NORTH-HOLLAND

TYPE ANALYSIS OF PROLOG USING TYPE GRAPHS*

PASCAL VAN HENTENRYCK, AGOSTINO CORTESI, AND BAUDOUIN Le CHARLIER

▷ Type analysis 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 optimization 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 contribution to achieve this result is 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. ◁

1. INTRODUCTION

Although Prolog is an untyped language, type analysis of the language is important since it allows the improvement of indexing, to specialize unification, and to pro-

*This paper is a revised and extended version of [27].

Address correspondence to Pascal Van Hentenryck, Brown University, Box 1910, Providence, RI 02912, or Agostino Cortesi, University of Venezia, Via Torino 155, I-30170 Mestre-VE, Italy, or Baudouin Le Charlier, University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium.

Received February 1994; revised June 1994; accepted July 1994..

THE JOURNAL OF LOGIC PROGRAMMING

0743-1066/95/$9.50
655 Avenue of the Americas, New York, NY 10010 SSDI 0743-1066(94)00021-W

duce 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 must 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 by the grammar

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

then at most two tests are necessary to select the appropriate clause to execute.[1] Note that a recursive type is needed because of the recursive call. Information about the functor of the second argument would only allow the specialization of 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 [12], was initiated by Mishra [19] and further developed by many authors (see [8] for a complete account). Mishra introduced the idea of argument closure in type inference of logic programs. This idea was used subsequently by Yardeni and Shapiro [30] who introduced the idea of approximating the traditional T_p operator by replacing substitutions by sets of substitutions and by using argument-closure to ignore interargument dependencies. This approach was further refined by Heintze and Jaffar [12] who introduced a more precise closure operator which, informally speaking, ignores intervariable dependencies instead of interargument dependencies. The resulting inference problem was shown to be decidable using a reduction to set constraints. By reducing the problem to the inference of (a subclass of) monadic logic programs, Fruehwirth et al. [8] 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 analysis is that the relationships between predicate variables 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. There are, however, solutions around this problem such as the combination with traditional abstract interpretation approaches. A type inference system based on this approach was developed by Heintze [11], 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 work of Bruynooghe and Janssens (e.g. [2,13]) which is based on a traditional abstract interpretation approach [5]. The key idea is to approximate a collecting semantics of the language by an abstract semantics

[1] Types are especially useful when combined with mode analysis, but they would specialize the code even if mode information is not available or not accurate enough.

where sets of substitutions are described by type graphs, i.e., disjunctive rational trees.[2] 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 abstract interpretation are the possibility of exploiting variable dependencies, the control offered to the designer to choose the tradeoff between accuracy and efficiency, and the ease with which type analysis can be combined with other analysis as required by applications such as compile-time garbage collection [21]. 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 endeavor. This approach has been implemented in a prototype system [13], 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 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 [17,7],[3] Pat is a generic pattern domain for structural information [4], and `Type` is a type graph domain [13]. 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 as compile-time garbage collection (e.g., [21]) and automatic termination analysis (e.g., [28]). The technical contribution to obtain this result is 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. 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 [20].

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 the type graph domain. Section 7 describes in detail the widening operator. Section 8 gives some details about the implementation. Section 9 reports the experimental results. Section 10 concludes the paper.

[2] Type graphs can, in fact, be seen as a data structure to represent tree automata or monadic logic programs. See Section 6.7 in this paper.

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

2. AN ILLUSTRATION OF THE FUNCTIONALITY OF THE TYPE SYSTEM

The purpose of this section is to illustrate the behavior 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 9.

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 $(\beta_{in}, p, \beta_{out})$ 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 tree grammars since there is a close analogy between grammars and type graphs (see Section 6.7). 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

$$\begin{array}{lll}
\texttt{T} & ::= & \texttt{[]} \mid \texttt{cons}(\texttt{T}_1,\texttt{T}). \\
\texttt{T}_1 & ::= & \texttt{c(Any)} \mid \texttt{d(Any)}. \\
\texttt{S} & ::= & \texttt{0} \mid \texttt{c(Any,S)} \mid \texttt{d(Any,S)}.
\end{array}$$

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

$$\begin{array}{lll}
\texttt{T} & ::= & \texttt{[]} \mid \texttt{cons}(\texttt{T}_1,\texttt{T}_2). \\
\texttt{T}_1 & ::= & \texttt{c(Any)} \\
\texttt{T}_2 & ::= & \texttt{cons}(\texttt{T}_3,\texttt{T}) \\
\texttt{T}_3 & ::= & \texttt{d(Any)} \\
\texttt{S} & ::= & \texttt{0} \mid \texttt{d}(\texttt{Any},\texttt{S}_1) \\
\texttt{S}_1 & ::= & \texttt{c(Any,S)}
\end{array}$$

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 nested lists and an accumulator. Given the input pattern get(Any), the analysis system returns the output pattern get(T) where

$$\begin{array}{lll}
\texttt{T} & ::= & \texttt{[]} \mid \texttt{cons}(\texttt{T}_1,\texttt{T}). \\
\texttt{T}_1 & ::= & \texttt{[]} \mid \texttt{cons}(\texttt{T}_2,\texttt{T}_1). \\
\texttt{T}_2 & ::= & \texttt{a} \mid \texttt{b}.
\end{array}$$

The analysis time is about 0.09 seconds. The example illustrates well how the nested 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

```
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) :- llist(X), reverse(X,Res).
```

FIGURE 1 A Prolog program manipulating nested 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.

information in arithmetic expressions. For the input pattern add(Any,Any), the analysis produces the optimal output pattern add(T,S) where

$$\begin{array}{lll}
\texttt{T} & \texttt{::=} & \texttt{T + T}_1 \mid \texttt{0.} \\
\texttt{T}_1 & \texttt{::=} & \texttt{T}_1 * \texttt{T}_2 \mid \texttt{1.} \\
\texttt{T}_2 & \texttt{::=} & \texttt{cst(Any)} \mid \texttt{par(T)} \mid \texttt{var(Any).} \\
\texttt{S} & \texttt{::=} & \texttt{[]} \mid \texttt{cons(Any,S).}
\end{array}$$

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 program on arithmetic expressions depicted in Figure 3, which requires the widening procedure to be rather sophisticated. For the input pattern add(Any,Any), the analysis produces the optimal output pattern add(T,S) where

$$\begin{array}{lll}
\texttt{T} & \texttt{::=} & \texttt{T}_1 \mid \texttt{T + T}_1\texttt{.} \\
\texttt{T}_1 & \texttt{::=} & \texttt{T}_2 \mid \texttt{T}_1 * \texttt{T}_2\texttt{.} \\
\texttt{T}_2 & \texttt{::=} & \texttt{cst(Any)} \mid \texttt{var(Any)} \mid \texttt{par(T).} \\
\texttt{S} & \texttt{::=} & \texttt{[]} \mid \texttt{cons(Any,S).}
\end{array}$$

The analysis time is about 0.56 seconds. The difficulty in this example is to prevent the widening operator from mixing the definition of T, T_1, and T_2 and replacing them

```
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.

by a rule subsuming them all but losing accuracy, e.g.,

```
T ::= T + T | T * T | cst(Any) | var(Any) | par(T).
```

To achieve this behavior, it is necessary to postpone the widening until the structure of the type appears clearly, as explained later in the paper. Consider now the following program:

```
succ([],[]).
succ([X|Xs],[s(X)|R]) :- succ(Xs,R).


gen([]).
gen([0 | L]) :- gen(X),succ(X,L).
```

Its success set, which cannot be represented exactly by a type graph, is the infinite set of lists

```
[]
[0]
[0,s(0)]
[0,s(0),s(s(0))]
...
```

The difficulty here is that the lists and the integers are increasing in size at the same time. Hence, the widening must infer both recursive structures simultaneously. For the input pattern gen(Any), our analysis produces the output pattern gen(T), where

```
T  ::= [] | cons(T1,T).
T1 ::= 0 | s(T1).
```

The analysis time is about 0.07 seconds. To conclude the positive examples, 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).
```

```
qsort(X1 , X2 ) :-
   qsort( X1 , X2 , [] ).

qsort([] , L , L ).
qsort([F|T] , O , A ) :-
   partition( T , F , Small , Big ),
   qsort( Small , O , [F|Ot] ),
   qsort( Big , Ot , A ).
```

FIGURE 4 The quicksort program.

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

The weakness of our analyzer appears when dealing with difference-lists or partially instantiated data-structures. Consider the quicksort program partially described in Figure 4. If the order of the two recursive calls is switched, the analyzer concludes that both arguments are of the type

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

However, in the order given, the analyzer only returns the type

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

for the second argument. The loss of precision comes from the fact that Ot is a variable when the first recursive call takes place, and hence no information can be deduced on its type. A remedy to this problem would consist of introducing variable-vertices in the type graphs and sophisticated equality constraints between the various nodes, as discussed in the conclusion.[4] We intentionally avoided to do so, since this adds even more complexity to the domain and the feasibility of the simpler case was not even demonstrated.

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 [17,4].

4. THE ABSTRACT INTERPRETATION FRAMEWORK

In this section, we briefly review our abstract interpretation framework. The framework is presented in detail in [17], and is close to the work of Marriott and

[4] This means moving from what Bruynooghe and Janssens call rigid type graphs to what they call integrated type graphs.

Søndergaard [18] and Winsborough [29]. It follows the traditional approach to abstract interpretation [5].

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 set 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 of abstracting a set of substitutions by a single abstract substitution, 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 set 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 of computing the least fixpoint or a postfixpoint of the abstract semantics. GAIA [17] is a top-down algorithm computing a small, but sufficient subset of least fixpoint (or of a postfixpoint) necessary to answer a user query. The algorithm uses memorization, 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 [22], and can be seen either as an implementation of Bruynooghe's framework [1] or as an instance of a general fixpoint algorithm [16]. In the experimental results, we use the prefix version of the algorithm [7].

5. THE GENERIC PATTERN DOMAIN

Motivation. In the system of Bruynooghe and Janssens [13], the type graph domain is enhanced by a same-value component to maintain equalities between subterms in order to improve accuracy. The generic pattern domain Pat($\mathcal{R}$) used here was motivated by the same considerations. Moreover, in addition to preserving equalities between subterms, Pat($\mathcal{R}$) also maintains sure type information (i.e., the functor associated with a subterm and its arguments) during and across clause executions. The main advantages of using Pat($\mathcal{R}$) are the resulting simplification of the design and implementation of the domain and the factorization of sure type information. The simplification of the implementation is due to the fact that Pat($\mathcal{R}$) updates any domain with sure type information and equalities between subterms. Hence, given GAIA and Pat($\mathcal{R}$), the type analysis only requires implementing the type graph domain (about 3,500 lines out of about 15,000 lines). The factorization of sure type information enables the analyzer to reduce the size of the type graphs. We have not tried to measure the impact of this feature experimentally. More generally, the results described in this paper should hold even if Pat($\mathcal{R}$) is replaced by the same-value component of Bruynooghe and Janssens, although, once again, we have not tried to verify this experimentally. In the rest of this section, we briefly recall the basic notions behind the generic abstract domain Pat($\mathcal{R}$). See [4] for a general account on Pat($\mathcal{R}$).

Informal Description of Pat($\mathcal{R}$). 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 [14], 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 an $\mathcal{R}$-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 $\langle t_1, \ldots, t_p \rangle$ 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 whether 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 7.

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 $1 \cdots n$ if n subterms are considered. For instance, the substitution $\{x \leftarrow t * a, x_2 \leftarrow a, x_3 \leftarrow y_1 \backslash [\,]\}$ will have 7 subterms. The association of indices to them could be, for instance,

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

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

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

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

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

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,T), (6,Any)} where T would be described as

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

Interaction Between Pat($\mathcal{R}$) *and the Type Graph Domain.* The interaction between Pat($\mathcal{R}$) and the type graph domain occurs mainly when Pat($\mathcal{R}$) is about to lose information, i.e., when a nonleaf subterm is about to be replaced with a leaf. The loss of information may happen in two situations:

1. when computing the result of a procedure from its clauses, i.e., in operation UNION of GAIA;
2. when applying a widening operation to avoid computing the solutions of infinitely many different input patterns, i.e., in operation WIDEN of GAIA.

In both situations, the operations receive two abstract substitutions and return an upper-bound of these substitutions. It may happen that the same subterm is bound to two different functors in the two substitutions. When computing an upper-bound of two terms with different functors, the indices appearing in the subtrees of these two terms are removed from Pat($\mathcal{R}$) and replaced by an equivalent type graph in Type.

6. THE TYPE GRAPH DOMAIN

In this section, we present the design of the domain Type. We use the type graph domain of Bruynooghe and Janssens [13] which can be seen as a data-structure to represent tree grammars or monadic logic programs. The result described here could be recast in terms of tree grammars and/or monadic predicates, but type graphs seem more appropriate and more intuitive to handle the algorithmic issues involved in the widening operator. The purpose of this section is to introduce the basic terminology on type graphs, their meaning, and a number of cosmetic and pragmatic restrictions. The connections with tree grammars and monadic logic programs is also made.

6.1. Type Graphs

Our type graphs are essentially what Bruynooghe and Janssens call rigid types, and readers are referred to [13] for a complete coverage of type graphs. Our presentation uses more algorithmic concepts to simplify the rest of the presentation.

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). A vertex v in a type graph g is associated with the following information:

- its type, denoted by type(v), which is an element of {Any, functor, or};
- its functor, denoted by functor(v), which is a string and is associated with the vertex only if type(v) = functor;
- its arity, denoted by arity(v), which is a natural number; the arity is strictly positive if type(v) = or and 0 if type(v) = Any.

In the following, type graphs are denoted by the letter g and vertices by the letter v, both possibly subscripted or superscripted. The successors of a vertex v are denoted by succ(v), and the ith successor of v is denoted by succ(v,i) for $1 \leq$ i $\leq$ arity(v). If v is successor of v' then v' is a predecessor of v. The set of predecessors of v is denoted by pred(v). We assume in the following that the successors of an or-vertex are sorted by functor names for simplicity.

Note that more types (e.g., Integer, Real) can be added easily without affecting the results described here.

6.2. Denotation

The denotation of a type graph g, denoted by Cc(g), is depicted in Figure 5. In the figure, ST denotes the set of all terms, SST the powerset of ST, and lfp is the least fixpoint operator. Note that the transformation $\mathcal{D}$ implicitly depends on the graph considered. Note also that, in the following, we also discuss the denotation of a vertex v in a graph g, i.e., lfp($\mathcal{D}$)(v), and we use Cc_g(v) to denote it.

6.3. Additional Definitions

The following definitions will be useful subsequently. We assume for simplicity an underlying type graph g.

A path is a sequence $\langle v_1, \ldots, v_n \rangle$ of vertices, satisfying $v_i \in$ succ(v_{i-1}) ($2 \leq i \leq n$). The length of path $\langle v_1, \ldots, v_n \rangle$ is n. The depth of a vertex v, denoted by depth(v), is the length of the shortest path from root(g) to v. A path from the root to a vertex can be uniquely identified by a sequence of integers, each integer

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

$\mathcal{D}$:  V → SST → V → SST
$\mathcal{D}\ \Phi$ v =
    if type(v) = Any then ST
    else if type(v) = or then $\bigcup_{1 \leq i \leq \texttt{arity(v)}} \Phi(\texttt{succ(v,i)})$
    else $\{\texttt{f}(\texttt{t}_1,\ldots,\texttt{t}_{\texttt{arity(v)}}) \mid \texttt{f} = \texttt{functor(v)} \,\&\, \texttt{t}_i \in \Phi(\texttt{succ(v,i)})\}$.
```

FIGURE 5 The denotation of type graphs.

denoting which successor to select. More formally, given a type graph g and a path p, the path p identifies the vertex `follow(root(g),p)` where

$$\texttt{follow(v,[])} = \texttt{v};$$
$$\texttt{follow}(\texttt{v},[\texttt{i}_1,\ldots,\texttt{i}_n]) = \texttt{follow}(\texttt{succ(v,i)},[\texttt{i}_2,\ldots,\texttt{i}_n]).$$

An ancestor of a vertex v is any vertex $\texttt{v'} \neq \texttt{v}$ on the shortest path from `root(g)` to v. The set of ancestors of v is denoted by `ancestor(v)`. By definition, this set is empty if the vertex is not connected to the root. A cycle is a path $\langle\ \texttt{v}_1,\ldots,\texttt{v}_n,\ \texttt{v}_1\ \rangle$. A cycle $\langle\ \texttt{v}_1,\ldots,\texttt{v}_n,\ \texttt{v}_1\ \rangle$ is *canonical* if $\texttt{depth}(\texttt{v}_1) < \cdots < \texttt{depth}(\texttt{v}_n)$.

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

$$\texttt{removeUnconnected}(\langle(\texttt{V},\texttt{E}),\texttt{r}\rangle) = \langle(\texttt{v'},\texttt{E'}),\texttt{r}\rangle \text{ where}$$
$$\texttt{v'} = \{\texttt{v} \mid \texttt{v} \in \texttt{V} \,\&\, \texttt{r} \in \texttt{ancester(v)}\} \cup \{\texttt{r}\}.$$
$$\texttt{E'} = \{(\texttt{v},\texttt{v'}) \mid (\texttt{v},\texttt{v'}) \in \texttt{E} \,\&\, \texttt{v},\texttt{v'} \in \texttt{v'}\}.$$

Finally, the principal functor set (or pf-set for short) of an or-vertex, v, denoted by `pf(v)`, is the set of functors of its successors, i.e.,

$$\texttt{pf(v)} = \{\texttt{functor(v')} \mid \texttt{v'} \in \texttt{succ(v)} \,\&\, \texttt{type(v')} = \texttt{functor}\}.$$

The definition is generalized to functor-vertices by defining `pf(v)` = { `functor (v)`} if v is a functor-vertex and to any-vertices by defining `pf(v)` = $\emptyset$ if v is an any-vertex.

6.4. Cosmetic Restrictions

Our system enforces a number of cosmetic restrictions on type graphs. These restrictions do not reduce the expressiveness of the type system, but enable us to simplify the algorithms and proofs of correctness. The cosmetic restrictions are as follows.

- **Flip-Flop:** Successors of functor-vertices are or-vertices. Successor of or-vertices are functor-vertices or any-vertices. The root of the graph is an or-vertex.
- **Or-Cycle:** In every canonical cycle $\langle\texttt{v}_1,\ldots,\texttt{v}_n,\texttt{v}_1\rangle$, the initial (and final) `vertex` $\texttt{v}_1$ is an or-vertex.
- **No-Sharing:** The graph obtained by removing the last edge of every canonical cycle is a tree.

- **Isolated-Any:** If $\{\mathtt{v}_1, \ldots, \mathtt{v}_n\}(\mathtt{n} > 1)$ are the successors of an or-vertex, then no $\mathtt{v}_i(1 \leq i \leq n)$ is an any-vertex.

The No-Sharing restriction implies that a type graph is a tree enhanced by a number of edges introducing cycles. The Or-Cycle restriction, in conjunction with the Flip-Flop restriction, implies that the last edge of a canonical cycle goes from a functor-vertex to an or-vertex. Note that some of these restrictions are related to some of the restrictions required by Bruynooghe and Janssens.

6.5. *Principal Functor Restriction*

We now introduce a restriction which reduces the expressiveness of type graphs. This restriction, called the principal functor restriction in [13], is used in many systems (e.g., [19, 30, 13]).

- **Principal Functor Restriction:** if $\{\mathtt{v}_1, \ldots, \mathtt{v}_n\}(\mathtt{n} > 1)$ are the successors of an or-vertex,

$$\mathtt{functor}(\mathtt{v}_i) \neq \mathtt{functor}(\mathtt{v}_j) \qquad (1 \leq i < j \leq n).$$

Informally speaking, this restriction requires the successors of an or-vertex to have different functors. It reduces the expressiveness of the type system since, for instance, any type graph including f(a,b) and f(b,a) in its denotation will also contain f(a,a) and f(b,b). This restriction in expressiveness has been formalized in the context of tree automata (see Section 6.7).

6.6. *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

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

6.7. *Relation to Regular Tree Grammars*

Type graphs can be seen as a data-structure to represent regular tree grammars [10, 25]. This correspondence is systematically exploited in this paper to display the results. It can be formalized by associating a non-terminal symbol $\mathtt{T}_\mathtt{v}$ with each vertex v. The grammar rule associated with an or-vertex v with successors $\mathtt{v}_1, \ldots, \mathtt{v}_n$ is simply

$$\mathtt{T}_\mathtt{v} ::= \mathtt{T}_{\mathtt{v}_1} \mid \cdots \mid \mathtt{T}_{\mathtt{v}_n}.$$

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

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

For our application, regular tree grammars need to be extended with a special terminal symbol Any which recognizes any tree. The rule associated with an any-vertex simply becomes

$$\mathtt{T}_v ::= \mathtt{Any}.$$

The initial symbol of the regular tree grammar is the nonterminal symbol associated with the root of the graph.

To define the tree generated by a regular tree grammar, it is useful to put the grammar in normal form [10]. In our context, this means that all the rules must of the form

$$\texttt{T ::= f(T}_1\texttt{,...,T}_n\texttt{).}$$

or of the form

```
T ::= Any.
```

A regular tree grammar can always be transformed into an equivalent grammar in normal form. We are now in a position to define the trees recognized (or generated) by a regular tree grammar.

A tree $\texttt{f(t}_1\texttt{,...,t}_n\texttt{)}$ is recognized by a nonterminal symbol T if there exists a rule

```
T ::= Any.
```

or if there exists a rule

$$\texttt{T ::= f(T}_1\texttt{,...,T}_n\texttt{).}$$

and t_i is recognized by $T_i(1 \leq i \leq n)$. The trees recognized (or generated) by a regular tree grammar are the set of all trees recognized by its initial symbol.

Note that a regular tree grammar satisfies the principal functor restriction if, for any nonterminal symbol T and functor f, there is atmost one rule of the form

$$\texttt{T ::= f(T}_1\texttt{,...,T}_n\texttt{).}$$

There are several points worth mentioning here. The trees recognized by regular tree grammars are exactly the trees recognized by nondeterministic bottom-up or top-down tree automata.[5] Moreover, the trees recognized by regular tree grammars satisfying the principal functor restriction are exactly the trees recognized by deterministic top-down tree automata. It is a basic result of tree automata theory that determinism is a real limitation of top-down tree automata. This comes from the fact that disjoint subtrees are recognized independently. This inability to share information between subtrees is compensated in nondeterministic tree automata by their ability to guess the correct relationships. See also [10] for a formal characterization of the set of trees recognized by deterministic tree automata in terms of a closure property.

Note also that, in the examples, we sometimes relax the cosmetic restrictions to improve clarity.

6.8. *Relation to Monadic Logic Programs*

Type graphs can also be related to monadic logic programs of [30, 8]. The logic program associated with a type graph succeeds for all well-typed terms. A simple way is to associate a procedure $\mathrm{p_v}$ with each vertex v. The procedure for an any-vertex is simply

```
any(X).
```

[5] Informally speaking, a state in a nondeterministic top-down tree automation corresponds to a nonterminal and every grammar rule corresponds to a transition. Note that, in the terminology of [10], frontier-to-root is used instead of bottom-up and root-to-frontier is used instead of top-down.

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

```
pv(f(X1,...,Xn)) :- pv1(X1),...,pvn(Xn).
```

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

```
pv(X) :- pv1(X).
...
pv(X) :- pvn(X).
```

Note 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)
```

6.9. Operations on Type Graphs

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

1. $g_1 \leq g_2$: returns true if $Cc(g_1) \subseteq Cc(g_2)$;
2. $g_1 \cap g_2$: returns g_3 such that $Cc(g_1) \cap Cc(g_2) \subseteq Cc(g_3)$.
3. $g_1 \cup g_2$: returns g_3 such that $Cc(g_1) \cup Cc(g_2) \subseteq Cc(g_3)$.

The first two operations are described in [13]. The first operation can be used directly, while the second needs to be adapted slightly to enforce our cosmetic restrictions, although there is no difficulty in doing so. Note that intersection is used for unification since our type graphs are downward-closed. The third operation is not described in [13] 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 principal functor restriction in the case of or-vertices by applying the algorithm recursively. Of course, memoization is used to guarantee termination.

Note also that, in the following, we often use operation $\leq$ on vertices to denote inclusion of their denotation. The algorithm is the same as for type graphs.

7. 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 [13] use a finite subdomain by restricting the number of occurrences of a functional symbol on the paths of the graphs. We adopted a different solution based on a widening operator as proposed in [5]. 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 show 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.

7.1. 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 that the result of a procedure 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 g_{old} (e.g., the previous result of a procedure) and a new graph g_{new} (e.g., the union of the new clause results) to produce a new graph g_{res} (e.g., the new result of the procedure). The main idea behind our widening operator is to consider two graphs

$$g_o = g_{old} \quad \text{and} \quad g_n = g_{old} \cup g_{new}$$

and to exploit the topology of the graphs to guess where g_n is growing compared to g_o. The key notion is the concept of topological clash which occurs in situations where

- an or-vertex v_o in g_o corresponds to an or-vertex v_n in g_n where $pf(v_o) \neq pf(v_n)$;
- an or-vertex v_o in g_o corresponds to an or-vertex v_n in g_n where $depth(v_o) <$ $depth(v_n)$.

In these cases, the widening operator tries to prevent the graph from growing by introducing a cycle in g_n. Given a clash (v_o, v_n) the widening searches for an ancestor v_a to v_n such that $pf(v_n) \subseteq pf(v_a)$. If such an ancestor is found and if $v_a \geq v_n$, a cycle can be introduced.

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

```
To ::= [] | cons(Any,T1).
T1 ::= [].
```

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

```
Tnew ::= [] | cons(Any,T2).
T2   ::= [] | cons(Any,T3).
T3   ::= [].
```

Taking the union of T_o and T_{new} produces the type graph described by T_{new}.

There is a topological clash between T_o and T_{new} for the path [2, 2], which corresponds to the nonterminals T_1 and T_2, respectively. The widening selects T_{new} as an ancestor and introduces a cycle producing the final result

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

Note also that an ancestor at any depth can be selected. For the first arithmetic program (See Figure 2), the widening applies to the type graphs T_o and T_n depicted

T_o ::= 0 | 0 + T_1.
T_1 ::= 1 | T_1 * T_2.
T_2 ::= cst(Any) | par(0) | var(Any).

T_n ::= 0 | T_3 + T_6.
T_3 ::= 0 | 0 + T_4.
T_4 ::= 1 | T_4 * T_5.
T_5 ::= cst(Any) | par(0) | var(Any).
T_6 ::= 1 | T_6 * T_7.
T_7 ::= cst(Any) | par(T_3) | var(Any).

FIGURE 6 Widening for the first arithmetic program.

in Figure 6. Consider the clash occurring for the path [2, 2, 2, 2, 2, 1] for T_o and T_n. An appropriate ancestor for T_3 is T_n, which is not a direct ancestor. This results in the optimal result T_r.

T_r ::= 0 | T_r + T_1.
T_1 ::= 1 | T_1 * T_2.
T_2 ::= cst(Any) | par(T_r) | 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 of 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 to the arithmetic program, the second iteration for the predicate basic/2 requires a widening for the first argument with the following two graphs:

T_o ::= cst(Any) | var(Any).
T_n ::= cst(Any) | par(0) | var(Any).

A topological clash is encountered, but there is no suitable ancestor. The result will simply be T_n in this case. Letting the graph grow in this case is of great importance to recover the structure of the type in its entirety.

The last case to consider appears when there is an ancestor v_a with a suitable pf-set, but unfortunately, $v_a \geq v_n$ is false. In this case, introducing a cycle would produce a graph T_r whose denotation may not include the denotation of T_n, and hence our widening operator cannot perform cycle introduction. Instead, the widening operation replaces v_a by a new or-vertex which is an upper bound to v_a and 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 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.

7.2. The Widening and Its Formal Properties

The following abbreviations will also be useful in this section.

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

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 $S \downarrow i = \{s \downarrow i \mid s \in S\}$.

7.2.1. 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 7.1. 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. Informally speaking, a topological clash occurs when two vertices in correspondence have different pf-sets or different depths.

Definition 7.2. Let $\texttt{g}_1,\texttt{g}_2$ be the two type graph 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:

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

The following proposition is an immediate consequence of the definitions. Informally speaking, it says that topological clashes only occur at or-vertices.

Proposition 7.1. Let $\texttt{g}_1,\texttt{g}_2$ be two type graphs such that $\texttt{g}_1 \leq \texttt{g}_2$. If $(\texttt{v}_1,\ \texttt{v}_2) \in \texttt{TC}(\texttt{g}_1,\ \texttt{g}_2)$, then $\texttt{OR}(\texttt{v}_1)\ \&\ \texttt{OR}(\texttt{v}_2)$. Moreover, if $(\texttt{v}_1,\ \texttt{v}_2)$ are not the roots of the graphs, there exists a unique tuple $(\texttt{v}_a,\ \texttt{v}_a')$, denoted by $\texttt{ca}(\texttt{v}_1,\ \texttt{v}_2)$, such that $\texttt{v}_1 \in \texttt{succ}(\texttt{v}_a)$ and $\texttt{v}_2 \in \texttt{succ}(\texttt{v}_a')$.

Note also that $(\texttt{v}_1,\ \texttt{v}_2) \in \texttt{TC}(\texttt{g}_1,\ \texttt{g}_2)$ implies that $\texttt{pf}(\texttt{v}_2) = \emptyset$ (i.e., the only successor of $\texttt{v}_2$ is an any-vertex) or $\texttt{pf}(\texttt{v}_1) \subseteq \texttt{pf}(\texttt{v}_2)$. Our widening operation focuses on a subset of topological clashes which lead to a growth in the graph.

Definition 7.3. 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:

```
replaceEdge(g,e,e') = removeUnconnected(g') where
   vertices(g') = vertices(g)
   edges(g') = edges(g) $\setminus$ {e} $\cup$ {e'}
   root(g') = root(g).

replaceVertex(g,v,v') = removeUnconnected(g') where
   vertices(g') = vertices(g) $\cup$ {$v_1,\ldots,v_n$} (n $\geq$ 1)
   edges(g') = edges(g) $\setminus$ $E_1 \cup E_2 \cup E_3$
   root(g') = root(g)
   vertices(g) $\cap$ {$v_1,\ldots,v_n$} = $\emptyset$
   $E_1$ = { ($v_a$,$v_b$) | ($v_a$,$v_b$) $\in$ edges(g) & $v_b$ = v }
   $E_2$ = { ($v_a$,$v_1$) | ($v_a$,$v_b$) $\in$ edges(g) & $v_b$ = v }
   ($v_a$,$v_b$) $\in E_3 \Rightarrow v_a \in$ {$v_1,\ldots,v_n$} & $v_b \in$ vertices(g')
   $v_1 \geq$ v, v'
   size(removeUnconnected(g')) < size(g).
```

FIGURE 7 The functions replaceEdge and replaceVertex.

$$\begin{aligned} \mathtt{WTC}(g_1,\ g_2) \;=\; & \{(v_1,\ v_2) \mid (v_1,\ v_2) \in \mathtt{TC}(g_1,\ g_2)\ \&\ \mathtt{pf}(v_2) \neq \emptyset\ \& \\ & ((\mathtt{pf}(v_1) \neq \mathtt{pf}(v_2)\ \&\ \mathtt{same-depth}(v_1,\ v_2)) \vee \\ & \mathtt{depth}(v_1) < \mathtt{depth}(v_2)))\} \end{aligned}$$

7.2.2. TRANSFORMATION RULES. The widening operator essentially consists of applying two transformation rules to eliminate (a subset of) widening clashes. The transformation rules nondeterministically produce a new type graph g_r from two type graphs g_o and g_n and $g_o \leq g_n$. 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 than or equal to v and v' and decreases the size of the graph. The formal definitions of the functions are given in Figure 7. In the second definition, $\{v_1, \ldots, v_n\}$ are the new vertices added to the graphs, v_1 is intended to replace v in the new graph, E_1 are the edges into v in the old graph that are replaced by new edges into v_1 in the new graph (the set E_2), and E_3 are the remaining new edges starting from new vertices. Note the last two conditions that guarantee that the denotation of v_1 is greater than the denotations of v, v' and that the size of the new graph is smaller than the size of the old graph. 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 7.4 [Cycle Introduction Rule]. Let g_o and g_n be two type graphs, and let

$$\begin{aligned} \mathtt{CI}(g_o,\ g_n) \;=\; & \{\langle(v,\ v_n)(v,\ v_a)\rangle \mid (v_o,\ v_n) \in \mathtt{WTCg}_o,\ g_n)\ \& \\ & v_a \in \mathtt{ancestor}(v_n)\ \&\ v_a \geq v_n\ \& \end{aligned}$$

$$\mathtt{depth}(\mathtt{v}_o) \geq \mathtt{depth}(\mathtt{v}_a) \ \& \ \ \mathtt{v} = \mathtt{ca}(\mathtt{v}_o,\ \mathtt{v}_n) \downarrow 2\}.$$

The cycle introduction rule can be specified as follows:

$$\begin{aligned}
&\mathtt{TR}_i(\mathtt{g}_o,\ \mathtt{g}_n) = \mathtt{g}_r\\
&\textbf{Precondition}: \mathtt{CI}(\mathtt{g}_o,\ \mathtt{g}_n) \neq \emptyset.\\
&\textbf{Postcondition}: \mathtt{g}_r = \mathtt{replaceEdge}\ (\mathtt{g}_n,\ \mathtt{e},\ \mathtt{e'})\\
&\qquad\qquad\qquad\qquad \mathtt{for\ some(e,\ e')} \in \mathtt{CI}(\mathtt{g}_o,\ \mathtt{g}_n).
\end{aligned}$$

The following result shows that the cycle introduction rule involves the widening clash entirely.

Proposition 7.2. Let $g_r = \mathit{replaceEdge}(g_n, e, e')$ for some $(e, e') \in \mathit{CI}(g_o, g_n)$. Then,

$$\#\mathtt{WTC}(\mathtt{g}_o,\ \mathtt{g}_n) < \#\mathtt{WTC}(\mathtt{g}_o,\ \mathtt{g}_r).$$

PROOF. Assume the notations of the definition of the cycle introduction rule. If $\mathtt{same\text{-}depth}(\mathtt{v}_o, \mathtt{v}_n)$, then the result is obviously true since $\mathtt{depth}(\mathtt{v}_a) < \mathtt{depth}(\mathtt{v}_n)$. Otherwise, either $\mathtt{depth}(\mathtt{v}_a) < \mathtt{depth}(\mathtt{v}_o)$, in which case the result is trivial, or $\mathtt{depth}(\mathtt{v}_a) = \mathtt{depth}(\mathtt{v}_o)$. In this last case, $(\mathtt{v}_o,\ \mathtt{v}_a) \in \mathtt{C}(\mathtt{g}_o,\ \mathtt{g}_n)$ by definition of the cosmetic restrictions, and thus $\mathtt{pf}(\mathtt{v}_o) = \mathtt{pf}(\mathtt{v}_a)$ by definition of the topological clashes. □

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 7.5. [Replacement Rule.] Let g_o and g_n be two type graphs, and let

$$\begin{aligned}
\mathtt{CR}(\mathtt{g}_o,\mathtt{g}_n) \ = \ &\{(\mathtt{v}_n,\mathtt{v}_a) \mid (\mathtt{v}_o,\mathtt{v}_n) \in \mathtt{WTC}(\mathtt{g}_o,\mathtt{g}_n)\ \&\\
&\mathtt{v}_a \in \mathtt{ancestor}\ \ (\mathtt{v}_n)\ \& \neg(\mathtt{v}_a \succeq \mathtt{v}_n)\ \&\\
&\mathtt{depth}(\mathtt{v}_o) \geq \mathtt{depth}(\mathtt{v}_a)\ \&\\
(\mathrm{pf}(\mathtt{v}_n) \ \subseteq \ &\mathtt{pf}(\mathtt{v}_a)\ \vee\ \mathtt{depth}(\mathtt{v}_o){<}\mathtt{depth}(\mathtt{v}_n))\ \}.
\end{aligned}$$

The replacement rule can be specified as follows:

$$\begin{aligned}
&\mathtt{TR}_r(\mathtt{g}_o,\ \mathtt{g}_n){=}\mathtt{g}_r\\
&\textbf{Precondition}: \mathtt{CR}(\mathtt{g}_o,\ \mathtt{g}_n){=}\emptyset.\\
&\textbf{Postcondition}: \mathtt{g}_r{=}\mathtt{replaceVertex}(\mathtt{g}_n, \mathtt{v}_a, \mathtt{v}_n)\\
&\quad \mathtt{for\ some}\ (\mathtt{v}_n,\ \mathtt{v}_a) \in \mathtt{CR}(\mathtt{g}_o,\ \mathtt{g}_n).
\end{aligned}$$

Note that this rule only applies when $\mathtt{pf}(\mathtt{v}_n) \subseteq \mathtt{pf}(\mathtt{v}_a)$ or when there is a depth-clash, and hence it leaves room for the expansion of type graphs before the widening applies. In the case of the depth-clash, there is at least one suitable ancestor so that the rule always applies. Note also that this rule may solve the widening clash by introducing new widening clashes in the graph. However, by definition of replaceVertex, the size of the resulting graph must decrease.

7.2.3. THE WIDENING OPERATION. We are now in a position to present the widening operation. The widening essentially applies the transformation rules until the sets CI and CR are empty.

Definition 7.6 [Widening Operator]. The widening operator $g_o \bigtriangledown g_n$ is defined as follows:

```
$g_o \bigtriangledown g_n$=
 if $g_n \le g_o$ then $g_o$ else widen($g_o$, $g_o \cup g_n$).
```

```
widen($g_o$,$g_n$)=
 if CI($g_o$,$g_n$) $\neq \emptyset$ then widen($g_o$,TR$_i$($g_o$,$g_n$))
 else if CR($g_o$,$g_n$) $\neq \emptyset$ then widen($g_o$,TR$_r$($g_o$,$g_n$))
else                    $g_n$.
```

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

Proposition 7.3 (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 = size(g) and n_2 = #WTC(g_o, g). The set of these pairs becomes a well-founded set with the following total ordering:

$$(n_1, n_2) < (n_1', n_2') \quad \text{if } n_1 < n_1' \text{ or} \\ \qquad\qquad n_1 = n_1' \text{ and } n_2 < n_2'.$$

Consider now the sequence of graphs $g_1, \ldots, g_i, \ldots$ occurring as second argument of widen. We have F(g_i) < F(g_{i-1}) since the cycle introduction rule satisfies size(g_i) $\le$ size(g_{i-1}) and #WTC(g_o, g_i)< # WTC(g_o, g_{i-1}) by Proposition 7.2, while the replacement rule satisfies size(g_i) < size (g_{i-1}). □

We now prove that $\bigtriangledown$ is a widening operator. The main ideas behind the proof can be described very informally as follows. $\bigtriangledown$ lets the graph grow, when a suitable ancestor to introduce a cycle cannot be found. However, each time $\bigtriangledown$ lets the graph grow, it introduces a vertex with a new pf-set (not included in the pf-sets of its ancestors) along a branch. Hence, the more $\bigtriangledown$ is applied, the easier it becomes to find a suitable ancestor along a given branch. Eventually, the graph cannot grow along this branch. This idea is captured by the concept of potential in the proof. In some cases, the potential does not decrease, but then we can show that the cycles involve vertices closer and closer to the root. This is the idea behind the set E in the proof. We now formalize these ideas.

Theorem 7.1. Operator $\bigtriangledown$ is a widening operator.

PROOF. We first show that $g_o \bigtriangledown g_n \ge g_o, g_n$. This follows immediately from the fact that widen is applied initially on g_o and $g_o \cup g_n$ and that the transformation rules cannot decrease the denotation.

Let $g'_0, \ldots, g'_i, \ldots$ be a sequence of type graphs and $g_0, \ldots, g_i, \ldots$ a sequence defined as

$$\begin{aligned} g_0 &= g'_o \\ g_{i+1} &= g_i \nabla g'_i (i \geq 0) \end{aligned}$$

We show that $g_0, \ldots, g_i, \ldots$ is stationary. To define a well-founded set, we assume without loss of generality that a is the number of function symbols in the analyzed program and we use the following notions. The potential of a vertex v, denoted by p(v), is defined as follows:

$$\begin{aligned} p(v) &= 2^a + 1 && \text{if } pf(v) = \emptyset. \\ p(v) &= \#\{s \mid s \subseteq pf(v')\text{and } v' \in && \text{otherwise.} \quad \text{ancestor}(v) \text{ or } v = v'\} \end{aligned}$$

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

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

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

$$\begin{aligned} &\langle n_1, \ldots, n_s\rangle < \langle n'_1, \ldots, n'_s\rangle \text{ if} \\ &\quad n_1 < n'_1 \text{ or} \\ &\quad \langle n_1\rangle = \langle n'_1\rangle \text{ and } n_2 < n'_2 \text{ or} \\ &\quad \ldots \\ &\quad \langle n_1, \ldots, n_{s-1}\rangle = \langle n'_1, \ldots, n'_{s-1}\rangle \text{ and } n_s < n'_s. \end{aligned}$$

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 to the natural number

$$\sum_{i=1}^{\infty} i\ E[g,i].$$

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

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

Consider the following sets:

$$\begin{aligned} CO &= C(g_i,\ g_{i+1}) \setminus TC(g_i, g_{i+1}). \\ TC &= TC(g_i,\ g_{i+1}). \\ NE &= \{v' \in g_{i+1} \mid \neg\exists v(v,v') \in C(g_i,\ g_{i+1})\}. \\ OE &= \{v \in g_i \mid \neg\exists v'\ C(g_i,\ g_{i+1})\}. \end{aligned}$$

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

$$P[CO \downarrow 1,\ k] = P[CO \downarrow 2,\ k] \qquad (2 \leq k \leq 2^a + 1)$$

Each tuple (v,v') in TC can only be of three different forms by definition on ∇ knowing that OR(v) & OR(v'):[6]

- **different-depth:** $\neg$ same-depth(v,v');
- **new-any-vertex:** pf(v) $\neq \emptyset$ and pf(v') = $\emptyset$;
- **different functor:** same-depth (v,v') & pf(v') $\neq \emptyset$ & pf(v) $\neq$ pf(v')

In the first case, we have depth(v') < depth(v) by definition of ∇. 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 ∇ and of the potential. Two cases must be distinguished now:

- **existence of a functor clash:** there exists a tuple (v,v') in TC such that pf(v') = $\emptyset$ or pf(v) $\neq$ pf(v');
- **depth clashes only:** all other cases.

We first prove that if there exists a functor clash, then $P(g)_i > P(g_{i+1})$. The existence of a tuple (v,v') in TC such that pf(v') = $\emptyset$ or pf(v) $\neq$ pf(v') implies the existence of k such that

$$P[CO{\downarrow}1 \cup TC{\downarrow}1,\ k] > P[CO{\downarrow}2 \cup TC{\downarrow}2,\ k].$$
$$P[CO{\downarrow}1 \cup TC{\downarrow}1,\ j] = P[CO{\downarrow}2 \cup TC{\downarrow}2,\ j](2 \leq j < k).$$

It remains to consider the sets NE and OE. Note that each vertex v" in NE has an ancestor v' such (v,v') $\in$ TC for some v by definition of ∇. Moreover, p(v) < p(v') $\leq$ p(v"). It then follows that

$$P[CO{\downarrow}1 \cup TC{\downarrow}1,k] > P[CO{\downarrow}2 \cup TC{\downarrow}2 \cup NE,\ k].$$
$$P[CO{\downarrow}1 \cup TC{\downarrow}1,j] = P[CO{\downarrow}2 \cup TC{\downarrow}2 \cup NE,\ j](2 \leq j < k).$$

and hence that $P(g_i) > P(g_{i+1})$.

Consider now the remaining case. Since there exists no tuple (v,v') in TC such that pf(v') = $\emptyset$ or pf(v) $\neq$ pf(v'), the fact that depth < depth(v') for all (v,v') in TC implies that

$$P[CO{\downarrow}1 \cup TC{\downarrow}1,\ j] \geq P[CO{\downarrow}2 \cup TC{\downarrow}2,j](2 \leq j \leq 2^a+1).$$

and that NE is the empty set. It follows that $P(g_i) \geq P(g_{i+1})$. We conclude the proof by showing that if $P(g_i) = P(g_{i+1})$, then $E(g_i) > E(g_{i+1})$. $P(g_i) = P(g_{i+1})$ implies that OE is the empty set. Assuming $g_i < g_{i+1}$, there must be at least one pair (v,v') $\in$ TC and, by definition of ∇, depth(v) > depth(v'). Hence, $E(g_i) > E(g_{i+1})$. □

8. IMPLEMENTATION DETAILS

The implementation was greatly simplified by the availability of GAIA and Pat($\mathcal{R}$) which enables us to focus on the type graph domain. The implementation of the type graph domain is based on a small number of principles and optimizations.

[6] Recall that an or-vertex has an empty pf-set only if it has as its only successor an any-vertex.

First, hash-tables are used in all algorithms to memorize pairs of vertices that have been encountered already. This is useful to guarantee termination of almost all operations, including inclusion, union, and intersection. Second, some of the cosmetic restrictions are relaxed in the implementation to allow for a more compact representation of the graphs. In particular, the Flip-Flop restriction is relaxed to allow a functor-vertex to have functor-vertices and/or any-vertices as successors. Finally, in the widening operation, the computation of the widening clashes is performed at the same time as their resolutions. This enables us to speed up the algorithms considerably since, in many cases, a single traversal (or a small number of traversals) of the graph would be enough.

The main open issues that remain in our implementation of the type graph domain are how to generalize the algorithms to remove the No-Sharing and the Or-Cycle restrictions and to apply the idea of caching [7]. Lifting these restrictions should allow for a more compact representation of the type graphs.

Another important issue concerns the integration of the domain into GAIA. In its current version, GAIA creates an input pattern for each new abstract substitution encountered for some goal. Although this is appropriate for multiple specialization [29], it may be very demanding when the domain is large and contains disjunctive information. Techniques to avoid explosion of the number of input patterns should be investigated. See Section 9 for more discussion on this topic.

9. 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[7] are hopefully representative of "pure" logic programs. KA is an alpha-beta program to play the game of kalah [23]. PR is a symbolic equation-solver [23]. CS is a program to generate a number of configurations representing various ways of cutting a wood board into small shelves [26]. DS is the generate and test equivalent of a disjunctive scheduling problem [6]. RE is the Prolog tokenizer and reader of O'Keefe and Warren. PG is a program written by Older to solve a specific mathematical problem. BR is a program taken from Gabriel benchmark. PL is a planning program from [23]. QU solves the n-queens problem. Finally, PE is 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 of the size of these programs, while Table 2 reports the number of nonrecursive, tail recursive, locally recursive (more than one recursive call or a nonterminal recursive call), and mutually recursive procedures in each of the benchmarks. In Table 1, the number of goals is the number of procedure calls in the program, while the size of the static call tree is essentially the size of the static call graph, except that some of the recursive calls are removed. This measure was introduced in [15] to simplify the complexity analysis. Four programs have only tail recursive procedures or nonrecursive procedures. Many programs have mutually recursive procedures and some have many of them. In general, the majority of procedures are nonrecursive,

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

TABLE 1. Sizes of the Programs

	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 2. Syntactic Form 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

and in many programs, most of the recursive procedures are tail recursive. Program PR contains locally recursive procedures due to their divide and conquer approach. Note that mutually recursive programs are generally difficult for type analysis, as mentioned, for instance, in [11].

Computation Times. In this section, we analyze 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. Informally speaking, the number of procedure (resp. clause) iterations is the number of times a procedure (resp. clause) is analyzed in the fixpoint algorithm (see [17] for a precise description of these measures). We also give the CPU time when the number of successors to or-vertices is restricted to 5 and 2, respectively. The algorithms are then generalized to replace an or-vertex with too many successors by an any-vertex. As can be seen, the analysis is fast (below 3 seconds) for all programs except RE, which takes about 117, 23, and 9 seconds, depending on the various restrictions. Note that PR is heavily mutually recursive, that CS manipulates heavily nested 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 tokenizer 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 graphs 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 analyzer. It is also important to stress that program RE seems to be a pathological benchmark for all abstract domains of which we are aware (e.g., [17, 3]), and is not even typical of a mutually recursive program since PR is heavily mutually recursive

TABLE 3. Computation Results

	KA	QU	PR	PE	CS	DS	PG	RE	BR	PL
CPU Time	1.52	0.01	2.51	2.73	1.01	0.72	0.39	117.15	0.38	0.31
Procedure Iterations	149	18	253	109	99	78	59	1052	72	50
Clause Iterations	290	35	791	569	190	142	123	3300	165	98
CPU Time (5)	1.27	0.01	2.35	2.06	0.97	0.61	0.37	23.00	0.38	0.28
CPU Time (2)	1.23	0.01	2.25	1.69	1.02	0.71	0.35	9.19	0.43	0.31

as well. The difficulty in RE comes from the combination of mutual recursion with an abundance of functional symbols (representing priorities and precedence of operators, for instance) which produces a rich spectrum of input patterns. Presently, our analyzer allocates a new input pattern whenever needed, which can be very demanding when the domain is as precise and as large as the type graph domain. The most satisfactory solution around this problem is probably to limit the number of input patterns for each procedure by collapsing them. Imposition of restrictions on the size of the graphs or the out-degree of vertices as shown in the table are another solution. Overall, the results are very encouraging, and seem to indicate that type graphs can be engineered to be practical. The tradeoff between efficiency and accuracy obviously remains 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 input/output patterns of the analysis under the following assumptions. First, no multiple specializations take place, i.e., a procedure is associated with a single version. This assumption is motivated by the inherent difficulty in comparing polyvariant analyses since they may have fundamentally different input/output patterns. Second, we consider the following tag information: NI (empty list), CO (cons), LI (list), ST (structure), DI (atom), and HY (structure or 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 factors, i.e., the pattern domain of [17] which can be seen as an instantiation of Pat($\mathcal{R}$) with mode and sharing. These components play no role in the type analysis, but allow the computation of freeness. This domain is roughly equivalent to the domain of Taylor [24]. 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 parentheses the number of arguments inferred by a principal functor analysis when this number is nonzero. 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. On average, the type analysis produces an improvement in 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 (as described in this paper) and freeness (as described in many other papers) analysis should produce significant improvement in code generation since the two analyses are complementary.

TABLE 4. Accuracy Results: Output Tags

	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

10. 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 accuracy. This has implications beyond type analysis since type graphs are used for a variety of other analyses 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

TABLE 5. Accuracy Results: Input 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

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 an ancestor must be selected and/or replaced. Finally, on the theoretical level, it would be interesting to characterize for which classes of programs our widening is optimal in accuracy.

Note that, recently, another practical type analysis has been proposed by Gallagher and de Waal [9]. The analysis is based on monadic logic programs, and uses bottom-up abstract interpretation with a normalization procedure to obtain a finite domain. The normalization has some similarities with our widening, but it may lose much accuracy by merging types with the same principal functors. This makes it impossible to handle nested structures with the same functors, a situation which occurs very frequently in practice (see, for instance, the program in Figure 1 and the tokenizer of Prolog in Section 2). Their analysis is about 30 times a slower on the common benchmarks (i.e., Press and Qsort), but they use a slightly slower machine and Prolog as the implementation language.

Stimulating discussions with David McAllester are gratefully acknowledged. Detailed comments from the three reviewers helped in improving the presentation and in establishing the connection with tree grammars. We are particularly grateful to one of the reviewers for identifying several limitations in the formalization of our algorithm. This research was partly supported by the Office of Naval Research under Grant N00014-91-J-4052 ARPA Order 8225, by the National Science Foundation under Grant numbers CCR-9357704, and by a National Young Investigator Award.

REFERENCES

1. Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal of Logic Programming* 10(2):91–124 (Feb. 1991).
2. Bruynooghe, M. and Janssens, G., An Instance of Abstract Interpretation: Integrating type and Mode Inferencing, in: *Proc. 5th International Conference on Logic Programming*, Seattle, WA, August 1988, MIT Press, Cambridge, pp. 669–683.
3. Corsini, M., Musumbu, K., Rauzy, A., and Le Charlier, B., Efficient Bottom-up Abstract Interpretation of Prolog by Means of Constraint Solving over Symbolic Finite Domains, in: *Proc. 5th International Conference on Programming Language Implementation and Logic Programming*, Tallinn, Estonia, Aug. 1993.
4. Cortesi, A., Le Charlier, B., and Van Hentenryck, P., Combinations of Abstract Domains for Logic Programming, in: *21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*, Portland, OR, Jan. 1994.
5. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: ACM Press (ed.) *Conf. Record 4th ACM Symposium on Programming Languages (POPL'77)*, Los Angeles, CA, Jan. 1977, pp. 238-252.
6. Dincbas, M., Simonis, H., and Van Hentenryck, P., Solving Large Combinatorial Problems in Logic Programming, *Journal of Logic Programming* 8(1–2):75–93 (Jan./Mar. 1990).
7. Englebert, V., Le Charlier, B., Roland, D., and Van Hentenryck, P., Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation, *Software Practice and Experience* 23(4) (Apr. 1993).

8. Fruehwirth, T., Shapiro, E., Vardi, M., and Yardeni, E., Logic Programs as Types for Logic Programs, in: *IEEE 6th Annual Symposium on Logic in Computer Science*, 1991, pp. 300–309.

9. Callagher, J. and de Waal, D. A., Fast and Precise Regular Approximation of Logic Programs, in: *11th International Conference on Logic Programming*, Genoa, Italy, June 1994.

10. Gecseg, F. and Steinby, M., *Tree Automata*, Akademiai Kiado, Budapest, 1984.

11. Heintze, N., Practical Aspects of Set-Based Analysis, in: *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, Nov. 1992.

12. Heintze, N. and Jaffar, J., A Finite Presentation Theorem for Approximating Logic Programs, in: *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1990, pp. 197–209.

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

14. Kanamori, T. and Kawamura, T., Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation, Technical Report, ICOT, 1987.

15. Le Charlier, B., Musumbu, K., and Van Hentenryck, P., A Generic Abstract Interpretation Algorithm and Its Complexity Analysis (Extended Abstract), in: *8th International Conference on Logic Programming (ICLP-91)*, Paris, France, June 1991, MIT Press, Cambridge, MA, pp. 64–78.

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

17. Le Charlier, B. and Van Hentenryck, P., Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog, *ACM Transactions on Programming Languages and Systems* 16(1):35–101 (Jan. 1994).

18. Marriott, K. and Søndergaard, H., Notes for a Tutorial on Abstract Interpretation of Logic Programs, North American Conference on Logic Programming, Cleveland, OH, Oct. 1989.

19. Mishra, P., Towards a Theory of Types in Prolog, in: *International Symposium on Logic Programming*, 1984, pp. 289–298.

20. Monsuez, B., Polymorphic Types and Widening Operators, in: *International Workshop on Static Analysis (WSA-93)*, Padova, Italy, Sept. 1993.

21. Mulkers, A., Winsborough, W., and Bruynooghe, M., Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs, in: *7th International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990, MIT Press, Cambridge, MA, pp. 747–764.

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

23. Sterling, L. and Shapiro, E., *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, MA, 1986.

24. Taylor, A., LIPS on MIPS: Results from a Prolog Compiler for a RISC, in: *7th International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990, MIT Press, Cambridge, MA, pp. 174–188.

25. Thomas, W., *Automata on Infinite Objects*, volume Handbook of Theoretical Computer

Science: Formal Models and Semantics, MIT Press, Cambridge, MA, 1990, pp. 133–161.

26. Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, Logic Programming Series, MIT Press, Cambridge, MA, 1989.
27. Van Hentenryck, P., Cortesi, A., and Le Charlier, B., Type Analysis of Prolog Using Type Graphs, in: *Proc. ACM–SIGPLAN Conference on Programming Language Design and Implementation (PLDI-94)*, Orlando, FL, June 1994.
28. Verschaetse, K. and De Schreye, D., 8th Deriving Termination Proofs for Logic Programs Using Abstract Procedures, in: *8th International Conference on Logic Programming (ICLP-91)*, Paris, France, June 1991.
29. Winsborough, W., Multiple Specialization Using Minimal-Function Graph Semantics, *Journal of Logic Programming* 13(4) (July 1992).
30. Yardeni, E. and Shapiro, E., A type System for Logic Programs, *Journal of Logic Programming* 10(2):125–153 (1991).