# An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs

Giorgio Levi, Fausto Spoto

**Abstract**

We apply the methodology of domain refinement to systematically derive domains for type analysis. Domains are built by iterative application of the Heyting completion operator to a given set of basic types. We give a condition on the type system which assures that two steps of iteration are sufficient to reach the fixpoint. Moreover, we provide a general representation for type domains through transfinite formulas. Finally, we show a subset of finite formulas which can be used as a computationally feasible implementation of the domains and we define the corresponding abstract operators.

*Keywords: Abstract interpretation, abstract domains, static analysis, type analysis, logic programming.*

## 1 Introduction

Type analysis for untyped logic programs is useful both to the programmer (for debugging and verification) and to the compiler (for code optimization). This is the motivation of many different proposals for type analysis. It is hard to compare the various techniques in terms of precision, efficiency and generality, because they use different methods and are often based on different assumptions.

There exist type inference techniques similar to those developed for (higher order) functional languages (see, for example, [15, 16]) and techniques inspired by program verification methods [1]. Finally, there are plenty of type analysis techniques based on abstract interpretation [8]. The most relevant feature of abstract interpretation is that the analysis can systematically be derived from the (concrete) semantics and is guaranteed to be correct. The starting point is always the definition of an abstract domain modeling a given type system. In principle, the theory would allow us to systematically derive from the concrete semantics the optimal abstract operations and the corresponding abstract semantics, which is by construction a correct approximation. However, in practical type analysis systems, ad-hoc non-optimal abstract operations and abstract semantics computation algorithms are often considered. The theory is then used to prove the correctness of the construction.

The basic step in every abstract interpretation approach to type analysis is the choice of the abstract domain, which defines how we assign types to terms. A ground type language does not allow one to handle type dependencies [4]. This is the case of [14, 18]. Some type dependencies among different arguments of a procedure can be expressed using type variables in the type language. This is a standard solution, used for instance in [3, 13, 19, 6]. The same solution is used in the framework of regular approximations of the success set in [11]. However, the use of type variables does not allow one to express

G. Levi and F. Spoto are with the Dipartimento di Informatica - Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. E-mail: {levi, spoto}@di.unipi.it. Phone: +39-50-887246 Fax: +39-50-887226

all type dependencies between argument positions. [5] is the only example of an analysis which *explicitly* expresses type dependencies. However, the underlying type language does not contain variables. Hence type dependencies arising from polymorphism can not be handled without an infinite set of dependencies. Moreover, [5] does not consider dependencies between different types, since the analysis is performed separately for every type. Finally, there is no approach which is able to express negative information on types. One of the contributions of this paper is to give a general technique for explicitly expressing general type dependencies and negative information on types.

We achieve this result by pushing forward the *systematic derivation* view of abstract interpretation by applying it to the design of the abstract domain. The systematic derivation of abstract domains can be based on the theory of domain refinement operators [9]. One example of such an operator is the Heyting completion [12], which was recently used to systematically reconstruct various domains for groundness analysis [17] and to show that the domain $\mathcal{POS}$ [2, 7] is indeed optimal.

In this paper, we apply the same methodology to the derivation of new type domains. We start by defining a basic domain of elementary polymorphic types (section 4). We then define a hierarchy of refined domains by iteratively applying the Heyting completion operator (section 5). We prove (in section 6) that, for a large class of elementary types, the refinement procedure derives the optimal domain after two steps of refinement only, as was the case for the groundness domain. The optimal domain can then be viewed as a version of $\mathcal{POS}$ for types, and is similar to the domains of type dependencies defined in [5].

Once we have defined the abstract domain, we are left with the problems of defining a domain representation, suitable for being implemented in an abstract analyzer, and of giving a precise algorithmic definition of the abstract operators. We tackle the above problems in two steps. In the first step we represent type domains by formulas in a fragment of transfinite logic (section 7) and we define correct approximations for the abstract operators (section 8). The results of the first step can effectively be used for type analysis only if the set of types is finite. In the more interesting case of elementary type domains containing infinitely many types, transfinite formulas are not finitely representable. Hence the representation by means of transfinite formulas is introduced essentially to establish some theoretical results to be used in the next step (section 9), where we use a representation in terms of finite formulas with type variables. The resulting domain turns out to be formed by logic programs. Using logic programs to represent abstract domains for logic programs is not new (see, for example, [10]). However, in our experiment, we succeed in providing a formal justification of the construction.

## 2 Preliminaries

### 2.1 Terms and Substitutions

Given a set of variables $V$ and a set of function symbols $\Sigma$ with associated arity, we define $\mathrm{terms}(\Sigma, V)$ as the set of terms built from $V$ and $\Sigma$ in the usual way. In the following sections, we will consider different signatures $\Sigma$ for the set of terms built from the functor symbols of a logic program and for the set of terms built from a type signature. To distinguish these sets, the former will be denoted as $U_V$, while the latter as $\mathrm{terms}(\Sigma, V)$, where $\Sigma$ will be a type signature and $V$ is a set of type variables. We will often abridge $V \cup \{x\}$ as $V \cup x$ and $V - \{x\}$ as $V - x$.

We define $\Theta_{V,U_W}$ as the set of idempotent substitutions $\theta$ such that $\mathrm{dom}(\theta) \subseteq V$ and $\theta(x) \in U_W$ for every $x \in V$. If $\theta \in \Theta_{V,Y_W}$ and $V' \subseteq V$, $\theta|_{V'}(x) = \theta(x)$ if $x \in V'$ and $\theta|_{V'}(x) = x$ if $x \notin V'$. A substitution $\theta \in \Theta_{V,U_W}$ is called grounding for a set of variables

$G \subseteq V$ if and only if $\theta(x)$ is ground for every $x \in G$. For every set of variables $V$, a preordering is defined on substitutions $\theta, \theta' \in \Theta_{V,U_W}$ as $\theta' \leq_V \theta$ if and only if there exists a substitution $\sigma \in \Theta_{W,U_W}$ such that $\theta' = \theta \circ \sigma$. When the set $V$ is clear from the context, we will often remove the subscript $V$. A set $S \subseteq \Theta_{V,U_W}$ of substitutions is downward closed if and only if $\theta \in S$ and $\theta' \leq \theta$ entail $\theta' \in S$. We define $\downarrow \{S\} = \{\theta | \theta \leq \theta' \text{ and } \theta' \in S\}$.

Given $t_1, t_2 \in U_V$, we define $t_1 \leq_V t_2$ if and only if $t_1 = t_2\theta$ for a suitable $\theta \in \Theta_{V,U_V}$. As usual, the subscript will be always removed. $S \subseteq U_V$ is called downward closed if and only if $t \in S$ and $t' \leq t$ entail $t' \in S$.

If $S$ is a set endowed with a relation $\leq$, $\mathcal{P}(S)$ is the set of all the subsets of $S$, while $\mathcal{P}\downarrow(S)$ is the set of all the downward closed (with respect to $\leq$) sets of $S$.

We will often define types as solutions of recursive equations over sets of terms. In such definitions, we will use classical $\lambda$-notation as well as a least fixpoint operator $\mu$.

## 2.2 Abstract interpretation

Abstract interpretation [8] is a theory developed to reason about the abstraction relation between two different semantics. The theory requires these semantics to be defined on domains which are posets. $(\mathcal{C}, \preceq)$ (the concrete domain) is the domain of the concrete semantics, while $(\mathcal{A}, \leq)$ (the abstract domain) is the domain of the abstract semantics. The partial order relations reflect an approximation relation. The two domains are related by a pair of functions $\alpha$ (*abstraction*) and $\gamma$ (*concretization*), which form a Galois connection.

Let $f : \mathcal{C}^n \to \mathcal{C}$ be an operator and assume that $\tilde{f} : \mathcal{A}^n \to \mathcal{A}$ is its abstract counterpart. Then $\tilde{f}$ is *(locally) correct* with respect to $f$ if and only if for all $x_1, \ldots, x_n \in \mathcal{A}$ we have $\alpha(f(\gamma(x_1), \ldots, \gamma(x_n))) \preceq \tilde{f}(x_1, \ldots, x_n)$. According to the theory, for each operator $f$, there exists an optimal (most precise) locally correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$, where $\alpha$ is extended to sets $S \in \mathcal{C}$ defining $\alpha(S) = \wedge_{s \in S}\alpha(s)$.

We briefly recall the equivalence between the Galois insertion and the closure operator approach to the design of abstract domains. Let $\langle L, \leq, \wedge, \vee, \top, \bot \rangle$ be a complete lattice. An upper closure operator on $L$ is an operator $\rho : L \mapsto L$ monotonic, idempotent and extensive. Each closure operator $\rho$ is uniquely determined by the set of its fixpoints, which is its image $\rho(L)$. A set $X \subseteq L$ is the set of fixpoints of a closure operator if and only if $X$ is a Moore family, i.e., $\top \in X$ and $X$ is completely $\wedge$–closed. For any $X \subseteq L$, we denote by $\curlywedge(X)$ the Moore–closure of $X$, i.e., the least subset of $L$ containing $X$ which is a Moore family of $L$.

The complete lattice of all abstract interpretations (identified up to isomorphism) of a domain $\mathcal{C}$ is isomorphic to the complete lattice of upper closure operators on $\mathcal{C}$.

A systematic approach to the development of abstract domains is based on the use of domain refinement operators. Intuitively, given an abstract domain $\mathcal{A}$, a domain refinement operator $R$ yields an abstract domain $R(\mathcal{A})$ which is more precise than $\mathcal{A}$. Classical domain refinement operations are reduced product and disjunctive completion [9].

Heyting completion was proposed in [12] as a powerful domain refinement operation. It allows us to include in a domain the information related to the propagation of the abstract property. Let $L$ be a complete lattice and $a, b \in L$. The relative pseudo–complement (or intuitionistic implication) of $a$ relative to $b$, if it exists, is the unique element $a \to b \in L$ such that for any $x \in L$: $a \wedge_L x \leq b$ if and only if $x \leq_L a \to b$. Relative pseudo–complements, when they exist, are uniquely given by $a \to b = \bigvee_L \{c | a \wedge_L c \leq_L b\}$. A complete lattice $A$ is a complete Heyting algebra if and only if it is relatively pseudo–

complemented, i.e., $a \to b$ exists for every $a, b \in A$. An example of complete Heyting algebra which will be used throughout this paper is $\langle \mathcal{P} \downarrow (\Theta_{V,U_V}), \subseteq, \cap, \cup, \Theta_{V,U_V}, \emptyset \rangle$. Given $a, b \in \mathcal{P} \downarrow (\Theta_{V,U_V})$, the intuitionistic implication $a \to b = \bigcup \{c \in \mathcal{P} \downarrow (\Theta_{V,U_V}) | a \cap c \subseteq b\}$ is also given by $a \to b = \{\theta \in \Theta_{V,U_V} |$ for all $\delta \leq \theta$ if $\delta \in a$ then $\delta \in b\}$. Note that this corresponds exactly to the definition of the concretization of implication in the case of the $\mathcal{POS}$ [2, 7] domain for groundness analysis. The concretization of $x \Rightarrow y$ is the set of substitutions such that every instance which grounds $x$ must also ground $y$. This is not by chance, as shown in [17]. Roughly speaking, an arrow $x \to y$ represents the set of substitutions such that the property of interest propagates from the variable $x$ to the variable $y$ for every possible instance.

Given the domains $\mathcal{D}$, $\mathcal{D}^1_V$ and $\mathcal{D}^2_V$ such that $\mathcal{D}^1_V \subseteq \mathcal{D}$ and $\mathcal{D}^2_V \subseteq \mathcal{D}$, we define

$$\mathcal{D}^1_V \xrightarrow{\curlywedge} \mathcal{D}^2_V = \curlywedge \{d_1 \to d_2 | d_1 \in \mathcal{D}^1_V \text{ and } d_2 \in \mathcal{D}^2_V\} \ .$$

This domain is called the Heyting completion of $\mathcal{D}^1_V$ with respect to $\mathcal{D}^2_V$ and contains all possible dependencies between an element of $\mathcal{D}^1_V$ and an element of $\mathcal{D}^2_V$.

## 3 The concrete domain

Since types are downward closed properties of substitutions, our concrete domain consists of downward closed sets of substitutions. Given a substitution $\theta$, its downward closure represents the set of substitutions which are *compatible* with $\theta$, i.e., which might be obtained by refining $\theta$ by further computation steps. For instance, if the computed substitution at a program point is $\{y \mapsto \mathtt{f}(x)\}$, then, as computation proceeds, the new substitution might be $\{y \mapsto \mathtt{f}(\mathtt{g}(w)), x \mapsto \mathtt{g}(w)\}$. With this interpretation in mind, a downward closed set of substitutions contains exactly all the substitutions which are compatible with the rest of the computation. For instance, if $S_1$ is the (downward closed) set of substitutions which are consistent with a procedure call $p_1$ and $S_2$ is the (downward closed) set of substitutions which are consistent with a procedure call $p_2$, then $S_1 \cap S_2$ is the (downward closed) set of substitutions which are consistent with the calls $p_1, p_2$ and $p_2, p_1$.

We endow downward closed sets of substitutions with two operations:

**unification:** if $S_1, S_2 \subseteq \Theta_{V,U_V}$, then $\text{unify}_V(S_1, S_2) = S_1 \cap S_2$;

**cylindrification:** if $S \subseteq \Theta_{V,U_V}$, its cylindrification with respect to a variable $x \in V$ is

$$\text{cyl}_V(S, x) = \left\{ \theta' \in \Theta_{V,U_V} \;\middle|\; \begin{array}{l} \text{there exist } \theta, \sigma \in \Theta_{V\cup\{n\},U_{V\cup\{n\}}} \\ \text{such that } \theta|_V = \theta', \theta \leq_{V\cup\{n\}} \sigma \text{ and } \sigma \in S[n/x] \end{array} \right\} , \tag{1}$$

where $n$ is a new variable ($n \notin V$), $S[n/x] = \{\sigma[n/x] | \sigma \in S\}$ and $\sigma[n/x](n) = \sigma(x)$, $\sigma[n/x](x) = x$ and $\sigma[n/x](y) = \sigma(y)[n/x]$ if $y \neq x$.

While the definition of unification is the classical one for the case of downward closed sets of substitutions (see for instance [17]), and is justified by the above considerations, it turns out that an *explicit* definition of concrete cylindrification on downward closed sets of substitutions was never given.

Definition (1) should be read as follows. In order to compute the cylindrification of a set $S$ of substitutions, we consider $x$ as a new variable $n$. Then we instantiate all the substitutions in $S$ and we select those instantiations $\theta$ such that $\theta|_V$ does not contain $n$. We try now to get some insight on the meaning of this definition. Consider a procedure

defined as $\mathtt{p(y)} : -\mathtt{y} = \mathtt{f(x)}$.. The set of substitutions which are consistent with the body of the procedure is $S = \downarrow\{\theta\}$, where $\theta = \{y \mapsto \mathtt{f}(x)\}$. Note that $\theta' = \{y \mapsto \mathtt{f}(\mathtt{g}(x))\} \notin S$, as long as we consider idempotent substitutions (or, equivalently, logic programming with occur–check). Consider now the set of substitutions which are consistent with the procedure call $\mathtt{p(y)}$. We have to consider $x$ in the body of the procedure as existentially quantified. Hence $x$ in the body of the procedure is *not* the same $x$ that we have outside the procedure `p`. This means that $\theta'$ is now consistent with the procedure call `p`. For instance, if we make a procedure call $\mathtt{p(y)}$, with a partial computed substitution $\{y \mapsto \mathtt{f}(\mathtt{g}(x))\}$, we obtain success, even when the occur–check is performed. Definition (1) should now be clear. We consider $x$ as a new variable $n$, then we instantiate the new variable in every possible way, including with terms which contain $x$.

The domain of downward closed sets of substitutions will be considered as our concrete domain. We will show in the following sections how some elements of this domain can be selected in order to get a hierarchy of type domains.

## 4 Basic domains for types

In this section we build a domain for type analysis which is able to model elementary monomorphic as well as polymorphic types. We assume a given set of functor symbols $\Sigma$ (the type signature) and a finite set of variables $V = \{x, y, z, \dots\}$ (variables of interest).

Given $\Sigma$, we define a related interpretation $I_{\Sigma,V}$ (denoted in the following simply by $I$). The domain of $I$ is $\mathcal{P}(U_V)$ ($U_V$ could be the set of terms over $V$ induced by the signature of the program). Functors are interpreted in a "user–defined" way. For instance the user can define:

$$\begin{aligned} I(\mathtt{top}) &= U_V \\ I(\mathtt{int}) &= \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \dots\} \\ I(\mathtt{list}) &= \lambda\alpha.\mu\beta.\{[]\} \cup \{[h|t] | h \in \alpha \text{ and } t \in \beta\} \\ I(\mathtt{tree}) &= \lambda\alpha.\mu t.\{\mathtt{void}\} \cup \{\mathtt{tree}(x,l,r) | x \in \alpha, l \in t \text{ and } r \in t\} \ . \end{aligned} \tag{2}$$

Note that in equations (2) `list` stands for the polymorphic list constructor, through the use of the $\lambda$-abstraction. Monomorphic lists can be defined as $I(\mathtt{list}') = \mu l.\{[]\} \cup \{[h|t] | h \in U_V, t \in l\}$. `tree` is the polymorphic tree constructor.

$I$ allows us to evaluate a type $t \in \text{terms}(\Sigma, \emptyset)$ into a set of terms:

$$\begin{aligned} [\![c]\!]I &= I(c) \quad \text{if } c \text{ has arity } 0 \\ [\![f(t_1, \dots, t_n)]\!]I &= [\![f]\!]I([\![t_1]\!]I, \dots, [\![t_n]\!]I) \quad \text{if } f \text{ has arity } n. \end{aligned}$$

According to the above definitions, `tree(int)` contains the terms `void`, `tree(2,void,void)` and `tree(3,tree(1,void,void),void)`. Moreover, $I(\mathtt{list(top)}) = I(\mathtt{list}')$.

A type system is a triple $\langle \Sigma, V, I \rangle$ [1]. Given a variable $x$ and a type $t \in \text{terms}(\Sigma, \emptyset)$, the set $x \in_{V,I} t = \{\theta \in \Theta_{V,U_V} | \theta(x) \in [\![t]\!]I\}$ is a *basic type property*. It represents the set of substitutions which bind $x$ to a term which belongs to the type $t$. The domain of basic types on variables $V$, $\mathcal{TYPE}^0_{\Sigma,V,I}$ (in the following simply $\mathcal{TYPE}^0$), is defined as follows:

$$\mathcal{TYPE}^0 = \bigwedge \{x \in_{V,I} t \, | x \in V, t \in \text{terms}(\Sigma, \emptyset)\} \ .$$

As already mentioned, the Moore family operator selects the least set of downward closed sets of substitutions which contains top and all basic type properties and is closed with

[1] Strictly speaking, we should include $U_V$ in a type system. This information will be left unspecified, in order to simplify the notation.

respect to intersection. The selected set is ordered with respect to the original ordering relation on downward closed sets of substitutions (set inclusion). Note that if $I$ yields downward closed sets for constants and downward closed sets transformers for symbols with arity greater than zero, then $\mathcal{TYPE}^0$ is a set of downward closed sets of substitutions. In the following, we will assume this hypothesis to be satisfied. An abstraction map from the set of downward closed sets of substitutions into $\mathcal{TYPE}^0$ can be defined in a standard way.

$\mathcal{TYPE}^0$ is able to model only conjunction of simple type properties, like for instance "$x$ is an integer and $y$ is a list of integers". It is not able to model type dependencies (directionality). This means that the set $\downarrow\{\theta\}$, where $\theta = \{x \mapsto [h|t]\}$ is abstracted into $\Theta_{V,U_V}$, the set of all the substitutions, in the type system defined by equations (2). $\mathcal{TYPE}^0$ is not able to model the directionality of $\theta$. Actually, if $h$ is of type `int` and $t$ is of type `list(int)`, then $x$ is of type `list(int)`, and vice versa. In the following section we will introduce directionality, by systematically refining $\mathcal{TYPE}^0$ by the Heyting completion operator.

## 5 A hierarchy of domains for directional types

We want now to build a (possibly infinite) hierarchy of type domains as follows:

$$\begin{array}{rcl} \mathcal{TYPE}^0 & = & \curlywedge\{x \in_{V,I} t \,|x \in V, t \in \mathrm{terms}(\Sigma,\emptyset)\} \ , \\ \mathcal{TYPE}^i & = & \mathcal{TYPE}^{i-1} \stackrel{\curlywedge}{\longrightarrow} \mathcal{TYPE}^{i-1} \text{ for } i \geq 1 \ . \end{array} \tag{3}$$

Note that we do not know whether this refinement chain is finite or not. In the following, we will show that under proper conditions on the type system this chain is finite. Namely, it converges at $\mathcal{TYPE}^2$.

Consider now a generic element of $\mathcal{TYPE}^1$. It has the form $(b_1^1 \to b_2^1) \cap \ldots \cap (b_1^n \to b_2^n) \cap \ldots$ where $b_1^i, b_2^i \in \mathcal{TYPE}^0$ for $i \geq 1$. The intersection can be finite as well as infinite. Moreover, every $b_j^i$ has the form $b_j^i = (x_1^{i,j} \in_{V,I} t_1^{i,j}) \cap \ldots \cap (x_{m_{i,j}}^{i,j} \in_{V,I} t_{m_{i,j}}^{i,j}) \cap \ldots$ where, again, the intersection can be finite as well as infinite. Hence $\mathcal{TYPE}^1$ is able to model directional types.

Assume the type system to contain two disjoint types $t_1$ and $t_2$. Namely, we require $[\![t_1]\!]I \cap [\![t_2]\!]I = \emptyset$. In such a case, $\mathcal{TYPE}^1$ would contain the element $(x \in_{V,I} t_1 \cap x \in_{V,I} t_2) \leftarrow y \in_{V,I} t$, where $t$ is a type and $x, y$ are two variables. The meaning of this element is that $y$ is not and will not be eventually bound to a term of type $t$. This is a form of intuitionistic negation. Note that the variable $x$ can be substituted with whatever other variable. Its name is irrelevant. Hence we could simply write $\bot \leftarrow y \in_{V,I} t$, where $\bot$ means failure. Moreover, this argument can be applied even if there are not disjoint types. We just need to add a distinguished type `bot` to the type system, whose interpretation is $I(\mathtt{top}) = \emptyset$. In such a case, $\bot$ would be an abridged form for $x \in_{V,I}$ `bot`. The introduction of negative information seems to be a distinguishing feature of our approach. This information is essentially useless in the case of groundness analysis. For instance, if we add the distinguished type `bot` to the basic domain for groundness containing the unique property `g` such that $I(\mathtt{g}) = \{t \in U_V \,|\, \mathrm{vars}(t) = \emptyset\}$, then an element of the form $\bot \leftarrow x \in_{V,I}$ `g` is $\bot$ itself. This is because every term can always be made ground.

Negative information is extremely powerful for generic type systems. In our concluding example of analysis (section 10) we will show a case where it plays a key role.

It is worth noting that polymorphism is treated in a "ground fashion". For instance, consider the type signature and the interpretation given by equations (2). In $\mathcal{TYPE}^1$ we

are able to say that if $x$ is of type $T$ then $y$ is of type $\texttt{list}(T)$. However, the element representing this information is an infinite intersection of the form

$$\begin{aligned}(x \in_{V,I} \texttt{int} &\rightarrow y \in_{V,I} \texttt{list(int)}) \cap \\ (x \in_{V,I} \texttt{list(int)} &\rightarrow y \in_{V,I} \texttt{list(list(int))}) \cap \\ (x \in_{V,I} \texttt{list(list(int))} &\rightarrow y \in_{V,I} \texttt{list(list(list(int)))}) \cap \ldots\end{aligned}$$

This observation means that the way elements are built in $\mathcal{TYPE}^i$ can not be used directly as a guide for devising a computationally effective representation for $\mathcal{TYPE}^i$. We have to represent a possibly infinite intersection in a finite way. This can be accomplished through the use of type variables in the representation, as it will be shown in section 9.

## 6 Well formed type systems

In this section we investigate a class of type systems which enjoy the property that the refinement chain (3) is finite. Namely, for these type systems the refinement chain converges at the second refinement step to a domain which contains, by construction, all possible type dependencies.

**Definition 1.** A type system $\langle \Sigma, V, I \rangle$ is called *well formed* if and only if, for every $\theta \in \Theta_{V,U_V}$, there exists a grounding substitution $\sigma$ for $V$, such that for every type $t \in \text{terms}(\Sigma, \emptyset)$, $\theta(x) \in [\![t]\!]I$ if and only if $(\theta(x))\sigma \in [\![t]\!]I$.

Roughly speaking, well formed type systems are such that terms which do not belong to types can be instantiated in such a way that they will definitively not belong to those types. It turns out that all sensible type systems are well formed. For instance, the type system INT_LIST_TOP with $\Sigma = \{\texttt{int}, \texttt{list}, \texttt{top}\}$, $I[\texttt{int}] = \mu i.i = \{\texttt{0}\} \cup \{\texttt{s}(j) | j \in i\}$, $I[\texttt{list}] = \lambda x.\mu l.l = \{[]\} \cup \{[h|t] | h \in x, t \in l\}$ and $I[\texttt{top}] = U_V$, is well formed.

Let $b_i, c_j, d_k$ be basic type properties for $i \in I$, $j \in J$ and $k \in K$, where $I, J, K$ are index sets. Let $B = \bigcap_{i \in I} b_i$, $C = \bigcup_{j \in J} c_j$ and $D = \bigcup_{k \in K} d_k$. Assume the following condition

**H1:** $(B \rightarrow C) \rightarrow D = (B \cup D) \cap (C \rightarrow D)$.

holds. Intuitively, hypothesis H1 means that deep arrows can be factorized into simpler arrows. The following results can be proved by extending similar proofs done in [17] for the groundness domains:

1. If $\langle \Sigma, V, I \rangle$ satisfies condition H1, then $\mathcal{TYPE}^2 = \mathcal{TYPE}^i$ for every $i \geq 2$;
2. If $\langle \Sigma, V, I \rangle$ satisfies condition H1, then $\mathcal{TYPE}^2$ can be obtained as implication between conjunctions of basic type properties and disjunctions of basic type properties. Formally, we have:

$$\mathcal{TYPE}^2 = \bigwedge \left\{ a \rightarrow o \,\middle|\, a \in \mathcal{TYPE}^0 \text{ and } o \in \mathcal{OR} \right\} \text{ , where}$$

$$\mathcal{OR} = \left\{ \bigcup_i \{x_i \in_{V,I} t_i\} \,\middle|\, \begin{array}{l} x_i \in V,\ t_i \in \text{terms}(\Sigma, \emptyset) \\ \text{and the union is not empty} \end{array} \right\} .$$

$\mathcal{OR}$ is able to model disjunction of basic type properties, while $\mathcal{TYPE}^2$ is able to model propagation of type information from conjunctions of basic type properties to disjunctions of basic type properties.

The importance of well formed type systems is that they satisfy condition H1:

**Proposition 2.** *Let $\langle \Sigma, V, I \rangle$ be a well formed type system. Then condition H1 holds.*

## 7 A hierarchy of intermediate representations

In this section we consider a hierarchy of representations for the above defined type domains. They are intermediate because they are not adequate for a direct implementation in an analysis tool. However, they will be used to obtain some theoretical results, and to devise an effective representation for our type domains.

We start by defining a fragment of transfinite logic which will be called $\mathcal{LOG}_{\Sigma,V}$ (in the following simply $\mathcal{LOG}$). It is the least set such that $false, true \in \mathcal{LOG}$, $(x \in t) \in \mathcal{LOG}$, for $x \in V$ and $t \in \text{terms}(\Sigma, \emptyset)$, if $S$ is a (possibly infinite) subset of $\mathcal{LOG}$ then $\wedge(S) \in \mathcal{LOG}$ and $\vee(S) \in \mathcal{LOG}$, if $s_1, s_2 \in \mathcal{LOG}$ then $s_1 \Rightarrow s_2 \in \mathcal{LOG}$ and if $s \in \mathcal{LOG}$ then $\neg s \in \mathcal{LOG}$.

Given a substitution $\theta \in \Theta_{V,U_V}$ and an interpretation $I$, we define the interpretation of a formula of $\mathcal{LOG}$ as follows:

$$
\begin{array}{rcl}
[\![false]\!]^I_\theta = 0 \quad \text{and} \quad [\![true]\!]^I_\theta = 1 & & \\
[\![x \in t]\!]^I_\theta = 1 & \text{iff} & \theta(x) \in [\![t]\!]I \text{ , for } x \in V \text{ and } t \in \text{terms}(\Sigma, \emptyset), \\
[\![\wedge(S)]\!]^I_\theta = 1 & \text{iff} & [\![s]\!]^I_\theta = 1 \text{ for every } s \in S, \\
[\![\vee(S)]\!]^I_\theta = 1 & \text{iff} & \text{there exists } s \in S \text{ such that } [\![s]\!]^I_\theta = 1 \text{ ,} \\
[\![s_1 \Rightarrow s_2]\!]^I_\theta = 1 & \text{iff} & \text{if } [\![s_1]\!]^I_\theta = 1 \text{ then } [\![s_2]\!]^I_\theta = 1, \\
[\![\neg s]\!]^I_\theta = 1 & \text{iff} & [\![s]\!]^I_\theta = 0 \text{ .}
\end{array}
$$

An interpretation $I$ induces an equivalence relation $\equiv_I$ on formulas. Namely, $\phi_1 \equiv_I \phi_2$ if and only if for every $\theta \in \Theta_{V,U_V}$, $[\![\phi_1]\!]^I_\theta = 1$ entails $[\![\phi_2]\!]^I_\theta = 1$ and vice versa. This equivalence will be called *logical* equivalence in the following.

Given $\phi \in \mathcal{LOG}$ and an interpretation $I$, we define the map $\gamma_I : \mathcal{LOG} \mapsto \mathcal{P}{\downarrow}\,(\Theta_{V,U_V})$ as $\gamma_I(\phi) = \{\theta \in \Theta_{V,U_V} | \text{for all } \sigma \leq \theta, [\![\phi]\!]^I_\sigma = 1\}$. $\gamma_I$ induces an equivalence relation on formulas defined as $\phi_1 \equiv_{\gamma_I} \phi_2$ if and only if $\gamma_I(\phi_1) = \gamma_I(\phi_2)$. This equivalence will be called $\gamma$-equivalence in the following. Note that if $\phi_1$ and $\phi_2$ are logically equivalent then they are $\gamma$-equivalent, by definition of $\gamma$. However, the converse, in general, does not hold.

We define now a hierarchy of representations as follows:

$$
\begin{array}{rcl}
\mathcal{LOG}^0 & = & \left\{ \wedge(S) \;\middle|\; \begin{array}{l} S = \bigcup_i \{x_i \in t_i\} \text{ ,} \\ x_i \in V,\ t_i \in \text{terms}(\Sigma, \emptyset) \end{array} \right\} /_{\equiv_I} \text{ ,} \\
\mathcal{LOG}^1 & = & \left\{ \wedge(S) \;\middle|\; \begin{array}{l} S = \bigcup_j \{s^1_j \Rightarrow s^2_j\} \text{ ,} \\ s^1_j, s^2_j \in \mathcal{LOG}^0 \end{array} \right\} /_{\equiv_I} \text{ ,} \\
\mathcal{LOG}^2 & = & \left\{ \wedge(S) \;\middle|\; \begin{array}{l} S = \cup_j \{a_j \Rightarrow o_j\} \text{ ,} \\ [a_j]_{\equiv_I} \in \mathcal{LOG}^0, \\ [o_j]_{\equiv_I} \in \mathcal{LOG}_{\mathcal{OR}} \end{array} \right\} /_{\equiv_I} \text{ , where} \\
\mathcal{LOG}_{\mathcal{OR}} & = & \left\{ \vee(S) \;\middle|\; \begin{array}{l} S = \bigcup_i \{x_i \in t_i\} \text{ is non empty,} \\ x_i \in V,\ t_i \in \text{terms}(\Sigma, \emptyset) \end{array} \right\} /_{\equiv_I} \text{ .}
\end{array}
$$

The following inclusions can easily be proved: $\mathcal{LOG}^0 \subseteq \mathcal{LOG}^1 \subseteq \mathcal{LOG}^2$.

We extend the concretization map on equivalence classes as $\gamma_I([\phi]_{\equiv_I}) = \gamma_I(\phi)$. This definition is well given because logical equivalence entails $\gamma$-equivalence. Note that $\gamma_I$ is monotonic. In the following, a formula will stand for its (logical) equivalence class.

It can be shown that every element of $\mathcal{TYPE}^0$ is the image through $\gamma_I$ of an element of $\mathcal{LOG}^0$, that every element of $\mathcal{TYPE}^1$ is the image through $\gamma_I$ of an element of $\mathcal{LOG}^1$ and that every element of $\mathcal{TYPE}^2$ is the image through $\gamma_I$ of an element of $\mathcal{LOG}^2$. Note

that this last result holds only for the particular form of the elements of $\mathcal{LOG}^2$. Hence in these three cases $\gamma_I$ is onto. We have already shown that it is monotonic. However, we know that it is not always one to one. If we make the assumption:

**H2:** $\gamma_I$ is one to one, i.e., logical equivalence is $\gamma$-equivalence;

then we conclude that the following isomorphisms hold: $\mathcal{TYPE}^0 \approx \mathcal{LOG}^0$, $\mathcal{TYPE}^1 \approx \mathcal{LOG}^1$ and $\mathcal{TYPE}^2 \approx \mathcal{LOG}^2$. The following result shows that well formed type systems satisfy condition H2. Hence the representations of this section are isomorphic to the type domains of section 5 for well formed type systems.

**Proposition 3.** *Let $\langle \Sigma, V, I \rangle$ be a well formed type system. Then condition H2 holds.*

## 8 Abstract operators on transfinite logic

In this section we give an explicit definition of correct abstract operators on the domains of transfinite logic formulas.

Let us first note that if $S_1$ is represented by $\phi_1$ and $S_2$ is represented by $\phi_2$, then $S_1 \cap S_2$ is represented by $\phi_1 \wedge \phi_2$. Moreover, this is the best possible approximation.

We consider now the approximation of the concrete cylindrification operator. Let $\mathcal{T} = \curlywedge\{[\![t]\!]I | t \in \text{terms}(\Sigma, \emptyset)\}$ be the least Moore family (with respect to set intersection) which contains all the types. The substitution of a type $t \in \mathcal{T}$ for a variable $x$ in a formula $\phi$ is defined as follows:

$$
\begin{array}{rcl}
false[t/x] & = & false \\
true[t/x] & = & true \\
(x \in t')[t/x] & = & \begin{cases} true & \text{if } t \subseteq [\![t']\!]I \\ false & \\ \text{otherwise} & \end{cases}
\end{array}
\qquad
\begin{array}{rcl}
(y \in t')[t/x] & = & (y \in t') \\
 & & \text{if } x \neq y \\
\wedge(S)[t/x] & = & \wedge(\{s[t/x] | s \in S\}) \\
\vee(S)[t/x] & = & \vee(\{s[t/x] | s \in S\}) \\
(s_1 \Rightarrow s_2)[t/x] & = & s_1[t/x] \Rightarrow s_2[t/x] \\
(\neg s)[t/x] & = & \neg(s[t/x]) \ .
\end{array}
$$

We define the abstract cylindrification operator on the logical domain as $\exists_x \phi = \vee(\{\phi[t/x] | t \in \mathcal{T}'\})$, where $\mathcal{T}' \subseteq \mathcal{T}$ is the set of types which are the most specific type of some term. Note that this subset is not empty because $\mathcal{T}$ is a Moore family. Hence every term has a most specific type. This definition is similar to the Schröder elimination used in the case of groundness analysis [2, 7]. Note that $x$ does not occur in $\exists_x \phi$. For generic type systems, $\exists_x$ is not a correct cylindrification operator. However, it turns out that $\exists_x$ is a correct abstract operator with respect to concrete cylindrification in the case of well formed type systems.

## 9 Logic programs as finite representations of type domains

Transfinite formulas can be used as a computationally effective representation domain if the set of types is finite. In such a case the set of transfinite formulas is isomorphic to a set of finite formulas and the abstract operators on the domain can correctly be approximated by effective algorithms. For instance, the operator $\exists_x$ becomes an algorithm for computing cylindrification. Even the equivalence test between two formulas becomes effective, though very expensive, being a classical $\mathcal{NP}$–complete problem.

A more interesting case is when we deal with an infinite set of basic types, i.e., when terms$(\Sigma, \emptyset)$ is infinite. In such a case, transfinite formulas are not finitely representable. However, the full power of transfinite formulas is seldom useful for our purposes. For instance, to express the relationship between the variables in the binding $x = [h|t]$ we must

write an infinite conjunction: $\bigwedge_{t\in\mathrm{terms}(\Sigma,\emptyset)} (x \in \mathtt{list}(t) \iff h \in t \wedge t \in \mathtt{list}(t))$. However, this could be expressed more compactly using type variables as $x \in \mathtt{list}(T) \iff h \in T \wedge t \in \mathtt{list}(T)$.

In this section, we introduce a domain of finite formulas with type variables and the corresponding abstract operators.

Let $V' = \{X, Y, Z, \dots\}$ be an infinite set of type variables. Type variables are denoted by uppercase letters to distinguish them from the variables of interest $V = \{x, y, z, \dots\}$. $\mathcal{REP}_{\Sigma,V,V',I}$ (in the following, simply $\mathcal{REP}$) is the least set of first order formulas containing $x_i \in t_i$ for $x_i \in V$ and $t_i \in \mathrm{terms}(\Sigma, V')$ and closed with respect to $\wedge$ and $\Rightarrow$, modulo the equivalence relation $\equiv_{\mathcal{REP}}$ defined as follows: $\phi_1 \equiv_{\mathcal{REP}} \phi_2$ if and only if $\gamma_{\mathcal{REP}}(\phi_1) \equiv_I \gamma_{\mathcal{REP}}(\phi_2)$, where $\gamma_{\mathcal{REP}}(\phi(X_1, \dots, X_2)) = \bigwedge_{t_1,\dots,t_n\in\mathrm{terms}(\Sigma,\emptyset)} \phi[t_1/X_1, \dots, t_n/X_n]$ (where $\phi(X_1, \dots, X_n)$ means that the type variables contained in $\phi$ are *exactly* $X_1, \dots, X_n$). Intuitively, the last formula is the transfinite formula represented by the finite formula $\phi$. Note that there exist transfinite formulas which can not be represented this way. Moreover, $\mathcal{REP}$ is not closed with respect to infinite $\wedge$. This means that we are not guaranteed to have optimal operators. Finally, in general $\mathcal{REP}$ is not finite and not even noetherian. For example, the instance of $\mathcal{REP}$ induced by the type system INT_LIST_TOP is not noetherian, because there exists an infinite chain $X \in \mathtt{top}$, $X \in \mathtt{list}(\mathtt{top})$, $X \in \mathtt{list}(\mathtt{list}(\mathtt{top}))$, and so on.

In order to make $\mathcal{REP}$ finite, we consider the approximate domain $\mathcal{REP}^k$, $k \in \mathbb{N}$, $k > 0$, whose formulas contain constraints of the form $x_i \in t_i$, such that $t_i \in \mathrm{terms}(\Sigma, V')$ and the depth of $t_i$ is less than or equal to $k$. We define $\gamma_{\mathcal{REP}^k} = \gamma_{\mathcal{REP}}$ and $\equiv_{\mathcal{REP}^k} = \equiv_{\mathcal{REP}}$ restricted to formulas in $\mathcal{REP}^k$.

Considering only constraints with bounded term depth does not boil down to the case of a finite set of types. In fact, type variables are free to assume any value, with arbitrary depth. As the concluding example will show, this restriction on the constraints does not introduce a big loss in precision, thanks to the use of type variables.

We give now algorithmic definitions for the two abstract operators and for the abstraction map.

**Abstract unification.** It can be shown that $\wedge$ is correct with respect to intersection of downward closed sets of substitutions. Given $\phi_1, \phi_2 \in \mathcal{REP}^k$, we have $\gamma_I(\gamma_{\mathcal{REP}^k}(\phi_1 \wedge \phi_2)) = \gamma_I(\gamma_{\mathcal{REP}^k}(\phi_1)) \cap \gamma_I(\gamma_{\mathcal{REP}^k}(\phi_2))$.

As a consequence, we have that $(x \in \mathtt{list}(T) \Leftarrow y \in T) \wedge (z \in T \Leftarrow w \in T) \equiv_{\mathcal{REP}^k} (x \in \mathtt{list}(T) \Leftarrow y \in T) \wedge (z \in D \Leftarrow w \in D)$. Actually, two occurrences of the same type variable in two different implications can be replaced by different type variables.

This suggests an interesting interpretation for formulas in $\mathcal{REP}^k$. Since $(A_1 \wedge \dots \wedge A_n) \Leftarrow B$ can be equivalently rewritten as $(A_1 \Leftarrow B) \wedge \dots \wedge (A_n \Leftarrow B)$, the elements of $\mathcal{REP}^k$ can be viewed as definite Horn clauses. We only need to interpret a constraint of the form $x \in t$ as an atom $x(t)$, where the variables of interest become predicate symbols. For instance, the abstract constraint $(x \in \mathtt{list}(T) \Leftarrow (y \in T \wedge z \in \mathtt{list}(T))) \wedge (y \in T \Leftarrow x \in \mathtt{list}(T)) \wedge z \in \mathtt{list}(T)$ can be seen as the logic program

$$\begin{array}{l} \mathtt{x(list(T))} :- \mathtt{y(T)}, \mathtt{z(list(T))}. \\ \mathtt{y(T)} :- \mathtt{x(list(T))}. \\ \mathtt{z(list(T))}. \end{array} \qquad (4)$$

The above remark is important when we look for a correct approximation of the cylindrification operator. We already have a non effective definition of the approximation: given $\phi$ and a variable $x$, we compute $\gamma_{\mathcal{REP}^k}(\phi)$, then we apply Schröder elimination. This

definition is not adequate because Schröder elimination destroys the clause structure of the elements of $\mathcal{REP}^k$, which is extremely important for representing in a compact way a possibly infinite quantification on types. If the elements of $\mathcal{REP}^k$ are represented as logic programs, cylindrification of an element $P$ with respect to a variable $x$ means computing a program $P'$ which expresses the same dependencies among the predicate symbols (variables of interest) different from $x$ in the same way as $P$ does, but does not contain $x$ anymore. The simplest technique for removing a predicate from a program is unfolding. Given two clauses $H_1 \Leftarrow B_1 \wedge \ldots \wedge B_n$ and $H_2 \Leftarrow C_1 \wedge \ldots \wedge C_m$, if $\theta = \mathrm{mgu}(B_i, H_2)$ exists, one of their unfoldings is $(H_1 \Leftarrow B_1 \wedge \ldots \wedge B_{i-1} \wedge C_1 \wedge \ldots \wedge C_m \wedge B_{i+1} \wedge \ldots \wedge B_n)\theta$. It can be shown that the unfoldings of two clauses are logical consequences of them. Note, however, that the unfolding of two clauses whose terms have depth less than or equal to $k$ can contain a term with depth greater than $k$.

**Abstract cylindrification.** We define the operator $\exists_x^{\mathcal{REP}^k}$ through the unfolding operation. Any element $P \in \mathcal{REP}^k$ is viewed as a set of clauses. In order to compute $\exists_x^{\mathcal{REP}^k} P$, we perform the following three steps:

1. we add to $P$ all the possible unfoldings of any clause containing $x$ in the body with any clause containing $x$ in the head. Let $P'$ be the resulting program;
2. we remove from $P'$ all the clauses containing $x$ thus obtaining $P''$;
3. we remove from $P''$ all the clauses which contain terms with depth greater than $k$, thus obtaining $\exists_x^{\mathcal{REP}^k} P$.

For instance, the abstract cylindrification of program (4) with respect to the variable `z` and for $k = 2$ is the program:

```
x(list(T)):-y(T).
y(T):-x(list(T)).
```

while the abstract cylindrification of the same program with respect to the variable `x` and for $k = 2$ is the program:

```
y(T):-y(T),z(list(T)).
z(list(T)).
```

which is $\equiv_I$-equivalent to the program `z(list(T))`..

Note that the algorithm for computing the abstract cylindrification introduces a loss in precision in the last two steps. In order to improve the precision, we can repeat the first step to decrease the number of clauses which contain $x$ in the body. However, the loss in precision of the third step can not be avoided. It can be shown that $\exists_x^{\mathcal{REP}^k}$ is indeed correct with respect to concrete cylindrification for well-formed type systems.

The algorithm for cylindrification uses concrete unification between type terms. This is not related to the unification operator of the domain. It is simply a consequence of the use of logic programs as abstract domains. However, since types are partially ordered with respect to a subtyping relation (for instance: `int` $\subseteq$ `top`), the unification procedure used in the unfolding step might be too coarse. For instance, if we have a clause whose head is $x \in$ `list(int)` and we try to unfold it in the body of a clause containing $x \in$ `list(top)`, the unification procedure fails. Actually, unfolding should be allowed because if $x$ is a list of integers then it is even a list of generic terms. Similarly, if we have a clause whose body contains $x \in T$, we can remove $x \in T$ from the body and instantiate the resulting clause with the substitution $\{T \mapsto$ `top`$\}$. This is correct because every term is always in `top`. This means that we could improve the precision of the cylindrification operator using *in its algorithmic definition* a unification procedure which embeds subtyping information.

**Abstraction map.** We define now a correct approximation of the abstraction of $\downarrow\{\theta\}$. We only need to find a formula $\phi \in \mathcal{RECP}^k$, such that $\downarrow\{\theta\} \subseteq \gamma_I\gamma_{\mathcal{RECP}^k}(\phi)$. Let $\alpha(\theta)$ be one such a formula. Assume we have a procedure `type`, which, for every term $u \in U_V$ with $\mathrm{vars}(u) = \{x_1, \ldots, x_n\}$, behaves as $u \overset{\texttt{type}}{\rightsquigarrow} \{\langle t^1, t^1_{x_1}, \ldots, t^1_{x_n}\rangle, \ldots, \langle t^m, t^m_{x_1}, \ldots, t^m_{x_n}\rangle\}$, where $t^i$ and $t^i_{x_j}$ belong to $\mathrm{terms}(\Sigma, V')$, i.e., they are types possibly containing type variables. We require that, for every $\sigma \in \Theta_{V,U_V}$, $u\sigma \in [\![t^i\sigma']\!]I$ for a suitable $\sigma' \in \Theta_{V',\mathrm{terms}(\Sigma,\emptyset)}$ grounding for $V'$, if and only if, for all $j = 1, \ldots, n$, $x_j\sigma \in [\![t^i_{x_j}\sigma']\!]I$.

Roughly speaking, `type`$(u)$ computes a finite set of possible types for $u$, and, for each possible type, it computes some necessary and sufficient conditions on the variables of $u$ in order for $u$ to belong to the type. Note that a straightforward definition of the `type` procedure can be automatically derived from the definition of types and that this definition is compositional with respect to addition of new types to the type system.

Given the procedure `type` and a substitution $\theta$, such that $x \in \mathrm{dom}(\theta)$, we define $\alpha_x(\theta) = \bigwedge_{i=1}^m \left(x \in t^i \iff \left(x_1 \in t^i_{x_1} \wedge \ldots \wedge x_n \in t^i_{x_n}\right)\right)$, where $\mathrm{vars}(\theta(x)) = \{x_1, \ldots, x_n\}$ and $\texttt{type}(\theta(x)) = \{\langle t^1, t^1_{x_1}, \ldots, t^1_{x_n}\rangle, \ldots, \langle t^m, t^m_{x_1}, \ldots, t^m_{x_n}\rangle\}$. Finally, we define $\alpha(\theta) = \bigwedge_{x\in\mathrm{dom}(\theta)} \alpha_x(\theta)$. It can be proved that, for every substitution $\theta \in \Theta_{V,U_V}$, $\downarrow\{\theta\} \subseteq \gamma_I\gamma_{\mathcal{RECP}^k}(\alpha(\theta))$.

For instance, if we consider the top type, integers, and polymorphic lists, we can implement `type` as a Prolog procedure `type(Term,Type)` which enumerates all possible types `Term` can take. Moreover, the variables of `Term` are bound to types to represent necessary and sufficient conditions for `Term` to belong to `Type`. For instance, `type([H|T],Type)` yields a computed answer substitution $\{\texttt{Type} \mapsto \texttt{list(S)}, \texttt{H} \mapsto \texttt{S}, \texttt{T} \mapsto \texttt{list(S)}\}$, meaning that `[H|T]` has type `list(S)` if and only if `H` has type `S` and `T` has type `list(S)`.

| meta-clause |
|---|
| `type(X,S):-var(X),!,X=S.` |
| the whole universe $U_V$ |
| `type(X,top).` |
| integers: $\mu i.i = \{\texttt{0}\} \cup \{\texttt{s}(t) \mid t \in i\}$ |
| $\texttt{type}(\texttt{X}, \texttt{int}) :- \texttt{X} = \texttt{0}.$ |
| $\texttt{type}(\texttt{X}, \texttt{int}) :- \texttt{X} = \texttt{s}(\texttt{I}), \texttt{type}(\texttt{I}, \texttt{int}).$ |
| polymorphic lists: $\lambda s.\mu l.l = \{[]\} \cup \{[h\mid t] \mid h \in s \text{ and } t \in l\}$ |
| $\texttt{type}(\texttt{X}, \texttt{list}(\texttt{S})) :- \texttt{X} = [].$ |
| $\texttt{type}(\texttt{X}, \texttt{list}(\texttt{S})) :- \texttt{X} = [\texttt{H}\mid\texttt{T}], \texttt{type}(\texttt{H}, \texttt{S}), \texttt{type}(\texttt{T}, \texttt{list}(\texttt{S})).$ |

The above algorithmic definition of the abstraction map can be improved by extracting from a substitution even the negative information that it contains. We just need to modify `type`. We can assume that `type`$(t)$ contains even pairs of the form $\langle t_i, \bot\rangle$, meaning that the term $t$ can never belong to the type $t_i$. For instance, $[H|T]$ can never be an integer, while $\texttt{s}(X)$ can. As a consequence, we define $\alpha_x(\theta) = \bigwedge_{i=1}^m \left(x \in t^i \iff \left(x_1 \in t^i_{x_1} \wedge \ldots \wedge x_n \in t^i_{x_n}\right)\right) \wedge \bigwedge_{i=1}^k (\bot \Leftarrow (t')^i)$, where $\mathrm{vars}(\theta(x)) = \{x_1, \ldots, x_n\}$ and $\texttt{type}(\theta(x)) = \{\langle t^1, t^1_{x_1}, \ldots, t^1_{x_n}\rangle, \ldots, \langle t^m, t^m_{x_1}, \ldots, t^m_{x_n}\rangle, \langle (t')^1, \bot\rangle, \ldots, \langle (t')^k, \bot\rangle\}$.

## 10 An example

We implemented in Prolog an abstract analyzer which uses the $\mathcal{RECP}^k$ abstract domain and which is parametric with respect to a given set of types. In this section, we show

how it behaves on the following program which computes the derivative of an expression involving the variable `x`:

```
int(0).
int(s(I)):-int(I).

der(x,s(0)).
der(X,0):-int(X).
der(X*Y,(DX*Y)+(X*DY)):-der(X,DX),der(Y,DY).
der(X+Y,DX+DY):-der(X,DX),der(Y,DY).
der(-(X),-(DX)):-der(X,DX).
der(X-Y,DX-DY):-der(X,DX),der(Y,DY).
der(X^K,DK*K*(X^(K-s(0)))):-der(K,DK).
der(exp(X),DX*exp(X)):-der(X,DX).
der(sin(X),DX*cos(X)):-der(X,DX).
der(cos(X),-(DX*sin(X))):-der(X,DX).
```

The types used in the analysis are the top type, denoted by `top`, integers, denoted by `int`, generic expressions on `x`, denoted by `expr`, and algebraic expressions on `x`, i.e., expressions on `x` which do not involve exponentiation or trigonometric functions, denoted by `algebraic`. We compute the abstract fixpoint of the above program through our analyzer. Then we evaluate the query (mode) `der(algebraic,top)` in the abstract fixpoint. We get the following set of constraints, where `var0` and `var1` stand for the first and for the second argument of the predicate `der`, respectively.

```
constraint 1:
bot:-var0(int).
var0(algebraic).
var0(expr).
var1(algebraic).
var1(expr).
var1(int).
var1(top).

constraint 2:
var0(algebraic).
var0(expr).
var0(int).
var1(algebraic).
var1(expr).
var1(int).
var1(top).

constraint 3:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr).
var1(expr).
var1(top).

constraint 4:
bot:-var0(int).
bot:-var1(int).
var0(algebraic).
var0(expr).
var1(algebraic).
var1(expr).
var1(top).

constraint 5:
bot:-var0(int).
bot:-var1(int).
var0(algebraic).
var0(expr).
var0(expr):-var1(expr).
var1(algebraic):-var0(algebraic).
var1(expr):-var0(expr).
var1(top).
```

```
constraint 6:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr):-var1(expr).
var1(expr):-var0(expr).
var1(top).

constraint 7:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr):-var1(expr).
var1(expr):-var0(expr).
var1(top).

constraint 8:
bot:-var0(algebraic).
bot:-var0(int).
bot:-var1(algebraic).
bot:-var1(int).
var0(algebraic).
var0(expr).
var1(expr).
var1(top).
```

Every constraint is a logic program. If the predicate `bot` is derivable from the logic program, then the constraint can be dropped since it is not satisfiable. In the case at hand, constraints 3, 6, 7 and 8 are dropped. ¿From the remaining four constraints, we derive the fact `var1(expr)`. This means that the second argument is bound to an expression. More interestingly, the same constraints allow us to derive the fact `var1(algebraic)`, i.e., the second argument is bound to an algebraic expression. Roughly speaking, our analyzer concludes that the derivative of an algebraic expression is an algebraic expression too. Note that this result was possible only through the use of negative information. Namely, the dropped constraints do not allow to derive the fact `var1(algebraic)`. Hence only if we remove them we can obtain the desired result.

## 11 Conclusions

We presented a polymorphic type analysis scheme based on abstract interpretation. The construction of the abstract domains is made through a formal methodology, namely domain refinement starting from a simple domain of elementary types. We have introduced an isomorphic representation of the elements of the domain by means of transfinite formulas. We have given sufficient conditions on the type systems which assure that the resulting type domains and type representations with transfinite formulas enjoy some desirable properties, namely factorization of deep type implications, identity between logical equivalence on the representation and concretization equality and correctness of the Schröder elimination procedure on the representation. Finally, we have shown how a finite domain of finite formulas (represented by definite Horn clauses) with type variables can be selected in order to make the analysis effective.

We are left with several important open problems. Some of the problems we are currently investigating are:

- the optimality of Schröder elimination;
- the automatic derivation and the use of subtyping information in the algorithm for abstract cylindrification;
- the relation, in terms of precision, between the domain of finite formulas with type variables and the domain of transfinite formulas;
- the definition of a better approximation of the abstraction map into formulas in $\mathcal{REP}^k$;
- the definition of a generic implementation based on a *type specification language* (as we have shown, a type analyzer can be constructed from a type specification in an automatic way);
- the comparison of our technique to other existing techniques for type analysis based on abstract interpretation. The domain refinement methodology we use should be useful to compare existing domains, as already shown in the case of groundness analysis.

# References


[1] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[2] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proc. Static Analysis Symposium, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag, 1994.

[3] R. Barbuti and R. Giacobazzi. A Bottom-up Polymorphic Type Inference in Logic Programming. *Science of Computer Programming*, 19(3):281–313, 1992.

[4] J. Boye. *Directional Types in Logic Programming*. PhD thesis, Linköping University (Sweden), 1996.

[5] M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In *Proc. of the first International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.

[6] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI–unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, June 1996. Extended version to appear in Theoretical Computer Science.

[7] A. Cortesi, G. Filè, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, 1996.

[8] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[9] G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view on abstract domain design. *ACM Computing Surveys*, 28(2):333–336, 1996.

[10] T. Frühwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[11] J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximation of Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press.

[12] R. Giacobazzi and F. Scozzari. Intuitionistic implication in abstract interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and Programs PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 175–189. Springer-Verlag, 1997.

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

[14] T. Kanomori and K. Horiuchi. Polymorphic Type Inference in Prolog by Abstract Interpretation. In *Logic Programming 87- Tokyo*, volume 315 of *Lecture Notes in Computer Science*, pages 195–214. Springer-Verlag, 1988.

[15] M. Kifer and J. Wu. A first–order theory of types and polymorphism in logic programming. In *IEEE Symposium on logic in Computer Science*, 1991.

[16] C. Pyo and U. S. Reddy. Inference of Polymorphic Types for Logic Programming. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 1115–1132. The MIT Press, 1989.

[17] F. Scozzari. Logical optimality of groundness analysis. In P. Van Hentenryck, editor, *Proceedings of International Static Analysis Symposium, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997.

[18] J. Xu and D. S. Warren. A Type Inference System for Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 604–619. The MIT Press, 1988.

[19] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10:125–135, 1991.