Analyzing Declarative Semantics of Logic Programs

Dmitri Boulanger
GMD-FIRST, Rudower Chaussee 5, 12489 Berlin, Germany
email: dmitri@first.gmd.de

Abstract Deductive databases and constraint systems have a transparent declarative semantics, which enables to read and to understand such programs ignoring their execution algorithms. An interactive toolkit, which is aimed at such "understanding", is presented. It enables generation of precise descriptions of the goal-relevant correct answers. Its ultimate goal is a problem-specific optimization, which can be derived from such descriptions. A flexible interface is used to specify required details of descriptions. The toolkit is reasonably efficient even if a fine-grained description is required.

Key Words: logic programming, declarative semantics, static analysis

1 Introduction

An approach to analyze logic-based software systems has been suggested in [1, 2, 3, 4]. It exploits the clean transparent declarative semantics of logic programming languages to provide a basis for verification and optimization of complex applications, such as constraint systems and deductive database queries. Recently an experimental toolkit to facilitate comprehensive analysis of the possibly non-ground success set of a program has been developed. The aim of the toolkit is a *problem-oriented* analysis, which can produce really useful hints to optimize applications. Let us to discuss some important details.

The set of correct answers is the most well-know *declarative* semantics of definite logic programs (see Lloyd [5]). It is an important theoretical notion[1], which provides a basic semantics of the declarative programming. However, it is rarely needed in practice. A goal-relevant declarative semantics is more appropriate because it captures *procedural* details of the usage of the program. This is achieved taking into consideration a *subset* of the set of correct answers. Such a subset is defined below.

First of all, assume the program has the predicate q as an "entry point", i.e. the program is considered as a module, which exports the predicate q. Formally, only entry point queries of the form $\leftarrow q(\overline{t})$ where $\overline{t}$ a term tuple, are supposed to be answered. In the sequel we always assume that "a program" means a logic program with the declared entry point predicate.

Next, we use a slight reformulation of the standard definition of the set of correct answers: an atom $p(\overline{t})$ is a *(correct) answer (atom)* iff $\theta = \{\overline{X}/\overline{t}\}$ is a *correct answer* substitution for the most general query $\leftarrow p(\overline{X})$ ($\overline{X}$ is a tuple of pairwise different variables). The set of such atoms is called the (possibly non-ground) *success set*. Then the **program $\mathcal{C}$-base** is a set of all answer atoms, which are *relevant* to the entry point q in the following sense:

- all entry point answers are relevant atoms
- given a relevant atom A and an instance of a program clause of the form $A \leftarrow A_1, \ldots, A_n$, $n \geq 0$, the atoms $A_1, \ldots, A_n$ are also relevant *iff* they are answer atoms

[1] It corresponds to the set of universally quantified atomic logical consequences of the program.

So, the $\mathcal{C}$-base is a set of answer atoms, which is recursively obtained from the set of the correct answers for the entry point. The above extends the definition suggested in [7][2].

If one would change the program in such a way that its $\mathcal{C}$-base remains unchanged, the modified program would have the same set of refutations of the entry point queries. Informally, the answers atoms, which are not in the $\mathcal{C}$-base, don't carry any important characterization of the program. To see this, consider the set of all refutations of all entry point answer atoms. In general each such a refutation consists of nested sub-refutations. A sub-refutation selects an atom in the current accumulated goal. Following Lloyd [5], the most essential features of set of all such sub-refutations can be characterized as follows: 1) an atom A has a refutation with the *computed* answer substitution θ iff the atom $A.\theta$ is an answer; 2) given an instance of a program clause of the form $A \leftarrow A_1, \ldots, A_n$, $n \geq 0$, the atom A is an answer iff the atoms A_k, $k = 1, \ldots, n$ are answers. Therefore, the $\mathcal{C}$-base is exactly the set of the atoms A, such that there exists an entry point answer atom, which has a refutation which at some point selects the atom A. On the other hand, it is not difficult to see that the $\mathcal{C}$-base is the set of computed answer atoms, which are generated by the sub-refutations of the refutations of the entry point queries.

Thus, the $\mathcal{C}$-base, being a declarative semantics, captures important procedural details of the usage of the program. In particular, it shows theoretically the best evaluation algorithms. As it follows from the above, a top-down evaluation algorithm might be the best one only if it selects atoms having refutations with answer atoms occurring in the $\mathcal{C}$-base. Similar, existing bottom-up evaluation algorithms generate a superset of the $\mathcal{C}$-base, i.e. the $\mathcal{C}$-base can be seen as a theoretical target to improve them.

The main motivation behind the toolkit is an optimization, which can be derived from *detailed* descriptions of the $\mathcal{C}$-base properties. Indeed, if such a description is available, it can be inserted into the program as additional constraints to reduce the search space. As an illustration, assume a source constraint system defines a complex object, whose components have to satisfy many constraints. Such a constraint may be: "The component x must be chosen from the set S". If a number of such constraints is large, it is rather difficult to see how they are used in the evaluation algorithm of the program, e.g. the top-down evaluation algorithm might pick up the above constraint too late and, therefore, it might produce too large search space. On the other hand, a problem-oriented analysis, which is specially designed to investigate the $\mathcal{C}$-base of the application, might reveal the importance of the above constraint, which suggests to move it more close to the goal to reduce the search space. Notice that such an analysis is *independent* from the evaluation algorithm. This helps to see a better one.

As a summary of the above, the toolkit, which is described below, is an interactive environment to investigate the $\mathcal{C}$-base of a program. The main feature is the generation of detailed problem-oriented descriptions. This is different from the traditional usage of static analysis in compilers, which need algorithms for a relatively simple but fast type inference.

The rest of the paper is organized as follows. In the next section a self-contained description of the basic notions of the toolkit is presented. This is a modification of the previous results, which have been made to explain essential elements of the recent implementation.

Next, a comprehensive illustration of the interactive problem-oriented usage of the toolkit is presented. It includes an investigation of different features of a program, combination of them to get more refined results and using of the obtained results to derive new constraints.

[2] The (Herbrand) program base, as it is defined in [7], is the ground subset of the $\mathcal{C}$-base.

2 Basics of the Toolkit

A program language and its interpretation play a fundamental role in deriving properties of the $\mathcal{C}$-base. Following Lloyd [5], terms, atoms and clauses in the program language are constructed from function symbols, predicates and variables (variables appear in upper case). An interpretation of function symbols is called a *pre-interpretation*. It has a non-empty *domain* (set) D, which is used to construct the set of *mappings*

$$\{\varphi_f : \mathsf{D} \times \cdots \times \mathsf{D} \mapsto \mathsf{D} \mid f \text{ is a function symbol}\}$$

A mapping φ_f above, which is associated with the function symbol f of the arity $n \geq 0$, is defined for any domain tuple $\overline{d}$ of the form $\langle d_1, \ldots, d_n\rangle$, i.e. φ_f is a *total function* $\overline{\mathsf{D}} \mapsto \mathsf{D}$.

Consider a pre-interpretation of the program language. The corresponding *domain language* is obtained extending the program language with the domain elements as new constants. In other words, in the domain language domain elements are allowed to occur in any place where a constant can appear. Let μ be a mapping such that $\mu(d) =_{\mathsf{def}} d$ for any domain element d, i.e. an interpretation of a domain element is the domain element itself. Next, for any function symbol f having the arity n, $n \geq 0$, and for any *ground* terms t_i, $i = 1, \ldots, n$, in the domain language, we define

$$\mu(f(t_1, \ldots, t_n)) \;=_{\mathsf{def}}\; \varphi_f(\mu(t_1), \ldots, \mu(t_n))$$

As a result, each ground term t in the domain language has the domain element $d = \mu(t)$ as an image (interpretation). Finally, the **representation** ρ extends μ (which is only defined for ground terms) to *non-ground* term tuples as follows:

$$\rho(\langle t_1, \ldots, t_n\rangle) =_{\mathsf{def}} \{\ \langle\, \mu(t_1.\vartheta), \ldots, \mu(t_n.\vartheta)\,\rangle\ \mid \vartheta \text{ is a variable assignment }\ \}$$

where ϑ assigns a domain element to *each* variable. A ground term $t.\vartheta$ is obtained replacing each variable X in the term t with the domain element assigned to X. As a result, *any* (possibly non-ground) term tuple $\overline{t}$ has the image $\rho(\overline{t})$, which is a *non-empty set* of domain tuples. Such images, or **representation sets**, are also defined for clauses and atoms in the straightforward way, e.g. $\rho(\, p(\overline{t}) \leftarrow p_1(\overline{t}_1), \ldots, p_n(\overline{t}_n)\,) \;=_{\mathsf{def}}$

$$\{\, p(\mu(\overline{t}.\vartheta)) \leftarrow p_1(\mu(\overline{t}_1.\vartheta)), \ldots, p_n(\mu(\overline{t}_n.\vartheta)) \mid \vartheta \text{ is a variable assignment }\},\ \ n \geq 0$$

The mapping ρ is a core of the mechanism below, which enables to generate abstractions of the $\mathcal{C}$-base. As a theoretical observation, one could show that representation sets in the Herbrand pre-interpretation are simply sets of ground instances, e.g. a term is represented as the set of its ground instances. On the other hand, choosing a pre-interpretation with a *finite* domain, one represents the set all terms in the program language as a finite set of finite sets of domain elements.

The next crucial notion is the **program base**, which is a subset of the *least model* of the program wrt the pre-interpretation. The least model itself is the minimal set of *true simple domain atoms* of the form $p(\overline{d})$, which enables to evaluate all program clauses as true[3]. The program base is its subset such that

- any entry point domain atom of the least model occurs in the program base

[3] Any definite program has the least model wrt any pre-interpretation (see Lloyd [5]).

- given a domain atom $p(\overline{d})$ in the program base and a program clause, which has the representation set containing the domain clause $p(\overline{d}) \leftarrow p_1(\overline{d}_1), ..., p_n(\overline{d}_n)$, $n \geq 0$, the atoms $p_1(\overline{d}_1)$, ..., $p_n(\overline{d}_n)$ are in the program base *iff* they are in the least model

Notice that the above is very similar with the definition of the $\mathcal{C}$-base, which has been discussed in detail in Section 1. It is possible to show that it is a reformulation of the $\mathcal{C}$-base definition in the context of an arbitrary pre-interpretation[4]. So, the main result is:

Proposition 2.1 *The atom $p(\overline{t})$ occurs in the program $\mathcal{C}$-base* only if *its representation set*

$$\rho(p(\overline{t})) = \{p(\overline{d}) \mid \overline{d} \in \rho(\overline{t})\}$$

is a subset of the program base. □

The above reformulates as follows: the set

$$\cup_A \{\rho(A) \mid \text{the atom } A \text{ belongs to the } \mathcal{C}\text{-base}\}$$

which is the ρ-image of the $\mathcal{C}$-base, is a subset of the program base, i.e. any program base can be seen as a safe approximation of the ρ-image of the $\mathcal{C}$-base. Obviously such an approximation is finite if the pre-interpretation is finite. One may expect that the ρ-image of the $\mathcal{C}$-base captures the $\mathcal{C}$-base properties very precisely. On the other hand, it is important to understand, that in general it is *not* decidable even if the pre-interpretation has a finite domain. For instance, as it is shown in [1], there exists a pre-interpretation having only two domain elements, such that the corresponding ρ-image of the $\mathcal{C}$-base gives precisely the so-called groundness dependencies, which are known to be undecidable.

Proposition 2.1 enables to derive $\mathcal{C}$-base properties. Firstly, one formulates and proves the so called **granularity lemma** of the form:

$$\rho(t) = \begin{cases} S_1 & \text{iff the term } t \quad \text{satisfies } \langle\text{type description}\rangle \\ \ldots & \text{iff the term } t \quad \text{satisfies } \langle\text{type description}\rangle \\ S_n & \text{iff the term } t \quad \text{satisfies } \langle\text{type description}\rangle \end{cases}$$

The above sets $S_k \subseteq \mathsf{D}$, $k = 1, \ldots, n$, $n \geq 1$ *must* form the *complete* set of representation sets of terms in the program language, i.e. there exists no other subset of the domain D, which is a representation set of a term. The type descriptions must be mutually exclusive conditions, which split the set of terms of the language into disjoint subsets, called **types**. A type has a unique representation set and each representation set defines a unique type. As a consequence, any term t in the program language has a unique type, which is determined computing the representation set $S = \rho(t)$.

Secondly, the $\mathcal{C}$-base properties, which are expressible by means of the types, are derived. It is enough to prove that a term occurs in the $\mathcal{C}$-base only if it has a type, which corresponds to the property. By Proposition 2.1, to do that, one verifies that representation sets of the types, which are *not* implied by the property, are *not* subsets of the program base.

The above is the backbone of the toolkit, which is aimed at generating **views**. A view is a description of the $\mathcal{C}$-base properties, which uses particular types as atomic basic notions. In principle, views, are obtained using any types, which are constructed using finite

[4]As a theoretical observation, one could show that the $\mathcal{C}$-base is the program base wrt an appropriate pre-interpretation (see Maher [6] for motivations). Also, the program base, as it is defined in [7], is obtained applying our definition in the context of the Herbrand pre-interpretation.

pre-interpretations (Proposition 2.1 holds for an arbitrary pre-interpretation). Following the above theory, the toolkit supports a view generation provided that it is organized in three steps: 1) choosing a **finite** pre-interpretation and proving the granularity lemma; 2) generating the program base 3) analyzing the program base to verify or to derive $\mathcal{C}$-base properties, which are expressible in terms of the types of the granularity lemma.

A pre-interpretation must be chosen to have a granularity lemma with the types to express the required properties. The toolkit supports an interactive development of such a pre-interpretation. First of all, a formal correctness of the pre-interpretation is checked, e.g. it is checked that all function symbols in the program language are assigned total mappings over the declared domain. Next, some user-declared semantic statements are tested and the complete set of representation sets is generated. This only yields a formal enumeration of types. It remains unclear which terms are described by which types. Therefore, a remaining part of the development of a pre-interpretation can be seen as a *semantic* debugging of the type descriptions. This means that for any representation set S one has to suggest and then to prove a statement, which describes all terms t such that $\rho(t) = S$. A form of this statement can be chosen taking into account a particular application (see examples in Section 3).

The base generator consists of a compiler and of a simple interpreter to run the compiled code. As it is suggested in [4], the compiler performs the so-called flattening, which produces a Datalog program (it is obtained as the ρ-image of the source program). As a second phase of the compilation, the flat program rewrites into a XSB-Prolog code, which significantly exploits tabulation. The first part of the code by backtracking constructs all refutations of the most general entry point query $q(\overline{X})$. As a side-effect, an approximation (superset) of the program base is accumulated in the tables. The second part of the code cleans the approximation to obtain exactly the program base.

Deriving properties is illustrated in the next section. It contains an analysis of a non-trivial program, which hopefully makes all the above points more clear.

3 Views of the Program PG

This section contains a detailed example of the problem-oriented analysis. A non-trivial program, which is shown in Figure 1, is used as a running example. What follows doesn't require understanding of this program. It is enough only to have a rather superficial hypothesis about the program properties. The toolkit enables to check and to refine them.

An immediate examination of the program `PG` shows that it likely processes arithmetic expressions, integers and lists of integers. This is a *hypothesis*, which might be true or false but it gives rise to a granularity of the $\mathcal{C}$-base view. Therefore, a pre-interpretation with the granularity lemma, which claims at least such *types* as arithmetic expressions, integers and lists of integers, is needed.

The language of `PG` has the constants 0/0, 1/0, []/0, functions $+/2$, $-/2$, $*/2$, ./2 and program-defined predicates

$$\{\texttt{pdsbm/2}, \texttt{pds/2}, \texttt{pds/4}, \texttt{iota/2}, \texttt{iota/3}, \texttt{check/5}, \texttt{dif/5}, \texttt{mergedelete/3}, \texttt{member/2}, \texttt{rev/3}\}$$

with the predicate `pdsbm/2` as an entry point. To construct a program base, the symbols above should be given interpretations as total functions over a finite domain. Also, since `PG` has the external predicates `is/2` and $>$ `/2`, which have no program definitions, a program base generation needs their models. Having a suitable granularity, such models can be easily suggested. So, the key step is choosing a pre-interpretation and proving its granularity lemma.

```
pdsbm(N,X)  :-  pds(N,[0,1|X]).
pds(N,[0|Ans])  :-  Mod is N*(N+1)+1, iota(Mod,[0|List]), pds(List,[0],Ans,Mod).
pds([],_,[],_).
pds(Unused,List,[Choice|Rest],Mod)  :-
        member(Choice,Unused),
        check(List,Choice,Unused,U3,Mod),
        pds(U3,[Choice|List],Rest,Mod).

iota(Mod,List)  :-  iota(0,Mod,List).
iota(K,K,[]).
iota(K,N,[K|List])  :-  K1 is K+1, iota(K1,N,List).

check([],_,L,L,_).
check([X|Y],Choice,Old,L3,Modulus)  :-
        dif([X|Y],Choice,Modulus,Ds,Dds),
        mergedelete(Ds,Old,L2),
        rev(Dds,[],Rds),
        mergedelete(Rds,L2,L3).

dif([],_,_,[],[]).
dif([S|Ss],Val,Mod,[D|Ds],[D2|D2s])  :-  D is Val-S, D2 is Mod-D, dif(Ss,Val,Mod,Ds,D2s).

mergedelete([],L,L).
mergedelete([D|Ds],[D|R],L)  :-  mergedelete(Ds,R,L).
mergedelete([D|Ds],[X|R],[X|L])  :-  D > X, mergedelete([D|Ds],R,L).

member(X,[X|_]).                              rev([],L,L).
member(X,[_|L])  :-  member(X,L).             rev([X|Xs],Y,L)  :-  rev(Xs,[X|Y],L).
```

Figure 1 *The program* `PG` *with the entry point* `pdsbm/2`*.*

3.1 J_1-View: Arithmetic Expressions, Integers and Lists of Integers

To verify the hypothesis, the pre-interpretation J_1 with the domain

$$\mathsf{D}_1 = \{\texttt{int}, \texttt{expr}, \texttt{list}, \texttt{smth}\}$$

is constructed. The constants 0/0 and 1/0 are mapped to `int` but the constant []/0 is mapped to `list` since the hypothesis says that lists and integers must have different types. Similar, the arithmetic functions $+/2$, $-/2$, and $*/2$ are assigned the same mapping φ_{arithm_op} which is different from the mapping $\varphi_{.}$ of the list constructor ./2:

$$\varphi_{arithm\_op}(d_1, d_2) = \begin{cases} \texttt{expr} & \text{iff } d_1, d_2 \in \{\texttt{int}, \texttt{expr}\} \\ \texttt{smth} & \text{otherwise} \end{cases}$$

$$\varphi_{.}(d_1, d_2) = \begin{cases} \texttt{list} & \text{iff } d_1 = \texttt{int},\ d_2 = \texttt{list} \\ \texttt{smth} & \text{otherwise} \end{cases}$$

An implementation of pre-interpretations similar to the above is guided by the toolkit. First of all, the program language is constructed and displayed. Then XSB-modules, which have as their interface the declarations of the predicates defining the domain and the mappings of the function symbol, are created. They should be completed in an interactive way to obtain the XSB-code of the pre-interpretation. Next, the correctness of the implementation of the pre-interpretation is checked. This operation is essentially a fix point computation, which constructs the mappings and checks them. Finally, the representation sets are generated. This is not a very trivial thing, however, an algorithm exists. A detailed technique for generating representation sets is out of the scope of the paper.

Toolkit provides various debugging commands to display some important features of the pre-interpretation. This is rather useful when developing complex pre-interpretations having large domains (50 - 90 domain elements). For instance, each mapping of the pre-interpretation can be investigated. Also, for each representation set a term having it as a representation can be seen.

Having the representation sets, the granularity lemma can be developed. Using the above mentioned commands, it is not that difficult to see that the above pre-interpretation has the granularity lemma, which says that: $\rho_1(t) =$

$$\begin{cases} \{\mathtt{int}\} & \text{iff the term } t \text{ is an integer} \\ \{\mathtt{expr}\} & \text{iff the term } t \text{ is an expression} \\ \{\mathtt{list}\} & \text{iff the term } t \text{ is a list of integers} \\ \{\mathtt{list}, \mathtt{smth}\} & \text{iff the term } t \text{ is a non-ground (possibly open) list} \\ & \quad \text{only consisting of integers and/or variables} \\ \{\mathtt{expr}, \mathtt{smth}\} & \text{iff the term } t \text{ is a non-ground expression} \\ \{\mathtt{int}, \mathtt{expr}, \mathtt{list}, \mathtt{smth}\} & \text{iff the term } t \text{ is a variable} \\ \{\mathtt{smth}\} & \text{iff the term } t \text{ is distinct from all above} \end{cases}$$

The type "an expression", which has the representation set $\{\mathtt{expr}\}$, defines the set of ground terms, which are only build from arithmetic functions and integers but not integers themselves, i.e. it has a rather simple type description (semantics). The type description of the type "a non-ground expression" is a bit less trivial. It is the set of non-ground non-variable terms, which might be instantiated to be expressions, e.g. $X+Y$. Similar, any term of the type, which has the representation set $\{\ \mathtt{list},\ \mathtt{smth}\}$, has an instance of the type "list of integers", e.g. the term $[X|Y]$ has a list of integers as its instance. The type with the representation $\{\mathtt{smth}\}$ covers ground and non-ground terms, e.g. terms $+([],[])$ and $+([],X)$ have the representation $\{\mathtt{smth}\}$. Their type is a type of "outsiders", because the hypothesis supposes that such terms are absent in the $\mathcal{C}$-base of the program.

Thus, there are three "ground" types, which only consist of ground terms, three "non-ground" types, which only consist of non-ground terms, and the "mixed" type with the representation set $\{\mathtt{smth}\}$. A term is non-ground only if it has the type representation set with the domain element `smth`.

It is very important to understand that the above granularity has to be proven. Such proofs are not always immediate. The main problem is the correctness of type descriptions. As an illustration, as it was explained above, the term $[X|Y]$ has the type "non-ground list" with the representation set $\{\mathtt{list}, \mathtt{smth}\}$, but the term $[X|X]$ has the representation set $\{\mathtt{smth}\}$. So, the type description of the type "non-ground list" in the granularity lemma above, should be modified to exclude open lists, which cannot be instantiated to be a list of integers. Namely, lists of the form $[t_1, \ldots, t_n|X]$, $n \geq 1$, where t_i is the variable X for at least one $i \in \{1, \ldots, n\}$, don't belong to the type "non-ground list". This correction is not obvious, but, as we shall see, its importance is rather limited.

A technique for proving granularities is out of the scope of the paper. However, some simple hints are useful. For instance, the domain of a pre-interpretation is always a representation set, which very often (but not always!) represents variables. Also, ground types are always have singleton representation sets. Moreover, it is possible automatically to find out all ground types (the toolkit supports this operation). A general method to develop type descriptions uses the simple relation $\rho(\bar{t}.\theta) \subseteq \rho(\bar{t})$, which holds for any representation ρ, for any term tuple $\bar{t}$ and any substitution θ (cf. the above discussion).

As soon as a granularity of the pre-interpretation is available, the models of the external predicates can be suggested. In particular, the models of the built-ins in `PG` are defined as:

$$\texttt{int is } d,\ d \in \{\texttt{int}, \texttt{expr}\} \quad | \quad \texttt{int} > \texttt{int}$$

Notice the models above significantly exploit the granularity lemma, i.e. it is taken into account that integers appear as a result of an evaluation of expressions and integers themselves. Typically a declarative analysis takes exceptions in built-ins as failures, since it is only aimed at analyzing the meaning the program rather than a particular algorithm to execute the program. So, above the exceptions are considered as failures. Formally, given the representation ρ, the model of the external predicate p is any superset of the set

$$\cup_{\bar{t}} \{\rho(\ p(\bar{t})\ ) \mid \text{the atom } p(\bar{t}) \text{ is true }\}$$

i.e. one must know the semantics of the external predicate p. The set above is the ρ-image of the success set of the predicate p. It makes the analysis of a program more precise, but any its superset can be used (cf. Section 2).

In fact, the pre-interpretation J_1 has been constructed to check that *no* term having a type with **smth** in its representation set occurs in the $\mathcal{C}$-base of `PG` . This is exactly what the hypothesis is saying. Since Figure 2 shows that the J_1-base of `PG` has no occurrences of **smth**, only integers, expressions and lists of integers occur in the $\mathcal{C}$-base. Expressions only occur in a few program points. This is not a trivial result. For instance, it exploits the fact that the goal $\leftarrow \texttt{iota}(t, X)$ has an infinite failure if the term t is not an integer. This effect is automatically discovered and exploited in the base generation.

```
pdsbm( expr list  )     pds( expr list )      pds(  list  list  list  int  )
pdsbm( int  list  )     pds( int  list )

iota( int list )       iota( int int list  )

check( list int list list int  )

dif( list  int int  list  list  )      mergedelete( list  list  list  )

member( int list )       rev( list  list list  )
```

Figure 2 *The base of the program* `PG` *wrt the pre-interpretation* J_1

The above is an example of a general method to prove program properties, i.e. a pre-interpretation with the types to express a property is constructed, and then one checks that representation sets of other types are not covered by the program base. This is an immediate consequence of Proposition 2.1 and of the granularity lemma. As the development of the above view shows, the complete granularity lemma is seldom needed, e.g. only part of it, which describes the ground types, has been used above. A development of a description of ground types can be automated since it is more simple than that for non-ground types. Such a situation is typical for declarative applications such as deductive database queries or constraint programs. On the other hand, if the domain element smth would occur, it would be a problem to explain what does it mean (see Section 3.2).

3.2 J_2-View: Non-negative Expressions and Lists of such Expressions

Now we would like to refine the hypothesis. Since it is already known that PG mostly processes integers, it sounds reasonable that it processes non-negative integers and the corresponding lists. It is possible to simplify the problem only focusing on whether a term evaluates to negative, zero or positive number rather than on its syntactic structure. So, the set

$$\mathsf{D}_2 = \{\mathtt{0}, +\mathtt{int}, \mathtt{list}, \mathtt{smth}\}$$

is chosen as a domain of the pre-interpretation J_2. The constants $0/0$, $1/0$ and $[]/0$ are mapped to the domain elements 0, +int and list, respectively. The arithmetic operations $+/2$, $-/2$, $*/2$ and the list constructor $./2$ are associated respectively with the mappings:

$$\varphi_+(d_1,d_2) = \begin{cases} \mathtt{0} & \text{iff } d_1 = d_2 = \mathtt{0} \\ +\mathtt{int} & \text{iff } d_1, d_2 \in \{\mathtt{0}, +\mathtt{int}\} \text{ but } \langle d_1, d_2\rangle \neq \langle \mathtt{0}, \mathtt{0}\rangle \\ \mathtt{smth} & \text{otherwise} \end{cases}$$

$$\varphi_-(d_1,d_2) = \begin{cases} \mathtt{0} & \text{iff } d_1 = d_2 = \mathtt{0} \\ +\mathtt{int} & \text{iff } d_1 = +\mathtt{int}, d_2 = \mathtt{0} \\ \mathtt{smth} & \text{otherwise} \end{cases}$$

$$\varphi_*(d_1,d_2) = \begin{cases} \mathtt{0} & \text{iff } d_1 = \mathtt{0} \text{ or } d_2 = \mathtt{0} \\ +\mathtt{int} & \text{iff } d_1 = d_2 = +\mathtt{int} \\ \mathtt{smth} & \text{otherwise} \end{cases}$$

$$\varphi_.(d_1,d_2) = \begin{cases} \mathtt{list} & \text{iff } d_1 \in \{\mathtt{0}, +\mathtt{int}\},\ d_2 = \mathtt{list} \\ \mathtt{smth} & \text{otherwise} \end{cases}$$

Now all function symbols in the language of the program PG are assigned pairwise different mappings to have a more refined model of arithmetic (cf. Section 3.1). Namely, the mapping φ_{arithm_op} of the pre-interpretation J_1 is split out in three different mappings to capture simple arithmetic expressions, which are evaluated into non-negative integers. However, integers themselves and arithmetic expressions are not distinguishable. The mapping $\varphi_.$ models lists of such expressions. As a result, the granularity lemma says that $\rho_2(t) =$

$$\begin{cases} \{\mathtt{0}\} & \text{iff } t \text{ is a simple zero-expression (is evaluated to 0)} \\ \{+\mathtt{int}\} & \text{iff } t \text{ is a simple positive expression} \\ \{\mathtt{list}\} & \text{iff } t \text{ is a list of non-negative simple expressions} \\ \{\mathtt{0}, \mathtt{smth}\} & \text{iff } t \text{ is a non-ground simple zero-expression} \\ \{\mathtt{0}, +\mathtt{int}, \mathtt{smth}\} & \text{iff } t \text{ is a non-ground simple non-negative expression} \\ \{\mathtt{list}, \mathtt{smth}\} & \text{iff } t \text{ is a non-ground (possibly open) list of ground} \\ & \quad \text{and/or non-ground simple expressions and/or variables} \\ \{\mathtt{0}, +\mathtt{int}, \mathtt{list}, \mathtt{smth}\} & \text{iff } t \text{ is a variable} \\ \{\mathtt{smth}\} & \text{iff } t \text{ is distinct from all above} \end{cases}$$

The above types with representation sets {0} and {+int} don't capture *all* arithmetic expressions, which are evaluated to non-negative integers. Indeed, they only cover a part of ground arithmetic expressions, which are simple enough to check that they are evaluated to non-negative integers. For instance, arithmetic expressions such as $(1 + 1) - 1$ are evaluated

```
    pdsbm( 0     list )        pds( 0     list )      pds( list  list list  +int  )
    pdsbm( +int  list )        pds( +int  list )
    pdsbm( smth  list )        pds( smth  list )
 %  pdsbm( list  list )     %  pds( list  list )

iota( +int list )        iota( 0     +int list  )
                         iota( +int  +int list  )

check( list 0     list  list  +int  )
check( list +int  list  list  +int  )

dif( list  0     +int list  list  )
                                         mergedelete( list  list  list  )
dif( list  +int  +int list  list  )

member( 0     list )        rev( list  list  list )
member( +int  list )
```

Figure 3 *The base of the program* PG *wrt the pre-interpretation* J_2

to non-negative integers but they have **smth** as a representation set because they are already too complex to decide the sign.

The above suggest the following J_2-models of the external predicates in the program PG :

0	is	0	0	is	smth	+int	> 0		smth	> +int
+int	is	+int	+int	is	smth	+int	> +int		+int	> smth
			smth	is	smth				smth	> 0
									0	> smth
									smth	> smth

Since now integers and expressions are not distinguishable, the evaluation of the domain element **smth** produces any term, which is not a list. In other words, the granularity does not fit very well the semantics of the above built-ins (cf. Section 3.1).

The program base of PG is shown in Figure 3. It indicates that we failed to prove that negative expressions are absent in the $\mathcal{C}$-base of PG . Indeed, the entry point query

$$\leftarrow \mathtt{pdsbm}(0 - ((1 + 1) * (1 + 1)), [3, 9])$$

succeeds.

The pre-interpretation J_2 has been constructed hoping that no term with **smth** in its representation set occur in the $\mathcal{C}$-base. The above view, if taken as it is, does not indicate this. Moreover, due to the occurrences of **smth** information about the first arguments of the predicates pdsbm/2 and pds/2 has been lost. Using the granularity lemma we can only report that pdsbm/2 and pds/2 might have a variable as a first argument, which, as Proposition 2.1 says, simply means any term.

Problems with the predicates pdsbm/2 and pds/2 don't disturb a nice picture for other predicates: they only use non-negative expressions and lists of non-negative expressions. This is a bit surprising because the body of the second clause of the predicate dif/5 has a rather complex arithmetic but the models of arithmetic operations, which has been used to construct the program base, are very primitive (it was expected that PG doesn't need complex arithmetic). However, it was derived that all lists occurring in the $\mathcal{C}$-base are constructed from the elements of the initial list, which is generated by the predicate `iota/2` and which only consists of non-negative expressions.

3.3 Putting J_1-View and J_2-View to-gather

Let us illustrate another important motivation behind the technology for the program-specific analysis. First of all, notice that Figure 3 has the domain atoms with the symbol % as a label. Those atoms are in contradiction with Figure 2, because it to-gather with the granularity lemma in Section 3.1, claim that that terms, which occur in the first argument of the predicates `pdsbm/2` and `pds/2`, must be expressions or integers. Therefore, the tuples with `list` in the first argument are results of the over-approximation of the ρ_2-image of the $\mathcal{C}$-base, and should be dropped because only the ρ-image of the $\mathcal{C}$-base is needed (cf. Section 2). However, the tuples with `smth` cannot be removed because in the context of J_2-view the representation set $\{$`smth`$\}$ corresponds to the type description "a negative integer or an arithmetic expression, which possibly evaluates to a negative integer".

As a consequence of the above, all predicates in PG except `pdsbm/2` and `pds/2` only use non-negative integers and lists of such integers (`pdsbm/2` and `pds/2` might need negative integers or expressions, which possibly evaluate in negative numbers). This immediately suggests the following optimization of the second clause of the predicate `dif/5`:

$$\begin{array}{lll} \mathtt{dif([S|Ss],Val,Mod,[D|Ds],[D2|D2s])} & \mathtt{:-} & \mathtt{D\ is\ Val\text{-}S},\ \boxed{\mathtt{D} \geq \mathtt{0}}, \\ & & \mathtt{D2\ is\ Mod\text{-}D},\ \boxed{\mathtt{D2} \geq \mathtt{0}}, \\ & & \mathtt{dif(Ss,Val,Mod,Ds,D2s)}. \end{array}$$

The insertion of the above constraints, for instance, enables to run the query $\leftarrow$ `pdsbm(7,X)` nearly twice faster. Such optimizations are rather typical since exactly the program base is generated, which, in turn, enables to insert precise constraints in important program points. The second clause of the predicate `dif/5` is an important point since it influences the construction of the search space.

Notice that the above properties of the $\mathcal{C}$-base of PG cannot be derived for the J_1-view and J_2-view, if they are considered separately. The effect is due to their combination. In fact, such a technology enables us to decompose a fine-grain view generation into a number of relatively simple views. Recent experiments with some applications have shown that their analysis needs fine-grain views, which, in turn, require pre-interpretations with large domains (about 40 domain elements). But it was not possible to compute the corresponding program bases due to explosions in the generation of the base approximation. Therefore, a technique to decompose a fine-grain view generation into a number of much more simple views is very important from the pragmatic point of view.

A support for the above decomposition method needs a uniform type description language, which allows to formulate type descriptions of different granularity lemmas using the same formalism. Notice that in the above examples this was rather simple since the types used are very transparent. When analyzing applications, which need application-oriented types, such a transparency might be lost. A interactive interface with a reasonable automation to combine several views, is a current direction of research.

4 Conclusion

An experimental toolkit for a fine-grained declarative goal-oriented analysis of logic programs has been presented. Fine-grained views generation is considered as a key element of the optimization of complex applications, which are implemented using declarative programming systems such as deductive databases and finite domain constraint solvers.

Experiments have shown that fine-grained views generation is a rather difficult task because of possible explosions during the base approximation generation. This problem has significantly influenced the architecture of the toolkit. Namely, the toolkit has been implemented to support a fine-grained view generation as an iterative process, which starts with relatively simple views. Each iteration should exploit the previous results to avoid explosions.

The toolkit is based on theoretical and practical results in [1, 2, 3, 4], which have been reconsidered during the implementation. The program base generator have been recently re-implemented to avoid magic set transformations replacing them with a special code with tabulation. The new interfaces to develop complex pre-interpretations in an interactive way have been implemented. Also a persistent environment to store, retrieve and process rather complex objects, which are needed in the incremental analysis have been designed using the DBMS Oracle coupled with the XSB-Prolog. Experiments have shown that it is efficient enough to fit the interactive fine-grain view generation (e.g. the above analysis of `PG` needs only a few minutes of working time).

The main target of the toolkit is experiments aimed at a fine-grained view generation of applications of the finite domain constraint solvers. These experiments were needed to see the complexity of problem-oriented analysis and to elaborate a methodology for the usage of the results to improve efficiency of applications. The results are promising since they show a draft of such a technology for an application-oriented optimization.

References

[1] Boulanger,D. Analysis of Logic-Based Systems, *Proc. International Workshop on Logic in Databases LID'96, LNCS* **1154** *(1996), 281-301.*

[2] Boulanger,D. Complete Analysis for Definite Logic Programs, *Proc. 11th Workshop Logische Programmierung, Technishe Universität Wien, GMD-Studien Nr.270, 1995, 101-110.*

[3] Boulanger,D., Bruynooghe,M. A Systematic Construction of Abstract Domains, *Proc. 1st International Static Analysis Symposium, LNCS* **864** *(1994), S61-77.*

[4] Gallagher,J., Boulanger,D., Sağlam,H. Practical Model-Based Static Analysis for Definite Logic Programs, *Proc. 1995 ISLP, 1995, 351-365.*

[5] Lloyd,J. Foundations of Logic Programming, *Springer, 1987.*

[6] Maher, M. Equivalence of Logic Programs, *Foundations of Deductive Databases and Logic Programming.* Morgan-Kaufmann, 1988, 627-658

[7] Mascellani,P. ,Pedreschi,D. The Declarative Side of Magic, *to be published.*