beta, \epsilon), \mathbf{0}/0\backslash\!| \rightarrow \mathit{false}) \qquad \textit{(f3)}$$

Formula (f2) in example 7.1 is logically equivalent to the conjunction of the following two formulae.

$$\forall.(|(\beta, \epsilon), \mathit{list}/1| \wedge \backslash\!|(\beta, 1), \mathbf{0}/0\backslash\!| \wedge \backslash\!|(\beta, \epsilon), \mathbf{0}/0\backslash\!| \rightarrow |(\beta, \epsilon), \mathit{list}/1|) \qquad \textit{(f4)}$$

$$\forall.(|(\beta, \epsilon), \mathit{list}/1| \rightarrow (\beta, 1) \preceq (\beta, 1) \wedge \mathit{list}(\mathbf{0}) \preceq \mathit{list}((\beta, 1)) \wedge (\beta, \epsilon) \preceq (\beta, \epsilon) \wedge (\beta, \epsilon) \preceq \mathit{list}((\beta, 1))) \qquad \textit{(f5)}$$

7.1 Entailment of one case condition by another

A primitive case condition in C is either of the form $|\tilde{\beta}, c/m|$ or of the form $\backslash\!|\tilde{\beta}, c/m\backslash\!|$. This enables us to separate primitive case conditions on one derived type parameter from those on another. $|\tilde{\beta}, c/m|$ restricts the principal constructor of $\tilde{\beta}$ to $\{c/m\}$ whilst $\backslash\!|\tilde{\beta}, c/m\backslash\!|$ restricts the principal constructor of $\tilde{\beta}$ to $\mathsf{Cons} \setminus \{c/m\}$. Each primitive case condition on $\tilde{\beta}$ restricts the principal constructor of $\tilde{\beta}$ to a subset of Cons. $\mathsf{C} \downarrow \tilde{\beta}$ is defined as the intersection of the subsets of Cons to which primitive case conditions in C restrict the principal constructor of $\tilde{\beta}$.

$$\mathit{true} \downarrow \tilde{\beta} \quad \stackrel{def}{=} \quad \mathsf{Cons}$$

$$false \downarrow \tilde{\beta} \stackrel{def}{=} \emptyset$$

$$(|\tilde{\beta}', c/m|) \downarrow \tilde{\beta} \stackrel{def}{=} \begin{cases} \{c/m\} & if\ \tilde{\beta}' \equiv \tilde{\beta} \\ \mathsf{Cons} & if\ \tilde{\beta}' \not\equiv \tilde{\beta} \end{cases}$$

$$(\backslash\!\!\backslash\tilde{\beta}', c/m\backslash\!\!\backslash) \downarrow \tilde{\beta} \stackrel{def}{=} \begin{cases} \mathsf{Cons} \setminus \{c/m\} & if\ \tilde{\beta}' \equiv \tilde{\beta} \\ \mathsf{Cons} & if\ \tilde{\beta}' \not\equiv \tilde{\beta} \end{cases}$$

$$(\mathsf{C}_1 \wedge \mathsf{C}_2) \downarrow \tilde{\beta} \stackrel{def}{=} (\mathsf{C}_1 \downarrow \tilde{\beta}) \cap (\mathsf{C}_2 \downarrow \tilde{\beta})$$

$\mathsf{C} \downarrow \tilde{\beta}$ is well defined for every derived type parameter including those not appearing in C. $\forall.\mathsf{C}$ iff $\mathsf{C} \downarrow \tilde{\beta} = \mathsf{Cons}$ for every $\tilde{\beta}$ and $\forall.\neg\mathsf{C}$ iff $\mathsf{C} \downarrow \tilde{\beta} = \emptyset$ for some $\tilde{\beta}$.

Now consider the truth value of $\forall.(\mathsf{C}_1' \rightarrow \mathsf{C}_2)$. If $\forall.\mathsf{C}_2$ then $\forall.(\mathsf{C}_1' \rightarrow \mathsf{C}_2)$ is true. If $\forall.\neg\mathsf{C}_2$ then $\forall.(\mathsf{C}_1' \rightarrow \mathsf{C}_2)$ is equivalent to $\neg\exists.\mathsf{C}_1'$ which is true if $\mathsf{C}_1' \downarrow \tilde{\beta} = \emptyset$ for some $\tilde{\beta}$. The case that neither $\forall.\mathsf{C}_2$ nor $\forall.\neg\mathsf{C}_2$ is dealt with according to the following lemma.

Lemma 7.3 *Let $\mathsf{C}_1', \mathsf{C}_2 \in \mathsf{Case}$, $\neg\forall.\mathsf{C}_2$, and $\neg\forall.\neg\mathsf{C}_2$. Then $\forall.(\mathsf{C}_1' \rightarrow \mathsf{C}_2)$ iff $\mathsf{C}_2 \downarrow \tilde{\beta} \supseteq \mathsf{C}_1' \downarrow \tilde{\beta}$ for every type parameter $\tilde{\beta}$ occurring in C_2.*

PROOF: Since $\neg\forall.\mathsf{C}_2$ and $\neg\forall.\neg\mathsf{C}_2$, C_2 contains at least one derived type parameter. Let $\|M\|$ denote the principle constructor of a mono-type $M \in \mathsf{Mono}$, $\mathsf{C} \in \mathsf{Case}$, and $\kappa \in \mathcal{A}_t$. $\kappa(\mathsf{C})$ iff $\|\kappa(\tilde{\beta})\| \in \mathsf{C} \downarrow \tilde{\beta}$ for any $\tilde{\beta}$ occurring in C. Therefore, $\forall.(\mathsf{C}_1' \rightarrow \mathsf{C}_2)$ iff $\mathsf{C}_2 \downarrow \tilde{\beta} \supseteq \mathsf{C}_1' \downarrow \tilde{\beta}$ for every type parameter $\tilde{\beta}$ occurring in C_2. ∎

Example 7.4 *Formula (f3) in example 7.2 holds since $\forall.\neg false$ holds and*

$$(|(\beta,\epsilon), \mathbf{0}/0| \wedge \backslash\!\!\backslash(\beta,\epsilon), \mathbf{0}/0\backslash\!\!\backslash) \downarrow (\beta,\epsilon) = \emptyset$$

This implies that $\bar{\nu}_1$ in example 7.1 is redundent since (f3) is logically equivalent to (f1) in example 7.1.

Formula (f4) in example 7.2 holds since

$$(|(\beta,\epsilon), list/1| \wedge \backslash\!\!\backslash(\beta,1), \mathbf{0}/0\backslash\!\!\backslash \wedge \backslash\!\!\backslash(\beta,\epsilon), \mathbf{0}/0\backslash\!\!\backslash) \downarrow (\beta,\epsilon) = |(\beta,\epsilon), list/1| \downarrow (\beta,\epsilon) = \{list/1\}$$

7.2 Entailment of a subtyping constraint by a case condition

The truth of $\forall.(\mathsf{C} \rightarrow \mathsf{S})$ is tested by a rewriting system in figure 6. A rewriting rule of the form $\mathsf{C} \rhd \mathsf{S} \leadsto \mathsf{S}'$ reads that S reduces to S' in the context C in one step. The rewriting rules are intended to allow rewriting of S into its equivalents in the context C in every possible way. In other words, it is intended that $\forall.(\mathsf{C} \rightarrow \mathsf{S})$ is true iff S can be rewritten into *true* in the context C.

The rules (a)-(c) remove those subtyping constraints that are always true. The rules (d)-(i) replace a primitive subtyping constraint by an equivalent conjunction of primitive subtyping constraints according to the definition of $\precsim$. The rule (e) removes a primitive subtyping constraint if its left hand side is $\mathbf{0}$ in the context C. The rule (f) removes a primitive subtyping constraint if its right hand side is $\mathbf{1}$ in the context C. In the rules (g)-(h), one side of the primitive subtyping constraint is a derived type parameter. The condition in the rule (g) ensures that any mono-type value of the derived type parameter on the right hand side is

$$
\begin{array}{ll}
(a) & \mathsf{C} \rhd \mathsf{S} \wedge \mathbf{0} \precsim \tau \rightsquigarrow \mathsf{S} \\
(b) & \mathsf{C} \rhd \mathsf{S} \wedge \tau \precsim \mathbf{1} \rightsquigarrow \mathsf{S} \\
(c) & \mathsf{C} \rhd \mathsf{S} \wedge \tilde{\beta} \precsim \tilde{\beta} \rightsquigarrow \mathsf{S} \\
(d) & \mathsf{C} \rhd \mathsf{S} \wedge c(\tau_1, \cdots, \tau_n) \precsim c(\tau_1', \cdots, \tau_n') \rightsquigarrow \mathsf{S} \wedge \bigwedge_{1 \leq i \leq n} \tau_i \precsim \tau_i' \\
(e) & \mathsf{C} \rhd \mathsf{S} \wedge \tilde{\beta} \precsim \tau \rightsquigarrow \mathsf{S} \ \mathit{if} \ \mathsf{C} \downarrow \tilde{\beta} = \{\mathbf{0}/0\} \\
(f) & \mathsf{C} \rhd \mathsf{S} \wedge \tau \precsim \tilde{\beta} \rightsquigarrow \mathsf{S} \ \mathit{if} \ \mathsf{C} \downarrow \tilde{\beta} = \{\mathbf{1}/0\} \\
(g) & \mathsf{C} \rhd \mathsf{S} \wedge c(\tau_1, \cdots, \tau_n) \precsim \tilde{\beta} \rightsquigarrow \mathsf{S} \wedge \bigwedge_{1 \leq i \leq n} \tau_i \precsim (\tilde{\beta}, i) \qquad \mathit{if} \ \mathsf{C} \downarrow \tilde{\beta} \setminus \{\mathbf{1}/0\} = \{c/n\} \\
(h) & \mathsf{C} \rhd \mathsf{S} \wedge \tilde{\beta} \precsim c(\tau_1, \cdots, \tau_n) \rightsquigarrow \mathsf{S} \wedge \bigwedge_{1 \leq i \leq n} (\tilde{\beta}, i) \precsim \tau_i \qquad \mathit{if} \ \mathsf{C} \downarrow \tilde{\beta} \setminus \{\mathbf{0}/0\} = \{c/n\} \\
(i) & \mathsf{C} \rhd \mathsf{S} \wedge \tilde{\beta} \precsim \tilde{\beta}' \rightsquigarrow \mathsf{S} \wedge \bigwedge_{1 \leq i \leq n} (\tilde{\beta}, i) \precsim (\tilde{\beta}', i) \\
& \qquad \mathit{if} \ \tilde{\beta} \not\equiv \tilde{\beta}' \wedge \mathsf{C} \downarrow \tilde{\beta} \setminus \{\mathbf{0}/0\} = \mathsf{C} \downarrow \tilde{\beta}' \setminus \{\mathbf{1}/0\} = \{c/n\} \ \mathit{for\ some} \ c/n \in \mathsf{Func}
\end{array}
$$

Figure 6: The rewriting system for test $\forall.(\mathsf{C} \rightarrow \mathsf{S})$

either **1** or has the same principal constructor as the type on the left hand side. Similarly, the condition rule (h) ensures that any mono-type value of the derived type parameter on the left hand side is either **0** or has the same principal constructor as the type on the right hand side. In the rule (i), two sides of the primitive subtyping constraint are different derived type parameters, its condition makes sure that either the mono-type value of the derived type parameter on the left hand side is **0** or the mono-type value of the derived type parameter on the right hand side is **1** or they have the the principal constructor.

Let $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathsf{S}'$ denote that S reduces to S' in context C in multiple steps. Formally, $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathsf{S}'$ iff $\mathsf{S}' \equiv \mathsf{S}$ or there exists S'' such that $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathsf{S}''$ and $\mathsf{C} \rhd \mathsf{S}'' \rightsquigarrow \mathsf{S}'$.

Lemma 7.5 $\forall.(\mathsf{C} \rightarrow \mathsf{S})$ *iff* $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathit{true}$.

PROOF: The lemma holds vacuously if $\mathsf{S} \equiv \mathit{true}$. Let $\mathsf{S} \not\equiv \mathit{true}$. By examining the rewriting rules (a)-(i), we have that if $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow \mathsf{S}'$ then $\forall.(\mathsf{C} \rightarrow (\mathsf{S} \leftrightarrow \mathsf{S}'))$.

The if part, that is, if $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathit{true}$ then $\forall.(\mathsf{C} \rightarrow \mathsf{S})$. $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathit{true}$ implies that $\forall.(\mathsf{C} \rightarrow (\mathsf{S} \leftrightarrow \mathit{true}))$. So, if $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathit{true}$ then $\forall.(\mathsf{C} \rightarrow \mathsf{S})$.

The only-if part, that is, if $\forall.(\mathsf{C} \rightarrow \mathsf{S})$ then $\mathsf{C} \rhd \mathsf{S} \rightsquigarrow^* \mathit{true}$. Let the size of a type, a primitive subtyping constraint or a subtyping constraint be defined as follows.

$$
\begin{array}{rcl}
\mathit{size}(\tilde{\beta}) & \stackrel{def}{=} & 0 \\
\mathit{size}(c(\tau_1, \cdots, \tau_n)) & \stackrel{def}{=} & 1 + \mathit{size}(\tau_1) + \cdots + \mathit{size}(\tau_n)
\end{array}
$$

$$size(\tau_1 \precsim \tau_2) \stackrel{def}{=} size(\tau_1) + size(\tau_2)$$
$$size(\mathsf{S}_1 \wedge \mathsf{S}_2) \stackrel{def}{=} size(\mathsf{S}_1) + size(\mathsf{S}_2)$$

Let $\forall.(\mathsf{C} \rightarrow \mathsf{S})$. $\mathsf{C} \rhd \mathsf{S} \leadsto^* true$ is proved by induction on the size of S.

Basis. $size(\mathsf{S}) = 0$ implies that each primitive subtyping constraint in S is a primitive subtyping constraint between two derived type parameters. Without loss of generality, assume the two type parameters are syntactically different since the rule (c) can be applied to remove primitive subtyping constraints of the form $\tilde{\beta} \precsim \tilde{\beta}$. Since $\forall.(\mathsf{C} \rightarrow \tilde{\beta} \precsim \tilde{\beta}')$, either (1) C implies $\tilde{\beta} = \mathbf{0}$ or (2) C implies $\tilde{\beta}' = \mathbf{1}$ or (3) $\mathsf{C} \downarrow \tilde{\beta} \setminus \{\mathbf{0}/0\} = \mathsf{C} \downarrow \tilde{\beta}' \setminus \{\mathbf{1}/0\} = \{c/n\}$ and C implies $\bigwedge_{1 \leq i \leq n} (\tilde{\beta}, i) \precsim (\tilde{\beta}', i)$. In the case (1), $\mathsf{C} \rhd \mathsf{S} \leadsto \mathsf{S}'$ by the rule (e). In the case (2), $\mathsf{C} \rhd \mathsf{S} \leadsto \mathsf{S}'$ by the rule (f). In the case (3), $\mathsf{C} \rhd \mathsf{S} \leadsto \mathsf{S}' \wedge \bigwedge_{1 \leq i \leq n} (\tilde{\beta}, i) \precsim (\tilde{\beta}', i)$ by the rule (i). Let S'' be S' in the cases (1) and (2) and be $\mathsf{S}' \wedge \bigwedge_{1 \leq i \leq n} (\tilde{\beta}, i) \precsim (\tilde{\beta}', i)$ in the case (3). We have $size(\mathsf{S}'') = 0$, $\forall.(\mathsf{C} \rightarrow \mathsf{S}'')$ and S'' be a conjunction of primitive subtyping constraints between two different derived type parameters. In other words, one of the rules (e), (f) and (i) is applicable to S'' if $\mathsf{S}'' \not\equiv true$. It now suffices to prove that repeated applications of these rules will reduces the number of primitive subtyping constraints into 0. The rules (e) and (f) decrease the number of primitive subtyping constraints. The rule (i) also decrease the number of primitive subtyping constraints when n in its condition is zero. The rule (i) keeps or increases the number of primitive subtyping constraints when $n \neq 0$. Note that the rule (i) introduces primitive constraints $(\tilde{\beta}, i) \precsim (\tilde{\beta}', i)$ when $n \neq 0$. $\forall.(\mathsf{C} \rightarrow ((\tilde{\beta}, i) \precsim (\tilde{\beta}', i)))$ because $\forall.(\mathsf{C} \rightarrow (\tilde{\beta} \precsim \tilde{\beta}'))$. $\forall.(\mathsf{C} \rightarrow ((\tilde{\beta}, i) \precsim (\tilde{\beta}', i)))$ implies that $(\tilde{\beta}, i)$ and $(\tilde{\beta}', i)$ appears in C. Since C contains only a finite number of derived type parameters and is not changed by rewriting, the rule (i) with $n \neq 0$ can only be applied for a number of finite times. Therefore, repeated applications of these rules will reduces the number of primitive subtyping constraints into 0.

Induction. $size(\mathsf{S}) \neq 0$ implies that at least one side of a primitive subtyping constraint in S is not a derived type parameter. At least one of the rules (a)-(h) can be applied to rewrite S since $\forall.(\mathsf{C} \rightarrow \mathsf{S})$. Let S' be a subtyping constraint and $\mathsf{C} \rhd \mathsf{S} \leadsto \mathsf{S}'$ due to the application of one of the rules (a)-(h). We have $size(\mathsf{S}') < size(\mathsf{S})$ and $\forall.(\mathsf{C} \rightarrow \mathsf{S}')$. By the induction hypothesis, we have $\mathsf{C} \rhd \mathsf{S}' \leadsto^* true$. So, $\mathsf{C} \rhd \mathsf{S} \leadsto^* true$. ∎

Example 7.6 *Consider formula (f5) in example 7.2. We have*

$$
\begin{array}{rl}
|(\beta,\epsilon), list/1| \rhd & (\beta,1)\precapprox(\beta,1) \wedge list(\mathbf{0})\precapprox list((\beta,1)) \wedge (\beta,\epsilon)\precapprox(\beta,\epsilon) \wedge (\beta,\epsilon)\precapprox list((\beta,1)) \\
& \leadsto (by\ rule\ (c)) \\
& list(\mathbf{0})\precapprox list((\beta,1)) \wedge (\beta,\epsilon)\precapprox(\beta,\epsilon) \wedge (\beta,\epsilon)\precapprox list((\beta,1)) \\
& \leadsto (by\ rule\ (d)) \\
& \mathbf{0}\precapprox(\beta,1) \wedge (\beta,\epsilon)\precapprox(\beta,\epsilon) \wedge (\beta,\epsilon)\precapprox list((\beta,1)) \\
& \leadsto (by\ rule\ (a)) \\
& (\beta,\epsilon)\precapprox(\beta,\epsilon) \wedge (\beta,\epsilon)\precapprox list((\beta,1)) \\
& \leadsto (by\ rule\ (c)) \\
& (\beta,\epsilon)\precapprox list((\beta,1)) \\
& \leadsto (by\ rule\ (h)) \\
& (\beta,1)\precapprox(\beta,1) \\
& \leadsto (by\ rule\ (c)) \\
& true
\end{array}
$$

Therefore, (f5) holds. This, together with example 7.4 which shows (f4) in example 7.2 holds, implies $\bar{\nu}_2 \trianglelefteq^\flat \bar{\nu}_3$ *in example 7.1.*

8 IMPLEMENTATION & EXAMPLES

We have implemented the abstract interpretation framework and the polymorphic type analysis described above in SWI-Prolog on a Sun SPARC IV station running SunOS operating system. The abstract interpretation framework is implemented using O'Keefe's least fixed-point algorithm [41]. Both the abstract interpretation framework and the polymorphic type analysis are implemented as meta-interpreters using ground representations for program variables and derived type parameters. The following examples present results of the polymorphic type analysis on some Prolog programs. **1** is written as top, **0** as bot, and β as beta. (β, ϵ) is also written as beta for β and (β, ϵ) have the same value for any assignement of mono-types to original type parameters. $X \mapsto T$ is written as X/T. A case condition C is written as $[(\tilde{\beta}_1, \mathsf{C} \downarrow \tilde{\beta}_1), \cdots, (\tilde{\beta}_n, \mathsf{C} \downarrow \tilde{\beta}_n)]$ where $\tilde{\beta}_1, \cdots, \tilde{\beta}_n$ are derived type parameters occuring in C. As a result, [ ] is logically equivalent to *true*. The type definitions for the function symbols occurring in these programs are those in example 4.3.

Example 8.1 *The following is an improved quicksort program from [8](p. 157) with polymorphic type analysis result. The result indicates that each variable at each program point has a value of an expected type when* iqsort *is called with its first argument being of type list*(β) *for any* $\beta \in$ Mono.

```
:-      %[([Xs/list(beta)], [])]
    iqsort(Xs, Ys)
        %[([Xs/list(beta),Ys/list(beta)], [])]

iqsort_aux([X|T], S, R) :-
        %[([X/beta,T/list(beta),S/list(beta)], [])]
```

```
    partition(T, X, B, A),
        %[([X/beta,T/list(beta),S/list(beta),B/list(beta),
        %                                A/list(beta)],[])]
    iqsort_aux(A, S, Sn),
        %[([X/beta,T/list(beta),S/list(beta),B/list(beta),
        %                       A/list(beta),Sn/list(beta)],[])]
    iqsort_aux(B, [X|Sn], R).
        %[([X/beta,T/list(beta),S/list(beta),R/list(beta),
        %          B/list(beta),A/list(beta),Sn/list(beta)], [])]
iqsort_aux([], S, S).
        %[([S/list(beta)], [])]
iqsort(L, R) :-
        %[([L/list(beta)], [])]
    iqsort_aux(L, [], R).
        %[([L/list(beta),R/list(beta)], [])]
partition([X|T], P, [X|B], A) :-
        %[([X/beta,T/list(beta),P/beta], [])]
    X =< P,
        %[([X/beta,T/list(beta),P/beta], [])]
    partition(T, P, B, A).
        %[([X/beta,T/list(beta),P/beta,B/list(beta),A/list(beta)], [])]
partition([X|T], P, B, [X|A]) :-
        %[([X/beta,T/list(beta),P/beta], [])]
    X > P,
        %[([X/beta,T/list(beta),P/beta], [])]
    partition(T, P, B, A).
        %[([X/beta,T/list(beta),P/beta,B/list(beta),A/list(beta)], [])].
partition([], _, [], []).
        %[([_/beta], [])]
```

Example 8.2 *The following is a buggy naive reverse program together with polymorphic type analysis result. The second literal in the body of the second clause for* rev/2 *is buggy and should be* append(T1, [H], L)*. Called with its first argument being of type list(β) for any $\beta \in$* Mono*,* rev/2 *succeeds with its second argument being either an empty list or a datum of type β. This is accurately inferred by the polymorphic type analysis. The polymorphic type analysis also infers that whenever* rev/2 *is called with its first argument being of type list(β) for any $\beta \in$* Mono*,* append/3 *is called with its second argument being of type β. This indicates that the program behaves unexpectedly in terms of type and that the polymorphic type analysis provides useful information for program debugging.*

```
:-      %[([X/list(beta)], [])]
    rev(X, Y).
        %[([X/list(bot),Y/list(bot)], []),
        % ([X/list(beta),Y/beta], [])     ]

append([], L, L).
        %[([L/beta], [])]
append([H|T], L, [H|TL]) :-
        %[([L/beta], [(beta, [top/0])]),
        % ([H/(beta,1),T/list((beta,1)),L/beta], [(beta, [list/1])]) ]
    append(T, L, TL).
```

```
        %[([H/(beta,1),T/list((beta,1)),L/beta,TL/list((beta,1))],
        %                                         [(beta, [list/1])]),
        % ([T/list(top),L/beta], [(beta, [top/0])])                   ]
rev([], []).
        %[([], [])]
rev([H|T], L) :-
       %[([H/beta,T/list(beta)], [])]
    rev(T, T1),
       %[([H/beta,T/list(bot),T1/list(bot)], []),
        % ([H/beta,T/list(beta),T1/beta], [])     ]
    append(T1, H, L).  %>> append(T1, [H], L).
        %[([H/beta,T/list(bot),L/beta,T1/list(bot)], []),
        % ([H/beta,T/list(beta),T1/list(top)], [(beta, [top/0])]),
        % ([H/beta,T/list(beta),L/list((beta,1)),T1/list((beta,1))],
        %                                         [(beta, [list/1])])   ]
```

Example 8.3 *The following is the pure Prolog program intended for inserting a natural number into an ordered list of natural numbers. The result of the polymorphic type analysis of the program indicates that, when* insert/3 *is called with its first argument being of type β and its second argument being of type list(β), it will fail if $\beta \neq$ nat and $\beta \neq \mathbf{1}$ and the second argument is not an empty list. The result also indicates that if the second argument is an empty list then the first argument can be of any type $\beta \in$* Mono. *This is correct since* insert([s(0)], [], [[s(0)]]) *is in the success set of the program.*

```
:-      %[([X/beta,Y/list(beta)],[])]
    insert(X,Y,Z).
        %[([X/beta,Y/list(bot),Z/list(beta)],[]),
        % ([X/nat,Y/list(top),Z/list(top)],[(beta,[top/0])]),
        % ([X/nat,Y/list(nat),Z/list(nat)],[(beta,[nat/0])]),
        % ([X/beta,Y/list(nat),Z/list(top)],[(beta,[top/0])])]

insert(X,[],[X]).
        %[([X/beta],[])]
insert(X,[Y|L],[X,Y|L]) :-
        %[([X/beta,Y/beta,L/list(beta)],[])]
    leq(X,Y).
        %[([X/nat,Y/nat,L/list(beta)],[(beta,[nat/0])]),
        % ([X/nat,L/list(beta)],[(beta,[top/0])])]
insert(X,[Y|L],[Y|L1]) :-
        %[([X/beta,Y/beta,L/list(beta)],[])],
    gt(X,Y),
        %[([X/nat,Y/nat,L/list(beta)],[(beta,[nat/0])]),
        % ([Y/nat,L/list(beta)],[(beta,[top/0])])],
    insert(X,L,L1).
        %[([X/beta,Y/nat,L/list(nat),L1/list(top)],[(beta,[top/0])]),
        % ([X/nat,Y/nat,L/list(nat),L1/list(nat)],[(beta,[nat/0])]),
        % ([X/nat,Y/nat,L/list(beta),L1/list(top)],[(beta,[top/0])])].
gt(s(X),0).
        %[([],[(beta,[top/0])]),([X/nat],[(beta,[nat/0])])]
```

```
gt(s(Y),s(X)) :-
        %[([],[(beta,[top/0])]),([Y/nat,X/nat],[(beta,[nat/0])])],
    gt(Y,X).
        %[([Y/nat,X/nat],[(beta,[nat/0])]),([X/nat],[(beta,[top/0])])]
leq(0,0).
        %[([],[(beta,[top/0])]),([],[(beta,[nat/0])])].
leq(0,s(X)).
        %[([],[(beta,[top/0])]),([X/nat],[(beta,[nat/0])])]
leq(s(X),s(Y)) :-
        %[([],[(beta,[top/0])]),([X/nat,Y/nat],[(beta,[nat/0])])],
    leq(X,Y).
        %[([X/nat,Y/nat],[(beta,[nat/0])]),([X/nat],[(beta,[top/0])])]
```

Example 8.4 *The following analysis result illustrates the limitation of the polymorphic type analysis. The type definitions are those in example 4.3. The analysis is unable to infer accurately the following property of the* append *program: if* append *is called with its first argument being a list of elements of type α and its second argument being a list of elements of type β, its third argument will be a list whose elements are either of type α or of type β when it succeeds. This is a consequence of using terms constructible from* Para *and* Cons *as type expressions. Codish and Demoen's work [10] is the only one in the literature that can infer the above property. Type expressions they use are terms constructible from* Para *and* $\mathsf{Cons} \cup \{+\}$ *with $+$ being interpreted as an upper bound of mono-types. The code that interprets $+$ must be an input to their type analysis system.*

```
:-      %[([X/list(alpha),Y/list(beta)],[])]
    append(X,Y,Z)
        %[([X/list(bot),Y/list(beta),Z/list(beta)],[]),
        % ([X/list(alpha),Y/list(beta),Z/list(alpha)],[(beta,[bot/0])]),
        % ([X/list(alpha),Y/list(beta),Z/list(top)],
        %     [(beta,[nat/0,list/1,top/0]),(alpha,[nat/0,list/1,top/0])])].

append([],L,L) :-
        %[([L/list(beta)],[])].
append([X|L1],L2,[X|L3]) :-
        %[([X/alpha,L1/list(alpha),L2/list(beta)],[])],
    append(L1,L2,L3),
        %[([X/alpha,L1/list(bot),L2/list(beta),L3/list(beta)],[]),
        % ([X/alpha,L1/list(alpha),L2/list(beta),L3/list(alpha)],[(beta,[bot/0])]),
        % ([X/alpha,L1/list(alpha),L2/list(beta),L3/list(top)],
        %     [(beta,[nat/0,list/1,top/0]),(alpha,[nat/0,list/1,top/0])])].
```

The implementation of the polymorphic type analysis has been run on Prolog programs drawn from the literature. The time performance on these programs is demonstrated in figure 7. The function symbols whose type definitions are given in example 4.3 are given the same type definitions. For the "tree sort" and the "Heapify Binary Trees" programs, the following type definitions apply.

$$void : tree(\beta)$$

$$tr(X, L, R) : tree(\beta) \quad \leftarrow \quad X : \beta, L : tree(\beta), R : tree(\beta)$$

For the "Quicksort using Difference Lists" program, the difference lists are typed according to the following type definition.

$$Xs \backslash Ys : dlist(\beta) \leftarrow Xs : list(\beta), Ys : list(\beta)$$

Finally, for the "Dictionary Lookup in a Binary Tree" program, the type of a dictionary mapping a key of type α into a value of type β are defined as follows.

$$\begin{aligned} void &: dict(\alpha, \beta) \\ dict(K, V, L, R) : dict(\alpha, \beta) &\quad \leftarrow \quad K : \alpha, V : \beta, L : dict(\alpha, \beta), R : dict(\alpha, \beta) \end{aligned}$$

Figure 7 indicates that the analyzer spent an average of 0.0668 seconds to process one program line. There are two main contributors to the low speed of the analyzer. Firstly, both the abstract interpretation framework and the polymorphic type analysis are implemented as meta-interpreters in a public domain Prolog system. Moreover, we use ground representations for program variables and type parameters. Using ground representations enables us to make the prototype implementation in a short time and to avoid possible difficulties that may arise due to improper use of meta-level and object-level variables. However, it also prevents us from taking advantages of built-in unification and forces us to code unification in Prolog. We believe that both the use of meta-programming and that of ground representations contribute significantly to the inefficiency of the analyzer. Secondly, the generality of polymorphic type analysis makes it necessary to do case analysis of type structures, which leads to representing an abstract substitution as a set of constrained variable poly-typing. That means that both abstract unification and least upper bound operations are more costly for polymorphic type analysis than for monomorphic type analysis. The abstract unification operator needs to process more items and to do case analysis of type structures and the least upper bound operator needs to do redundancy checking. However, as shown by examples, the polymorphic type analysis also gives preciser analysis result than the monomorphic type analysis because the abstract domain for the polymorphic type analysis is more expressive than that for the monomorphic type analysis.

9 CONCLUSION

We have formalised a polymorphic analysis as a representation of many monomorphic analyses and developed a polymorphic type analysis by lifting a monomorphic type analysis. The notion of polymorphic abstract interpretation provides a basis for understanding and designing polymorphic analyses. It is language-independent and applicable to other analyses. The benefit of a polymorphic analysis derives from its generality. The result of a polymorphic analysis can be instantiated by assigning parameters values from an underlying domain. This can be done for as many times as necessary. The polymorphic type analysis is designed by devising a domain of abstract substitutions and an abstract unification operator on the domain of abstract substitutions. An abstract substitution is a set of constrained variable

Name	*Source*	*Lines*	*Goal*	*Input*	*Time* (sec)
Connectivity	$p.220, [44]$	26	$connected(X, Y)$	$[([X \mapsto (\beta, \epsilon)], true)]$	1.33
Merge	$p.158, [44]$	12	$merge(\mathrm{Xs}, \mathrm{Ys}, \mathrm{Zs})$	$[([\mathrm{Xs} \mapsto list((\beta, \epsilon)), \mathrm{Ys} \mapsto list((\beta, \epsilon))], true)]$	1.48
Buggy Reverse	$Ex.$ 8.2	8	$rev(X, Y)$	$[([X \mapsto list((\beta, \epsilon))], true)]$	0.77
Buggy Quicksort	$p.3, [30]$	26	$qs(Li, Lo)$	$[([Li \mapsto list((\beta, \epsilon))], true)]$	3.73
Improved Quicksort	$p.157, [8]$	15	$iqsort(\mathrm{Xs}, \mathrm{Ys})$	$[([\mathrm{Xs} \mapsto list((\beta, \epsilon))], true)]$	1.90
Tree Sort	[15]	23	$treesort(\mathrm{Xs}, \mathrm{Ys})$	$[([\mathrm{Xs} \mapsto list((\beta, \epsilon))], true)]$	1.97
List Difference	$p.21, [30]$	10	$diff(X, Y, Z)$	$[([Y \mapsto list((\beta, \epsilon)), Z \mapsto list((\beta, \epsilon))], true)]$	0.38
Insertion	$Ex.$ 8.3	14	$insert(X, Y, Z)$	$[([X \mapsto (\beta, \epsilon), Y \mapsto list((\beta, \epsilon))], true)]$	1.47
Interchange Sort	$p.162, [44]$	16	$sort(\mathrm{Xs}, \mathrm{Ys})$	$[([\mathrm{Xs} \mapsto list((\beta, \epsilon))], true)]$	1.13
Quicksort with differece lists	$p.244, [44]$	15	$quicksort(\mathrm{Xs}, \mathrm{Ys})$	$[([\mathrm{Xs} \mapsto list((\beta, \epsilon))], true)]$	1.52
Dictionary Lookup in binary trees	$p.250, [44]$	9	$lookup(K, D, V)$	$[([K \mapsto (\beta, \epsilon), D \mapsto dict((\beta, \epsilon), (\alpha, \epsilon))], true)]$	0.68
Permutation Sort	$p.55, [44]$	16	$sort(\mathrm{Xs}, \mathrm{Ys})$	$[([\mathrm{Xs} \mapsto list((\beta, \epsilon))], true)]$	0.63
Heapify binary trees	$p.62, [44]$	20	$heapify(T, H)$	$[([T \mapsto tree((\beta, \epsilon))], true)]$	2.88
Exponentiation by multiplication	$p.39, [44]$	17	$exp(N, X, Y)$	$[([N \mapsto (\beta, \epsilon), X \mapsto (\beta, \epsilon)], true)]$	1.63
Factorials	$p.39, [44]$	16	$factorial(N, F)$	$[([N \mapsto (\beta, \epsilon)], true)]$	0.72

Figure 7: Performance of the Polymorphic Type Analysis on Example Programs

poly-typings. Under an assignement of mono-types to original type parameters, the abstract substitution can be interpreted as the disjunction of the instances of its members under the assignment. This leads to a precise expression of type information in presence of original type parameters which serves as place holders for type information provided after analysis.

In the logic programming community, research into type analysis by means of abstract interpretation has been mainly focused on monomorphic type analyses except the work of Barbuti and Giacobazzi [2] and the work of Codish and Demoen [10].

Yardeni and Shapiro [49] infer a type for a program by computing the least fixed-point of an abstract immediate consequence operator that approximates the normal immediate consequence [46]. A type is a recursively enumerable set of ground atoms that is tuple-distributive. A set of terms is tuple-distributive if it is equal to the set of terms constructed recursively by permuting each argument position among all terms that have the same function symbol [37]. Heintze and Jaffar [19, 20] show that Yardeni and Shapiro's approach is equivalent to Mishra's [37] in the sense that a type in one approach has its counterpart in the other. Mishra's approach that is also adopted in [1, 16, 19, 20, 37, 42, 48, 50] infers descriptive types by finding a distinguished model for a set of formulae that captures set-theoretic relationships between the atoms appearing in the program. Both Yardeni and Shapiro's approach and Mishra's infer monomorphic type information and disregard variable dependencies that are essential to perform precise analyses. In addition, it is difficult to integrate these type analyses with other analyses.

Bruynooghe, Janssens, Callebaut, and Demoen [6] and Janssens and Bruynooghe [24] use type graphs to denote sets of terms. [6] uses restricted types to denote sets of ground terms. A restricted type is a rooted graph whose nodes are either labelled with a type and hence called a type node or labelled with a function symbol and thus called a functor node. The root of the graph is a type node. The successors of a type node are functor nodes labelled with different function symbols. A functor node labelled with a function symbol f/n has n ordered type nodes as its successors with each corresponding to an argument of f/n. There is a path from the root to any other node in the graph and the number of occurrences of the same function symbol labelling the functor nodes in any acyclic path in the graph is bounded by a constant. The denotation of a type node is the union of those of its successors. The denotation of a functor node labelled with f/n is the set of terms $f(t_1, \cdots, t_n)$ with t_i being in the denotation of the i-th successor of the functor node. Since restricted types can only denote sets of ground terms, rigid types and integrated types are introduced to denote sets of terms containing non-ground terms [24]. A rigid type differs from a restricted type in that a type node may be labelled with a primitive type that denotes a fixed set of terms. For instance, a type node may be labelled with *int* which denotes the set of integers. There is a special primitive type **1** that denotes Term. All other primitive types denote sets of ground terms. An integrated type is an enrichment of a rigid type in that an extra primitive type var which denotes Vars is allowed. The primitive type var captures freeness information. If a term is of type var then it is a variable. Since integrated types are not downwards closed, type analysis with integrated types is accomplished together with sharing and aliasing in [24] to ensure the safeness of analysis. Van Hentenryck, Cortesi and Le Charlier [47] also use type graphs to denote sets of terms in type analysis. They integrate rigid types into the generic abstract domain $\mathsf{Pat}(\Re)$ [12]. $\mathsf{Pat}(\Re)$ enhances the parameter abstract domain $\Re$ with structure patterns and equality constraints between subterms. By replacing $\Re$ with the domain of rigid types, [47] is able to capture both non-recursive type information in terms of structure patterns and recursive type information in terms of rigid types. Both the work in [6, 24] and that in [47] integrate type analysis with other analyses such as sharing and aliasing analyses. Since our polymorphic type analysis is an application of an abstract interpretation framework, it can be easily integrated with other analyses as in [6, 24] and [47]. The major difference between our work and [6, 24] and [47] is that our polymorphic type analysis infers polymorphic type information while their type analyses do not. In addition, our abstract domain captures type information more precisely than [6, 24] and [47]. An abstract substitution in the polymorphic type analysis is a set of variable poly-typings. For a fixed assignment of mono-types to original type parameters, the abstract substitution becomes a set of variable mono-typings which is a disjunctive description of a set of substitutions. In [6, 24], the type information part of an abstract substitution corresponds to a variable mono-typing with rigid/integrated types playing the role of mono-types. The type graphs assigned to the same variables are merged when a (least) upper bound of two abstract substitutions is computed, which causes loss of precision. In [47], type information in a set of substitutions is described by assigning each variable of interest a term-like structure whose leaf nodes are rigid types. The structure patterns and equality constraints between subterms may alleviate to some extent the loss of precision due to the merging of rigid types, but rigid types are also merged when a (least) upper bound of two abstract substitutions is computed. In our polymorphic type analysis, variable poly-typings in two abstract substitutions are simply added together with redundant variable poly-typings removed when the least upper bound of the two abstract substitutions is computed.

Horiuchi and Kanamori [22, 27] and Kanamori and Kawamura [28] use prescriptive types to denote sets of terms by giving each function symbols a type definition in the same way as we have done in the paper. Though type definitions are polymorphic, their type analysis only infers monomorphic type information. In addition, their abstract domain for monomorphic type analysis captures type information less precisely than our abstract domain for polymorphic type analysis for the same reason as we have given for type analysis in [6, 24] and that in [47].

Both the work of Barbuti and Giacobazzi [2] and that of Codish and Demoen [10] infer a polymorphic type approximation of the success set of the program. The approximation inferred by Barbuti and Giacobazzi's system is not complete in the sense that only well-typed atoms refutable by well-typed SLD-refutations are considered. They argue that "ill-typed goals do not correspond to the philosophy of the program". Ignoring ill-typed goals enables them to perform much of work on abstract unification by executing the type program composed of type clauses as a pure Prolog program. However, this corresponds to imposing a type system on top of an untyped language, which is still a controversial thing to do. Codish and Demoen combine abstract compilation and multiple incarnations of the abstract domain Prop to do polymorphic type analysis. The incarnations of Prop define meanings of types and capture interactions between types. We believe that more work is needed on automated derivation of these incarnations from type definitions since these incarnations seem to be inputs to their analyser according to [10].

The polymorphic type analysis that we have presented is based on the concept of polymorphic abstract interpretations and is amicable to adaptations to both top-down and bottom-up abstract interpretation frameworks. It is also fully automated in that its only inputs are the program to be analysed and type definitions for the function symbols in the program.

References

[1] H. Azzoune. Type inference in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the ninth International Conference on Automated Deduction*, pages 258–277, Argonne, Illinois, USA, May 23-26 1988. Springer-Verlag. Springer-Verlag Lecture Notes in Computer Science 310.

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

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Meo. A compositional semantics for logic programs. *Journal of Theoretical Computer Science*, 122(1-2):3–47, 1994.

[4] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the first International Logic Programming Conference*, pages 129–133, Marseille, France, 1982.

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

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

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM computing surveys*, 17(4):471–522, 1985.

[8] W.F. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[9] M. Codish, D. Dams, and Yardeni E. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 79–93, Paris, France, 1991. The MIT Press.

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

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

[12] A. Cortesi, B. Le Charlier and P. van Hentenryck. Combinations of Abstract Domains for Logic Programming. *The Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239, Portland, Oregon, January 17-21, 1994.

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

[14] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[15] M. A. Covington, D. Nute, and A. Vellino. *PROLOG PROGRAMMING IN DEPTH*. Scott, Foresman & Co., 1988.

[16] P.W. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, Cambridge,Massachusetts, 1992.

[17] M. Falaschi, G. Levi, and C. Palamidessi. Declarative modelling of the operational behavior of logic programs. *Theoretical Computer Science*, 69:289–318, 1989.

[18] T.W. Früehwirth. Using meta-interpreters for polymorphic type checking. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 339–351, Leuven, Belgium, April 4-6 1990.

[19] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *The seventh Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 17-19 1990. The ACM Press.

[20] N. Heintze and J. Jaffar. Semantic types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, Cambridge,Massachusetts, 1992.

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

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

[23] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(1, 2, 3 and 4):291–314, 1992.

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

[25] N.D. Jones and H. Søndergarrd. A semantics-based framework for abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 123–142. Ellis Horwood Limited, 1987.

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

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

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

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

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

[31] L. Lu. Type analysis of logic programs in the presence of type definitions. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95)*, pages 241–252, La Jolla, California, June 21-23 1995. The ACM Press.

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

[33] K. Marriott and H. Søndergaard. Bottom-up dataflow analysis of normal logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):181–204, 1992.

[34] K. Marriott. Frameworks for abstract interpretation. *Acta Informatica*, 30(2):103–129, 1993.

[35] K. Marriott, H. Søndergaard and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[36] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 181–198. Ellis Horwood Limited, 1987.

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

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

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

[40] U. Nilsson. Towards a framework for the abstract interpretation of loic programs. In P. Deransart, B. Lorho, and J. Małuszynski, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82, Orléans, France, 1988. Springer-Verlag.

[41] R. A. O'Keefe. Finite fixed-point problems. In J.-L. Lassez, editor, *Proceedings of the fourth International Conference on Logic programming*, volume 2, pages 729–743. The MIT Press, 1987.

[42] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Journal of Theoretical Computer Science*, 34(1):227–240, 1984.

[43] H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. In *Proceedings of the European symposium on programming*, pages 324–338. Springer-Verlag, 1986.

[44] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[45] J. Tiuryn. Type inference problems: A survey. In B. Roven, editor, *Proceedings of the Fifteenth International Symposium on Mathematical Foundations of Computer Science*, pages 105–120, Berlin,Heidelberg, 1990. Springer-Verlag.

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

[47] P. van Hentenryck, A. Cortesi and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[48] J. Xu and D.S. Warren. A type inference system for Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 604–619. The MIT Press, 1988.

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

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