Type Analysis of Prolog Using Type Graphs

P. Van Hentenryck[1]
A. Cortesi[2]
B. Le Charlier[3]

Technical Report No. CS-93-52

November, 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]University of Venezia Via Torino 155, 1-30170 Mestre-VE (Italy)
[3]University of Mamur 21 rue Grandgagnage, B-5000 Numar (Belgium)

Type Analysis of Prolog Using Type Graphs[1]

P. Van Hentenryck
Brown University
Box 1910
Providence RI 02912 (USA)
pvh@cs.brown.edu

A. Cortesi
University of Venezia
Via Torino 155
I-30170 Mestre-VE (Italy)
cortesi@moo.dsi.unive.it

B. Le Charlier
University of Namur
21 rue Grandgagnage
B-5000 Namur (Belgium)
ble@info.fundp.ac.be

Abstract

Type analyis of Prolog is of primary importance for high-performance compilers, since type information may lead to better indexing and to sophisticated specializations of unification and built-in predicates to name a few. However, these optimizations often require a sophisticated type inference system capable of inferring disjunctive and recursive types and hence expensive in computation time.

The purpose of this paper is to describe a type analysis system for Prolog based on abstract interpretation and type graphs (i.e. disjunctive rational trees) with this functionality. The system (about 15,000 lines of C) consists of the combination of a generic fixpoint algorithm, a generic pattern domain, and a type graph domain. The main contribution of the paper is to show that this approach can be engineered to be practical for medium-sized programs without sacrificing accuracy. The main technical contributions to achieve this result are (1) a novel widening operator for type graphs which appears to be accurate and effective in keeping the sizes of the graphs, and hence the computation time, reasonably small; (2) the use of the generic pattern domain to obtain a compact representation of equality constraints between subterms and to factor out sure structural information.

1 Introduction

Although Prolog is an untyped language, type analysis of the language is important since it enables to improve indexing, to specialize unification, and to produce more efficient code for built-in predicates to name a few. However, to provide compilers with sufficiently precise information, type analyses must be rather sophisticated and contain disjunctive and recursive types. Consider for instance the simple program to insert an element in a binary tree:

```
insert(E,void,tree(void,E,void)).
insert(E,tree(L,V,R),tree(Ln,V,R)) :- E ≤ V, insert(E,L,Ln).
insert(E,tree(L,V,R),tree(L,V,Rn)) :- E > V, insert(E,R,Rn).
```

If compilers are given the information that the first argument is not a variable and that the type `T` of the second argument is described the grammar

[1]This research was partly supported by the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225 and the National Science Foundation under grant numbers CCR-9357704 and a National Young Investigator Award.

```
T ::= void | tree(T,Any,T)
```

then at most two tests are necessary to select the appropriate clause to execute. Note that a recursive type is needed because of the recursive call. Information about the functor of the second argument would only enable to specialize the first call to `insert`.

Extensive research has been devoted to type inference in logic programming, although few systems have actually been developed. A popular line of research, called the *cartesian closure* approach in [9], was initiated by [15] and further developed in many authors (See [7] for a complete account). The key idea is to approximate the traditional T_p operator by replacing substitutions by sets of substitutions and using a cartesian closure operator to ignore inter-variable and inter-argument dependencies. This approach originated in type checking applications but can be used for type analysis as well. The type inference problem in this approach was shown to be decidable by Heintze and Jaffar [9] using a reduction to set constraints. By reducing the problem to the inference of (a subclass of) monadic logic programs, Fruehwirth et al. [7] gave an exponential lower bound for type checking and an exponential algorithm for type inference. The appealing feature of this approach is that the problem is amenable to precise characterization and hence its properties can be studied more easily. Its limitation for type analyis is that the relationships between predicate arguments are ignored which may entail a loss of precision and makes it difficult to integrate the system with other analyses such as modes and sharing. A type inference system based on this approach was developed by Heintze [8] and the experimental results (on programs up to 32 clauses) indicate that there is hope to make this approach practical.

Another line of research is the approach of Bruynooghe and Janssens (e.g. [2, 10]) which is based on a traditional abstract interpretation approach [4]. The key idea is to approximate a collecting semantics of the language by an abstract semantics where sets of substitutions are described by type graphs, i.e. disjunctive rational trees. A fixpoint algorithm is then used to compute the least fixpoint or a postfixpoint of the abstract semantics. The problem of inferring the set of principal functors for an argument in a program is undecidable and the result of the analysis is thus an approximation as is traditional in abstract interpretation. The appealing features of this approach is the possibility of exploiting variable dependencies and the ease with which type analysis can be combined with other analyses as required by applications such as compile-time garbage collection [17]. The drawback is that the result of the analysis is more difficult to characterize formally as the design of the abstract domain is an experimental endeavour. This approach has been implemented in a prototype system [10] but experimental results have only been reported on very small programs and were not very encouraging. Hence the practicability of this approach remains open. Note also that the two approaches, which use fundamentally different algorithms, are not directly comparable in accuracy, since the accuracy of the abstract interpretation approach depends upon the choices made in the abstract domain.

The purpose of this paper is to describe the design and implementation of a type system based on the second approach. The system is best described as `GAIA(Pat(Type))`, where `GAIA` is a generic top-down fixpoint algorithm for Prolog [12, 6][2], `Pat` is a generic pattern domain for structural information [3], and `Type` is a type graph domain. The main contribution of the system (about 15,000 lines of C) is to show that type analysis based on abstract interpretation and type graphs can be engineered to be practical, at least for medium-sized programs (up to 450 lines of Prolog). It also shows that type graphs can be practical and this is of importance for many applications such

[2] `GAIA` is available by anonymous ftp from Brown University.

as compile-time garbage collection (e.g. [17]) and automatic termination analysis (e.g. [21]). The technical contributions to obtain this result are (1) a novel widening operator for type graphs which appears to be accurate and effective in keeping the sizes of the graphs, and hence the computation time, reasonably small; (2) the use of the pattern domain to obtain a compact representation of equality constraints between subterms and to factoring sure structural information. This should be contrasted with Bruynooghe and Janssens's approach which restricts attention to a finite domain to guarantee the finiteness of the analysis and represents all information in the type graph domain. Note also that the use of widening operators for type inference has been recently investigated in the context of functional programming but the technical details of this work are fundamentally different [16].

The rest of this paper is organized as follows. Section 2 illustrates the functionality of the system on a variety of small but representative examples. Section 3 gives an overview of the paper. Sections 4 and 5 briefly review our abstract interpretation framework and the generic pattern domain. Section 6 describes our design of the type graph domain. Section 7 reports the experimental results. Section 8 concludes the paper.

2 An Illustration of the Functionality of the Type System

The purpose of this section is to illustrate the behaviour of the type analysis system on a number of examples. It should give the reader an intuitive idea of the accuracy and efficiency of the type analysis system. The examples are small for clarity but they represent abstractions of existing procedures and illustrate many aspects of Prolog programming. Results on medium-sized programs are given in Section 8.

Our type analysis system receives as input a Prolog program and an input pattern, i.e. a predicate symbol and some type information on each of the arguments. The input pattern gives information on how the program is used, i.e. it specifies the top-level goal and the type properties satisfied by the arguments. In this section, for simplicity, all input patterns are of the form `p(Any...,Any)`, where `Any` represents the set of all terms. The output of the system is an output pattern, i.e. a predicate symbol and some type information on each of the arguments. The output pattern represents type information of the arguments on success of the predicate. The system also returns a set of tuples $(\beta_{in}, \mathtt{p}, \beta_{out})$ which represent the input and output patterns for a predicate symbol `p` needed to compute the result. Note that the system performs a polyvariant analysis, i.e. there may be multiple tuples associated with the same predicate symbol. In the following, we mainly show the top-level result for simplicity. The results are presented as context free grammars, since there is a close analogy between grammars and type graphs (see Section 6.1). Consider first the traditional naive reverse program

```
nreverse([],[]).
nreverse([F|T],Res) :- nreverse(T,Trev), append(Trev,[F],Res).

append([],X,X).
append([F|T],S,[F|R]) :- append(T,S,R).
```

For an input pattern `nreverse(Any,Any)`, the system produces the output pattern `nreverse(T,T)` where `T` is defined as follows:

```
T   ::= [] | cons(Any,T).
```

In other words, both arguments should be lists after execution of `nreverse`. The analysis also concludes that the first argument to `append` is always a list. Note that the system has no predefined notion of list: `[]` and `cons/2` are uninterpreted functors. The analysis time for this example is about 0.01 seconds. Consider now the following program which is an abstraction of a procedure used in the parser of Prolog.

```
process(X,Y) :- process(X,0,Y).

process([],X,X).
process([c(X1)|Y],Acc,X) :- process(Y,c(X1,Acc),X).
process([d(X1)|Y],Acc,X) :- process(Y,d(X1,Acc),X).
```

The program is interesting because it contains a sophisticated form of accumulator, a traditional Prolog programming technique. For the input pattern `process(Any,Any)`, the analysis returns the output pattern `process(T,S)` such that

```
T   ::= [] | cons(T1,T).
T1  ::= c(Any) | d(any).
S   ::= 0 | c(Any,S) | d(Any,S).
```

The first argument is inferred to be a list with two types of elements while the second argument captures perfectly the structure of the accumulator. The analysis time is about 0.34 seconds. Consider now a slight variation of the program to introduce two mutually recursive procedures:

```
process(X,Y) :- process(X,0,Y).

process([],X,X).
process([c(X1)|Y],Acc,X) :- other_process(Y,c(X1,Acc),X).

other_process([d(X1)|Y],Acc,X) :- process(Y,d(X1,Acc),X).
```

For the input pattern `process(Any,Any)`, the analysis returns the output pattern `process(T,S)` such that

```
T   ::= [] | cons(c(Any),cons(d(Any),T)).
S   ::= 0 | d(Any,c(Any,S)).
```

Once again the types of the accumulator and of the list are inferred perfectly and the analysis time is about 0.08 seconds. Consider now the example depicted in Figure 1 which contains imbricated lists and an accumulator. Given the input pattern `get(Any)`, the analysis system returns the output pattern `get(T)` where

```
T   ::= [] | cons(T1,T).
T1  ::= [] | cons(T2,T1).
T2  ::= a | b.
```

```
llist([]).
llist([F|T]) :- list(F), llist(T).

list([]).
list([F|T]) :- p(F), list(T).

p(a).  p(b).

reverse(X,Y) :- reverse(X,[],Y).

reverse([],X,X).
reverse([F|T],Acc,Res) :- reverse(T,[F|Acc],Res).

get(Res) :- lt(X), reverse(X,Res).
```

Figure 1: A Prolog Program Manipulating Imbricated Lists

```
add(0,[]).
add(X + Y,Res) :- add(X,Res1), mult(Y,Res2), append(Res1,Res2,Res).

mult(1,[]).
mult(X * Y,Res) :- mult(X,Res1), basic(Y,Res2), append(Res1,Res2,Res).

basic(var(X),[X]).
basic(cst(C),[]).
basic(par(X),Res) :- add(X,Res).
```

Figure 2: A Prolog Program Manipulating Arithmetic Expressions

```
add(X,Res) :- mult(X,Res).
add(X + Y,Res) :- add(X,R1), mult(Y,R2), append(R1,R2,Res).

mult(X,Res) :- basic(X,Res).
mult(X * Y,Res) :- mult(X,R1), basic(Y,R2), append(R1,R2,Res).

basic(var(X),[X]).
basic(cst(X),[]).
basic(par(X),Res) :- add(X,Res).
```

Figure 3: Another Prolog Program Manipulating Arithmetic Expressions

The analysis time is about 0.09 seconds. The example illustrates well how the imbricated list structure is inferred by the system and preserved when used inside the accumulator of `reverse`. Consider the program depicted in Figure 2 which collects information in arithmetic expressions. For the input pattern `add(Any,Any)`, the analysis produces the output pattern `add(T,S)` where

```
T   ::= T + T1 | 0.
T1  ::= T1 * T2 | 1.
T2  ::= cst(Any) | par(T) | var(Any).
S   ::= [] | cons(Any,S).
```

The interesting point in this example is that the rule for T_2 contains an occurrence of `T` showing that our analysis can generate grammars with mutually recursive rules. The analysis time is about 0.11 seconds. Consider now the even more sophisticated program on arithmetic expressions depicted in Figure 3. For the input pattern `add(Any,Any)`, the analysis produces the output pattern `add(T,S)` where

```
T   ::= T2 | T + T1 | T * T2.
T1  ::= T2 | T1 * T2.
T2  ::= cst(Any) | var(Any) | par(T).
S   ::= [] | cons(Any,S).
```

The analysis time is about 0.45 seconds. Finally, we would like to mention the analysis of the tokenizer of Prolog which produces the result

```
T   ::= [] | cons(T1,T).
T1  ::= '(' | ')' | ',' | '[' | ']' | '{' | '|' | '}' |
        atom(Any) | integer(Any) | string(T2) | var(Any,Any)
T2  ::= [] | cons(Any,T2).
```

The analysis time is about 0.42 seconds and the interesting point was the ability of the widening to preserve the string type.

3 Overview of the Type Analysis System

Our type analysis system can be described as `GAIA(Pat(Type))` where

1. `GAIA(`$\mathcal{R}$`)` is a generic fixpoint algorithm for Prolog which, given an abstract domain $\mathcal{R}$, computes the least fixpoint (finite domains) or a postfixpoint (infinite domains) of an abstract semantics based on $\mathcal{R}$;

2. `Pat(`$\mathcal{R}$`)` is a generic pattern domain which enhances any domain $\mathcal{R}$ with structural information and equality constraints between subterms;

3. `Type` is the type graph domain to represent type information.

The next three sections are devoted to each of the subsystems with a special emphasis on `Type` since the other two systems have been presented elsewhere.

4 The Abstract Interpretation Framework

In this section, we briefly review our abstract interpretation framework. The framework is presented in details in [12] and is close to the work of Marriott and Sondergaard [14] and Winsborough [22]. It follows the traditional approach to abstract interpretation [4].

Concrete Semantics As is traditional in abstract interpretation, the starting point of the analysis is a collecting semantics for the programming language. Our concrete semantics is a collecting fixpoint semantics which captures the top-down execution of logic programs using a left-to-right computation rule and ignores the clause selection rule. The semantics manipulates sets of substitutions which are of the form $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ for some $n \geq 0$. Two main operations are performed on substitutions: unification and projection. The semantics associates to each of the predicate symbol p in the program a set of tuples of the form $(\Theta_{in}, p, \Theta_{out})$ which can be interpreted as follows:

> "the execution of $p(x_1, \ldots, x_n)\theta$ with $\theta \in \Theta_{in}$ produces a sequence of substitutions $\theta_1, \ldots, \theta_n, \ldots$, all of which belongs to Θ_{out}."

Abstract Semantics The second step of the methodology is the abstraction of the concrete semantics. Our abstract semantics consists in abstracting a set of substitutions by a single abstract substitutions, i.e. an abstract substitution represents a set of substitutions. As a consequence, the abstract semantics associates with each predicate symbol p a set of tuples of the form $(\beta_{in}, p, \beta_{out})$ which can be read informally as follows:

> "the execution of $p(x_1, \ldots, x_n)\theta$ with θ satisfying the property described by β_{in} produces a sequence of substitutions $\theta_1, \ldots, \theta_n, \ldots$, all of which satisfying the property described by β_{out}."

The abstract semantics assumes a number of operations on abstract substitutions, in particular unification, projection, and upper bound. The first two operations are simply consistent approximations of the corresponding concrete operations. The upper bound operation is a consistent abstraction of the union of sets of substitutions.

The Fixpoint Algorithm The last step of the methodology consists in computing the least fixpoint or a postfixpoint of the abstract semantics. The fixpoint algorithm `GAIA` [12] is a top-down fixpoint algorithm computing a small, but sufficient, subset of least fixpoint (or of a postfixpoint) necessary to answer a user query. The algorithm uses memoization, a dependency graph to avoid redundant computation, the abstract operations of the abstract semantics, and the ordering relation on the abstract domain. It has many similarities with `PLAI` [18] and can be seen either as an implementation of Bruynooghe's framework [1] or as an instance of a general fixpoint algorithm [13]. In the experimental results, we use the prefix version of the algorithm [6].

5 The Generic Pattern Domain

In this section, we briefly recall the basic notions behind the generic abstract domain Pat($\mathcal{R}$). The main significance of Pat($\mathcal{R}$) for type analysis is the reduction in the size of the type graphs by factoring out sure structural information and, more importantly, by providing a compact representation of equalities between subterms. This allows the domain `Type` to be kept as simple as possible and should be contrasted with the approach of [10] where both information are handled in the same domain.

The key intuition behind Pat($\mathcal{R}$) is to represent information on some subterms occurring in a substitution instead of information on terms bound to variables only. More precisely, Pat($\mathcal{R}$) may associate the following information with each considered subterm: (1) its *pattern* which specifies the main functor of the subterm (if any) and the subterms which are its arguments; (2) its *properties* which are left unspecified and are given in the domain $\mathcal{R}$. A subterm is said to be a leaf iff its pattern is unspecified. In addition to the above information, each variable in the domain of the substitutions is associated with one of the subterms. Note that the domain can express that two arguments have the same value (and hence that two variables are bound together) by associating both arguments with the same subterm. This feature produces additional accuracy by avoiding decoupling terms that are equal but it also contributes in complicating the design and implementation of the domain. It should be emphasized that the pattern information is optional. In theory, information on all subterms could be kept but the requirement for a finite analysis makes this impossible for almost all applications. As a consequence, the domain shares some features with the depth-k abstraction [11], although Pat($\mathcal{R}$) does not impose a fixed depth but adjusts it dynamically through upper bound and widening operations.

Pat($\mathcal{R}$) is thus composed of three components: a pattern component, a same value component, and a $\mathcal{R}$-component. The first two components provide the skeleton which contains structural and same-value information but leaves unspecified which information is maintained on the subterms. The $\mathcal{R}$-domain is the generic part which specifies this information by describing properties of a set of tuples $< t_1, \ldots, t_p >$ where $t_1, \ldots, t_p$ are terms. As a consequence, defining the $\mathcal{R}$-domain amounts essentially to defining a traditional domain on substitutions. In particular, it should contain operations for unification, projection, upper bound, and ordering. The only difference is that the $\mathcal{R}$-domain is an abstraction of a concrete domain whose elements are sets of tuples (of terms) instead of sets of substitutions. This difference is conceptual and does not fundamentally affect the nature or complexity of the $\mathcal{R}$-operations.

The implementation of the abstract operations of Pat($\mathcal{R}$) is expressed in terms of the $\mathcal{R}$-domain operations. In general, the implementations are guided by the structural information and call the $\mathcal{R}$-domain operations for basic cases. Pat($\mathcal{R}$) can be designed in two different ways, depending upon the fact that we maintain information on all terms or only on the leaves. For `Pat(Type)`, we only maintain type information on the leaves. Since Pat($\mathcal{R}$) and `Type` are both infinite domains, a widening operation is needed as well. This operation is simply the upper-bound operation on Pat($\mathcal{R}$) with the upper bound operation on the subdomain replaced by a widening operation. The widening operation on `Type` is the critical design decision in `Type` and is discussed in Section 6.3.

The identification of subterms (and hence the link between the structural component and the $\mathcal{R}$-domain) is a somewhat arbitrary choice. In Pat($\mathcal{R}$), subterms are identified by integer indices, say $1 \ldots n$ if n subterms are considered. For instance, the substitution $\{x_1 \leftarrow t{*}a \ , \ x_2 \leftarrow a \ , \ x_3 \leftarrow y_1 \backslash [\,]\}$ will have 7 subterms. The association of indices to them could be for instance

$$\{(1, t * a), (2, t), (3, a), (4, a), (5, y_1 \setminus [\,]), (6, y_1), (7, [\,])\}.$$

The *pattern component* (possibly) assigns to an index an expression $f(i_1, \ldots, i_n)$, where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. If it is omitted, the pattern is said to be undefined. To represent the following set of substitutions

$$\{ \ \{x_1 \leftarrow t * a \ , \ x_2 \leftarrow a \ , \ x_3 \leftarrow y_1 \setminus [\,]\} \ , \ \ \{x_1 \leftarrow t * a \ , \ x_2 \leftarrow b \ , \ x_3 \leftarrow y_1 \setminus [\,]\} \ \}$$

the (most precise) pattern component will make the following associations

$$\{(1, 2 * 3), (2, t), (3, a), (5, 6 \setminus 7), (7, [\,])\}.$$

Note that no information is associated with subterm 4, since this information differs in the substitutions. The *same value* component, in this example, maps x_1 to 1, x_2 to 4, and x_3 to 5.

As mentioned previously, the $\mathcal{R}$-domain associates some properties with the leaves. In the case of the `Pat(Type)`, we could have the association $\{(4, \mathtt{T})\}$ where `T` would be described as

```
T ::= a | b.
```

6 The Type Graph Domain

In this section, we present the design of the domain `Type`. We start with type graphs and then define the domain, its operations, and the widening operator.

6.1 Type Graphs

Our type graphs are essentially what Bruynooghe and Janssens call rigid types and readers are referred to [10] for a complete coverage of type graphs. Our presentation uses more algorithmic concepts to simplify the rest of the presentation.

Definition and Denotation A type graph `g` is a rooted graph $\langle$ `G, r` $\rangle$, where `G = (V,E)` is a directed graph such that, for any vertex `v` in `G`, the successors of `v` are ordered and `r` is a distinguished vertex called the root of `g` and denoted by `root(g)`. In the following, type graphs are denoted by the letter `g` and vertices by the letter `v`, both possibly subscripted or superscripted. A vertex `v` in a type graph `g` is associated with the following information:

$$\texttt{Cc}(\langle(\texttt{V},\texttt{E}),\texttt{r}\rangle) = \texttt{lfp}(\mathcal{D})(\texttt{r}).$$

$$\mathcal{D}: \quad (\texttt{V} \rightarrow \texttt{SST}) \rightarrow (\texttt{V} \rightarrow \texttt{SST})$$
$$\mathcal{D}(\Phi) = \{ (\texttt{v},\texttt{T}) \mid \texttt{v} \in \texttt{V} \ \& \ \texttt{T} = \texttt{Denot}(\texttt{v},\Phi) \}.$$

$$\begin{array}{l} \texttt{Denot}(\texttt{v},\Phi) = \\ \quad \texttt{if type(v) = Any then ST} \\ \quad \texttt{else if type(v) = or then } \bigcup_{1 \le i \le \texttt{arity(v)}} \Phi(\texttt{succ}(\texttt{v},\texttt{i})) \\ \quad \texttt{else } \{\texttt{f}(\texttt{t}_1,\ldots,\texttt{t}_{\texttt{arity(v)}}) \mid \texttt{f} = \texttt{functor}(\texttt{v}) \ \& \ \texttt{t}_i \in \Phi(\texttt{succ}(\texttt{v},\texttt{i}))\}. \end{array}$$

Figure 4: The Denotation of Type Graphs

`type(v)`: an element of {`Any, functor, or`};
`functor(v)`: a string if `type(v) = functor`;
`arity(v)`: a natural number if `type(v)` $\in$ {`functor,or`}.

If `type(v) = or`, `arity(v)` $>$ `1`. The successors of a vertex `v` are denoted by `succ(v)` and the `i`th successor of `v` by `succ(v,i)` for `1` $\leq$ `i` $\leq$ `arity(v)`. More types (e.g. `Integer, Real, Ground`) can be added easily without affecting the results described here.

The denotation of a type graph `g`, denoted by `Cc(g)`, is depicted in Figure 4. In the figure, `ST` denotes the set of all terms, `SST` the powerset of `ST`, and `lfp` is the least fixpoint operator. Φ is a function from vertices to sets of terms which is described by its table in the result of the transformation $\mathcal{D}$. Note also that, in the following, we also talk about the denotation of a vertex `v` in a graph `g`, i.e. $\texttt{lfp}(\mathcal{D})(\texttt{v})$, and we use $\texttt{Cc}_g\texttt{(v)}$ to denote it.

Relation to Context-Free Grammars Type graphs can easily be related to context-free grammars and we exploited this correspondence in the informal introduction to display the results. A simple idea is to associate a non-terminal symbol $\texttt{T}_\texttt{v}$ with each vertex `v`. The rule associated with an or-vertex `v` with successors $\texttt{v}_1,\ldots,\texttt{v}_n$ is simply

$$\texttt{T}_v \quad ::= \texttt{T}_{\texttt{v}_1} \ | \ \ldots \ | \ \texttt{T}_{\texttt{v}_n}.$$

The rule associated with a functor-vertex having `f` as functor and $\texttt{v}_1,\ldots,\texttt{v}_n$ as successors is simply

$$\texttt{T}_v \quad ::= \texttt{f}(\texttt{T}_{\texttt{v}_1},\ldots,\texttt{T}_{\texttt{v}_n}).$$

The rule associated with an any-vertex is simply

$$\texttt{T}_v \quad ::= \texttt{Any}.$$

In the presentation of the results, we generally apply some partial evaluation of the grammar to improve its readability.

Relation to Monadic Logic Programs Type graphs can also be related to monadic logic programs of [7]. The logic program associated with a type graph succeeds for all well-typed terms. A simple way is to associate a procedure p_v with each vertex v. The procedure for an any-vertex is simply

```
any(X).
```

The procedure for a functor-vertex having f as functor and $v_1,\ldots,v_n$ as successors is simply

$p_v(f(X_1,\ldots,X_n))$:- $p_{v_1}(X_1)$, ..., $p_{v_n}(X_n)$.

The procedure associated with an or-vertex with successors $v_1,\ldots,v_n$ is simply

$p_v(X)$:- $p_{v_1}(X)$.
...
$p_v(X)$:- $p_{v_n}(X)$.

Note that this rewriting shows that inferring even the principal functors of an argument is undecidable, since the halting problem for a program prog(Input,Output) can be expressed as the type inference problem:

```
p(a,Input) :- prog(Input,Output).
p(b,Input).
```

Additional Definitions The following definitions will be useful subsequently. We assume for simplicity an underlying type graph g. The depth of a vertex v, denoted by depth(v), is the shortest path from root(g) to v. An ancestor of a vertex v is any vertex v' $\neq$ v on the shortest path from root(v) to v. The set of ancestors of v is denoted by ancestor(v). It is sometimes convenient to identify a vertex by a path, i.e. a sequence of integers. Given a type graph g and a path p, the vertex is obtained by follow(root(g),p) where

follow(v,[]) = v;
follow(v,[$i_1,\ldots,i_n$]) = follow(succ(v,i_1),[$i_2,\ldots,i_n$]).

The size of a graph g, denoted by size(g), is simply the number of vertices and edges in the graph. The vertices (resp. the edges) of g are denoted by vertices(g) (resp. edges(g)). We also use the function removeUnconnected to remove the vertices which are not connected to the root. It is defined as

removeUnconnected($\langle$(V,E),r$\rangle$) = $\langle$(V',E'),r$\rangle$ where
  V' = { v | v $\in$ V & r $\in$ ancestor(v) }
  E' = { (v,v') | (v,v') $\in$ E & v,v' $\in$ V' }.

Pragmatic Restrictions Our system enforces a number of pragmatic restrictions on type graphs for efficiency and convenience reasons:

1. the successors of an or-vertex are functor-vertices;
2. if v_1 and v_2 are successors of an or-vertex v, functor(v_1) $\neq$ functor(v_2);

3. if `v'` is a successor of a functor-vertex `v`, `v'` is an or-vertex or `depth(v')` $>$ `depth(v)`;

4. for any vertex `v`, `ancestor(v)` $\cap$ `predecessor(v) =` $\{$`v'`$\}$ and there is a single edge (`v`,`v'`).

Restrictions 1,2, and 4 are adaptations of similar restrictions in [10]. The second restriction actually reduces the expressive power of type graphs and is called the principal functor restriction in [10] (pf-restriction for short). Restriction 3 requires cycles to start at functor-vertices and to end at or-vertices while restriction 4 prevents subgraphs from sharing. When these restrictions are adopted, it makes sense to refer to the pf-set of an or-vertex as the set of all functors of its successors. The pf-set of an or-vertex `v` is denoted `pf(v)`. The pf-set of a functor-vertex is simply the singleton set containing its functor. We will also assume in the following that the successors of or-vertices with the same pf-set are ordered in such a way that successors with the same functor are at the same position.

The Domain The abstract domain `Type` simply abstracts a set of term tuples of the form $\langle \mathtt{t}_1, \ldots, \mathtt{t}_n \rangle$ by an abstract tuple $\langle \mathtt{g}_1, \ldots, \mathtt{g}_n \rangle$. The concretization function is simply given by

$$Cc(\langle \mathtt{g}_1, \ldots, \mathtt{g}_n \rangle) \;=\; \{\langle \mathtt{t}_1, \ldots, \mathtt{t}_n \rangle \mid \mathtt{t}_i \in Cc(\mathtt{g}_i)\ (1 \le i \le n)\}.$$

6.2 Operations on Type Graphs

The abstract operations of `Type` can be obtained immediately from three operations on type graphs:

1. $\mathtt{g}_1 \le \mathtt{g}_2$: returns true iff $\mathtt{Cc(g_1)} \subseteq \mathtt{Cc(g_2)}$;

2. $\mathtt{g}_1 \cap \mathtt{g}_2$: returns $\mathtt{g}_3$ such that $\mathtt{Cc(g_3)} \subseteq \mathtt{Cc(g_1)} \cap \mathtt{Cc(g_2)}$;

3. $\mathtt{g}_1 \cup \mathtt{g}_2$: returns $\mathtt{g}_3$ such that $\mathtt{Cc(g_1)} \cup \mathtt{Cc(g_2)} \subseteq \mathtt{Cc(g_3)}$.

The first two operations are described in [10]. Note that intersection is used for unification since our type graphs are downward-closed. The third operation is not described in [10] which uses an indirect approach: first, an or-vertex is created with the two inputs as successors; then a compaction algorithm is applied to satisfy the restrictions. Our system uses a direct implementation which does not raise any difficulty. It is only necessary to take care of the pf-restriction in the cases functor/or and or/or by applying the algorithm recursively. Of course, memoization is used to guarantee termination. Note also that, in the following, we often use operation $\le$ on vertices to denote inclusion of their denotation. The algorithm is the same as for type graphs.

6.3 The Widening Operator

The main difficulty in the type graph domain comes from the fact that the domain is infinite and does not satisfy the ascending chain property. In fact, it is not even a cpo. To overcome this difficulty, Bruynooghe and Janssens [10] use a finite subdomain by restricting the number of occurrences of a functional symbol on the paths of the graphs. We adopted a different, less syntactic, solution based on a widening operator as proposed in [4]. The design of widening operators is experimental in nature and it affects both the performance and accuracy of the analysis. The examples given previously in the paper shows that our widening operator leads to accurate results and is effective in keeping the graph sizes small. The purpose of this section is to describe the widening operator informally and formally.

6.3.1 Informal Presentation

In abstract interpretation of Prolog, widening needs to be applied in two different situations:

1. when the result of a procedure is updated;
2. when a procedure is about to be called.

In the first case, widening avoids the result of a procedure to be refined infinitely often while, in the second case, widening avoids an infinite sequence of procedure calls. Hence, the widening operator is always applied to an old graph $\mathtt{g}_{old}$ (e.g. the previous result of a procedure) and a new graph $\mathtt{g}_{new}$ (e.g. the union of the new clause results) to produce a new graph $\mathtt{g}_{res}$ (e.g. the new result of the procedure).

The main idea behind our widening operator is to consider two graphs,

$$\mathtt{g}_o = \mathtt{g}_{old} \text{ and } \mathtt{g}_n = \mathtt{g}_{old} \cup \mathtt{g}_{new}$$

and to exploit the topology of the graphs to guess where $\mathtt{g}_n$ is growing compared to $\mathtt{g}_o$. The key notion is the concept of topological clash which occurs in situations where

- a functor-vertex in $\mathtt{g}_o$ corresponds to an or-vertex in $\mathtt{g}_n$;
- an or-vertex $\mathtt{v}_o$ in $\mathtt{g}_o$ corresponds to an or-vertex $\mathtt{v}_n$ in $\mathtt{g}_n$ where $\mathtt{pf(v}_o\mathtt{)} \neq \mathtt{pf(v}_n\mathtt{)}$;
- an or-vertex $\mathtt{v}_o$ in $\mathtt{g}_o$ corresponds to an or-vertex $\mathtt{v}_n$ in $\mathtt{g}_n$ where $\mathtt{depth(v}_o\mathtt{)} < \mathtt{depth(v}_n\mathtt{)}$.

In these three cases, the widening operator tries to prevent the graph from growing by introducing a cycle in $\mathtt{g}_n$. Given a clash $(\mathtt{v}_o,\mathtt{v}_n)$, the widening searches for an ancestor $\mathtt{v}_a$ to $\mathtt{v}_n$ such that $\mathtt{pf(v}_n\mathtt{)} \subseteq \mathtt{pf(v}_a\mathtt{)}$. If such an ancestor is found and if $\mathtt{v}_a \geq \mathtt{v}_n$, a cycle can be introduced.

Consider for instance `append/3`. The second iteration has produced the following type graph for the first argument

$\mathtt{T}_o$ `::= [] | cons(Any,[]).`

The union of the clause results for the third iteration gives the following type graph for the first argument

$\mathtt{T}_{new}$ `::= [] | cons(Any,cons(Any,[])).`

Taking the union of $\mathtt{T}_o$ and $\mathtt{T}_{new}$ produces the type graph described by

$\mathtt{T}_n$ `::= [] | cons(Any,T).`
`T ::= [] | cons(Any,[]).`

There is a topological clash between $\mathtt{T}_o$ and $\mathtt{T}_n$ for the path [2,2] which corresponds to `[]` and `T` respectively. The widening selects $\mathtt{T}_n$ as an ancestor and introduces a cycle producing the final result

$\mathtt{T}_r$ `::= [] | cons(Any,`$\mathtt{T}_r$`).`

```
To  ::= 0 | 0 + T1.
T1  ::= 1 | T1 * T2.
T2  ::= cste(Any) | par(0) | var(Any).

Tn  ::= 0 | T3 + T6.
T3  ::= 0 | 0 + T4.
T4  ::= 1 | T4 * T5.
T5  ::= cste(Any) | par(0) | var(Any).
T6  ::= 1 | T6 * T7.
T7  ::= cste(Any) | par(T3) | var(Any).
```

Figure 5: Widening for the First Arithmetic Program

Note also that an ancestor at any depth can be selected. For the first arithmetic program (see Figure 2), the widening applies to the the type graphs $\mathtt{T}_o$ and $\mathtt{T}_n$ depicted in Figure 5. Consider the clash occurring for the path [2,2,2,2,2,1] for $\mathtt{T}_o$ and $\mathtt{T}_n$. An appropriate ancestor for $\mathtt{T}_3$ is $\mathtt{T}_n$ which is not a direct ancestor. This results in the optimal result $\mathtt{T}_r$

```
Tr  ::= 0 | 0 + T1.
T1  ::= 1 | T1 * T2.
T2  ::= cste(Any) | par(Tr) | var(Any).
```

When no ancestor with a suitable pf-set can be found, the widening operator simply allows the graph to grow. Termination will be guaranteed because this growth necessarily adds along the branch a pf-set which is not a subset of any existing pf-set in the branch. This case of course happens frequently in early iterations of the fixpoint. Returning the arithmetic program, the second iteration for the predicate `basic/2` requires a widening for the first argument with the following two graphs:

```
To  ::= cste(Any) | var(Any).
Tn  ::= cste(Any) | par(0) | var(Any).
```

A topological clash of type 2 is encountered but there is no suitable ancestor. The result will simply be $\mathtt{T}_n$ in this case.

The last case to consider appears when there is an ancestor $\mathtt{v}_a$ with a suitable pf-set but unfortunately $\mathtt{v}_a \geq \mathtt{v}_n$ is false. In this case, introducing a cycle would produce a graph $\mathtt{T}_r$ whose denotation may not include the denotation of $\mathtt{T}_n$ and hence our widening operator cannot perform cycle introduction. Instead the widening operation replaces $\mathtt{v}_a$ by a new or-vertex which is an upper bound to $\mathtt{v}_a$ and $\mathtt{v}_n$ but decreases the overall size of the type graph. The widening operator is then applied again on the resulting graph.

As a consequence, our widening operator is best viewed as a sequence of transformations on $\mathtt{T}_n$, which are of two types: (1) cycle introduction; (2) vertex replacement, until no more topological clashes can be resolved. We now formalize these notions.

7 The Widening and its Formal Properties

To simplify presentation, we assume, without loss of generality, that type graphs are such that their roots are or-vertices and that the successors of or-vertices (resp. functor-vertices) are functor-vertices (resp. or-vertices). This assumption requires replacing some functor-vertices (resp. any-vertices) by or-vertices with a single successor which is the original functor-vertex (resp. any-vertex). This convention is purely a matter of presentation, all our algorithms being defined on the original graphs. The following abbreviations will also be useful in this section.

$$\begin{array}{lll} \texttt{OR}(\texttt{v}_1) & \equiv & \texttt{type}(\texttt{v}_1) \texttt{ = or.} \\ \texttt{same-depth}(\texttt{v}_1,\texttt{v}_2) & \equiv & \texttt{depth}(\texttt{v}_1) \texttt{ = depth}(\texttt{v}_2)\texttt{.} \\ \texttt{same-pf}(\texttt{v}_1,\texttt{v}_2) & \equiv & \texttt{pf}(\texttt{v}_1) \texttt{ = pf}(\texttt{v}_2)\texttt{.} \end{array}$$

We also use $\langle n_1, \ldots, n_i, \ldots, n_p \rangle \downarrow i$ to denote element n_i. This notation is generalized to sets of tuples by defining $S \downarrow i \;=\; \{s \downarrow i \mid s \in S\}$.

Topological Clashes As mentioned previously, the key idea behind our widening operator is to exploit the topology of the graphs to guess where the sequence is growing. We can establish a correspondence between the vertices of two graphs as follows.

Definition 1 The *correspondence set* between two type graphs $\texttt{g}_1$ and $\texttt{g}_2$, denoted by $\texttt{C}(\texttt{g}_1,\texttt{g}_2)$, is the smallest relation R closed by the following two rules:

- $(\texttt{root}(\texttt{g}_1),\texttt{root}(\texttt{g}_2)) \in \texttt{R}$
- $(\texttt{v}_1,\texttt{v}_2) \in \texttt{R} \;\&\; \texttt{same-depth}(\texttt{v}_1,\texttt{v}_2) \;\&\; \texttt{same-pf}(\texttt{v}_1,\texttt{v}_2) \Rightarrow$
  $(\texttt{succ}(\texttt{v}_1,\texttt{i}),\texttt{succ}(\texttt{v}_2,\texttt{i})) \in \texttt{R} \; (1 \leq \texttt{i} \leq \texttt{arity}(\texttt{v}_1))$.

The set of topological clashes can now be defined in a simple way.

Definition 2 Let $\texttt{g}_1$,$\texttt{g}_2$ be two type graphs such that $\texttt{g}_1 \leq \texttt{g}_2$. The set of topological clashes between $\texttt{g}_1$ and $\texttt{g}_2$, denoted $\texttt{TC}(\texttt{g}_1,\texttt{g}_2)$, is defined as follows:

$$\texttt{TC}(\texttt{g}_1,\texttt{g}_2) = \{ \; (\texttt{v},\texttt{v'}) \mid (\texttt{v},\texttt{v'}) \in \texttt{C}(\texttt{g}_1,\texttt{g}_2) \;\&\; \neg\; (\texttt{same-depth}(\texttt{g}_1,\texttt{g}_2) \;\&\; \texttt{same-pf}(\texttt{g}_1,\texttt{g}_2)) \; \}.$$

The following proposition is an immediate consequence of the definitions.

Proposition 3 Let $\texttt{g}_1$, $\texttt{g}_2$ be two type graphs such that $\texttt{g}_1 \leq \texttt{g}_2$. If $(\texttt{v},\texttt{v'}) \in \texttt{TC}(\texttt{g}_1,\texttt{g}_2)$, then $\texttt{OR}(\texttt{v}) \;\&\; \texttt{OR}(\texttt{v'})$. Moreover, there exists a unique tuple $(\texttt{v}_a,\texttt{v}_{a'}) \in \texttt{C}(\texttt{g}_1,\texttt{g}_2)$, denoted by $\texttt{ca}(\texttt{v},\texttt{v'})$, such that $\texttt{v} \in \texttt{succ}(\texttt{v}_a)$ and $\texttt{v'} \in \texttt{succ}(\texttt{v}_{a'})$.

Our widening operation focuses on a subset of topological clashes which lead to a growth in the graph.

Definition 4 Let $\texttt{g}_1$,$\texttt{g}_2$ be two type graphs such that $\texttt{g}_1 \leq \texttt{g}_2$. The set of widening clashes between $\texttt{g}_1$ and $\texttt{g}_2$, denoted $\texttt{WTC}(\texttt{g}_1,\texttt{g}_2)$, is defined as follows

$$\begin{array}{l} \texttt{TWC}(\texttt{g}_1,\texttt{g}_2) = \{ \; (\texttt{v},\texttt{v'}) \mid (\texttt{v},\texttt{v'}) \in \texttt{TC}(\texttt{g}_1,\texttt{g}_2) \;\&\; \texttt{pf}(\texttt{v'}) \neq \{\texttt{Any}\} \;\& \\ \qquad\qquad\qquad\qquad\qquad (\texttt{pf}(\texttt{v}) \neq \texttt{pf}(\texttt{v'}) \vee \texttt{depth}(\texttt{v}) < \texttt{depth}(\texttt{v'})) \; \} \end{array}$$

```
replaceEdge(g,e,e') = removeUnconnected(g') where
   vertices(g') = vertice(g)
   edges(g') = edges(g) \ {e} ∪ {e'}
   root(g') = root(g).

replaceVertex(g,v,v') = removeUnconnected(g') where
   vertices(g') = vertice(g) ∪ {v1,...,vn} (n ≥ 1)
   edges(g') = edges(g) \ E1 ∪ E2 ∪ E3
   root(g') = root(g)
   vertice(g) ∩ {v1,...,vn} ≠ ∅
   E1 = { (va,vb) | (va,vb) ∈ edges(g) & vb = v }
   E2 = { (va,v1) | (va,vb) ∈ edges(g) & vb = v }
   (va,vb) ∈ E3 ⇒ va ∈ {v1,...,vn} & vb ∈ vertices(g')}
   size(removeUnconnected(g')) < size(g).
```

Figure 6: The Functions `replaceEdge` and `replaceVertex`

Transformation Rules The widening operator essentially consists in applying two transformation rules to eliminate (a subset of) topological clashes. The transformation rules nondeterministically produce a new type graph g_r from two type graphs g_o and g_n with $g_o \leq g_n$. They are defined in terms of two functions: `replaceEdge` and `replaceVertex`. Informally speaking, `replaceEdge(g,e,e')` replaces edge `e` by edge `e'` in the graph while `replaceVertex(g,v,v')` replaces vertex `v` by a new vertex greater or equal than `v` and `v'` and decreases the size of the graph. The formal definitions of the functions are given in Figure 6.
The first operation is straightforward. The second operation can be implemented easily by making v_1 an any-vertex. It is however possible to obtain much more precision by using a variant of the union operation which avoids creating or-vertices which would lead to a growth in size. Note also that the case where $v' \geq v$ can be handled in a straightforward manner. We are now ready to specify the transformation rules. The cycle introduction rule introduces a cycle in the graph by replacing edges to a vertex by edges to one of its ancestors.

Definition 5 [Cycle Introduction Rule] Let g_o and g_n be two type graphs and let

$$CI(g_o,g_n) = \{ \langle (v,v_n),(v,v_a) \rangle \mid (v_o,v_n) \in WTC(g_o,g_n) \ \&\ v_a \in ancestor(v_n) \ \&\ v_a \geq v_n \ \&\ depth(v_o) \geq depth(v_a) \ \&\ v = ca(v_o,v_n){\downarrow}2 \}.$$

The cycle introduction rule can be specified as follows:

$TR_i(g_o,g_n) = g_r$
Precondition: $CI(g_o,g_n) \neq \emptyset$.
Postcondition: g_r = replaceEdge(g_n,e,e') for some (e,e') $\in CI(g_o,g_n)$.

The replacement rule applies when a cycle cannot be introduced because the denotation of the ancestor is not greater than the vertices in the clash. It replaces the ancestor by an upper bound of the vertices.

Definition 6 [Replacement Rule] Let $\mathtt{g}_o$ and $\mathtt{g}_n$ be two type graphs and let

$$\mathtt{CR(g}_o\mathtt{,g}_n\mathtt{)} = \{ \mathtt{(v}_n\mathtt{,v}_a\mathtt{)} \mid \mathtt{(v}_o\mathtt{,v}_n\mathtt{)} \in \mathtt{WTC(g}_o\mathtt{,g}_n\mathtt{)}\ \&\ \mathtt{v}_a \in \mathtt{ancestor(v}_n\mathtt{)}\ \&$$
$$\neg\mathtt{(v}_a \geq \mathtt{v}_n\mathtt{)}\ \&\ \mathtt{pf(v}_n\mathtt{)} \subseteq \mathtt{pf(v}_a\mathtt{)}\ \&\ \mathtt{depth(v}_o\mathtt{)} \geq \mathtt{depth(v}_a\mathtt{)} \}.$$

The replacement rule can be specified as follows:

$\mathtt{TR}_r\mathtt{(g}_o\mathtt{,g}_n\mathtt{)} = \mathtt{g}_r$
Precondition: $\mathtt{CR(g}_o\mathtt{,g}_n\mathtt{)} \neq \emptyset$.
Postcondition: $\mathtt{g}_r = \mathtt{replaceVertex(g}_n\mathtt{,v}_a\mathtt{,v}_n\mathtt{)}$ for some $\mathtt{(v}_n\mathtt{,v}_a\mathtt{)} \in \mathtt{CR(g}_o\mathtt{,g}_n\mathtt{)}$.

Note that this rule only applies when $\mathtt{pf(v}_n\mathtt{)} \subseteq \mathtt{pf(v}_a\mathtt{)}$ and hence it leaves room for the expansion of type graphs before the widening applies.

The Widening Operation We are now in position to present the widening operation. The widening essentially applies the transformation rules until the sets CI and CR are empty.

Definition 7 [Widening Operator] The widening operator $\mathtt{g}_o \bigtriangledown \mathtt{g}_n$ is defined as follows:

$\mathtt{g}_o \bigtriangledown \mathtt{g}_n$ =
  if $\mathtt{g}_n \leq \mathtt{g}_o$ **then** $\mathtt{g}_o$ **else** $\mathtt{widen(g}_o\mathtt{,g}_o \cup \mathtt{g}_n\mathtt{)}$.

$\mathtt{widen(g}_o\mathtt{,g}_n\mathtt{)}$ =
    if $\mathtt{CI(g}_o\mathtt{,g}_n\mathtt{)} \neq \emptyset$ **then** $\mathtt{widen(g}_o\mathtt{,TR}_i\mathtt{(g}_o\mathtt{,g}_n\mathtt{))}$
  else if $\mathtt{CR(g}_o\mathtt{,g}_n\mathtt{)} \neq \emptyset$ **then** $\mathtt{widen(g}_o\mathtt{,TR}_r\mathtt{(g}_o\mathtt{,g}_n\mathtt{))}$
  else $\mathtt{g}_n$.

We now prove two important results on operation $\bigtriangledown$.

Proposition 8 [Termination] Operation $\bigtriangledown$ terminates.

Proof We define a function F which maps a graph g to a pair of natural numbers (n_1, n_2) such that $n_1 = \mathtt{size(g)}$ and $n_2 = \#\mathtt{WTC(g}_o\mathtt{,g)}$. The set of these pairs becomes a well-founded set with the following total ordering:

$$(n_1, n_2) < (n'_1, n'_2) \text{ if } n_1 < n'_1 \text{ or } n_1 = n'_1 \text{ and } n_2 < n'_2.$$

Consider now the sequence of graphs $\mathtt{g}_1,\ldots,\mathtt{g}_i,\ldots$ occurring as second argument of widen. We have $\mathtt{F(g}_i\mathtt{)} < \mathtt{F(g}_{i-1}\mathtt{)}$, since the cycle introduction rule satisfies $\mathtt{size(g}_i\mathtt{)} \leq \mathtt{size(g}_{i-1}\mathtt{)}$ and $\#\mathtt{WTC(g}_o\mathtt{,g}_i\mathtt{)} < \#\mathtt{WTC(g}_o\mathtt{,g}_{i-1}\mathtt{)}$ while the replacement rule satisfies $\mathtt{size(g}_i\mathtt{)} < \mathtt{size(g}_{i-1}\mathtt{)}$. □

Theorem 9 Operator $\bigtriangledown$ is a widening operator.

Proof We first show that $\mathtt{g}_o \bigtriangledown \mathtt{g}_n \geq \mathtt{g}_o, \mathtt{g}_n$. This follows immediately from the fact that `widen` is applied initially on $\mathtt{g}_o$ and $\mathtt{g}_o \cup \mathtt{g}_n$ and that the transformation rules cannot decrease the denotation.

Let $\mathtt{g}_0\text{'},\ldots,\mathtt{g}_i\text{'},\ldots$ be a sequence of type graphs and $\mathtt{g}_0,\ldots,\mathtt{g}_i,\ldots$ a sequence defined as

$$\begin{array}{lcl} \mathtt{g}_0 & = & \mathtt{g}_0\text{'} \\ \mathtt{g}_{i+1} & = & \mathtt{g}_i \bigtriangledown \mathtt{g}_i\text{'}\ (\mathtt{i} \geq 0) \end{array}$$

We show that $\mathtt{g}_0,\ldots,\mathtt{g}_i,\ldots$ is stationary. To define a well-founded set, we assume without loss of generality that `a` is the number of function symbols in the analysed program and we use the following notions. The `potential` of a vertex `v` in a graph `g`, denoted by `p(v,g)`, is defined as follows:

$$\begin{array}{ll} \mathtt{p(v,g)} = \mathtt{a + 1} & \text{if } \mathtt{pf(v)} = \{\mathtt{Any}\}. \\ \mathtt{p(v,g)} = \# \bigcup \{ \mathtt{pf(v')} \mid \mathtt{v'} \in \mathtt{ancestor(v)} \text{ or } \mathtt{v = v'} \} & \text{otherwise.} \end{array}$$

We use `P[S,i,g]` to denote the number of vertices `v` in `S` such that `p(v,g) = i` and we define a function `P` which associates a graph with a `a`-tuple

$$\langle\ \mathtt{P[vertices(g),2,g]},\ldots,\mathtt{P[vertices(g),a+1,g]}\ \rangle.$$

The set of these tuples is a well-founded set for the following total ordering

$$\begin{array}{l} \langle n_1,\ldots,n_a\rangle < \langle n'_1,\ldots,n'_a\rangle \text{ if} \\ \quad n_1 < n'_1 \text{ or} \\ \quad \langle n_1\rangle = \langle n'_1\rangle \text{ and } n_2 < n'_2 \text{ or} \\ \quad \ldots \\ \quad \langle n_1,\ldots,n_{a-1}\rangle = \langle n'_1,\ldots,n'_{a-1}\rangle \text{ and } n_a < n'_a. \end{array}$$

We also denote by `E[g,i]` the number of edges in `g` whose destination is an or-vertex at depth `i` and define a function `E` which maps a graph in the natural number

$$\sum_{\mathtt{i}=0}^{\infty} \mathtt{i}\ \mathtt{E[g,i]}.$$

We prove that the sequence is stationary by associating with each graph `g` a pair $\langle m_p, m_e\rangle = \langle \mathtt{P(g)}, \mathtt{E(g)}\rangle$. The set of these these pairs is a well-founded set for the following total ordering:

$$\begin{array}{l} (m_p, m_e) < (m'_p, m'_e) \text{ if } m_p < m'_p \text{ or} \\ \qquad\qquad\qquad\qquad m_p = m'_p \text{ and } m_e < m'_e. \end{array}$$

We first show that $\mathtt{F(g}_i\mathtt{)} \geq \mathtt{F(g}_{i+1}\mathtt{)}$. Consider the following sets:

$$\begin{array}{l} \mathtt{CO} = \mathtt{C(g}_i\mathtt{,g}_{i+1}\mathtt{)} \setminus \mathtt{TC(g}_i\mathtt{,g}_{i+1}\mathtt{)}. \\ \mathtt{TC} = \mathtt{TC(g}_i\mathtt{,g}_{i+1}\mathtt{)}. \\ \mathtt{NE} = \{\ \mathtt{v'} \in \mathtt{g}_{i+1} \mid \neg\exists\ \mathtt{v}\ \mathtt{(v,v')} \in \mathtt{C(g}_i\mathtt{,g}_{i+1}\mathtt{)}\ \}\ . \end{array}$$

Each tuple `(v,v')` in `CO` satisfies `p(v) = p(v')`. Hence

$$\mathtt{N[\ CO}\!\downarrow\!\mathtt{1\ ,\ k\ ,\ g}_i\mathtt{\ ] = N[\ CO}\!\downarrow\!\mathtt{2\ ,\ k\ ,\ g}_{i+1}\mathtt{\ ]\ (\ 2} \leq \mathtt{k} \leq \mathtt{a + 1).}$$

Each tuple (v,v') in TC can only be of three different forms by definition of $\bigtriangledown$ knowing that OR(v) & OR(v'):

1. $\neg$ same-depth(v,v');
2. pf(v) $\neq$ {Any} and pf(v') = {Any};
3. same-depth(v,v') & pf(v') $\neq$ {Any} & pf(v') $\neq$ pf(v).

In the first case, we have depth(v') $<$ depth(v) by definition of $\bigtriangledown$. Hence, p(v') has already been accounted for in CO$\downarrow$2. In the second case, p(v') $>$ p(v) by definition of the potential. In the third case, p(v') $>$ p(v) by definition of $\bigtriangledown$ and of the potential. Hence, if case 2 or 3 occurs, there exists a k such that

$$N[\ CO{\downarrow}1 \cup TC{\downarrow}1\ ,\ k\ ,\ g_i\ ] > N[\ CO{\downarrow}2 \cup TC{\downarrow}2\ ,\ k\ ,\ g_{i+1}\ ].$$
$$N[\ CO{\downarrow}1 \cup TC{\downarrow}1\ ,\ j\ ,\ g_i\ ] = N[\ CO{\downarrow}2 \cup TC{\downarrow}2\ ,\ j\ ,\ g_{i+1}\ ]\ (2 \leq j < k).$$

If only case 1 occurs, then

$$N[\ CO{\downarrow}1 \cup TC{\downarrow}1\ ,\ j\ ,\ g_i] = N[\ CO{\downarrow}2{\cup}TC{\downarrow}2\ ,\ j\ ,\ g_{i+1}]\ (2 \leq j \leq a+1).$$

Finally, note that each vertex v'' in NE has an ancestor v' such (v,v') $\in$ TC for some v by definition of $\bigtriangledown$. Moreover, p(v) $<$ p(v') $\leq$ p(v''). Hence it follows that F(g_i) $\geq$ F(g_{i+1}) by definition of F.

We conclude the proof by showing that if F(g_i) = F(g_{i+1}) then E(g_i) > E(g_{i+1}). If F(g_i) = F(g_{i+1}), only case 1 could occur. Since $g_i < g_{i+1}$, there must be a least one such case (v,v') $\in$ TC and, by definition of $\bigtriangledown$, depth(v) $>$ depth(v'). Hence E(g_i) > E(g_{i+1}). $\Box$

8 Experimental Evaluation

We now describe the experimental result of our type system. We first describe the benchmarks and discuss the efficiency and accuracy of the analysis.

The Benchmarks The benchmark programs[3] are hopefully representative of "pure" logic programs. KA is an alpha-beta program to play the game of kalah [19]. PR is a symbolic equation-solver [19]. CS is a program to generate a number of configurations representing various ways of cutting a wood board into small shelves [20]. DS is the generate and test equivalent of a disjunctive scheduling problem [5]. RE is the Prolog tokeniser and reader of R.O'keefe and D.H.D.Warren. PG is a program written by W. Older to solve a specific mathematical problem. BR is a program taken from Gabriel benchmark. PL is a planning program from [19]. QU solves the n-queens problem. Finally, PE is a the *peephole* optimizer of SB-Prolog, written by Debray. We will also prefix some programs by L to indicate that the input query assigns lists to some arguments. Finally, we will also use the arithmetic programs discussed previously and denote them by AR and AR1. Table 1 gives some indication on the size of these programs while Table 2 reports the number of non-recursive, tail recursive, locally recursive (more than one recursive call or a nonterminal recursive call), and mutually recursive procedures in each of the benchmarks. Four programs have only tail recursive

[3] The benchmarks are available by anonymous ftp from Brown University.

	KA	QU	PR	PE	CS	DS	PG	RE	BR	PL
Number of Procedures	44	5	52	19	32	28	10	42	20	13
Number of Clauses	82	9	158	168	55	52	18	163	45	26
Number of Program Points	475	38	742	808	336	296	93	820	207	94
Number of Goals	84	8	130	90	57	60	17	168	37	29
Static Call Tree Size	73	5	75	80	46	47	11	144	21	25

Table 1: Sizes of the Programs

	KA	QU	PR	PE	CS	DS	PG	RE	BR	PL
Tail recursive	12	4	12	6	9	14	6	6	11	4
Locally recursive	0	0	5	0	1	0	0	0	1	0
Mutually recursive	7	0	8	4	2	0	0	16	0	0
Non-recursive	25	1	27	9	20	14	4	20	8	9

Table 2: Syntactic Form of the Programs

procedures or non-recursive procedures. Many programs have mutually recursive procedures and some have many of them. In general, the majority of procedures are non-recursive and in many programs, most of the recursive procedures are tail recursive. Program `PR` contains locally recursive procedures due to their divide & conquer approach.

Computation Times In this section, we analyse the efficiency of our type system experimentally. Table 3 describes the CPU time (on a Sun Sparc-10), the number of procedure iterations and the number of clause iterations. We also give the CPU time when the number of successors to or-vertices is restricted to 5 and 2 respectively. As can be seen, the analysis is very fast (below 3 seconds) for all programs except `RE` which takes about 153, 20, and 10 seconds depending on the various restrictions. Note that `PR` is heavily mutually recursive, that `CS` manipulates heavily imbricated lists, and that `PE` has large disjunctions, yet the running time of these programs is excellent. Program `RE` is time-consuming, since it manipulates large graphs (the result of the tokeniser shown previously is only the first step), is heavily mutually recursive, and contains an accumulator-based procedure (very much like the `process` predicate shown previously) in the middle of the recursion. This procedure is actually where the time goes since it is expensive in itself, is applied on the largest graphes occurring in the program, and is recomputed each time a new approximation for the main predicate is obtained. Program `RE` is a worst case scenario for our analyser, although the time remains acceptable. If more efficiency is desirable, there are various ways of speeding up the computation, including the use of a monovariant analysis (instead of the polyvariant analysis used here) or the imposition of restrictions on the size of the graphs or vertices as shown in the table. Overall, the results are very encouraging and seems to indicate that type graphs can be engineered to be practical. The tradeoff between efficiency and accuracy remains obviously an important topic for further research.

Accuracy To give an idea of the accuracy of the system, we measure tag information that can be extracted from the analysis under following assumptions. First, no multiple specializations take place, i.e. a procedure is associated with a single version. Second, we consider the following tag information: `NI` (empty list), `CO` (cons), `LI` (list), `ST` (structure), `DI` (atom), and `HY` (structure or

	KA	QU	PR	PE	CS	DS	PG	RE	BR	PL
CPU Time	1.66	0.01	2.64	2.82	1.14	0.77	0.39	152.38	0.43	0.31
Procedure Iterations	142	18	236	100	96	78	51	1075	70	45
Clause Iterations	276	35	740	548	182	142	107	3369	161	88
CPU Time (5)	1.40	0.01	2.50	2.32	1.14	0.77	0.39	26.28	0.43	0.31
CPU Time (2)	1.36	0.01	2.48	1.74	1.14	0.77	0.39	10.25	0.43	0.31

Table 3: Computation Results

	Type Graphs (Principal Functors)						Comparison					
Programs	NI	CO	LI	ST	DI	HY	A	AI	AR	C	CI	CR
AR	0	0	6	1	0	3	10	10	1.00	5	5	1.00
AR1	0	0	6	4	0	0	10	10	1.00	5	5	1.00
CS	0	31 (30)	23	0	0	0	93	24	0.26	33	12	0.37
DS	0	5 (4)	29	0	1 (1)	0	59	30	0.51	29	13	0.45
BR	0	8 (8)	13	2 (2)	10 (10)	0	59	13	0.22	20	11	0.55
KA	0	11 (11)	20	27	13 (1)	2	124	34	0.27	45	22	0.49
LDS	0	5 (4)	39	0	1 (1)	0	61	40	0.66	31	23	0.56
LPE	0	6 (6)	25	8 (3)	6	4	63	40	0.66	19	19	1.00
LPL	0	9 (9)	10	7 (3)	0	1	33	15	0.45	14	8	0.57
PE	0	6 (6)	23	8 (3)	6	4	63	38	0.60	19	19	1.00
PG	0	6 (6)	14	0	0	0	31	14	0.45	10	7	0.70
PL	0	9 (9)	5	7 (3)	0	1	33	10	0.30	14	8	0.57
PR	0	19 (19)	24	24 (20)	10 (6)	0	144	32	0.22	53	22	0.41
QU	0	1 (1)	6	0	0	0	11	6	0.55	5	4	0.80
RE	2 (2)	6 (6)	28	1 (1)	8 (2)	3	123	37	0.30	43	27	0.63
Mean									0.50			0.67

Table 4: Accuracy Results: Output Tags

	Type Graphs						Comparison					
Programs	NI	CO	LI	ST	DI	HY	A	AI	AR	C	CI	CR
AR1	0	0	2	0	0	0	10	2	0.20	5	1	0.20
AR	0	0	2	0	0	0	10	2	0.20	5	1	0.20
CS	0	9 (8)	14	0	0	0	93	15	0.16	33	10	0.30
DS	0	2 (1)	15	0	1 (1)	0	59	16	0.27	29	12	0.41
BR	0	0	5	1 (1)	10 (10)	0	59	5	0.08	20	5	0.25
KA	0	2 (2)	13	18 (18)	7 (1)	2	124	21	0.17	45	18	0.40
LDS	0	2 (1)	23	0	1 (1)	0	61	24	0.39	31	13	0.42
LPE	0	0	18	5 (3)	0	0	63	20	0.32	19	14	0.74
LPL	0	3 (3)	12	4 (3)	0	1	33	14	0.42	14	10	0.71
PE	0	0	8	5 (3)	0	0	63	10	0.16	19	6	0.32
PG	0	5 (5)	7	0	0	0	31	7	0.22	10	5	0.50
PL	0	3 (3)	1	4 (3)	0	1	33	3	0.09	14	3	0.21
PR	0	9 (9)	18	9 (7)	5 (3)	0	144	22	0.15	53	19	0.36
QU	0	0	2	0	0	0	11	2	0.18	5	2	0.40
RE	1	2 (2)	10	1 (1)	5 (2)	3	123	16	0.13	43	14	0.33
Mean									0.21			0.38

Table 5: Accuracy Results: Input Tags

atom). For each program, we extract the tag of each procedure argument. These tags will allow us to generate more efficient code by avoiding tests and specializing indexing. Hence the analysis should infer as many tags as possible. In addition, we compare the information so obtained with the information produced by an analysis preserving only principal functors, i.e. the pattern domain of [12]. The type analysis described here is always more precise than the pattern domain and the gain can come from disjunctive and recursive types. Note also that when the pattern domain infers a single functor for an argument, so does our type analysis. The results are described in Tables 4 and 5 for the output and input tags respectively. A column is associated with each tag and contains the number of arguments whose tag corresponds to the column. We also give in parenthesis the number of arguments inferred by a principal functor analysis when this number is non-zero. Columns `A`, `AI` and `AR` represent the number of arguments, the numbers of arguments for which the type analysis improves over the functor analysis (i.e. infer more tag information) and the ratio between the last two figures. The last three figures collect the same information at the clause level with the understanding that a clause is improved if any of its arguments is inferred more precisely. The results indicate that type analysis significantly improves a principal functor analysis. In the average, the type analysis produces an improvement on about 50% of the output tags and about 21% of the input tags. The tag information is improved in 67% of the clauses (output) and 38% of the clauses (input). Most of the improvement is divided into the tags `LI`, `DI`, `ST`, and `HY` with a majority of the tags being lists. The results also show that the combination of type and freeness analysis should produce significant improvement in code generation, since the two analyses are complementary.

9 Conclusion

In this paper, we have described a sophisticated type analysis system for Prolog. The system is based on abstract interpretation and uses three main components: a fixpoint algorithm, a generic pattern domain, and the type graph domain of Bruynooghe and Janssens. The main contribution of our work is to show that type analysis of Prolog based on type graphs can be engineered to be practical without sacrificing efficiency. This has implications beyond type analysis since type graphs are used for a variety of other analysis such as termination and compile-time garbage collection. The key technical contribution of this work is a novel widening operator which appears to be rather accurate and effective in keeping the sizes of the graphs, and hence the computation time, reasonably small.

There are many ways to extend this work. A natural extension is to consider integrated type graphs which allow variable-vertices and should enable difference-list programs to be handled precisely. Another extension consists of providing a database of types that the widening can use whenever necessary. Finally, on the theoretical level, it would be interesting to characterize for which classes of programs our widening is optimal in accuracy.

10 Acknowledgments

Stimulating discussions with David MacAllester are gratefully acknowledged.

References

[1] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[2] M Bruynooghe and G Janssens. An Instance of Abstract Interpretation: Integrating Type and Mode Inferencing. In *Proc. Fifth International Conference on Logic Programming*, pages 669–683, Seattle, WA, August 1988.

[3] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *21st Annual ACM SIGPLAN–SIGACT Symposium on Principles Of Programming Languages*, Portland, OR, January 1994.

[4] P Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, CA, 1977.

[5] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[6] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. *Software Practice and Experience*, 23(4), April 1993.

[7] T. Fruehwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic Programs as Types for Logic Programs. In *IEEE 6th Annual Symposium on Logic in Computer Science*, pages 300–309, 1991.

[8] N. Heintze. Practical Aspects of Set-based Analysis. In *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, November 1992.

[9] N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 197–209, 1990.

[10] G. Janssens and M. Bruynooghe. Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.

[11] T. Kanamori and T. Kawamura. Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation. Technical report, ICOT, 1987.

[12] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems.* To appear. An extended abstract appeared in the Proceedings of Fourth IEEE International Conference on Computer Languages (ICCL'92), San Francisco, CA, April 1992.

[13] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, CS Department, Brown University, 1992.

[14] K. Marriott and H. Sondergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. North American Conference on Logic Programming, Cleveland, Ohio, 1989.

[15] P. Mishra. Towards a Theory of Types in Prolog. In *International Symposium on Logic Programming*, pages 289–298, 1984.

[16] B. Monsuez. Polymorphic Types and Widening Operators. In *International Workshop on Static Analysis (WSA-93)*, Padova, Italy, September 1993.

[17] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Seventh International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990.

[18] K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, 1992.

[19] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Ma, 1986.

[20] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[21] K. Verschaetse and D. De Schreye. Deriving Termination Proofs for Logic Programs Using Abstract Procedures. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[22] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(4), 1992.