More Precise Yet Efficient Type Inference for Logic Programs

Claudio Vaucheret and Francisco Bueno

Technical University of Madrid (UPM), Spain
{claudio,bueno}@fi.upm.es

Abstract. Type analyses of logic programs which aim at inferring the types of the program being analyzed are presented in a unified abstract interpretation-based framework. This covers most classical abstract interpretation-based type analyzers for logic programs, built on either top-down or bottom-up interpretation of the program. In this setting, we discuss the widening operator, arguably a crucial one. We present a new widening which is more precise than those previously proposed. Practical results with our analysis domain are also presented, showing that it also allows for efficient analysis.

1 Introduction

In type analyses, the widening operation has much influence in the results. If the widening is too aggressive in making approximations then the analysis results may be too imprecise. On the other hand, if it is not sufficiently aggressive then the analysis may become too inefficient.

Widening operators are aimed at identifying the recursive structure of the types being inferred. All widening operators already proposed in the literature are based on locating type nodes with the same functors, which are possible sources of recursion. However, they disregard whether such nodes come in fact from a recursive structure in the program or not. This may originate an unnecessary loss of precision, since the widening result may then impose a recursive structure on the resulting type in argument positions where the concrete program is in fact not recursive. We propose a widening operator to try to remedy this problem.

We present our widening operator for regular type inference in an analysis framework based on abstract interpretation of the program. In order for the paper to be self contained, we first revisit regular types (Section 2) and, in particular, deterministic ones. We focus on deterministic types for ease of presentation; however, there is nothing in our widening which prevents it to be applicable also to non-deterministic types. The abstract interpretation framework is set up in Section 3. Section 4 reviews previous widenings in the literature, and Section 5 presents ours. In Section 6 experimental results are presented, and Section 7 concludes and discusses future work.

2 Regular Types

A *regular type* [3] is a type representing a class of terms that can be described by a regular term grammar. A *regular term grammar*, or grammar for short, describes a set of finite terms constructed from a finite alphabet $\mathcal{F}$ of *ranked function symbols* or *functors*. A grammar $G = (S, \mathcal{T}, \mathcal{F}, \mathcal{R})$ consists of a set of non-terminal symbols $\mathcal{T}$, one distinguished symbol $S \in \mathcal{T}$, and a finite set $\mathcal{R}$ of productions $T \longrightarrow rhs$, where $T \in \mathcal{T}$ is a non-terminal and the right hand side rhs is either a non-terminal or a term $f(T_1, \ldots, T_n)$ constructed from an n-ary function symbol $f \in \mathcal{F}$ and n non-terminals.

The non-terminals $\mathcal{T}$ are *types* describing (ground) terms built from the functors in $\mathcal{F}$. The concretization $\gamma(T)$ of a non-terminal T is the set of terms derivable from its productions, that is,

$$\gamma(T) = \bigcup_{(T \longrightarrow rhs) \in \mathcal{R}} \gamma(rhs)$$

$$\gamma(f(T_1, \ldots, T_n)) = \{f(t_1, \ldots, t_n) \mid t_i \in \gamma(T_i)\}$$

The types of interest are each defined by one grammar: each T_i is defined by a grammar $(T_i, \mathcal{T}_i, \mathcal{F}, \mathcal{R}_i)$, so that for any two types of interest T_1 and T_2 on $\mathcal{F}$, $\mathcal{T}_1 \cap \mathcal{T}_2 = \emptyset$. Sometimes, we will be interested in types defined by non-terminals of a grammar $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$ other than the distinguished non-terminal T. This is formalized by defining a type $T_i \in \mathcal{T}$ as the grammar

$$(T_i, \{T \in \mathcal{T} \mid T_i \xrightarrow{reach^*}_{\mathcal{R}} T\}, \mathcal{F}, \{(T \longrightarrow rhs) \in \mathcal{R} \mid T_i \xrightarrow{reach^*}_{\mathcal{R}} T\}) \qquad (1)$$

where all the non-terminals are renamed apart, $\xrightarrow{reach^*}_{\mathcal{R}}$ is the reflexive and transitive closure of $\xrightarrow{reach}_{\mathcal{R}}$ and

$$T_i \xrightarrow{reach}_{\mathcal{R}} T_j \text{ iff } T_i \longrightarrow_{\mathcal{R}} T_j \text{ or } T_i \longrightarrow_{\mathcal{R}} f(\ldots, T_j, \ldots).$$

A grammar is in *normal form* if none of the right hand sides are non-terminals. A particular class of grammars are deterministic ones. A grammar is *deterministic* if it is in normal form and for each non-terminal T the function symbols are all distinct in the right hand sides of the productions for T.

Deterministic grammars are less expressive than non-deterministic ones. Deterministic grammars can only express sets of terms which are *tuple-distributive*; informally speaking, which are "closed under exchange of arguments". I.e., if the set contains two terms of the same functor, then it also contains terms with the same principal functor obtained by exchanging subterms of the previous two terms in the same argument positions. Basically, no dependencies between arguments of a term can be expressed with deterministic grammars.

Example 1. Consider the type T denoting the set $\{f(a, b), f(c, d)\}$, which is non-deterministic,

$$\begin{array}{lll} T \longrightarrow f(A, B) & A \longrightarrow a & C \longrightarrow c \\ T \longrightarrow f(C, D) & B \longrightarrow b & D \longrightarrow d \end{array}$$

A deterministic type T' with a concretization which included $\gamma(T)$ would also have to include $\{f(c,b), f(a,d)\}$, that is,

$$\begin{array}{lll} T' \longrightarrow f(AC,BD) & AC \longrightarrow a & BD \longrightarrow b \\ & AC \longrightarrow c & BD \longrightarrow d \end{array}$$

To facilitate the presentation non-terminals with a single production will often be "inlined" and multiple right hand sides combined so that T above will be written $T \longrightarrow f(a,b) \mid f(c,d)$ and T' as

$$T' \longrightarrow f(AC,BD) \;\; AC \longrightarrow a \mid c \;\; BD \longrightarrow b \mid d$$

To be able to describe terms containing numbers and variables we introduce two distinguished symbols **num** and **any**, plus an additional $\perp$. The concretization of **num** is the set of all numbers, the concretization of **any** is the set of all terms (including variables), and the concretization of $\perp$ is the empty set of terms. These symbols are non-terminals but they are considered terminals to the effect of regarding a grammar as deterministic.

Let $\mathcal{G}$ be the set of all grammars, if T_1, T_2 belong to $\mathcal{G}$, the relation $T_1 \equiv T_2 \Leftrightarrow \gamma(T_1) = \gamma(T_2)$ is an equivalence relation. The quotient set $\mathcal{G}/\equiv$ is a complete lattice with top element **any** and bottom element $\perp$ based on the relation of *containment*, or type *inclusion*: for every $\overline{T_1}, \overline{T_2} \in \mathcal{G}/\equiv$, $\overline{T_1} \sqsubseteq \overline{T_2} \Leftrightarrow \gamma(T_1) \subseteq \gamma(T_2)$. We will denote $\overline{T_i}$ simply by T_i.

The least upper bound is given by type *union*, $(T_1 \sqcup T_2)$, and the greatest lower bound by type *intersection*, $(T_1 \sqcap T_2)$ [3]. It can be shown that intersection describes term unification:

$$t_1^* \subseteq \gamma(T_1) \wedge t_2^* \subseteq \gamma(T_2) \wedge t_1\theta = t_2\theta \Rightarrow (t_1\theta)^* \subseteq \gamma(T_1 \sqcap T_2)$$

where t^* denotes the set of ground terms which are instances of the term t.

3 Abstract Domain for Type Inference

In an abstract interpretation-based type analysis, a type is used as an abstract description of a set of terms. Given variables of interest $\{x_1, \ldots, x_n\}$, any substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ can be approximated by an *abstract substitution* $\{x_1 \leftarrow T_{x_1}, \ldots, x_n \leftarrow T_{x_n}\}$ where $t_i \in \gamma(T_{x_i})$ and each type $T_{x_i} \in \mathcal{G}/\equiv$. We will write abstract substitutions as tuples $\langle T_1, \ldots, T_n \rangle$, and sometimes also abbreviate a tuple simply as T^n.

Concretization is lifted up to abstract substitutions straightforwardly,

$$\gamma(\langle T_1, \ldots, T_n \rangle) = \{ \; \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\} \mid t_i \in \gamma(T_i) \; \}$$

as well as the equivalence relation $\equiv$. Additionally, we consider a distinguished abstract substitution $\perp$ as a representative of any $\langle T_1, \ldots, T_n \rangle$ such that there is $T_i = \perp$. Of course, $\gamma(\perp) = \emptyset$.

An ordering on the domain is obtained as the natural element-wise extension of the ordering on types:

$$\begin{aligned} \bot &\sqsubseteq T^n \\ \langle T_1, \ldots, T_n \rangle &\not\sqsubseteq \bot \\ \langle T_1, \ldots, T_n \rangle &\sqsubseteq \langle T'_1, \ldots, T'_n \rangle \iff \forall_{1 \leq i \leq n} T_i \sqsubseteq T'_i \end{aligned}$$

The domain is a lattice with bottom element $\bot$ and top element $\langle T_1, \ldots, T_n \rangle$ such that $T_1 = \ldots = T_n =$ **any**. The greatest lower bound and least upper bound domain operations are lifted also element-wise, as follows,

$$\begin{aligned} \bot \sqcup T^n = T^n \sqcup \bot &= T^n \\ \langle T_1, \ldots, T_n \rangle \sqcup \langle T'_1, \ldots, T'_n \rangle &= \langle T_1 \sqcup T'_1, \ldots, T_n \sqcup T'_n \rangle \\ \bot \sqcap T^n = T^n \sqcap \bot &= \bot \\ \langle T_1, \ldots, T_n \rangle \sqcap \langle T'_1, \ldots, T'_n \rangle &= \langle T_1 \sqcap T'_1, \ldots, T_n \sqcap T'_n \rangle \end{aligned}$$

Using the adjoint α of γ as abstraction function, it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion, where Θ is the domain of concrete substitutions and Ω that of abstract substitutions.

The following abstract unification operator can be shown to approximate the concrete one. Let $x = t$ be a concrete unification equation, with x a variable,t any term, and T^n the current abstract substitution, and let y_j, $j = 1, \ldots, m$ be the variables of t, the new abstract substitution is:

$$amgu(T^n, x = t) = T^n[T_x/T'_x, T_{y_1}/T'_{y_1}, \ldots, T_{y_m}/T'_{y_m}] \qquad (2)$$

with each T replaced by T' in the tuple, $T'_x = T_x \sqcap t\mu$, $\mu = \{y_1 \leftarrow T_{y_1}, \ldots, y_m \leftarrow T_{y_m}\}$, and $solve(t, T'_x) = \{y_1 = T'_{y_1}, \ldots, y_m = T'_{y_m}\}$, a set of equations that define the types of the variables of a term $t \in \gamma(T'_x)$, obtained as:

$$solve(t, T) = \begin{cases} \{t = T\} & \text{if } t \text{ is a variable} \\ \bigcup_{T \longrightarrow f(T_1, \ldots, T_n)} \; \bigcup_{i=1,\ldots,n} solve(t_i, T_i) & \text{if } t \text{ is } f(t_1, \ldots, t_n) \end{cases}$$

In this abstract interpretation-based setting, analysis with a monotonic semantic function can be easily shown correct. However, it is not guaranteed to terminate, since Ω has infinite ascending chains. To guarantee termination, a widening operator is required.

Example 2. Consider the following program which defines the regular type lists of lists of numbers:

```
list_of_lists([]).                    num_list([]).
list_of_lists([L|Ls]):-               num_list([N|Xs]):-
        num_list(L),                          number(N),
        list_of_lists(Ls).                    num_list(Xs).
```

For the argument of `num_list`, without a widening operator, an analysis would obtain the following first three approximations:

$$T_0 \longrightarrow [] \quad T_1 \longrightarrow [] \mid .(\mathbf{num}, T_0) \quad T_2 \longrightarrow [] \mid .(\mathbf{num}, T_1)$$

where each T_i represents a list of i numbers. Analysis will never terminate, since it would keep on obtaining a new type representing a list with one more number. A widening operator would be required that over-approximates some type T_l to something like

$$T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

which is the expected type, and allows termination of the analysis.

4 Widenings

The widening operation is required to guarantee that an analysis terminates when the abstract domain has infinite ascending chains as is the case of regular types.

Functor Widening This is probably the simplest widening operator which still keeps information from the recursive structure of the program that "produces" the corresponding terms. The idea behind it is to create a type and a production for each functor symbol in the original type. All arguments of the function symbols are replaced with the new types [10].

Example 3. Consider predicate `list_of_lists` of Example 3.2, its argument should ideally have the following type:

$$T_{ll} \longrightarrow [] \mid .(T_l, T_{ll}) \quad T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

but the functor widening will yield

$$T \longrightarrow [] \mid \mathbf{num} \mid .(T, T)$$

Type Jungle Widening A type jungle is a grammar where each functor always has the same arguments. It was originally proposed as a finite type domain [9] , since in a domain where all grammars are of the type jungle class all ascending chains are finite. However, it can be used as a subdomain to provide a widening operator.

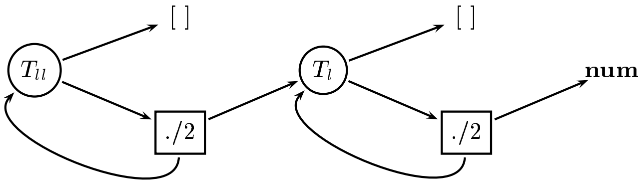
Example 4. Applying this widening to the previous type T_{ll}, the following will be obtained:

$$T \longrightarrow [] \mid .(T_1, T) \quad T_1 \longrightarrow [] \mid \mathbf{num} \mid .(T_1, T)$$

Note that this widening is strictly more precise than the functor widening. In the example, the new type captures the upper level of lists, but it loses precision when describing the type of the list elements. This is due to the restriction of forcing functors to always have the same arguments.

Shortening A grammar can be seen as a graph where the nodes correspond to the non-terminals (or-nodes) and to the right hand sides of productions (and-nodes), and the edges correspond to the production relation or the relation between a functor and its arguments in a right hand side of a production. Given an or-node, its *principal functors* are the functors appearing in its children nodes.

Example 5. The type T_{ll} of the previous examples can be seen as the graph:



Gallagher and de Waal [6] defined a widening which avoids having two or-nodes, which have the same principal functors, connected by a path. If two such nodes exist, they are replaced by their least upper bound.

Example 6. In the above example graph, nodes T_{ll} and T_l have the same principal functors ([] and .) so that they are replaced, yielding:

$$T \longrightarrow [\,] \mid .(T_1, T) \quad T_1 \longrightarrow [\,] \mid \mathbf{num} \mid .(\mathbf{num}, T)$$

Note the precision improvement with respect to the result in the previous example. Note also that still the result is imprecise.

Restricted Shortening Saglam and Gallagher [11] propose a more precise variant of the previous widening. Shortening is restricted so that two or-nodes T and T' which are connected by a path from T to T' and have the same principal functors are replaced only if $T' \sqsubseteq T$. If this is the case, only T' needs be replaced, since the least upper bound is T.

Example 7. Continuing previous examples, since nodes T_{ll} and T_l have the same principal functors but $T_l \not\sqsubseteq T_{ll}$, the widening operation will make no change. In this case, the most precise type is achieved.

Note, however, that restricted shortening does not guarantee termination in general (and thus, it is not, strictly speaking, a widening). There are cases in which analysis may not terminate using only this widening operator [10].

Depth Widening Janssens and Bruynooghe [8] proposed a type analysis in which the widening effect is achieved by a "pruning" of the type depth up to a certain bound. A parameter k establishes the maximum number of occurrences of a functor in-depth in a type. The idea is similar to the well-known depth-k abstraction for term structure analysis. The resulting type analysis uses normal restricted type graphs, which are basically deterministic types satisfying the depth limit. Obviously, precision of this analysis depends on the value of the parameter k.

Example 8. The widening of our previous type T_{ll} with k=1 will yield the same result than the functor widening (Example 4.3), whereas with k=2 will yield the same result as restricted shortening (Example 4.7).

Topological Clash Widening Van Hentenryck et al. [12] proposed the first widening operator that takes into account two consecutive approximations to the type being inferred. After merging the two —i.e., calculating their least upper bound, the result is compared with the previous approximation to try to "guess" where the type is growing. This is done by locating *topological clashes*: functors that differ or appear at different depth in each type graph. The clashes are resolved by replacing them with the recently calculated least upper bound.

Example 9. Consider the program:

```
sorted([]).
sorted([_X]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

and the moment during analysis when the final widening is performed. The resulting type for the argument of `sorted/1` is the one on the left below for the first two clauses, and the one on the right for the last one:

$$T_0 \longrightarrow [] \mid .(\mathbf{any}, []) \quad \begin{array}{l} T_1 \longrightarrow .(\mathbf{num}, .(\mathbf{num}, T_l)) \\ T_l \longrightarrow [] \mid .(\mathbf{num}, T_l) \end{array}$$

Their least upper bound is T_u on the left below, which exhibits a clash with T_0 in the second argument of functor ./2. Thus, the result of widening is T_s:

$$T_u \longrightarrow [] \mid .(\mathbf{any}, T_l) \quad T_s \longrightarrow [] \mid .(\mathbf{any}, T_s)$$

All widening operators are based on locating recursive structures in the type definitions where there are nodes with the same functors. This may originate an unnecessary loss of precision, since the widening may impose a recursive structure on the resulting type in argument positions where the concrete program is in fact not recursive. In the following section we present a new widening operator that tries to remedy this problem.

5 Structural Type Widening

In this section we define an extended domain for type analysis which incorporates a widening operator aimed at improving the precision of the analysis. The domain is defined so as to keep track of information on the program structure, so that recursion on the types produced by the analysis is imposed by the widening operator only in the cases where it corresponds to a recursive structure in the program being analyzed. To this end, type names will be used.

A *type name* is roughly a (distinguished) non-terminal that represents a type produced during the analysis. Type names are created for each variable in each argument of each variant of each program atom for each predicate (note how this is different from, for example, set-based analyses [1], where variants are not taken into account).

Type names provide information on how types are being formed from other types during analysis. This makes it possible to precisely identify places where to impose recursion on the types: in a subterm of the type which happens to refer to the name of that type. To this end, type names contain references to the position of its constituent types. To determine positions, selectors are used, as defined below.

Definition 1 (selector). *Define t/s, the subterm of a concrete term t referenced by a* selector *s, inductively as follows. The empty selector ϵ refers to the term t, that is, $t/\epsilon = t$. If $t/s = t'$, t' is a compound term $f(t_1, \ldots, t_i, \ldots, t_n)$ (where f is an n-ary function symbol) then $t/s \cdot (f.i) = t_i$, $1 \leq i \leq n$.*

For every two selectors s, p, if $t/s = t'$ and if t'/p exists then $t/s \cdot p = t'/p$. The initial ϵ of a non-empty selector will often be omitted, so $\epsilon \cdot p$ will be written simply as p.

We define a set of type names $\mathcal{N}$ such that $\mathcal{N} \cap \mathcal{G} = \emptyset$ and a set $2^{\mathcal{N} \times \mathcal{G}}$ of relations $\mathcal{X} \in 2^{\mathcal{N} \times \mathcal{G}}$ between type names and types, of the form $\mathcal{X} \subseteq \mathcal{N} \times \mathcal{G}$.

Definition 2 (label). *Let $\mathcal{X}$ a relation between type names and types. Given a type name N, a selector s, and a type name N', a tuple $\langle s, N' \rangle$ is a* label *of N iff $(N, T) \in \mathcal{X}$, $(N', T') \in \mathcal{X}$, and $T' \sqsubseteq T/s$.*

Labels of a type name N indicate subterms of the type T defining N where other type names occur.

Example 10. Let a relation $\mathcal{X}$ such that $\{(A, T_1), (B, T_2)\} \subseteq \mathcal{X}$, and let grammars $(T_1, \mathcal{T}_1, \mathcal{F}, \mathcal{R}_1)$ and $(T_2, \mathcal{T}_2, \mathcal{F}, \mathcal{R}_2)$, such that the only rule for T_1 is $(T_1 \longrightarrow f(b)) \in \mathcal{R}_1$ and $(T_2 \longrightarrow g(c, T_3)) \in \mathcal{R}_2$, $(T_3 \longrightarrow b \mid f(b)) \in \mathcal{R}_2$. Consider a label $\langle (g.2), A \rangle$ of B. We have that $T_1 \sqsubseteq T_2/(g.2) = T_3$.

Definition 3 (type descriptor). *Let $\mathcal{G}$ a set of types (regular term grammars), $\mathcal{N}$ a set of type names, and $\mathcal{X} \subseteq \mathcal{N} \times \mathcal{G}$. A* type descriptor *is a tuple (N, E, T) where $N \in \mathcal{N}$, $T \in \mathcal{G}$, $(N, T) \in \mathcal{X}$, and E is a set of labels of N.*

In the new domain, type descriptors will be used instead of types. Let $\mathcal{D}$ be the set of all type descriptors from given sets of types $\mathcal{G}$ and of type names $\mathcal{N}$. Concretization is defined as $\gamma((N, E, T)) = \gamma(T)$. The domain ordering and operations on $\mathcal{D}$ are the same as on $\mathcal{G}$ except for type names. In this case, they have to take into account the possible labels of the type name.[1]

Inclusion $(N_1, E_1, T_1) \sqsubseteq (N_2, E_2, T_2) \Leftrightarrow T_1 \sqsubseteq T_2 \wedge E_1 \subseteq E_2$.

Union $(N, E, T) = (N_1, E_1, T_1) \sqcup (N_2, E_2, T_2) \Leftrightarrow T = T_1 \sqcup T_2 \wedge E = E_1 \cup E_2$.

Intersection $(N, E, T) = (N_1, E_1, T_1) \sqcap (N_2, E_2, T_2) \Leftrightarrow T = T_1 \sqcap T_2 \wedge E = E_1 \cup E_2$.

[1] Note that these operations do not manipulate the type names: they are assigned independently during analysis. In particular, the name N of the type resulting from union and intersection is always a new name.

Again, we may be interested in types defined by non-terminals other than the distinguished non-terminal T of a grammar $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$. A type descriptor (N_i, E_i, T_i), where $T_i \in \mathcal{T}$, is formally defined from (N, E, T) as follows: T_i is the grammar of Equation 1, N_i is a new type name, and

$$E_i = \{\langle p, N' \rangle \mid \langle s \cdot p, N' \rangle \in E \wedge T/s = T_i\}.$$

Abstract substitutions for variables of interest $\{x_1, \ldots, x_n\}$ are now defined as tuples of the form $\langle (N_1, E_1, T_{x_1}), \ldots, (N_n, E_n, T_{x_n}) \rangle$. Concretization and the domain ordering and operations are lifted to abstract substitutions element-wise, in the same way as in Section 3, including the widening operator defined below. If now Ω is the domain of type descriptors, it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion, where α is the adjoin of γ. Abstract unification is defined as in Equation 2, but using type descriptors instead of types. During unification, all type names in the "input" abstract substitution T^n to *amgu* are preserved; in the labels, the selectors for those names are changed so as to refer to the resulting type graph instead of to that of T^n.

Definition 4 (structural widening). *The widening between an approximation T_2 to type name N and a previous approximation T_1 to N is $(N, E_1, T_1) \bigtriangledown (N, E_2, T_2) = (N, E_1 \cup E_2, T)$, such that T is defined by $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$ where $\mathcal{T} = \{T_i \mid T \longrightarrow^*_{\mathcal{R}} T_i\}$, and $\mathcal{R}$ is obtained by the following algorithm:*

$T' := T_1 \sqcup T_2$ defined by $(T', \mathcal{T}', \mathcal{F}, \mathcal{R}')$
$\mathcal{S} := \{s \mid (s, N) \in E_1 \cup E_2\}$
$Seen := \emptyset$
`for each` $(T' \longrightarrow f(A_1, \ldots, A_n)) \in \mathcal{R}'$ `add to` $\mathcal{R}$ `production`
    $T \longrightarrow f(\mathtt{widen}(A_1, \mathcal{R}', (f.1)), \ldots, \mathtt{widen}(A_n, \mathcal{R}', (f.n)))$

$\mathtt{widen}(N, \mathcal{R}', Sel)$:
    `if` $N =$ **any** `return` **any**
    `if` $\exists M \langle N, M \rangle \in Seen$ `return` M
    `let` M `a new non-terminal`
    $Seen$ `:=` $Seen \cup \{\langle N, M \rangle\}$
    `for each` $(N \longrightarrow f(A_1, \ldots, A_n)) \in \mathcal{R}'$ `add to` $\mathcal{R}$ `production`
        $M \longrightarrow f(\mathtt{widen}(A_1, \mathcal{R}', Sel \cdot (f.1)), \ldots, \mathtt{widen}(A_n, \mathcal{R}', Sel \cdot (f.n)))$
    `if` Sel $\in \mathcal{S}$ `then`
        `add to` $\mathcal{R}$ `production` $M \longrightarrow T$
    `return` M

Structural widening basically identifies subterms of the new type $T_1 \sqcup T_2$ where a reference to the type N being widened appears, and makes this "self-reference" explicit in the definition of the new type. Note that the widening operation starts with the least upper bound and, basically, adds new grammar rules to that type. Therefore, the result is always a correct approximation of such an upper bound. This justifies its correctness. Moreover, this approach based on type names is potentially more precise than any of the previous widening operators discussed, as the following examples show:

Example 11. Consider program `sorted` in Example 4.9. A top-down analysis with topological clash was roughly described there. Let us now look at analysis using restricted shortening. The resulting type happens to be the same one.

Analysis of program atom `sorted([Y|L])` approximates variable `Y` always as **num**, both in the calls and in the successes. The first two success approximations for variable `L` are $[]$ and $.(\mathbf{num}, [])$. Their lub (and widening) is:

$$T_1 \longrightarrow [] \mid .(\mathbf{num}, [])$$

The next approximation to the type of `L` is $.(\mathbf{num}, T_1)$. Its lub with T_1 is $T_2 \longrightarrow [] \mid .(\mathbf{num}, T_1)$, and since T_2 and T_1 have the same functors, and T_1 is included in T_2, the widening of T_2 is:

$$T_3 \longrightarrow [] \mid .(\mathbf{num}, T_3)$$

i.e., list of numbers. The next approximation to the type of `L` is $.(\mathbf{num}, T_3)$ (i.e., a list with at least one number). It is included in T_3, so fixpoint is reached.

The success of principal goal `sorted(X)` is approximated after analyzing the two non-recursive clauses by $T_4 \longrightarrow [] \mid .(\mathbf{any}, [])$. Analysis of the third clause yields $.(\mathbf{num}, .(\mathbf{num}, T_3))$. Its lub with T_4 is $T_5 \longrightarrow [] \mid .(\mathbf{any}, T_3)$. The widening of T_5 finds that T_5 and T_3 have the same functors and $T_3 \sqsubseteq T_5$, since $\mathbf{num} \sqsubseteq \mathbf{any}$. Thus, the result of widening is:

$$T_6 \longrightarrow [] \mid .(\mathbf{any}, T_6)$$

i.e., list of terms. This is the final result after one more iteration. Note that the information about successes where the tail of lists of length greater than one is a list of numbers is lost.

Let us now consider structural widening. Analysis of atom `sorted([Y|L])` always approximates the type of `Y` by $(N_{13}, \emptyset, \mathbf{num})$. For variable `L` the two first approximations are $(N_{14}, \emptyset, [])$ and $(N_{14}, E_{14}, .(\mathbf{num}, []))$, where the set of labels is $E_{14} = \{\ (\texttt{'.'}.1, N_{13}),\ (\texttt{'.'}.2, N_{14})\ \}$. The result of widening is (N_{14}, E_{14}, T_1) where T_1 is defined as:

$$T_1 \longrightarrow [] \mid .(\mathbf{num}, T_1)$$

i.e., list of numbers. This is the final result after one more iteration.

The success of principal goal `sorted(X)` is approximated after analyzing the two non-recursive clauses by $(N_3, \emptyset, T_2)$ where $T_2 \longrightarrow [] \mid .(\mathbf{any}, [])$. Analysis of the third clause yields $(N_3, E_3, .(\mathbf{num}, .(\mathbf{num}, T_1)))$, where

$$E_3 = \{\ (\texttt{'.'}.2 \cdot \texttt{'.'}.1, N_{13}),\ (\texttt{'.'}.2 \cdot \texttt{'.'}.2, N_{14})\ \}$$

Its widening with the previous approximation T_2 is (N_3, E_3, T_3), where

$$T_3 \longrightarrow [] \mid .(\mathbf{any}, T_1)$$

which amounts to their lub, since the widening operator does not produce any change, because N_3 is not among its own labels. Therefore, the final result, after one more iteration, is T_3, where indeed lists of length greater than one have a tail which is a list of numbers.

However, structural widening does not guarantee termination. It is effective as long as the new approximation is built from the previous approximation of the type being inferred. This case is identified, in essence, by locating a reference to the type name of the previous approximation within the definition of the new one. However, there are contrived cases in which a type is constructed during analysis which loses the reference to the previous approximation. In these cases, a more restrictive widening has to be applied to guarantee termination.

Example 12. Consider the program:

```
main:- p(a).      p(a).                      q(a,f(a)).
                  p(X):- q(X,Y), p(Y).       q(f(Z),f(L)):- q(Z,L).
```

The calling substitution for atom `p(Y)` is the sequence

$$T_1 \longrightarrow f(a) \quad T_2 \longrightarrow f(f(a)) \quad T_3 \longrightarrow f(f(f(a))) \quad \ldots$$

whereas the type $T \longrightarrow f(a) \mid f(T)$ correctly describes such calls. However, the analysis is not able to infer such a type.

The problem in the above example is that none of the approximations T_i contains a reference to the previous approximation. This is originated in the program fact for predicate `q/2` which causes the loss of the reference to the previous approximation because of the double occurrence of constant `a`.

In our analysis, termination is guaranteed by a bound on the number of times the widening operation can be applied to a type name. A counter is associated to each type name, so that when the bound is reached a more restrictive widening that guarantees termination is applied.

6 Analysis Results

We have implemented analyses based on most of the widenings discussed in this paper, including structural widening. The implementation is in Prolog and has been incorporated to the CiaoPP system [7], which uses the top-down analysis algorithm of PLAI. The analysis of [6], based on regular approximations, which uses a bottom-up algorithm, is also incorporated into the system. This analysis uses shortening. We want to compare the top-down and bottom-up approaches with the same widening and similar implementation technology,[2] as well as the precision and efficiency, within the same analysis framework, of the widening operators previously discussed.

We have used two sets of benchmark programs: the one used in the PLAI framework and that used in the GAIA [2] framework. A summary of the benchmarking follows. The analysis times in miliseconds are shown in Table 1. The first column (`rul`) is for the regular approximation analysis and the other three for the PLAI-based analyses: column `short` for shortening, column `clash` for topological clash, and column `struct` for structural widening.

[2] Similar in the programming technique. Of course, the regular approximation method is rather different from the method of program interpretation on an abstract domain: Evaluating this difference is part of the aim of the comparison.

Program	`rul`	`short`	`clash`	`struct`
aiakl	568	469	529	900
bid	1480	2209	2529	4730
boyer	3450	3890	4989	9629
browse	758	380	389	539
cs_o	3840	1889	2689	2580
cs_r	18549	10720	24479	19560
disj_r	4468	1819	6399	2440
gabriel	1549	1430	1870	1760
grammar	330	160	160	190
hanoiapp	620	719	1889	1150
kalah_r	1520	79	79	89
mmatrix	310	190	209	119
occur	380	219	330	289
palin	590	840	980	850
pg	839	2020	2980	3990
plan	1138	819	960	1009
progeom	979	1840	2530	3640
qsort	310	590	659	680
qsortapp	369	1000	2898	1210
queens	329	179	190	180
query	720	360	370	410
serialize	478	810	969	899
witt	2929	4890	1399	1169
zebra	560	3490	14958	12830

Table 1. Timing results

Table 2 shows results in terms of precision. The precision of `struct` is never improved by any of the others. The improved precision of `struct` has been measured as follows. The left subcolumns under `rul`, `short`, and `clash` show the number of types with a more precise definition inferred by `struct`. The right subcolumns show the number of types where the previous ones appear (and are thus, also, more precise). The former are types directly inferred from program predicates; the latter are types which are defined from the former, due to the data flow in the program.

The following conclusions can be drawn from the tables. First, the regular approximation approach seems to behave better in terms of efficiency than the program interpretation approach, at least for the bigger programs. This conclusion, however, has to be taken with some care, since the current implementation of `rul` performs some caching of the type grammars that the PLAI-based analysis does not. This should be subject of a more thorough evaluation, which is out of the scope of this paper. The fact that it improves in bigger programs seems to suggest that the effect of this caching is most surely not negligible.

Regarding the analyses based on program interpretation, it can be concluded that the better the precision the worse the efficiency: `short` takes less than

Program	rul		short		clash	
aiakl	1	1	1	1		
bid	9	12	9	12		
cs_o	4	18	4	18	2	9
cs_r	4	28	4	28	2	19
disj_r	6	13	6	13		
mmatrix	2	2	2	2		
occur	1	1	1	1		
palin	2	4	2	4		
pg	1	1	1	1		
qsort	1	1	1	1		
serialize	2	4	2	4		
zebra	3	3	3	3	1	1

Table 2. Precision results

`clash`, and this one takes less than `struct`; this one is more precise than `clash`, which is more precise than `short`. This conclusion seems evident at first sight, but it is not: in analysis, an improvement in precision can very well trigger an improvement in efficiency. This can also be seen in the tables in some cases, the most significant probably being `zebra`. Overall, one can arguably conclude that the efficiency loss found is not a high price in exchange for the gain in precision.

We have also carried out another test. For practical purposes, the CiaoPP system includes a back-end to the analysis that simplifies the types inferred, in the sense that equivalent types are identified, so that they are then reduced to a single type. This facilitates the interpretation of the output. It is the case that the structural widening includes certain amount of type simplification, so that the analysis creates less different types which are in fact equivalent. For this reason, we have included the same tests as above, but adding now the times taken in the back-end simplification phase.

The times including the simplification are shown in table 3. The columns read as before. It can be seen that in this case structural widening outperforms all of the other analyses, except, in some cases, `rul`.

It also can be observed that `rul` behaves usually better than `short` also when simplification is included. This seems to suggest that incorporating our widening into the regular approximation approach would probably give the best results in practice.[3]

7 Conclusions

We have presented a new widening operator on regular types within an abstract interpretation-based characterization of type inference. The idea behind it is similar to set-based analyses [4, 1] in that we assign and fix type names, but it

[3] This, however, may not be trivial. It is subject for future work.

Program	rul	short	clash	struct
aiakl	697	3009	3738	1409
bid	2899	31278	35949	15259
boyer	19620	201169	206917	92117
browse	987	2848	2987	1698
cs_o	11958	17389	32959	4878
cs_r	50760	303430	238788	30169
disj_r	6508	18598	26077	6408
gabriel	2098	13388	22379	5208
grammar	759	3169	3169	1279
hanoiapp	840	3988	13738	3378
kalah_r	2069	1187	1188	888
mmatrix	757	1769	2078	488
occur	530	1647	2628	767
palin	997	8520	11878	2180
pg	1349	15380	22870	7370
plan	1587	6167	6559	2288
progeom	1358	12800	17598	6679
qsort	520	3439	4168	1409
qsortapp	569	7789	9669	2900
queens	457	1128	1138	429
query	1627	22458	22788	11818
serialize	937	8429	11957	2217
witt	3438	188419	42699	25709
zebra	717	55100	189949	44540

Table 3. Timing results (including simplification)

is applied here with more generality. The most comparable aproach among the set-based analyses would be [5]. It can be seen as a generalization of the idea of "guessing" the growth of the types during analysis which is behind [12]. Instead of guessing, our technique determines exactly where the type is growing. The resulting widening operator has been presented on deterministic regular types. However, its extension to non-deterministic regular types should be straightforward.

Our operator is more precise than previous approaches, but it is still efficient. This has been shown with (preliminary) practical results. However, it does not guarantee termination. We are currently working on the non-termination problem. A moded type domain will help in this. The idea is to enhance abstract unification so that it is able to identify the "transference" of type names from the input to the output types, so that the names are not dropped. This will remedy the problem of Example 12 and, hopefully, allow us to prove termination of analyses with the proposed widening operator.

Finally, this work has revealed two issues that may be worth investigating for practical purposes: the impact on the efficiency of analysis of the different implementation techniques for different analysis methods, on one hand, and of the simplification of types, on the other hand.

Acknowledgements

We would like to thank John Gallagher for very useful discussions, and Pedro López for his help with the implementation and the availability of his library on type manipulation. This work has been partially supported by Spanish MCYT project EDIPIA TIC99-1151 and by EU funded projects AMOS IST-2001-34717 and ASAP IST-2001-38059.

References

1. W. Charatonik, A. Podelski, and J.-M. Talbot. Paths vs. Trees in Set-based Program Analysis. In *Principles of Programming Languages*, pages 330–338. ACM Press, January 2000.
2. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
3. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
4. T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In *Proc. LICS'91*, pages 300–309, 1991.
5. J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, LNCS, pages 243–261. Springer-Verlag, January 2002.
6. J.P. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
7. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
8. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.
9. T. Lindgren and P. Mildner. The Impact of Structure Analysis on Prolog Compilation. Technical Report 140, Computing Science Departament, Uppsala University, April 1997.
10. P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study.* PhD thesis, Computing Science Department - Uppsala University, 1999.
11. H. Saglam and J. Gallagher. Approximating Logic Programs Using Types and Regular Descriptions. Technical Report CSTR-94-19, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1994.
12. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.