# Backward Type Inference Generalises Type Checking

Lunjin Lu[1] and Andy King[2]

[1] Oakland University, MI 48309, USA.
[2] University of Kent at Canterbury, CT2 7NF, UK.

**Abstract.** This paper presents a backward type analysis for logic programs. Given type signatures for a collection of selected predicates such as builtin or library predicates, the analysis infers type signatures for other predicates such that the execution of any query satisfying the inferred type signatures will not violate the type signatures for the selected predicates. Thus, the backward type analysis generalises type checking in which the programmer manually specifies type signatures for all predicates that are checked for consistency by a type checker.

## 1 Introduction

This paper focuses on the inference of type signatures for predicates in a logic program that ensure the execution of the program with a query satisfying the inferred type signatures will be free from type errors. This generalises type checking in which the programmer declares type signatures for all predicates in the program and a type checker verifies if the program is well-typed with respect to these type signatures, that is, these type signatures are consistent with the operational semantics of the program. For instance, for the quicksort program in Prolog, a type checker would require the programmer to declare type signatures for $quicksort/2$, $partition/4$ and $append/3$ and then check if the program is well-typed with respect to these and the type signatures for builtin predicates $> /2$ and $=< /2$ stipulated in the Prolog language manual. In contrast, type signature inference will infer that if $quicksort/2$ is called with a list of numbers as the first argument then the execution of the program will not violate the type signatures of $> /2$ and $=< /2$; the programmer need not declare types for $quicksort/2, partition/4$ nor $append/3$.

Type analyses traditionally propagate type information in the direction of program execution and cannot infer type signatures (see section 7 for a short survey). One notable exception is the backward analysis framework of [15] that can, in principle, be instantiated with a type domain. However, the framework propagates information backwards using the (intuitionistic) logical implication operator and this requires domains to be closed under Heyting completion which is a strong requirement by any standard. The backward type analysis proposed in this paper is based on a novel abstract semantics (that will be presented in a companion paper) that relaxes this requirement. The backward type analysis is obtained by specialising the new abstract semantics with domain operators for type inference and in particular type inference using an abstract domain [18] which is particularly precise since it is disjunctive and supports nondeterministic regular types. The resulting analysis is the first analysis that infers type signatures for a logic program. It gives the programmer the flexibility not to declare and

maintain type signatures for predicates that are subject to frequent modifications during program development. In the extreme situation, the programmer may choose to leave unspecified type signatures for all user-defined predicates and let the analyser to infer type signatures from builtin and library predicates. One application of the new analysis is automatic program documentation; type signatures provide valuable information for both program development and maintenance. Another application is in bug detection. The inferred type signature for a predicate can be compared with that intended by the programmer and any discrepancy indicates the possible existence of bugs.

The remainder of the paper is organised as follows. Section 2 motivates the analysis with a worked example. Section 3 introduces the fundamental concepts used in the paper and outlines the abstract semantics on which the backward type analysis is based. Sections 4 and 5 present the abstract domain and the abstract operations for the backward type analysis. A prototype implementation and analysis examples are presented in section 6 and related work is discussed in section 7. Section 8 concludes. Proofs are omitted due to space limitations.

## 2 Worked Example

This section informally presents a backward type analysis that deduces typing properties of the call which, if satisfied, guarantee that the execution of the call can never encounter a type error. Consider the following partition program.

$pt(xs, x, ys, zs)$ :- ⓪$xs = [y|xs']$, $ys = [y|ys']$,①$y =< x$,②$pt(xs', x, ys', zs)$.③
$pt(xs, x, ys, zs)$ :- ④$xs = [z|xs']$, $zs = [z|zs']$, ⑤$z > x$, ⑥$pt(xs', x, ys, zs')$.⑦
$pt(xs, x, ys, zs)$ :- ⑧$xs = [\,]$, $ys = [\,]$, $zs = [\,]$.⑨

The program is in a normalised form in which the arguments of atoms are distinct variables. Textual points of interest in the program are numbered.

The analysis presupposes the existence of type assertions that are type constraints which must be respected by program execution. A type constraint is a disjunction of primitive type constraints. A primitive type constraint is a conjunction of atomic type constraints of the form $x \in R$ where $x$ is a variable and $R$ a type that denotes a set of terms closed under instantiation; $x$ is said to be of type $R$ if $x \in R$ (see section 4). The type assertions are given by a map $\mathcal{A}$ from atoms to type constraints. The type assertions for the partition program are as follows where type $num$ denotes the set of numbers.

$$\begin{aligned} \mathcal{A}(x > y) &= x \in num \wedge y \in num \\ \mathcal{A}(x =< y) &= x \in num \wedge y \in num \\ \mathcal{A}(pt(xs, x, ys, zs)) &= true \end{aligned}$$

The above type assertions stipulate that the only program states in which $x > y$ or $x =< y$ can be called are those in which both $x$ and $y$ are numbers. For the partition program, the only non-trivial type assertions arise from builtins. This would change if the programmer introduced type assertions for the purposes of verification.

The analysis also requires a set of type rules that gives the denotations of types. For the partition program, type rules are $list(\beta) \rightarrowtriangle [\,]$ and $list(\beta) \rightarrowtriangle [\beta|list(\beta)]$ that define polymorphic lists, According to these two type rules, $[1, 2]$ is of type $list(num)$ while $[1|2]$ is not.

The analysis is formulated in terms of abstract interpretation that iteratively refines and updates a function until stability is reached. The function maps a demand $\langle p(\boldsymbol{x}), \phi_R \rangle$ to a type constraint $\phi_L$ where $\boldsymbol{x}$ denotes a tuple of distinct variables and $\phi_R$ a type constraint. The guarantee is that if the execution of $p(\boldsymbol{x})$ begins in a state satisfying $\phi_L$, then the execution will respect type assertions in $\mathcal{A}$ and, if it succeeds, terminate in a state satisfying $\phi_R$. $\phi_R$ is called a type post-condition and $\phi_L$ a type pre-condition for $\langle p(\boldsymbol{x}), \phi_R \rangle$. The type pre-condition for the demand $\langle p(\boldsymbol{x}), true \rangle$ specifies the type signature for $p(\boldsymbol{x})$. Note that there may be more than one demand for a predicate.

The analysis is performed by computing a greatest fixpoint (gfp). One starts with assuming no call causes a type error and then checks this assumption by reasoning backwards over all clauses. If a type assertion is violated, type pre-conditions for demands are strengthened (made smaller), and the whole process is repeated until the assumptions turn out to be valid (the gfp is reached). A series $D_i : i \geq 0$ of iterates is computed. Each iterate $D_i$ that is computed corresponds to a function that maps a demand to a type pre-condition. The iterate contains only those demands whose type pre-conditions are required for the analysis to proceed. If a demand is not in the iterate, its type pre-condition is assumed to be $true$.

The process is exemplified by computing a type pre-condition for a demand $\langle pt(xs, x, ys, zs), true \rangle$. The initial iterate is

$$D_0 = \{\langle pt(xs, x, ys, zs), true \rangle \mapsto true\}$$

Each clause for $pt/4$ is used to compute the type pre-condition for the demand $\langle pt(xs, x, ys, zs), true \rangle$ in $D_1$. Illustrated below are steps in reasoning backwards over (from right to left) the body of the first clause for $pt/4$. The type constraint for each program point in the clause is listed. The comment following the program point explains the action taken to compute the type constraint for the next program point to the left. Types $num$ and $list(num)$ are abbreviated as $n$ and $l(n)$ respectively.

$$\begin{array}{r} true \ \textcircled{3} \\ true \ \textcircled{2} \text{ builtin predicate} \\ (x \in n \wedge y \in n) \ \textcircled{1} \text{ abstract unification} \\ (x \in n \wedge y \in n) \vee (xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n)) \ \textcircled{0} \end{array}$$

The demand $\langle pt(xs', x, ys', zs), true \rangle$ is a variant of $\langle pt(xs, x, ys, zs), true \rangle$ which is in $D_0$. Thus, the type constraint for point ② is $true$ which is obtained by applying $D_0$ and renaming. The type constraint $(x \in n \wedge y \in n)$ at point ① results from the type assertion for $=<$ in $\mathcal{A}$. The type constraint at point ⓪ is obtained by performing a backward abstract unification. An informal explanation follows. The conjunct $(x \in n \wedge y \in n)$ derives from the fact that a type constraint that holds in the program state before unification also holds in the program state after unification. The conjunct $xs \in l(n) \wedge x \in n$ is derived as follows. Both $xs$ and $[y|xs']$ are of the same type after unification $xs = [y|xs']$. If $xs \in l(n)$ then $[y|xs'] \in l(n)$ which implies $y \in n$. Therefore, if $(xs \in l(n) \wedge x \in n)$ holds at point ⓪ then $(x \in n \wedge y \in n)$ holds at point ①. The conjunct $(x \in n \wedge ys \in l(n))$ is derived similarly. The type constraint at point ⓪ is *universally* quantified with respect to variables $\{y, xs', ys'\}$ to compute a type constraint that is only defined in terms of the variables in the head of the clause. This

gives type constraint $(xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n))$ which is strictly stronger than the type constraint at point ⓪. More generally, universal quantification of a type constraint $\phi$ obtains a type constraint $\phi'$ that is at least as strong as $\phi$ in that a program state $\theta$ satisfies $\phi$ whenever $\theta$ satisfies $\phi'$. By restricting a type constraint in this way to variables in the head, the type constraints are kept small and manageable. Universal quantification is required because weakening type constraints compromises safety whereas strengthening the type constraints preserves safety.

Processing the second and third clauses for $pt/4$ gives type constraints $(xs \in l(n) \wedge x \in n) \vee (x \in n \wedge zs \in l(n))$ and $true$ respectively. The three type constraints obtained are conjoined to give the new type pre-condition $(xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n) \wedge zs \in l(n))$ for $\langle pt(xs, x, ys, zs), true\rangle$. Conjunction is required since type assertions must be respected no matter which clause is selected. Thus,

$$D_1 = \{\langle pt(xs, x, ys, zs), true\rangle \mapsto (xs \in l(n) \wedge x \in n) \vee (x \in n \wedge ys \in l(n) \wedge zs \in l(n))\}$$

Omitting the details of the computation of $D_2$, the gfp is reached at $D_1$ since $D_2 = D_1$. The gfp expresses safe call pattern; it states that $pt/4$ cannot generate a type error if either it is called with a list of numbers as its first argument and a number as its second argument or it is called with a number as its second argument and lists of numbers as its third and fourth arguments. Manual inspection might miss the latter.

## 3 Preliminaries

We now recall some terminology of logic programming and that of abstract interpretation. The reader is referred to [16] and [6] for more detailed exposition.

### 3.1 Basic Concepts

Let $\Sigma$ be a set of function symbols, $\mathcal{V}$ a denumerable set of variables. We assume that $\Sigma$ contains at least one function symbol of arity 0. $\mathsf{Term}(\Sigma, V)$ denotes the set of terms that can be constructed from $\Sigma$ and $V$ where $V \subseteq \mathcal{V}$.

The object resulting from renaming a variable $x$ in object $o$ to another variable $y$ is denoted $\rho_{x \mapsto y}(o)$. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be vectors of variables of the same dimension. $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}(o)$ is the result of simultaneously replacing elements of $\boldsymbol{x}$ in $o$ with corresponding elements of $\boldsymbol{y}$. An equation over $V$ is a formula of the form $t_1 = t_2$ with $t_1, t_2 \in \mathsf{Term}(\Sigma, V)$. An equational constraint is a finite set (conjunction) of equations. The set of all equational constraints is denoted as $Eqn$ whereas the set of all idempotent substitutions is denoted $Sub$. Let $Sub_{fail} = Sub \cup \{fail\}$. Given $E \in Eqn$, $mgu : Eqn \mapsto Sub_{fail}$ returns either a most general unifier for $E$ if $E$ is unifiable or $fail$ otherwise. For brevity, let $mgu(t_1, t_2) \doteq mgu(t_1 = t_2)$. The function composition operator $\circ$ is defined as $f \circ g \doteq \lambda x.f(g(x))$. $Sub$ is ordered by *less general than*, that is, $\theta_1 \leq \theta_2$ iff there exists $\delta \in Sub$ such that $\theta_1 = \delta \circ \theta_2$. The ordering $\leq$ extends to $Sub_{fail}$ by $fail \leq \theta$ for all $\theta \in Sub_{fail}$. The equivalence between substitutions is defined as $\theta_1 \sim \theta_2$ iff $\theta_1 \leq \theta_2$ and $\theta_2 \leq \theta_1$. Furthermore, let $\theta \circ fail \doteq fail$ and $fail \circ \theta \doteq fail$ for any $\theta \in Sub_{fail}$.

Let $VI$ be the set of variables in the program, $dom(\theta)$ the domain of an idempotent substitution $\theta$, and $Var(o)$ the set of variables in the syntactic object $o$. Existential

quantification $\exists_x : Sub_{fail} \mapsto Sub_{fail}$ for $x \in VI$ is defined as $\exists_x(fail) \doteq fail$ and, for $\theta \in Sub$, $\exists_x(\theta) \doteq \{y \mapsto \rho(\theta(y)) \mid y \in dom(\theta) \wedge y \neq x\}$ where $\rho = \{x \mapsto z\}$ and $z \notin (VI \cup Var(\theta))$. Thus, $\exists_x(\theta)$ projects out $x$ if $x \in dom(\theta)$ and otherwise renames $x$ to a fresh variable if $x \in Var(\theta)$. Universal quantification $\forall_x : Sub_{fail} \mapsto Sub_{fail}$ for $x \in VI$ is defined as $\forall_x(\theta) \doteq$ (if $x \notin Var(\theta)$ then $\theta$ else $fail$) . If $x \notin Var(\theta)$ then $\theta$ does not constrain $x$, hence $\forall_x(\theta) = \theta$. Observe that $\forall_x(\theta) \leq \theta \leq \exists_x(\theta)$.

Let $\boldsymbol{x} = \{x_1, x_2, \cdots, x_n\}$ be a subset of $VI$. Define $\exists_{\boldsymbol{x}}(\theta) \doteq \exists_{x_1}(\exists_{x_2}(\cdots \exists_{x_n}(\theta)))$ and $\forall_{\boldsymbol{x}}\theta \doteq \forall_{x_1}(\forall_{x_2}(\cdots \forall_{x_n}(\theta)))$. Also, $\bar{\exists}_{\boldsymbol{x}}(\theta) \doteq \exists_{VI \setminus \boldsymbol{x}}(\theta)$ and $\bar{\forall}_{\boldsymbol{x}}(\theta) \doteq \forall_{VI \setminus \boldsymbol{x}}(\theta)$.

Let $\langle C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \top^C, \bot^C \rangle$ be a complete lattice and $S \subseteq C$. $S$ is a Moore family iff $\top^C \in S$ and $(s_1 \sqcap^C s_2) \in S$ for any $s_1, s_2 \in S$. Let $\langle D, \sqsubseteq^D \rangle$ be a poset. A function $\gamma : D \mapsto C$ is a concretization function iff $\gamma$ is monotone and $\gamma(D)$ is a Moore family. A concretization function from $D$ to $C$ induces a Galois connection between $D$ and $C$ [6]. The Kleene closure of a set $S$ is denoted as $S^*$.

### 3.2 Abstract Semantics

The backward type analysis is based on a novel abstract semantics for backward analysis of logic programs that is sketched below so that the paper is self-contained. The abstract semantics is a (lower) approximation to a collecting semantics that maps a call $p(\boldsymbol{x})$ and a set $\Theta$ of substitutions into a set $\Xi$ of substitutions such that, for any $\xi \in \Xi$, if $\delta$ is a computed answer for $\xi(p(\boldsymbol{x}))$ then $\delta \circ \xi \in \Theta$, i.e., $\{\Xi\}p(\boldsymbol{x})\{\Theta\}$ is a valid partial correctness formula. Note that $\Xi = \emptyset$ is valid solution. In a more precise solution, $\Xi$ contains more substitutions while maintaining partial correctness. The collecting semantics is defined on the concrete domain $\langle \wp(Sub), \subseteq \rangle$ and in terms of six operations: $\cap : \wp(Sub) \times \wp(Sub) \mapsto \wp(Sub)$, $uf_{bw} : Eqn \times \wp(Sub) \mapsto \wp(Sub)$, $ex_{bw} : VI^* \times \wp(Sub) \times \wp(Sub) \mapsto \wp(Sub)$, $\exists_x : \wp(Sub) \mapsto \wp(Sub)$ for $x \in VI$, $\forall_x : \wp(Sub) \mapsto \wp(Sub)$ for $x \in VI$ and $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}} : \wp(Sub) \mapsto \wp(Sub)$ where $\cap$ is the set intersection operation and the others are defined as

$$
\begin{aligned}
uf_{bw}(E, \Theta) &\doteq \{\xi \mid mgu(\xi(E)) \circ \xi \in \Theta\} \\
ex_{bw}(\boldsymbol{x}, \Omega, \Theta) &\doteq \left\{ \xi \;\middle|\; \begin{array}{l} (\bar{\exists}_{\boldsymbol{x}}(\xi) \in \Omega) \\ \wedge\ \forall \delta \in Sub.(\exists \theta \in \Theta.(\delta \circ \bar{\exists}_{\boldsymbol{x}}(\xi) \sim \bar{\exists}_{\boldsymbol{x}}(\theta)) \rightarrow (\delta \circ \xi \in \Theta)) \end{array} \right\} \\
\exists_x(\Theta) &\doteq \{\exists_x(\theta) \mid \theta \in \Theta\} \\
\forall_x(\Theta) &\doteq \{\theta \in \Theta \mid \forall_x(\theta) \sim \theta\} \\
\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}(\Theta) &\doteq \{\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}(\theta) \mid \theta \in \Theta\}
\end{aligned}
$$

The collecting semantics guarantees that $\{\Omega\}p(\boldsymbol{x})\{\bar{\exists}_{\boldsymbol{x}}(\Theta)\}$ is a valid partial correctness formula for some $p(\boldsymbol{x})$ before $ex_{bw}(\boldsymbol{x}, \Omega, \Theta)$ is called to extend $\Omega$ to obtain a set $\Xi$ of substitutions such that $\{\Xi\}p(\boldsymbol{x})\{\Theta\}$ is also valid. This is guaranteed by requiring, for each $\xi \in \Xi$, (1) $\bar{\exists}_{\boldsymbol{x}}(\xi) \in \Omega$ and (2) if $\delta$ is a computed answer for $(\bar{\exists}_{\boldsymbol{x}}(\xi))(p(\boldsymbol{x}))$ then $\delta \circ \xi \in \Theta$.

The abstract semantics is parameterised by an abstract domain $\langle Z, \sqsubseteq^Z \rangle$ and six operations: $\sqcap^Z : Z \times Z \mapsto Z$, $uf^Z_{bw} : Eqn \times Z \mapsto Z$, $ex^Z_{bw} : VI^* \times Z \times Z \mapsto Z$, $\exists^Z_x : Z \mapsto Z$, $\forall^Z_x : Z \mapsto Z$ and $\rho^Z_{\boldsymbol{x} \mapsto \boldsymbol{y}} : Z \mapsto Z$ with $\sqcap^Z$ being the greatest lower bound

operation on $Z$. The correctness of a backward analysis is guaranteed by requiring that there be a concretization function $\gamma^Z : Z \mapsto \wp(Sub)$ and that each operation on $Z$ safely approximates its corresponding operation on $\wp(Sub)$ from below with respect to $\gamma^Z$. For instance, $\exists_x^Z : Z \mapsto Z$ approximates $\exists_x : \wp(Sub) \mapsto \wp(Sub)$ from below iff $\gamma^Z(\exists_x^Z(z)) \subseteq \exists_x(\gamma^Z(z))$ for all $z \in Z$. Note that $\sqcap^Z$ safely approximates $\cap$ from below with respect to $\gamma^Z$ since $\gamma^Z$ is a concretization function. Furthermore, if $\gamma^Z$ is additive, that is, $\gamma^Z(S) = \bigcup_{s\in S} \gamma^Z(s)$ for any $S \subseteq Z$ then it turns out that the precision of the abstract semantics can be improved because the disjunction of two or more pre-conditions for a demand is then a pre-condition for the demand. An abstract domain $\langle Z, \sqsubseteq^Z\rangle$ with an additive $\gamma^Z$ can be constructed from a pre-order $\langle K, \sqsubseteq^K\rangle$ with a monotone $\gamma^K : K \mapsto \wp(Sub)$ satisfying $\bigcup \gamma^K(K) = Sub$ as follows. Let $\approx, \preceq \subseteq \wp(K) \times \wp(K)$ be defined as $\mathcal{S}_1 \approx \mathcal{S}_2 \doteq \bigcup_{s_1\in\mathcal{S}_1} \gamma^K(s_1) = \bigcup_{s_2\in\mathcal{S}_2} \gamma^K(s_2)$ and $\mathcal{S}_1 \preceq \mathcal{S}_2 \doteq \bigcup_{s_1\in\mathcal{S}_1} \gamma^K(s_1) \subseteq \bigcup_{s_2\in\mathcal{S}_2} \gamma^K(s_2)$. Define $Z \doteq \wp(K)_{/\approx}$ and $\sqsubseteq^Z \doteq \preceq_{/\approx}$. It follows that $\langle Z, \sqsubseteq^Z\rangle$ is a complete lattice. Let $\gamma^Z : Z \mapsto \wp(Sub)$ be $\gamma^Z([\mathcal{S}]_\approx) \doteq \bigcup_{s\in\mathcal{S}} \gamma^K(s)$. It can be verified that $\gamma^Z$ is an additive concretization function. Under this construction, $ex_{bw}^Z : VI^* \times Z \times Z \mapsto Z$ can be replaced with $ex_{bw}^Z : VI^* \times Z \times K \mapsto Z$. Thus, the design of the backward type analysis can be accomplished by (i) designing a pre-order $\langle \mathsf{VT}, \sqsubseteq^{\mathsf{VT}}\rangle$ of primitive type constraints and a monotone $\gamma^{\mathsf{VT}} : \mathsf{VT} \mapsto \wp(Sub)$ such that $\bigcup \gamma^{\mathsf{VT}}(\mathsf{VT}) = Sub$, (ii) constructing an abstract domain $\langle \mathsf{DVT}, \sqsubseteq^{\mathsf{DVT}}\rangle$ of type constraints and corresponding concretization function $\gamma^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \wp(Sub)$, (iii) designing abstract operations $uf_{bw}^{\mathsf{DVT}}$, $ex_{bw}^{\mathsf{DVT}}$, $\exists_x^{\mathsf{DVT}}$, $\forall_x^{\mathsf{DVT}}$ and $\rho_{\boldsymbol{x}\mapsto\boldsymbol{y}}^{\mathsf{DVT}}$ and (iv) showing that they safely approximate (from below) corresponding operations on $\wp(Sub)$ with respect to $\gamma^{\mathsf{DVT}}$.

## 4 Abstract Domain

The domain for the backward type analysis is the same as that for a forward type analysis [18]. A type system is first introduced on which the domain is based.

### 4.1 Types

A (monomorphic) type is a ground term constructed from a set $\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}$ of type constructors. It is assumed that $(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}) \cap \Sigma = \emptyset$. The denotations of $\sqcap, \sqcup, \mathbf{1}$ and $\mathbf{0}$ are fixed whilst the denotations of those in $\mathsf{Cons}$ are determined by type rules [7]. The set of all types is then $\mathsf{RT} \doteq \mathsf{Term}(\mathsf{Cons} \cup \{\sqcap, \sqcup, \mathbf{1}, \mathbf{0}\}, \emptyset)$. A type parameter is a variable that ranges over $\mathsf{RT}$. Let $\mathsf{Para}_m \doteq \{\beta_1, \cdots, \beta_m\}$ and $\mathsf{Para} \doteq \bigcup_{c\in\mathsf{Cons}} \mathsf{Para}_{arity(c)}$. A general type over $\mathsf{Para}_m$ is either a type parameter in $\mathsf{Para}_m$ or of the form $d(\beta_1, \cdots, \beta_k)$ with $d \in \mathsf{Cons}$ and $\beta_1, \cdots, \beta_k$ being different type parameters in $\mathsf{Para}_m$. Note that general types are polymorphic. The set of all general types over $\mathsf{Para}_m$ is denoted $\mathsf{GT}_m$ and $\mathsf{GT} \doteq \bigcup_{c\in\mathsf{Cons}} \mathsf{GT}_{arity(c)}$. A type rule is of the form $c(\beta_1, \cdots, \beta_m) \twoheadrightarrow f(\tau_1, \cdots, \tau_n)$ where $f \in \Sigma$ and $\{c(\beta_1, \cdots, \beta_m), \tau_1, \cdots, \tau_n\} \subseteq \mathsf{GT}_m$. The set of all type rules is denoted $\Delta$. It is assumed that each function symbol $f \in \Sigma$ occurs in at least one type rule in $\Delta$ and that each type constructor $c \in \mathsf{Cons}$ occurs in the lefthand side of at least one type rule in $\Delta$.

*Example 1.* Let $\Sigma = \{0, s(), [\,], [\mid]\}$ and $\mathsf{Cons} = \{nat, even, odd, list()\}$. Then $\{\beta, list(\beta)\} \subseteq \mathsf{GT}$ and $list(even \sqcup list(odd)) \in \mathsf{RT}$. The following type rules define natural numbers, even numbers, odd numbers and lists. $\Delta = \{nat \rightarrowtriangle 0, nat \rightarrowtriangle s(nat), even \rightarrowtriangle 0, even \rightarrowtriangle s(odd), odd \rightarrowtriangle s(even), list(\beta) \rightarrowtriangle [\,],\ list(\beta) \rightarrowtriangle [\beta | list(\beta)]\}$. ∎

A type valuation is a mapping $\Bbbk = \{\beta_1 \mapsto R_1, \cdots, \beta_n \mapsto R_n\}$ where $\beta_i \in \mathsf{Para}$ and $R_i \in \mathsf{RT}$. The domain of $\Bbbk$ is defined $dom(\Bbbk) \doteq \{\beta_1, \cdots, \beta_n\}$. Let $\Bbbk(\beta) \doteq \mathbf{0}$ for any $\beta \notin dom(\Bbbk)$. Let $\mathsf{VL}$ denote the set of all type valuations. The application of a type valuation $\Bbbk$ to a general type $\tau$ is to replace simultaneously each occurrence of $\beta$ in $\tau$ with $\Bbbk(\beta)$. A special type valuation $\top^{\mathsf{VL}}$ is introduced and defined $\top^{\mathsf{VL}}(\tau) \doteq \mathbf{1}$ for any general type $\tau$. Let $ground(\Delta)$ be the set of all ground instances of rules in $\Delta$ plus rules of the form $\mathbf{1} \rightarrow f(\mathbf{1}, \cdots, \mathbf{1})$ for every $f \in \Sigma$. More exactly,

$$ground(\Delta) \doteq \{\Bbbk(\delta) \mid \delta \in \Delta \wedge \Bbbk \in (\mathsf{Para} \mapsto \mathsf{RT})\} \cup \{\mathbf{1} \mapsto f(\mathbf{1}, \cdots, \mathbf{1}) \mid f \in \Sigma\}$$

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

$$\begin{array}{ll}
[\![\mathbf{1}]\!]_\Delta \doteq \mathsf{Term}(\Sigma, \mathcal{V}) & [\![\mathbf{0}]\!]_\Delta \doteq \emptyset \\
[\![R_1 \sqcap R_2]\!]_\Delta \doteq [\![R_1]\!]_\Delta \cap [\![R_2]\!]_\Delta & [\![R_1 \sqcup R_2]\!]_\Delta \doteq [\![R_1]\!]_\Delta \cup [\![R_2]\!]_\Delta \\
[\![c(R_1, \cdots, R_m)]\!]_\Delta \doteq & \\
\quad \{f(t_1, \cdots, t_n) \mid \exists (c(R_1, \cdots, R_m) \rightarrowtriangle f(T_1, \cdots, T_n)) \in ground(\Delta).\ t_i \in [\![T_i]\!]_\Delta\} &
\end{array}$$

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

*Example 2.* Let $\Delta$ be that in Ex. 1. Then $[\![nat]\!]_\Delta = \{0, s(0), s(s(0)), \cdots\}$, $[\![list(\mathbf{0})]\!]_\Delta = \{[\,]\}$, and $[\![list(\mathbf{1})]\!]_\Delta = \{[\,], [x], \cdots\}$ where $x \in V$. ∎

**Proposition 1.** *Let $R \in \mathsf{RT}$. If $t \in [\![R]\!]_\Delta$ then $\xi(t) \in [\![R]\!]_\Delta$ for all $\xi \in Sub$. That is, types are closed under instantiation.*[1] ∎

Let $R_1$ and $R_2$ be two types. The set $\mathsf{RT}$ of types is ordered by $R_1 \sqsubseteq R_2$ iff $[\![R_1]\!]_\Delta \subseteq [\![R_2]\!]_\Delta$ and equivalence between types is defined by $R_1 \equiv R_2$ iff $[\![R_1]\!]_\Delta = [\![R_2]\!]_\Delta$.

### 4.2 Abstract Domain

The abstract domain is obtained by choosing a representation of type constraints informally introduced in section 2. Let $\mathsf{VT} \doteq (VI \mapsto \mathsf{RT})$ and $\sqcap^{\mathsf{VT}}$ and $\sqsubseteq^{\mathsf{VT}}$ be the point-wise extensions of the type constructor $\sqcap$ and the pre-order $\sqsubseteq$ respectively. Let $\top^{\mathsf{VT}} \doteq \{x \mapsto \mathbf{1} \mid x \in VI\}$. $\langle \mathsf{VT}, \sqsubseteq^{\mathsf{VT}} \rangle$ is a pre-order since $\langle \mathsf{RT}, \sqsubseteq \rangle$ is a pre-order. Members of $\mathsf{VT}$ represent primitive type constraints. For instance, if $VI = \{x, y\}$ then $\{x \mapsto nat, y \mapsto list(nat)\}$ constrains $x$ to be of type $nat$ and $y$ to be of type $list(nat)$. The denotation of a primitive type constraint is given by $\gamma^{\mathsf{VT}} : \mathsf{VT} \mapsto \wp(Sub)$ defined as $\gamma^{\mathsf{VT}}(\mu) \doteq \{\theta \mid \forall x \in VI.(\theta(x) \in [\![\mu(x)]\!]_\Delta)\}$. Observe that $\bigcup \gamma^{\mathsf{VT}}(\mathsf{VT}) = Sub$ since $\gamma^{\mathsf{VT}}(\top^{\mathsf{VT}}) = Sub$. A type constraint – a disjunction of primitive type constraints

[1] This would change if set complement were a type constructor.

– is represented as a set containing exactly those primitive type constraints in the disjunction. For instance, if $VI = \{x, y\}$ and $\mathcal{S} = \{\{x \mapsto nat, y \mapsto list(nat)\}, \{x \mapsto list(nat), y \mapsto nat\}\}$ then $\mathcal{S}$ denotes $\{\theta \mid \theta(x) \in [\![nat]\!]_\Delta \wedge \theta(y) \in [\![list(nat)]\!]_\Delta\} \cup \{\xi \mid \xi(x) \in [\![list(nat)]\!]_\Delta \wedge \xi(y) \in [\![nat]\!]_\Delta\}$.

There may be many type constraints that denote the same set of substitutions. Type constraints that have the same denotation are identified as follows. Let relation $\preccurlyeq$ on $\wp(\mathsf{VT})$ be defined as $\mathcal{S}_1 \preccurlyeq \mathcal{S}_2 \doteq \bigcup_{\mu \in \mathcal{S}_1} \gamma^{\mathsf{VT}}(\mu) \subseteq \bigcup_{\nu \in \mathcal{S}_2} \gamma^{\mathsf{VT}}(\nu)$. Define relation $\approx$ on $\wp(\mathsf{VT})$ by $\mathcal{S}_1 \approx \mathcal{S}_2 \doteq (\mathcal{S}_1 \preccurlyeq \mathcal{S}_2) \wedge (\mathcal{S}_2 \preccurlyeq \mathcal{S}_1)$. Observe that $\approx$ is an equivalence relation on $\wp(\mathsf{VT})$.

Let $\mathsf{DVT} \doteq \wp(\mathsf{VT})_{/\approx}$ and $\sqsubseteq^{\mathsf{DVT}} \doteq \preccurlyeq_{/\approx}$. The abstract domain $\langle \mathsf{DVT}, \sqsubseteq^{\mathsf{DVT}}, \sqcup^{\mathsf{DVT}}, \sqcap^{\mathsf{DVT}}, \top^{\mathsf{DVT}}, \bot^{\mathsf{DVT}} \rangle$ is a complete lattice where $[\mathcal{S}_1]_\approx \sqcup^{\mathsf{DVT}} [\mathcal{S}_2]_\approx = [\mathcal{S}_1 \cup \mathcal{S}_2]_\approx$, $[\mathcal{S}_1]_\approx \sqcap^{\mathsf{DVT}} [\mathcal{S}_2]_\approx = [\{\{x \mapsto (\mu(x) \sqcap \nu(x)) \mid x \in VI\} \mid \mu \in \mathcal{S}_1 \wedge \nu \in \mathcal{S}_2\}]_\approx$, $\bot^{\mathsf{DVT}} = [\emptyset]_\approx$ and $\top^{\mathsf{DVT}} = [\{\top^{\mathsf{VT}}\}]_\approx$. Define $\gamma^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \wp(Sub)$ by $\gamma^{\mathsf{DVT}}([\mathcal{S}]_\approx) \doteq \bigcup_{\mu \in \mathcal{S}} \gamma^{\mathsf{VT}}(\mu)$. $\gamma^{\mathsf{DVT}}(\mathsf{DVT})$ is a Moore family [18]. Thus, $\gamma^{\mathsf{DVT}}$ is a concretization function from $\langle \mathsf{DVT}, \sqsubseteq^{\mathsf{DVT}} \rangle$ to $\langle \wp(Sub), \subseteq \rangle$. Type constraints are closed under instantiation because types are closed under instantiation.

## 5 Abstract Operations for Backward Type Analysis

The design of the backward type analysis is now completed by defining those operations required by the abstract semantics outlined in section 3.2.

### 5.1 Simple Operations: $\exists_{\boldsymbol{x}}^{\mathsf{DVT}}$, $\forall_{\boldsymbol{x}}^{\mathsf{DVT}}$, $\rho_{\boldsymbol{x} \mapsto \boldsymbol{y}}^{\mathsf{DVT}}$ and $ex_{\boldsymbol{bw}}^{\mathsf{DVT}}$

The construction begins with the simple operations. Let $x \in VI$, $\boldsymbol{x} = \{x_1, \cdots, x_n\}$ a subset of $VI$ and $\mu \in \mathsf{VT}$. The existential quantification operation $\exists_x^{\mathsf{VT}} : \mathsf{VT} \mapsto \mathsf{VT}$ returns $\{y \mapsto \mathbf{0} \mid y \in VI\}$ if $\mu(x)$ denotes the empty set of terms; otherwise, it removes the constraint on $x$ from $\mu$.

$$\exists_x^{\mathsf{VT}}(\mu) \doteq \begin{cases} \{y \mapsto \mathbf{0} \mid y \in VI\} & \text{if } \mu(x) \equiv \mathbf{0} \\ \mu \circ \{x \mapsto \mathbf{1}\} & \text{otherwise} \end{cases}$$

Let $\exists_{\boldsymbol{x}}^{\mathsf{VT}}(\mu) \doteq \exists_{x_1}^{\mathsf{VT}}(\exists_{x_2}^{\mathsf{VT}}(\cdots \exists_{x_n}^{\mathsf{VT}}(\mu)))$ and $\bar{\exists}_{\boldsymbol{x}}^{\mathsf{VT}}(\mu) \doteq \exists_{VI \setminus \boldsymbol{x}}^{\mathsf{VT}}(\mu)$. It follows that $\gamma^{\mathsf{VT}}(\exists_x^{\mathsf{VT}}(\mu) \sqcap^{\mathsf{VT}} \bar{\exists}_x^{\mathsf{VT}}(\mu)) = \gamma^{\mathsf{VT}}(\mu)$. The existential quantification operation $\exists_x^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined as $\exists_x^{\mathsf{DVT}}([\mathcal{S}]_\approx) \doteq [\{\exists_x^{\mathsf{VT}}(\mu) \mid \mu \in \mathcal{S}\}]_\approx$. It removes the constraint on $x$ from those primitive type constraints that describe a non-empty set of substitutions. Let $\exists_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \exists_{x_1}^{\mathsf{DVT}}(\exists_{x_2}^{\mathsf{DVT}}(\cdots \exists_{x_n}^{\mathsf{DVT}}(\phi)))$ and $\bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \exists_{VI \setminus \boldsymbol{x}}^{\mathsf{DVT}}(\phi)$. The universal quantification operation $\forall_x^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined as $\forall_x^{\mathsf{DVT}}([\mathcal{S}]_\approx) \doteq [\{\mu \mid (\mu \in \mathcal{S}) \wedge (\mu(x) \equiv \mathbf{1})\}]_\approx$. It removes primitive constraints that place any constraint on $x$. Let $\forall_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \forall_{x_1}^{\mathsf{DVT}}(\forall_{x_2}^{\mathsf{DVT}}(\cdots \forall_{x_n}^{\mathsf{DVT}}(\phi)))$ and $\bar{\forall}_{\boldsymbol{x}}^{\mathsf{DVT}}(\phi) \doteq \forall_{VI \setminus \boldsymbol{x}}^{\mathsf{DVT}}(\phi)$. Existential and universal quantifications differ in their direction of approximation in that $\forall_x^{\mathsf{DVT}}(\phi) \sqsubseteq^{\mathsf{DVT}} \phi \sqsubseteq^{\mathsf{DVT}} \exists_x^{\mathsf{DVT}}(\phi)$. The renaming operation $\rho_{\boldsymbol{z} \mapsto \boldsymbol{y}}^{\mathsf{DVT}} : \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined as $\rho_{\boldsymbol{z} \mapsto \boldsymbol{y}}^{\mathsf{DVT}}(\phi) \doteq [\boldsymbol{y}/\boldsymbol{z}]\phi$ where $[\boldsymbol{y}/\boldsymbol{z}]\phi$ results from substituting $\boldsymbol{y}$ for $\boldsymbol{z}$ simultaneously. The operation $ex_{bw}^{\mathsf{DVT}}(\boldsymbol{x}, \phi, \mu)$ is applied to $\phi \in \mathsf{DVT}$ and $\mu \in \mathsf{VT}$ when the

execution of an atom $p(\boldsymbol{x})$ in a state satisfying $\phi$ can only succeed in a state satisfying $\bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}([\{\mu\}]_{\approx})$. The following definition of $ex_{bw}^{\mathsf{DVT}} : VI^* \times \mathsf{DVT} \times \mathsf{VT} \mapsto \mathsf{DVT}$ ensures that $ex_{bw}^{\mathsf{DVT}}(\boldsymbol{x}, \phi, \mu)$ is a type constraint $\psi$ such that the execution of $p(\boldsymbol{x})$ in any state satisfying $\psi$ succeeds only in a state satisfying $\mu$.

$$ex_{bw}^{\mathsf{DVT}}(\boldsymbol{x}, \phi, \mu) \doteq \phi \sqcap^{\mathsf{DVT}} \bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}([\{\mu\}]_{\approx})$$

It is straightforward to show that the abstract quantification and renaming operations are correct, but the $ex_{bw}^{\mathsf{DVT}}$ operator is more subtle. Therefore let $\theta \in \gamma^{\mathsf{DVT}}(\phi \sqcap^{\mathsf{DVT}} \bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}([\{\mu\}]_{\approx}))$ and suppose $\theta(p(\boldsymbol{x}))$ succeeds with a computed answer $\zeta$. Since $\theta \in \gamma^{\mathsf{DVT}}(\phi)$, $\zeta \circ \theta \in \gamma^{\mathsf{DVT}}(\bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}([\{\mu\}]_{\approx}))$ by assumption. Since $\theta \in \gamma^{\mathsf{DVT}}(\bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}([\{\mu\}]_{\approx}))$ and type constraints in $\mathsf{DVT}$ are closed under instantiation, $\zeta \circ \theta \in \gamma^{\mathsf{DVT}}(\bar{\exists}_{\boldsymbol{x}}^{\mathsf{DVT}}([\{\mu\}]_{\approx}))$. Thus, $\zeta \circ \theta \in \gamma^{\mathsf{DVT}}([\{\mu\}]_{\approx})$ by the definitions of $\gamma^{\mathsf{DVT}}$ and $\gamma^{\mathsf{VT}}$. The remaining operation $uf_{bw}^{\mathsf{DVT}} : Eqn \times \mathsf{DVT} \mapsto \mathsf{DVT}$ is the most complex; it simulates the (reverse) effect of unification.

### 5.2 Backward Abstract Unification: $uf_{bw}^{\mathsf{DVT}}$

In forward type analysis [18], abstract unification takes as inputs an equation $E$ and a type constraint $\phi$ (an upper approximation) and produces as output a type constraint $\psi$ (another upper approximation) which describes $mgu(\theta(E)) \circ \theta$ whenever $\phi$ describes $\theta$. In backward type analysis, abstract unification takes as inputs an equation $E$ and a type constraint $\psi$ (a lower approximation) and produces as output a type constraint $\phi$ (another lower approximation) which describes $\theta$ whenever $\psi$ describes $mgu(\theta(E)) \circ \theta$. Backward abstract unification is defined as a rewriting system which in turn is formulated in terms of the operations $type\_low$ (lower approximation to a type) and $tc\_low$ (lower approximation to a type constraint). These operations are themselves defined with the auxiliary operations introduced below.

**Conjoining type valuations** Given two type valuations $\Bbbk_1$ and $\Bbbk_2$, $\curlywedge : \mathsf{VL} \times \mathsf{VL} \mapsto \mathsf{VL}$ defined below gives another type valuation.

$$\Bbbk_1 \curlywedge \Bbbk_2 \doteq \begin{cases} \Bbbk_2 & \text{if } (\Bbbk_1 = \top^{\mathsf{VL}}); \\ \Bbbk_1 & \text{else if } (\Bbbk_2 = \top^{\mathsf{VL}}); \\ \{\beta \mapsto \Bbbk_1(\beta) \sqcap \Bbbk_2(\beta) \mid \beta \in dom(\Bbbk_1) \cap dom(\Bbbk_2)\} & \text{otherwise} \end{cases}$$

By the definitions of $[\![\cdot]\!]_{\Delta}$ and $\curlywedge$, it follows that $(\Bbbk_1 \curlywedge \Bbbk_2)(\tau) = \Bbbk_1(\tau) \sqcap \Bbbk_2(\tau)$ for any $\tau \in \mathsf{GT}$ and that $\curlywedge$ is commutative and associative with respect to $\equiv$.

**Approximating type valuations** Given a type $R$, a general type $\tau$, the function $tvs\_low$ : $\mathsf{RT} \times \mathsf{GT} \mapsto \wp(\mathsf{VL})$ defined below returns a set of type valuations that instantiate $\tau$ to types smaller than or equal to $R$.

$$tvs\_low(R,\tau) \doteq \begin{cases} \{\top^{\mathsf{VL}}\} & \text{if } R = \mathbf{1} \\ \{\{\tau \mapsto R\}\} & \text{else if } \tau \in \mathsf{Para} \\ tvs\_low(R_1,\tau) \cup tvs\_low(R_2,\tau) & \text{else if } R = R_1 \sqcup R_2 \\ \{\Bbbk_1 \curlywedge \Bbbk_2 \mid \Bbbk_i \in tvs\_low(R_i,\tau)\} & \text{else if } R = R_1 \sqcap R_2 \\ \{\{\beta_j \mapsto R_j \mid 1 \le j \le m\}\} & \text{else if } \left(\begin{smallmatrix} R = c(R_1,\cdots,R_m) \\ \tau = c(\beta_1,\cdots,\beta_m) \end{smallmatrix}\right) \\ \{\} & \text{otherwise} \end{cases}$$

The first two and the last two branches in the definition of $tvs\_low$ are obvious. The two recursive calls in the third branch returns two sets of type valuations members of which instantiate $\tau$ to types smaller than or equal to $R_1$ and $R_2$ respectively. Therefore, the union of two sets consists of type valuations that instantiate $\tau$ to types smaller than or equal to $R_1 \sqcup R_2$. The fourth branch applies $\curlywedge$ to each pair consisting of a type valuation from $tvs\_low(R_1,\tau)$ and a type valuation from $tvs\_low(R_2,\tau)$ resulting in a type valuation that instantiates $R$ to a type smaller than or equal to $R_1 \sqcap R_2$.

*Example 3.* Let $\Delta$ be that in Ex. 1. Then $tvs\_low(nat,\beta) = \{\{\beta \mapsto nat\}\}$ and $tvs\_low(list(nat) \sqcup list(list(nat)), list(\beta)) = \{\{\beta \mapsto nat\}\} \cup \{\{\beta \mapsto list(nat)\}\} = \{\{\beta \mapsto nat\}, \{\beta \mapsto list(nat)\}\}$. ∎

**Lemma 1.** *Let $R$ be a type and $\tau$ a general type. Then $[\![\sqcup_{\Bbbk \in tvs\_low(R,\tau)} \Bbbk(\tau)]\!]_\Delta \subseteq [\![R]\!]_\Delta$.* ∎

**Approximating types** Given a primitive type constraint $\mu$ and a term $t$, $type\_low : \mathsf{VT} \times \mathsf{Term}(\Sigma, VI) \mapsto \mathsf{RT}$ gives a type that describes a subset of those terms $\theta(t)$ for which $\mu$ describes $\theta$.

$$type\_low(\mu,t) \doteq \begin{cases} \mu(t) & \text{if } t \in VI \\ \sqcup \left\{ (\Bbbk_1 \curlywedge \cdots \curlywedge \Bbbk_n)(\tau) \middle| \begin{array}{c} R_i = type\_low(\mu,t_i) \\ \wedge\ (\tau \twoheadrightarrow f(\tau_1,\cdots,\tau_n)) \in \Delta \\ \wedge\ \Bbbk_i \in tvs\_low(R_i,\tau_i) \end{array} \right\} & \text{if } t = f(t_1,\cdots,t_n) \end{cases}$$

**Lemma 2.** *Let $\mu \in \mathsf{VT}$, $t \in \mathsf{Term}(\Sigma, VI)$ and $\theta \in Sub$. If $\theta(t) \in [\![type\_low(\mu,t)]\!]_\Delta$ then $\theta \in \gamma^{\mathsf{VT}}(\mu)$.* ∎

*Example 4.* Continuing with Ex.3, let $\mu = \{t_1 \mapsto nat, t_2 \mapsto list(nat) \sqcup list(list(nat))\}$ and $t = [t_1|t_2]$. Let $R_1 = type\_low(\mu,t_1)$ and $R_2 = type\_low(\mu,t_2)$. Then $R_1 = nat$ and $R_2 = list(nat) \sqcup list(list(nat))$. The only applicable type rule is $list(\beta) \twoheadrightarrow [\beta|list(\beta)]$. By Ex. 3, $tvs\_low(R_1,\beta) = \{\{\beta \mapsto nat\}\}$ and $tvs\_low(R_2, list(\beta)) = \{\{\beta \mapsto nat\}, \{\beta \mapsto list(nat)\}\}$. Let $\Bbbk_1 = \{\beta \mapsto nat\}$, $\Bbbk_{21} = \{\beta \mapsto nat\}$ and $\Bbbk_{22} = \{\beta \mapsto list(nat)\}$. Then $tvs\_low(R_1,\beta) = \{\Bbbk_1\}$ and $tvs\_low(R_2, list(\beta)) = \{\Bbbk_{21}, \Bbbk_{22}\}$. Thus, $type\_low(\mu,[t_1|t_2]) = \sqcup\{(\Bbbk_1 \curlywedge \Bbbk_{21})(list(\beta)), (\Bbbk_1 \curlywedge \Bbbk_{22})(list(\beta))\} = list(nat) \sqcup list(nat \sqcap list(nat)) \equiv list(nat)$ since $nat \sqcap list(nat) \equiv \mathbf{0}$. Observe that if $\theta([t_1|t_2]) \in [\![type\_low(\mu,[t_1|t_2])]\!]_\Delta = [\![list(nat)]\!]_\Delta$ then $\theta \in \gamma^{\mathsf{VT}}(\mu)$. ∎

**Approximating type constraints** Let $R \in \mathsf{RT}$ and $t \in \mathsf{Term}(\Sigma, VI)$. The operation $tc\_low : \mathsf{RT} \times \mathsf{Term}(\Sigma, VI) \mapsto \mathsf{DVT}$ defined below gives a type constraint $\phi$ such that $\theta(t) \in [\![R]\!]_\Delta$ for every $\theta$ described by $\phi$:

$$\begin{aligned}
tc_low(R, x) &\doteq [\{\top^{\mathsf{VT}} \circ \{x \mapsto R\}\}]_{\approx} \\
tc_low(R_1 \sqcup R_2, t) &\doteq tc_low(R_1, t) \sqcup^{\mathsf{DVT}} tc_low(R_2, t) \\
tc_low(R_1 \sqcap R_2, t) &\doteq tc_low(R_1, t) \sqcap^{\mathsf{DVT}} tc_low(R_2, t) \\
tc_low(R, f(t_1, \cdots, t_n)) &\doteq \sqcup^{\mathsf{DVT}}_{(R \twoheadrightarrow f(R_1, \cdots, R_n)) \in ground(\Delta)} \sqcap^{\mathsf{DVT}}_{1 \leq i \leq n} tc_low(R_i, t_i)
\end{aligned}$$

Observe that if $R = \mathbf{0}$ and $t = f(t_1, \cdots, t_n)$ then $tc\_low(R, t) = \bot^{\mathsf{DVT}}$ via the fourth definition because $\mathbf{0} \twoheadrightarrow f(R_1, \cdots, R_n)$ is not in $ground(\Delta)$. Also note that if $R = \mathbf{1}$ and $t = f(t_1, \cdots, t_n)$ then $tc\_low(R, t) = \top^{\mathsf{DVT}}$ since $\mathbf{1} \twoheadrightarrow f(\mathbf{1}, \cdots, \mathbf{1})$ is in $ground(\Delta)$.

*Example 5.* Let $\Delta$ be that in Ex. 1 and $VI = \{x, y\}$. Then $tc\_low(list(odd), [x|y]) = tc\_low(odd, x) \sqcap^{\mathsf{DVT}} tc\_low(list(odd), y) = [\{\{x \mapsto odd, y \mapsto \mathbf{1}\}\}]_{\approx} \sqcap^{\mathsf{DVT}} [\{\{x \mapsto \mathbf{1}, y \mapsto list(odd)\}\}]_{\approx} = [\{\{x \mapsto odd, y \mapsto list(odd)\}\}]_{\approx}$ and $tc\_low(list(even), [x|y]) = [\{\{x \mapsto even, y \mapsto list(even)\}\}]_{\approx}$. Thus, $tc\_low(list(odd) \sqcup list(even), [x|y]) = tc\_low(list(odd), [x|y]) \sqcup^{\mathsf{DVT}} tc\_low(list(even), [x|y]) = [\{\{x \mapsto odd, y \mapsto list(odd)\}, \{x \mapsto even, y \mapsto list(even)\}\}]_{\approx}$. ∎

**Lemma 3.** *Let $R \in \mathsf{RT}$ and $t \in \mathsf{Term}(\Sigma, VI)$. Then $\theta(t) \in R$ for any $\theta \in \gamma^{\mathsf{DVT}}(tc\_low(R, t))$.* ∎

**Propagating type constraints backwards by rewriting** The abstract unification is based on a rewriting relation that propagates a type constraint $\psi$ over an equational constraint $E$ in solved form. The relation $\rightsquigarrow \subseteq (\mathsf{VT} \times Eqn) \times (\mathsf{VT} \times Eqn)$ is constructed from from $type\_low$ and $tc\_low$ and is defined as follows:

$$\langle \mu, E \rangle \rightsquigarrow \langle \mu, \emptyset \rangle \tag{1}$$

$$\langle \mu, E \cup \{x = t\} \rangle \rightsquigarrow \langle (\top^{\mathsf{VT}} \circ \{x \mapsto type\_low(\mu, t)\}) \sqcap^{\mathsf{VT}} \exists^{\mathsf{VT}}_{Var(t)}(\mu), E \rangle \tag{2}$$

$$\langle \mu, E \cup \{x = t\} \rangle \rightsquigarrow \langle \exists^{\mathsf{VT}}_x(\mu) \sqcap^{\mathsf{VT}} \nu, E \rangle \text{ where } \nu \in \mathcal{S} \text{ and } [\mathcal{S}]_{\approx} = tc\_low(\mu(x), t) \tag{3}$$

The relation $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$. The first rule is justified because $\gamma^{\mathsf{VT}}(\mu)$ is downward closed and hence $\xi = mgu(\theta(E)) \circ \theta$ is described by $\mu$ if $\theta$ is described by $\mu$. Each rewriting step by the other rules infers a primitive type constraint $\mu'$ from $\mu$ and an equation $x = t$ such that if $\mu'$ describes $\theta$ then $\mu$ describes $\xi = mgu(\theta(x), \theta(t)) \circ \theta$. Consider rule (2) first. Let $R = type\_low(\mu, t)$ and $\theta(t) \in [\![R]\!]_\Delta$. Then $\theta(y) \in \gamma^{\mathsf{VT}}(\mu(y))$ for all $y \in Var(t)$ by Lemma 2, in other words, if the $x$ is constrained to be of type $R$ then the type constraints on $y \in Var(t)$ can be removed from $\mu$ since they are implied by the type of $x$. Now consider rule (3). Let $[\mathcal{S}]_{\approx} = tc\_low(\mu(x), t)$, $\nu \in \mathcal{S}$ and $\theta \in \gamma^{\mathsf{VT}}(\nu)$. Then $\theta(t) \in \gamma^{\mathsf{VT}}(\mu(x))$ by Lemma 3. Thus, the type constraint on $x$ in $\mu$ is implied by $\nu$, hence can be removed if $\nu$ is asserted.

*Example 6.* Let $VI = \{x, y, h, l_1, l_2\}$, $E = \{x = [h|l_1], y = [h|l_2]\}$ and $\mu_0 = \{x \mapsto list(nat),\ y \mapsto list(nat), h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$. Then

$\langle \mu_0, E\rangle \leadsto^{(x=[h|l_1],2)} \langle \mu_1, \{y = [h|l_2]\}\rangle \leadsto^{(y=[h|l_2],2)} \langle \mu_2, \emptyset\rangle$ and $\langle \mu_0, E\rangle \leadsto^{(y=[h|l_2],3)} \langle \mu_3, \{x = [h|l_1]\}\rangle \leadsto^{(x=[h|l_1],2)} \langle \mu_4, \emptyset\rangle$ where each step is labelled with the selected equation and the selected rewriting rule and

$$\mu_1 = \{x \mapsto list(nat), y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto \mathbf{1}, l_2 \mapsto list(nat)\}$$
$$\mu_2 = \{x \mapsto list(nat), y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto \mathbf{1}, l_2 \mapsto \mathbf{1}\}$$
$$\mu_3 = \{x \mapsto list(nat), y \mapsto \mathbf{1}, h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$$
$$\mu_4 = \{x \mapsto list(nat), y \mapsto \mathbf{1}, h \mapsto \mathbf{1}, l_1 \mapsto \mathbf{1}, l_2 \mapsto list(nat)\}$$

Moreover $\langle \mu_1, \{y = [h|l_2]\}\rangle \not\leadsto \langle \mu_4, \emptyset\rangle$ and $\langle \mu_3, \{x = [h|l_1]\}\rangle \not\leadsto \langle \mu_2, \emptyset\rangle$. Observe that $\gamma^{\mathsf{VT}}(\mu_2) \not\sqsubseteq \gamma^{\mathsf{VT}}(\mu_4)$ and $\gamma^{\mathsf{VT}}(\mu_4) \not\sqsubseteq \gamma^{\mathsf{VT}}(\mu_2)$. This shows that $\leadsto$ does not have the diamond property with respect to the selection of the equation. ∎

**Proposition 2.** *Let $E \in Eqn$ and $\mu, \nu \in \mathsf{VT}$ such that $\langle \mu, E\rangle \leadsto^* \langle \nu, \emptyset\rangle$. Then $mgu(\theta(E)) \circ \theta \in \gamma^{\mathsf{VT}}(\mu)$ for every $\theta \in \gamma^{\mathsf{VT}}(\nu)$.* ∎

**Backward Abstract Unification** The backward abstract unification operation $uf_{bw}^{\mathsf{DVT}} : Eqn \times \mathsf{DVT} \mapsto \mathsf{DVT}$ is defined

$$uf_{bw}^{\mathsf{DVT}}(E, [\mathcal{S}]_{\approx}) \doteq [\{\nu \mid \exists \mu \in \mathcal{S}.(\langle \mu, E\rangle \leadsto^* \langle \nu, \emptyset\rangle)\}]_{\approx}$$

$uf_{bw}^{\mathsf{DVT}}(E, [\mathcal{S}]_{\approx})$ rewrites $\langle \mu, E\rangle$ in every possible way for primitive constraint $\mu$ in $\mathcal{S}$ and collects the resulting primitive constraints. Proposition 2 can be interpreted as saying that correctness is not compromised by barring either rule (1), rule (2) or rule (3). Maximising the number of rules used maximises $uf_{bw}^{\mathsf{DVT}}(E, [\mathcal{S}]_{\approx})$ and therefore maximises precision by minimising the resulting type constraint.

*Example 7.* Continuing with Ex. 6, let

$$\mu_5 = \{x \mapsto \mathbf{1}, y \mapsto list(nat), h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$$
$$\mu_6 = \{x \mapsto list(nat), y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto list(nat), l_2 \mapsto \mathbf{1}\}$$
$$\mu_7 = \{x \mapsto \mathbf{1}, y \mapsto list(nat), h \mapsto \mathbf{1}, l_1 \mapsto list(nat), l_2 \mapsto \mathbf{1}\}$$
$$\mu_8 = \{x \mapsto \mathbf{1}, y \mapsto \mathbf{1}, h \mapsto nat, l_1 \mapsto list(nat), l_2 \mapsto list(nat)\}$$

Then $\langle \mu_0, E\rangle \leadsto^* \langle \mu_i, \emptyset\rangle$ for each $0 \leq i \leq 8$. It can be verified that there is no other primitive type constraint $\nu$ such that $\langle \mu_0, E\rangle \leadsto^* \langle \nu, \emptyset\rangle$. Thus, by the definition of $uf_{bw}^{\mathsf{DVT}}$, $uf_{bw}^{\mathsf{DVT}}(E, [\{\mu_0\}]_{\approx}) = [\{\mu_i \mid 0 \leq i \leq 8\}]_{\approx} = [\{\mu_2, \mu_4, \mu_7, \mu_8\}]_{\approx}$ Since $\mu_0 \sqsubseteq^{\mathsf{VT}} \mu_1 \sqsubseteq^{\mathsf{VT}} \mu_2$, $\mu_3 \sqsubseteq^{\mathsf{VT}} \mu_4$, $\mu_5 \sqsubseteq^{\mathsf{VT}} \mu_7$ and $\mu_6 \sqsubseteq^{\mathsf{VT}} \mu_7$. ∎

The following result states the correctness of $uf_{bw}^{\mathsf{DVT}}$, which together with correctness of $ex_{bw}^{\mathsf{DVT}}, \exists_x^{\mathsf{DVT}}, \forall_x^{\mathsf{DVT}}$ and $\rho_{\boldsymbol{x}\mapsto\boldsymbol{y}}^{\mathsf{DVT}}$ implies the correctness of the backward type analysis.

**Theorem 1.** *Let $E \in Eqn$ and $\phi \in \mathsf{DVT}$. Then $mgu(\theta(E)) \circ \theta \in \gamma^{\mathsf{DVT}}(\phi)$ for every $\theta \in \gamma^{\mathsf{DVT}}(uf_{bw}^{\mathsf{DVT}}(E, \phi))$.* ∎

## 6 Prototype Analyser and Examples

In order to evaluate the usefulness of the analysis presented in sections 4 and 5, a backward type analyser has been constructed for inferring type pre-conditions. The analyser is coded in SICStus Prolog 3.8.3. The analyser takes, as input, a program written in a declarative subset of ISO Prolog, type definitions for function symbols occurring in the program, type assertions for selected user-defined predicates, and a set of initial demands each consisting of an atom paired with a type post-condition. The analyser outputs a type constraint that is a type pre-condition for each initial demand. The safety result of the analysis ensures that if the atom in an initial demand is executed in a state satisfying the inferred pre-condition for the demand, then the execution will not violate any type assertion and the post-condition in the demand will be satisfied by any state in which the execution succeeds. The analyser supports pre-defined types such as $atom$, $float$, $int$, $num$ and $string$ with usual meanings.

The implementation of abstract operations follows section 5 closely. The top-level of the analyser was straightforward to implement as it is essentially a fixpoint computation. The only subtlety is in handling the builtins. For each builtin, it is necessary to specify a type assertion that is strong enough for avoiding a type error. This is a lower approximation (the *Assertion* column of table 1). It is also necessary to specify an operation that transforms a type post-condition for a builtin to a type pre-condition. These operations output lower approximations (the *Operation* column of table 1). Table 1 displays type constraints instead of their representations in $\mathsf{DVT}$ for improved readability.

Type assertions for some builtins are given in Table 1; most are easily verified but some warrant further explanation. Unification `=/2` is modelled by the abstract unification operation $uf_{bw}^{\mathsf{DVT}}$. Builtins such as `>/2` and $put/1$ that never instantiate their arguments, are modelled by the identity function $\lambda\phi.\phi$ since any type constraint that holds when the builtin succeeds must also hold before the builtin is called. The builtin $fail/0$ is modelled by the constant function that always returns $true$ ($\top^{\mathsf{DVT}}$) since any type post-condition of $fail/0$ is vacuously satisfied.

Consider a builtin to which a call $p(\boldsymbol{x})$ will definitely instantiate $x$ in $\boldsymbol{x}$ to a term of type $R$ upon success. Let $\mu$ be a conjunct in the type post-condition for the call. Observe that $\mu$ is a type pre-condition for the demand $\langle p(\boldsymbol{x}), \mu\rangle$. This type pre-condition, however, can be weakened if $R{\sqsubseteq}\mu(x)$. Specifically, if $\exists_x^{\mathsf{VT}}.\mu$ holds before the execution of the call and $R{\sqsubseteq}\mu(x)$ then $\mu$ holds upon success of the execution. In other words, $\exists_x^{\mathsf{VT}}.\mu$ is a type pre-condition for the demand $\langle p(\boldsymbol{x}), \mu\rangle$ if $R{\sqsubseteq}\mu(x)$. Thus define $relax : \mathsf{VT} \times VI \times \mathsf{RT}$ by

$$relax(\mu, x, R) \doteq \text{ if } R{\sqsubseteq}\mu(x) \text{ then } \exists_x^{\mathsf{VT}}.\mu \text{ else } \mu$$

and $relax : \mathsf{DVT} \times VI \times \mathsf{RT}$ by $relax([\mathcal{S}]_{\approx}, x, R) \doteq [\{relax(\mu, x, R) \mid \mu \in \mathcal{S}\}]_{\approx}$. The operation for the builtin $p(\boldsymbol{x})$ transforms $\phi$ to $relax(\phi, x, R)$ for each $x$ in $\boldsymbol{x}$ that is definitely of type $R$ upon success of the execution of $p(\boldsymbol{x})$. For instance, the operation for $is(x_1, x_2)$ is $\lambda\phi.relax(\phi, x_1, num)$ since $x_1$ is definitely of type $num$ upon success of $is(x_1, x_2)$. Note that the execution of $is(x_1, x_2)$ does not instantiate $x_2$. The operation for $read(x_1)$ is $\lambda\phi.\phi$ since $x_1$ does not necessarily belong to any type other than $\mathbf{1}$ upon the success of $read(x_1)$.

| *Builtin* | *Assertion* | *Operation* |
|---|---|---|
| $abort, fail, false$ | $true$ | $\lambda\phi.true$ |
| !, $x_1$@<$x_2$, $x_1$@>$x_2$, $x_1$=<@$x_2$, $x_1$@>=$x_2$, $x_1$==$x_2$, $x_1$\==$x_2$, $x_1$\=$x_2$, $compound(x_1)$, $display(x_1)$, $ground(x_1)$, $listing$, $listing(x_1)$, $nl$, $nonvar(x_1)$, $portray\_clause(x_1)$, $print(x_1)$, $read(x_1)$, $repeat$, $true$, $var(x_1)$, $write(x_1)$, $writeq(x_1)$, $float(x_1)$, $string(x_1)$, $atom(x_1)$, $atomic(x_1)$, $integer(x_1)$, $number(x_1)$ | $true$ | $\lambda\phi.\phi$ |
| $x_1$=$x_2$ | $true$ | $\lambda\phi.uf_{bw}^{\mathsf{DVT}}(\{x_1 = x_2\}, \phi)$ |
| $format(x_1)$, $format(x_1, x_2)$, $format(x_0, x_1, x_2)$ | $\phi_1$ | $\lambda\phi.\phi$ |
| $is(x_1, x_2)$ | $\phi_2$ | $\lambda\phi.relax(\phi, x_1, num)$ |
| $erase(x_1)$, $put(x_1)$, $tab(x_1)$ | $\phi_3$ | $\lambda\phi.\phi$ |
| $x_1$<$x_2$, $x_1 > x_2$, $x_1$=<$x_2$, $x_1$>=$x_2$, $x_1$=:=$x_2$, $x_1$=\=$x_2$ | $\phi_4$ | $\lambda\phi.\phi$ |
| $length(x_1, x_2)$ | $true$ | $f_1$ |
| $compare(x_1, x_2, x_3)$ | $true$ | $\lambda\phi.relax(\phi, x_1, atom)$ |
| $name(x_1, x_2)$ | $\phi_5$ | $f_2$ |

**Table 1.** Type assertions and abstract operations for builtins where $\phi_1 = (x_1 \in atom \sqcup string \sqcup list(int))$, $\phi_2 = (x_2 \in num)$, $\phi_3 = (x_1 \in int)$, $\phi_4 = (x_1 \in num \wedge x_2 \in num)$, $\phi_5 = (x_1 \in atom \sqcup int \vee x_2 \in string)$, $f_1 = \lambda\phi.relax(relax(\phi, x_1, list(\mathbf{1})), x_2, int)$ and $f_2 = \lambda\phi.relax(relax(\phi, x_1, atom \sqcup int), x_2, string)$ .

The analyser forces termination by limiting the number of types that may occur in demands and thereby limiting the number of demands. This is achieved by depth-$k$ abstraction – a technique that is frequently applied in type analysis [1, 3, 18]. Sub-terms at depth $k$ in a type $R$ are replaced with $\mathbf{0}$ (rather than $\mathbf{1}$), resulting a type $R'$ such that $R' \sqsubseteq R$. Thus, application of depth-$k$ abstraction to types in the post-condition $\phi$ in a demand $\langle p(\boldsymbol{x}), \phi\rangle$ obtains a post-condition $\phi'$ which is at least as strong as $\phi$. Safety is preserved because a pre-condition for the stronger $\langle p(\boldsymbol{x}), \phi'\rangle$ is also a pre-condition for the weaker $\langle p(\boldsymbol{x}), \phi\rangle$.

The analyser has been applied to some standard Prolog benchmarks which can be found at http://www.oakland.edu/∼l2lu/Benchmarks-BT.zip. Inferred type signatures for the predicates in the smaller benchmarks, are given in table 2 which also displays type constraints instead of their representations in DVT. The type assertions for each benchmark program are exactly those for the builtins that are called in the program although type assertions may be provided for user-defined predicates as well. The type signatures were obtained by analysing the program with a set of initial demands – one demand with the type post-condition $true$ for each predicate in the program. The results have been verified by hand and, though sometimes surprising, appear to be optimal. For instance, the first conjunct in the inferred type signature $\phi_2$ for predicate $partition/4$ indicates that the execution of $partition(x_1, x_2, x_3, x_4)$ is free from type errors if $x_2$ is a number, and $x_3$ and $x_4$ are lists of numbers when the execution starts. The analyser can thus infer type signatures that may well be missed by manual inspection. The main weakness of the prototype is that its front end does not currently support control features such as ; and -> or the meta-programming builtins, though this is not a fundamental limitation of the analysis itself.

| *Benchmark* | *Predicate* | *Inferred Type Signature* |
|---|---|---|
| merge | $merge(x_1, x_2, x_3)$ | $(x_1 \in list(num) \wedge x_2 \in list(num))$ |
| heapify | $adjust(x_1, x_2, x_3, x_4)$ | $\phi_1$ |
| | $heapify(x_1, x_2)$ | $(x_1 \in tree(num))$ |
| | $greater(x_1, x_2)$ | $(x_1 \in num \wedge x_2 \in tree(num))$ |
| quicksort | $partition(x_1, x_2, x_3, x_4)$ | $\phi_2$ |
| using | $iqsort\_aux(x_1, x_2, x_3)$ | $(x_1 \in list(num))$ |
| stack | $iqsort(x_1, x_2)$ | $(x_1 \in list(num))$ |
| quicksort | $append(x_1, x_2, x_3)$ | $true$ |
| | $partition(x_1, x_2, x_3, x_4)$ | $\phi_2$ |
| | $quicksort(x_1, x_2)$ | $(x_1 \in list(num))$ |
| quicksort | $partition(x_1, x_2, x_3, x_4)$ | $\phi_2$ |
| using | $quicksort\_dl(x_1, x_2)$ | $(x_1 \in list(num))$ |
| difference list | $quicksort(x_1, x_2)$ | $(x_1 \in list(num))$ |
| treesort | $tree\_to\_list\_aux(x_1, x_2, x_3)$ | $true$ |
| | $tree\_to\_list(x_1, x_2)$ | $true$ |
| | $insert(x_1, x_2, x_3)$ | $\phi_3$ |
| | $insert\_list(x_1, x_2, x_3)$ | $(x_1 \in list(num) \wedge x_2 \in tree(num))$ |
| | $list\_to\_tree(x_1, x_2)$ | $(x_1 \in list(num))$ |
| | $treesort(x_1, x_2)$ | $(x_1 \in list(num))$ |
| lookup | $lookup(x_1, x_2, x_3)$ | $x_1 \in num \wedge x_2 \in dictionary(num, \mathbf{1})$ |
| exp | $exp(x_1, x_2, x_3)$ | $(x_1 \in num \wedge x_2 \in num)$ |
| factorial | $factorial(x_1, x_2)$ | $x_1 \in num$ |

**Table 2.** Precision of the Backward Type Analysis where $\phi_1 = (x_1 \in num \wedge x_4 \in tree(num)) \vee (x_1 \in num \wedge x_2 \in tree(num) \wedge x_3 \in tree(num))$, $\phi_2 = (x_2 \in num \wedge x_3 \in list(num) \wedge x_4 \in list(num)) \vee (x_1 \in list(num) \wedge x_2 \in num)$ and $\phi_3 = (x_1 \in num \wedge x_3 \in tree(num)) \vee (x_1 \in num \wedge x_2 \in tree(num))$.

## 7 Related work

The literature on types in logic programming is vast so this section provides some initial pointers to related work on type analysis. Analysis can be performed either with [1–3, 11, 13, 14, 17, 18] or without [4, 8–10, 12, 20] type definitions provided by the programmer. The former are easy for the programmer to understand whereas the latter are useful in compiler optimisation but can be more difficult for the programmer to interpret.

All the above type analyses propagate type information in the direction of program execution and compute upper approximations to the set of reachable program states. In contrast, the backward type analysis presented in this paper propagates type information in the reverse direction of program execution and computes lower approximations to the set of program states from which the execution will not violate any type assertions.

In a related work [15], the authors present an abstract semantics for backward analysis of logic programs and specialise it to the groundness domain Pos [5, 19] to infer safe modes for queries which ensure that the program will not generate an instantiation error. Analysis is performed by first computing an upper approximation to the success set of the program and then a lower approximation to the set of programs states (substitutions) that will not violate any moding requirement. Both phases of analysis require

the domain to be equipped with a (computable) intuitionistic implication operator. Although [3] presents a type domain that is a complete Heyting algebra, its intuitionistic implication operator is left unspecified.

Pedreschi and Ruggieri [21] develop a calculus of weakest pre-conditions and weakest liberal pre-conditions, the latter of which is essentially a reformulation of Hoare's logic. Weakest liberal pre-conditions are characterised as the greatest fixpoint of a co-continuous operator on the space of interpretations. The work is motivated by, among other things, the desire to infer the absence of ill-typed arithmetic. Our work takes these ideas forward to show how abstract interpretation can infer weakest liberal pre-conditions.

## 8 Summary

A novel backward type analysis for logic programs has been presented. The analysis generalises type checking and is able to infer type constraints for predicates in a program that, if satisfied, guarantee that the execution of the program will not violate any type assertions. The analysis can relieve the programmer from the tedium of declaring all types, infer valuable program documentation and also aid in debugging. Algorithms for the domain operations have been specified that can be translated directly into implementation and a prototype implementation has demonstrated that useful type information can be inferred. Contrary to what was first thought [15], the work also shows that a domain for backward type analysis need not be a complete Heyting algebra.

**Acknowledgement** The work of King was supported, in part, by EPSRC grant GR/MO8769; the work of Lunjin Lu is supported, in part, by the National Science Foundation (CCR-0131862).

## References

1. R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):133–181, 1992.
2. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In *Proceedings of International Static Analysis Symposium*, pages 281–297. Springer-Verlag, 1994.
3. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.
4. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–208, 1995.
5. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–168, 1996.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languages*, pages 238–252. The ACM Press, 1977.
7. P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–189. The MIT Press, 1992.

8. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
9. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of Principles of Programming Languages*, pages 197–209. The ACM Press, 1990.
10. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–155. The MIT Press, 1992.
11. K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In *Proceedings of Conference on Logic Programming*, pages 195–214. Springer, 1988.
12. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(1–4):205–258, 1992.
13. T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 704–707. Morgan Kaufmann, 1985.
14. T. Kanamori and T. Kawamura. Abstract interpretation based on OLDT resolution. *Journal of Logic Programming*, 15(1 & 2):1–30, 1993.
15. A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4&5):517–547, 2002.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
17. L. Lu. A polymorphic type analysis in logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.
18. L. Lu. A precise type analysis of logic programs. In *Proceedings of International Conference on Principles and Practice of Declarative Programming*, pages 214–225. The ACM Press, 2000.
19. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
20. P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298. The IEEE Computer Society Press, 1984.
21. D. Pedreschi and S. Ruggieri. Weakest preconditions for pure Prolog programs. *Information Processing Letters*, 67(3):145–150, 1998.