Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Symbolic Finite Domains

Marc-Michel Corsini, Kaninda Musumbu, Antoine Rauzy
Laboratoire de Recherche en Informatique (LaBRI)
351 cours de la Libération, F-33405 (France)
Email: {corsini, musumbu, rauzy}@labri.greco-prog.fr

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

Abstract

The subject of this paper is Abstract Interpretation of Logic Programs, based on Constraint Solving over Finite Domains. More specifically, we use Toupie, a Constraint language based on the μ-calculus. Fixpoint computation is the basic execution mechanism of this language. Therefore it is very convenient and efficient for Abstract Interpretation of Logic programs. The following topics are covered in this paper. 1) The semantics and implementation of Toupie are described. 2) A class of abstract domains for Logic programs is defined. This class can be viewed as an extension of the domain `Prop` [1] to a limited class of first order formulae with equality. It happens that the elements of this domain coincide with the objects manipulated by Toupie, i.e. 0^+ formulae. It is also shown that 0^+ formulae can be used to define a completely different abstract domain for `possible sharing` analysis. 3) Transformation rules which allow the transformation of flat logic programs into Toupie programs are given. It is proved that execution of the Toupie programs on this abstract domains provides a correct bottom-up abstract interpretation of the corresponding Logic programs. That is: it provides a complete description of the success patterns for the program. 4) An experimental evaluation of the resulting analysis tool was performed and is thoroughly described. The system was used to analyse the benchmark programs used for the same purpose in [2, 3, 4, 5]. Four different instances of the generic domain were evaluated. Two variants of `Prop` were used for `groundness` and `covering` analysis. Two other instanciations were used for simple (but useful) type analyses. The experimental results shows that the system is very efficient and accurate on the benchmark programs. This seems to demonstrate that the proposed approach to abstract interpretation provides a versatile alternative to previous proposals for a significant class of analyses.

Finally, we explain in the conclusion that the approach can be extended to top-down Abstract Interpretation of logic programs along three possible ways that we will investigate in the future.

1 Introduction

Abstract Interpretation of logic and CLP programs is currently a very active field of research. This is due to the fact that the declarative nature of those languages make them amenable to a large variety of optimizations. Moreover, optimizing logic and CLP programs is a very important issue to make them competitive with procedural languages for large-scale applications.

Since the early work of Mellish [6], many frameworks have been proposed, for instance [7, 8, 9, 10, 11, 12, 13, 14, 15], and a large variety of applications and abstract domains have been developped,

for example [16, 17, 6, 18, 19, 20, 15]. Considerable progress has also been realised in terms of the algorithms (e.g., [8, 21, 4, 22, 10, 2, 23]), and the implementations (e.g., [4, 24, 3, 2, 25]).

In the Abstract Interpretation framework, the simulation of the unification can be seen as the problem of solving a set of equations over a so-called *abstract domain*. As numerous abstract domains are finite, it seems relevant to investigate the application of constraint programming languages to this field of research. In this paper, we investigate this issue in a comprehensive manner, including a semantic framework, two abstract domains, an implementation, and an experimental evaluation. Preliminary results were presented in [26, 27]. We only address *bottom-up* abstract interpretation but we sketch, in the conclusion, three possible methods to extend the method to top-down abstract interpretation. To the best of our knowledge, this work provides the first practical attempt to apply constraint programming techniques to Abstract Interpretation of logic programs.

The paper is organised as follows:

Section 2 presents the constraint language used as the basis of our work. Toupie [28] is a constraint language over symbolic finite domains, based on the μ-calculus [29]. The semantics of the language is formally defined and its main implementation features are described: the decision diagrams [30], the caching technique and the fixpoint algorithm.

Section 3 describes, in an informal way, how any pure Prolog program can be automatically translated in Toupie for the sake of Abstract Interpretation.

Section 4 describes a generic abstract domain ($\mathtt{O}^+$) whose abstract substitutions are exactly the kind of constraints handled by Toupie. The meaning of abstract substitutions is semantically defined by means of a concretization function. Operationally however, instantiating the domain amounts to rewrite in Toupie an abstract version of the built-in predicates of Prolog (mainly: unification). We show how this can be done systematically. Four sample domains are described.

Section 5 is a formal presentation of the various concepts presented in section 3. Moreover, we establish the correctness of the translation proposed.

Section 6 proposes a detailed experimental evaluation of the obtained Abstract Interpretation system. The evaluation uses the set of benchmarks which was extensively used for similar experiments in [4, 3, 31, 5]. The system is evaluated from the point of view of time and space efficiency. The results are excellent on the benchmarks and (seem to) suggest that the approach is practical. In particular, measurements on the size and number of generated decision diagrams seem to show that the potentially exponential behaviour of the system never occurs in practice. Accuracy of the analysis was also evaluated.

Section 7 describes another abstract domain easily implementable with Toupie. This domain can be used for `possible sharing` analysis.

Finally, section 8 provides the conclusion and explain how the approach can be extended to top-down abstract interpretation of logic and CLP programs.

2 Toupie: Syntax, Semantics and Implementation

2.1 Introduction

Toupie is a finite domain μ–calculus model checker that uses extended decision diagrams to represent relations and formulae. The propositional μ-calculus is a language that permits the description of

properties of finite states machines, see [29]. The idea is to introduce a least fixpoint operator μ, and to model the behaviour of finite state automata by means of Boolean formulae.

Toupie is a constraint language designed on an extension of μ-calculus for symbolic finite domains (i.e. arithmetic is not built-in in the language). In addition to the classical functionalities of symbolic finite domain constraint languages, one can define, in Toupie, relations (predicates) as fixpoints of equations. This gain in expressiveness is coupled with a practical efficiency that comes from the management of the relations via Decision Diagrams:

- Decision Diagrams encode the relations in a very compact manner by means of subtrees sharing.
- The algorithm that computes logical operations between two Decision Diagrams uses a generalized caching mechanism: no computation is performed twice.

In recent papers [26, 27], we have demonstrated that such a language can model and solve difficult problems, such as AI Puzzles with very impressive running times.

The rest of this section is organized as follows: subsection 2.2 and 2.4 give a formal presentation of the Toupie language. Subsection 2.5 presents the Decision Diagrams. Finally, subsection 2.6 describes the fixpoint algorithm which implements the semantics.

2.2 Syntax

A number of different versions of the μ-calculus have been proposed. Hereunder, we give the syntax of Toupie programs. For a more complete presentation the interested reader may see [32] and [33].

The first part of a Toupie program is a domain declaration (the domain in which the variables will be interpreted). It is of the form:

`domain {k1, k2, ... , kr}`

The interpretation domain is thus a set of constants (the symbols of constants have the same syntax as those in Prolog).

Now, there are two syntactic categories: *formulae* and *predicate definitions*. A Toupie program is a set of predicate definitions, having different predicate symbols for head (in reality, two predicates with distinct arities may use the same symbol, but for sake of simplicity we use in the remaining the former definition). A Toupie query is a formula. Formulae have the following form:

- The two constants 0 and 1.
- `(X1=X2)` or `(X1=k)` or `(X1#X2)` or `(X1#k)` where `X1` and `X2` are variables and `k` is a constant symbol belonging to the domain.
- `P(X1,..., Xn)` where `P` is an n-ary predicate variable and `X1,...,Xn` are individual variables.
- `~f, f & g, f | g, f <=> g, ...` where `f` and `g` are formulae and `~, &, |, <=>` denote the logical connectives $\neg, \wedge, \vee, \Longleftrightarrow$.
- `forall X1,..., Xn f` or `exist X1,..., Xn f` where `X1,...,Xn` are variables and `f` is a formula.

Predicate definitions are as follows:

`P(X1,...,Xn) += f` where `P` is an n-ary predicate variable, and `X1,...,Xn` are individual variables, and `f` is a formula. The token `+=` denotes a least fixpoint definition[1].

As in Prolog, the unquantified variables occurring in the right members of fixpoint definitions are assumed to be existentially quantified in front of the right hand side formula of the definition.

2.3 An example

In order to illustrate the previous subsection, we present the Toupie program for the well-known boatman's problem. The sad story is the following, a poor old man wants to cross the river with his wolf, his goat and cabbage; alas his two-places boat is too small for all his belongings, the river is too deep to cross it by foot and the closest bridge is far from there. He cannot leave alone the wolf and the goat nor the goat and the cabbage. To solve this terrible problem, the poor guy could try the program depicted figure 1. The four variables `Man, Wolf, Goat, Cabbage` are the respective positions of the man, wolf, goat and cabbage on the river's banks. The initial state is described by `((Man=left) & (Wolf=left) & (Goat=left) & (Cabbage=left))`. The forbidden states are described by `~(((Wolf=Goat) | (Goat=Cabbage)) & (Man#Goat))`. The set of accessible states are computed via fixed point, and our ol'man reaches safely the other bank with his goods since the call of **reachable** succeeds with the four arguments settled with `right` (see the session sample depicted figure 2).

```
domain {left,right}

reachable(Man,Wolf,Goat,Cabbage) += (
  ~(((Wolf=Goat) | (Goat=Cabbage)) & (Man#Goat))
  &
   (
    ((Man=left) & (Wolf=left) & (Goat=left) & (Cabbage=left))
    |
    ((M2#Man) & reachable(M2,Wolf,Goat,Cabbage))
    |
    ((M2=W2) & (Man=Wolf) & (M2#Man) & reachable(M2,W2,Goat,Cabbage))
    |
    ((M2=G2) & (Man=Goat) & (M2#Man) & reachable(M2,Wolf,G2,Cabbage))
    |
    ((M2=C2) & (Man=Cabbage) & (M2#Man) & reachable(M2,Wolf,Goat,C2))
   )
  )
```

Figure 1: The wolf, the goat and the cabbage

[1] Greatest fixpoints are also definable but we ignore them for simplicity.

```
2p |= load wgc.2p

2p ->  loading file wgc.2p

2p |= reachable(Man,Wolf,Goat,Cabbage) ?

{Man=left, Wolf=left, Goat=left}
{Man=left, Wolf=left, Goat=right, Cabbage=left}
{Man=left, Wolf=right, Goat=left}
{Man=right, Wolf=left, Goat=right}
{Man=right, Wolf=right, Goat=left, Cabbage=right}
{Man=right, Wolf=right, Goat=right}

2p |= reachable(right,right,right,right) ?

1

2p |=
```

Figure 2: A session sample

2.4 Semantics

The semantics of Toupie formulae is determined with respect to a structure $S = \langle Const, \mathcal{V} \rangle$ where *Const* is the interpretation domain defined at the beginning of the program, $\mathcal{V}$ is a denumerable set of variables including all the variables of the program.

Definition 1 [Individual Variable Assignments]
An *individual variable assignment* is a mapping from $\mathcal{V}$ into *Const*.

Definition 2 [Relation]
A *relation* on S is a mapping from $\mathcal{V} \rightarrow Const$ into $\mathcal{B}$, where $X \rightarrow Y$ stands for the set of mappings from X to Y and $\mathcal{B}$ stands for the Boolean values.

Definition 3 [Predicate Variable Interpretations]
Let Pr be the set of predicates occurring in the program. A *predicate variable interpretation* is a mapping from Pr into $(\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$, where $\mathbb{N}$ stands for the set of natural numbers.

This definition avoids the complications due to the different arities of the predicates. For a predicate of arity n, it suffices to consider that the corresponding function depends only on the first n numbers.

The semantics of a formula is thus a relation, and the semantics of a predicate (defined with a fixpoint equation) is a mapping from $(\mathbb{N} \rightarrow Const)$ into $\mathcal{B}$.

A Toupie program P assigns a meaning to a set of predicate symbols Pr. The semantics of the program is defined as the least fixpoint of a transformation $\mathcal{T}$. Let us denote $\mathcal{PR}$ the set

$Pr \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$ of predicate variable interpretations, and $\mathcal{RE}$ the set $(\mathcal{V} \rightarrow Const) \rightarrow \mathcal{B}$ of relations.

The program defines a continuous transformation:

$$\mathcal{T} : \mathcal{PR} \rightarrow \mathcal{PR}$$

Each formula `f` defines a function:

$$\mathcal{T}[\![\texttt{f}]\!] : \mathcal{PR} \rightarrow \mathcal{RE}$$

And each equation defines a function:

$$\mathcal{T}[\![\texttt{Eq}]\!] : \mathcal{PR} \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$$

The definition of $\mathcal{T}$ will use the following notation.

Definition 4 Let $f : A \rightarrow B$ be a function. Let $a_1, \ldots, a_n$ be distinct elements of A and $b_1, \ldots, b_n$ be arbitrary elements of B. We note

$$f[a_1/b_1, \ldots, a_n/b_n]$$

the function $g : A \rightarrow B$ such that $ga_i = b_i$ $(1 \leq i \leq n)$ and $ga = fa$ $(\forall a \notin \{a_1, \ldots, a_n\})$.
The notation

$$[a_1/b_1, \ldots, a_n/b_n]$$

stands for $f[a_1/b_1, \ldots, a_n/b_n]$ where f is an arbitrary function.

We are now in position to define the semantic function $\mathcal{T}$. Let π be a predicate variable interpretation, α be an individual variable assignment, and σ be an element of $(\mathbb{N} \rightarrow Const)$. $\mathcal{T}$ is defined inductively on the structure of formulae in the following way :

- $\mathcal{T}[\![1]\!]\ \pi\ \alpha\ = 1$ and $\mathcal{T}[\![0]\!]\ \pi\ \alpha\ = 0$.
- $\mathcal{T}[\![X_i = X_j]\!]\ \pi\ \alpha\ = \alpha(X_i) = \alpha(X_j)$.
- $\mathcal{T}[\![X_i = k]\!]\ \pi\ \alpha\ = \alpha(X_i) = k$.
- $\mathcal{T}[\![f \mid g]\!]\ \pi\ \alpha\ = \mathcal{T}[\![f]\!]\ \pi\ \alpha \vee \mathcal{T}[\![g]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![f\ \&\ g]\!]\pi\alpha\ = \mathcal{T}[\![f]\!]\ \pi\ \alpha \wedge \mathcal{T}[\![g]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![\forall\ X\ f]\!]\ \pi\ \alpha\ = \bigwedge_{k \in Const}(\mathcal{T}[\![f]\!]\ \pi\ \alpha[X/k])$.
- $\mathcal{T}[\![\exists\ X\ f]\!]\ \pi\ \alpha\ = \bigvee_{k \in Const}(\mathcal{T}[\![f]\!]\ \pi\ \alpha[X/k])$.
- $\mathcal{T}[\![P(X_{i_1}, \ldots, X_{i_r})]\!]\ \pi\ \alpha = \pi(P)(\alpha(X_{i_1}), \ldots, \alpha(X_{i_r}))$.
- $\mathcal{T}[\![P(X_1, \ldots, X_n)\texttt{+=}f\ ]\!]\ \pi\ \sigma = \mathcal{T}[\![f]\!]\ \pi\ [X_1/\sigma(1), \ldots, X_n/\sigma(n)]$.
- Finally, the transformation associated with the program is:
  $\mathcal{T}[\![Eq_1 \ldots Eq_n]\!]\ \pi = \pi[p_1/\mathcal{T}[\![Eq_1]\!]\ \pi, \ldots, p_n/\mathcal{T}[\![Eq_n]\!]\ \pi]$
  where the p_i are the predicates defined by the equations Eq_i.

Definition 5 [Denotation of a Toupie Formula wrt a Program] Let P be a Toupie program. Let f be a Toupie formula. Let D be the set of free variables occurring in f. By definition, the *denotation* of f wrt P is the function $\mathcal{D}[\![f]\!] : (D \rightarrow Const) \rightarrow \mathcal{B}$ such that, for all $\alpha \in (D \rightarrow Const)$,

$$\mathcal{D}[\![f]\!]\alpha \;=\; \mathcal{T}[\![f]\!](\mu(\mathcal{T}[\![P]\!]))\alpha',$$

where α' is any variable assignment such that

$$\alpha' X = \alpha X \ (\forall X \in D).$$

(The underlying program is kept implicit.)

2.5 Implementation of Constraints: Decision Diagrams

Decision Diagrams (DD for short) used in Toupie to encode relations, are an extension for symbolic finite domains of the Bryant's Binary Decision Diagrams [34, 30]. Thus, we review here only basic definitions and properties.

2.5.1 Shannon's Decomposition of Relations

In order to present Decision Diagrams, we need to introduce the *case* connective:

Definition 6 [case connective]
Let $Const = \{k_1, \ldots, k_r\}$ be the interpretation domain, X be a variable, and $f_1, \ldots, f_r$ be formulae. Then:

$$case(X, f_1, \ldots, f_r) = ((X = k_1) \wedge f_1) \vee \ldots \vee ((X = k_r) \wedge f_r)$$

Definition 7 [Shannon's Normal Form]
A formula f is in Shannon's normal form if one of the following points holds:

- $f = 0$ or $f = 1$,
- $f = case(X, f_1, \ldots, f_r)$, where X is a variable and $f_1 \ldots f_r$ are formulae in Shannon's normal form wherein X does not occur.

Property 8 [Shannon's Decomposition]
Let $V = \{X_1, \ldots X_n\}$ be a set of variables. Then, for any n-ary relation $R : (V \rightarrow Const) \rightarrow \mathcal{B}$ there exists a formula in Shannon's normal form encoding R.

Note that it may exist several formulae (even in Shannon's normal form) that are the syntactical expressions of a n-ary relation.

2.5.2 Reduced Ordered Decision DAGs

We can now define Decision DAGs:

Definition 9 [Decision DAGs]
Let $Const = \{k_1, \ldots, k_r\}$ be the interpretation domain, $V = \{X_1, \ldots, X_n\}$ be a set of variables. A *Decision DAG* F is a directed acyclic graph such that:

- F has two leaves 0 and 1.
- Each internal node of F is labelled by a variable X belonging to V and has r outedges labelled by $k_1, \ldots, k_r$.
- If a node labelled with the variable X is reachable from a node labelled with the variable Y then $X \neq Y$.

Now, it is clear that a DD encodes a formula in Shannon's normal form: the leaves of the DAG encode the corresponding Boolean constants and each internal node encodes a *case* connective.

We can know define reduced ordered decision DAGs:

Definition 10 [Reduced Ordered Decision DAGs]
Let $<$ be a total order on the variables $X_1, \ldots, X_n$. A *Reduced Ordered Decision DAG* F is a decision DAG such that:

- If a node labelled with the variable X is reachable from a node labelled with the variable Y then $X > Y$.
- Any node has at least two distincts sons ($case(X, f, \ldots, f) \uplus f$).
- Two distinct nodes F and G are syntactically distinct, i.e. they are either labelled by different variables or there exists an index $i \in 1..r$ such that the i-nth son of F is distinct of the i-nth son of G (reduction by means of maximum sharing of the sub-graphs)

In the remaining, we will consider only Reduced Ordered Decision DAGs and call them Decision DAGs (or DD for short).

Example 11 Assume the domain is $\{a, b\}$ and $p(X, Y, Z) = \{\langle a, a, a\rangle, \langle a, a, b\rangle, \langle a, b, a\rangle, \langle b, a, a\rangle, \langle b, b, a\rangle\}$; then the DD associated with p for the order $X < Y < Z$ is pictured figure 3. It encodes the formula: $case(X, case(Y, 1, case(Z, 1, 0)), case(Z, 1, 0))$ which is equivalent to p.

Property 12 [Canonicity]
Let R be a n-ary relation on the variables $X_1, \ldots, X_n$ and let $<$ be a total order on these variables. Then, there exists *exactly one* Reduced Ordered Decision DAG encoding R.

The proof is done by induction on the number of variables.

This property is fundamental: the test of equality between two relations encoded by means of two DDs is thus reduced to a test between the addresses of the DDs.

2.5.3 Logical Operations on DDs

Decision Diagrams are also very efficient for performing logical operations on relations. The following property holds:

Property 13 [Induction Principle]
Let $\odot$ be any binary logical operation and let $p = case(X, p_1, \ldots, p_r)$ and $q = case(X, q_1, \ldots, q_r)$ be two formulae in Shannon's normal form. Then, the following equality holds:

$$case(X, p_1, \ldots, p_r) \odot case(X, q_1, \ldots, q_r) = case(X, p_1 \odot q_1, \ldots, p_r \odot q_r)$$

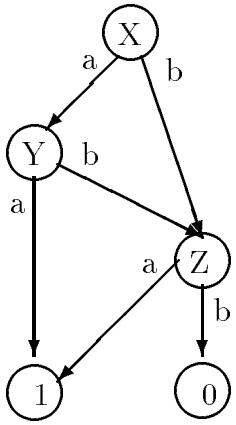

Figure 3: a Decision Diagram

Proof

$$
\begin{aligned}
& case(X, p_1, \ldots, p_r) \odot case(X, q_1, \ldots, q_r) \\
= {} & (X = k_1) \wedge (case(X, p_1, \ldots, p_r) \odot case(X, q_1, \ldots, q_r)) \\
& \vee \\
& \ldots \\
& \vee \\
& (X = k_r) \wedge (case(X, p_1, \ldots, p_r) \odot case(X, q_1, \ldots, q_r)) \\
= {} & (X = k_1) \wedge (p_1 \odot q_1) \vee \ldots \vee (X = k_r) \wedge (p_r \odot q_r) \\
= {} & case(X, p_1 \odot q_1, \ldots, p_r \odot q_r)
\end{aligned}
$$

□

It is easy to induce an effective procedure from this principle.

2.5.4 Memory Management for DDs

Decision Diagrams encode relations over finite domains in a very compact way by means of the sharing of the subtrees. This sharing is automatically performed by storing the nodes in a hashtable: each time a node $case(X, p_1, \ldots, p_r)$ is required, one first looks up the table and the node is created only if it does not belong to the table.

Another very important point that makes DDs efficient in practice is that the computation procedure uses a learning mechanism: each time a computation $p \odot q$ is performed, the result is memorized in an hashtable. Thus, this computation is never performed twice. Since the time required to an access in the hashtable is quasi-linear, the overhead due to this memorization is negligible. On the other hand, the improvement obtained in this way is often very big in practice, and becomes more and more important as the size of the problem grows up.

2.6 The Fixpoint Algorithm

The Toupie formulae are evaluated with a bottom-up lazy strategy that can also be seen as a top-down computation with memoization (see [35]). It means that for evaluating $f\&g$ one begins by computing the Decision DAGs associated with f and g and then one computes the binary operation & between these two DAGs in order to obtain the result. The fixpoint equations are evaluated in the same way, when needed: the DD associated with a predicate is computed at the first call of this predicate.

Sometimes a few lines of code are worth than long discurses. Figure 4 depicts the algorithm that computes the decision DAG associated with a fixpoint definition, as it is written for Toupie. The general principle is as follows: The formulae are encoded by means of syntactic trees. Each node corresponds to a variable or a connective and has an attribute "value" which is a pointer on a decision dag. The function `compute_dag_of_formula` goes trough the tree in a top-down way and computes the DD associated with a node by means of the DD associated with its sons.

In order to detect the recursive calls, each predicate definition has a status in `IDLE`, `ACTIVE`, `REACHED`. Intuitively, a predicate definition is `IDLE` when the predicate has never been called, `ACTIVE` when the computation of its value is in progress and `REACHED` when it is finished. This status is necessary to detect the mutually recursive calls and sufficient to reach the fixpoints by iterating the computation at each level of stack of calls.

With the above principle, the algorithm may perform many useless iterations and becoming quadratic (for the number of iterations) in the following case:

Assume the following program:

$$
\begin{aligned}
p_1(\vec{X}) \texttt{+=} (\ldots p_2(\vec{Y}) \ldots) \\
p_2(\vec{X}) \texttt{+=} (\ldots p_3(\vec{Y}) \ldots) \\
\ldots \\
p_n(\vec{X}) \texttt{+=} (\ldots p_1(\vec{Y}) \ldots)
\end{aligned}
$$

Assume too that the p_i occurring in the right members are the only predicate calls of the fixpoint definitions. Then, after a first computation of p_n the value of this predicate will change only if the one of p_1 changes. Thus, it is not necessary to recompute p_n in order to reach the fixpoint of p_{n-1}. The same thing holds for p_{n-1} and p_{n-2} $\ldots p_3$ and p_2.

In order to capture this phenomenon, the attribute `dependencies` has been added to predicate definitions. It encodes, for a predicate, the list of predicates calling it. The value of a predicate p is not recomputed until the one of another predicate q having p in its dependency graph has changed.

Now, the exact meaning of the attribute status can be defined:

- `IDLE` An equation is in this status when either it has never been called at the current step of the computation, or it has reached a local fixpoint but another predicate sent it a re-initialization message (with the function `update_fixed_point_status_in_list` described figure 5).
- `ACTIVE` An equation is in this status when the computation of its fixpoint is in progress.
- `REACHED` An equation is in this status when the computation of a local fixpoint is over, that is to say that a new iteration will not change its value. The global fixpoint is reached when all the local fixpoints are reached.

```
type_decision_dag compute_dag_of_fixed_point_equation(
                    type_fixed_point_equation self)
{
  type_decision_dag  new_value;

  if (self->fixed_point_status==IDLE) {
    self->fixed_point_status=ACTIVE;
    while (self->fixed_point_status!=REACHED) {
      new_value=compute_dag_of_formula(self->body);
      if (new_value==self->decision_dag)
        self->fixed_point_status=REACHED;
      else {
        self->decision_dag=new_value;
        update_fixed_point_status_in_list(self->dependencies);
        }
      }
    }

  return(self->decision_dag);
}
```

Figure 4: The Toupie fixpoint algorithm

```
void update_fixed_point_status_in_list(type_list dependencies)
{
  type_list current;
  type_fixed_point_equation called;

  current=dependencies;
  while (! isemptylist(current)) {
    called=(type_fixed_point_equation) carlist(current);
    if (called->fixed_point_status==REACHED) {
      called->fixed_point_status=IDLE;
      update_fixed_point_status_in_list(called->dependencies);
      }
    current=current->cdr;
    }
}
```

Figure 5: updating status with the dependency graph

3 Informal presentation

Before going on the formal definitions, theorems and proofs of our abstract interpretation with Toupie, we first give an intuition of the process.

In our framework, an abstract interpretation consists (roughly speaking) in executing a program over a finite abstract domain — which is built from a finite set of constants representing a partition of the concrete Herbrand domain — in order to get information about the concrete executions of the program. Our intention is to use the abstract domain as the interpretation domain for the translated Toupie program.

The domain `Prop`: The (abstract) domain[2] we consider along this section, is a slight variant of the domain `Prop` [36, 1, 5]. It uses two constants `g` (for ground) and `ng` (for nonground). All the theoretical results can be found in [1]. In `Prop` an abstract substitution over the set $D = \{x_1, \ldots, x_n\}$ of variables is thus a function from $D \rightarrow \{\texttt{g}, \texttt{ng}\}$ into the set $\mathcal{B}$ of Boolean values.

The intuition behind the domain `Prop` is that a substitution θ is abstracted by a function f such that $f(x_i) =$ `g` if and only if for all instances θ' of θ, θ' grounds x_i. As an example, the formula ($x =$ `g` $\wedge\, y =$ `g`) is a description for the substitution $\theta = \{x/a, y/b\}$, where x and y are variables whilst a and b are (Prolog) constants.

Flat programs: For technical reasons, the Toupie translation rules assume that logic programs are given in flat form. Any pure logic program can be translated into its flat form in a straightforward manner, thus it is not a restriction to only consider flat programs.

The correctness of the method is formally proved later in the section 5.

As an example, the flat version of the quicksort program using difference lists is given in figure 6.

The basic idea is to write explicitely all the unifications with two builtins $X = Y$ where X and Y are variables or $X = t$ where X is a variable and t a flat term *i.e.* a term with a functor and only variables as arguments.

The Toupie translation: To translate a (flat) Prolog program into a Toupie program is rather simple. To each Prolog literal corresponds a Toupie relation. Each sequence of Prolog literals is translated into a conjunction of relations. Finally each set of clauses with same head is viewed as a disjunction of conjunction of relations. The figure 7 will enlight the process.

When considering Toupie as a tool for abstract interpretation of logic programs, there are two main difficulties :

1. one has to deal with the explicit unification of the form $X = Y$ or $X = t$ where X,Y are variables and t is any (flat) prolog term.

2. The computation rule for Toupie is Bottom-Up, whilst the one for Prolog is Top-Down.

The translation of the builtin $X = Y$ is straightforward, whereas the translation of the builtin $X = t$ heavily depends on the domain under interest, as an example figure 7 depicts the automated translation of the quicksort program for the domain `Prop` wherein the `pr_` prefix stands for predicate and the `bi_` prefix for built-in.

For the second problem, one has to formally prove the correctness of the computation, see section 5.3 below.

To conclude this section, we give an example of a possible session with Toupie.

The solution for the query `pr_qsort(L1,L2) ?` is { `{L1=g, L2=g}`,`{L1=ng, L2=ng}` } which means that `L1=L2` in the interpretation domain `Prop`.

[2]the abstract domain for the Prolog program, and the domain for the Toupie program

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

partition([], _, [], []).
partition([ X5 | X6 ], X2 , [X5 | X7] , X4) :-
     X5 <= X2,
     partition( X6 , X2 , X7 , X4 ).
partition([ X5 | X6 ], X2 , [X5 | X7] , X4) :-
     X5 > X2,
     partition( X6 , X2 , X3 , X7 ).

qsort([] , X, X).
qsort([X4 | X5] , X2 , X3 ) :-
     partition( X5 , X4 , X6 , X7 ),
     qsort( X6 , X2 , [ X4 | X9 ] ),
     qsort( X7 , X9 , X3 ).
```

```
qsort(X1 , X2 ) :-
     X3 = [],
     qsort( X1 , X2 , X3 ).

partition(X1 , X2 , X3 , X4 ) :-
     X1 = [],
     X3 = [],
     X4 = [].
partition(X1 , X2 , X3 , X4 ) :-
     X1 =  [ X5 | X6 ] ,
     X3 =  [ X5 | X7 ] ,
     X5 <= X2,
     partition( X6 , X2 , X7 , X4 ).
partition(X1 , X2 , X3 , X4 ) :-
     X1 =  [ X5 | X6 ] ,
     X4 =  [ X5 | X7 ] ,
     X5 > X2,
     partition( X6 , X2 , X3 , X7 ).

qsort(X1 , X2 , X3 ) :-
     X1 = [],
     X3 = X2.
qsort(X1 , X2 , X3 ) :-
     X1 =  [ X4 | X5 ] ,
     partition( X5 , X4 , X6 , X7 ),
     qsort( X6 , X2 , X8 ),
     X8 =  [ X4 | X9 ] ,
     qsort( X7 , X9 , X3 ).
```

Figure 6: The original and flat versions of Quicksort

```
domain {g, ng}

pr_qsort(X1,X2) +=
     bi_nil(X3) & pr_qsort(X1,X2,X3)

pr_partition(X1,X2,X3,X4) +=
     bi_nil(X1) & bi_nil(X3) & bi_nil(X4)
    |
     bi_cons(X1,X5,X6)  & bi_cons(X3,X5,X7) &
     bi_ari(X5,X2)      & pr_partition(X6,X2,X7,X4)
    |
     bi_cons(X1,X5,X6)  & bi_cons(X4,X5,X7) &
     bi_ari(X5,X2)      & pr_partition(X6,X2,X3,X7)

pr_qsort(X1,X2,X3) +=
     bi_nil(X1) & X3 = X2
    |
     bi_cons(X1,X4,X5)  & pr_partition(X5,X4,X6,X7) &
     pr_qsort(X6,X2,X8) & bi_cons(X8,X4,X9) &
     pr_qsort(X7,X9,X3)

bi_nil(X1) += X1 = g

bi_ari(X1,X2) += X1 = g & X2 = g

bi_cons(X1,X2,X3) +=
     X1 = g & X2 = g & X3 = g
    |
     X1=ng & ( X2 = ng | X3 = ng)
```

Figure 7: Toupie translation of Quicksort

```
2p |= load quick_prop.2p

2p ->  loading file quick_prop.2p

2p |= display domain

domain {g,ng}

2p |= pr_qsort(L1,L2) ?

{L1=g, L2=g}
{L1=ng, L2=ng}

2p |= pr_qsort(g,L) ?

{L=g}

2p |= pr_partition(X1,X2,X3,X4) ?

{X1=g, X3=g, X4=g}

2p |= (  ( X1=X2 ) &  pr_partition(X1,X2,X3,X4) ) ?

{X1=g, X2=g, X3=g, X4=g}
```

Figure 8: Quicksort analysis with Toupie + `Prop`

The solution for the query `pr_partition(X1,X2,X3,X4) ?` is `{X1=g, X3=g, X4=g}` , this means that any assignment for `X2` , wrt the domain, is correct, or in other words, the solution is `{ {X1=g, X2=g, X3=g, X4=g}, {X1=g, X2=ng, X3=g, X4=g} }`.

4 O^+: a Generic Abstract Domain for Logic Programs

This section describes a generic abstract domains for logic programs which can be viewed as an extension of the domain `Prop` [36, 1]. `Prop` has proven to be useful for a large variety of applications including *groundness* [36, 5], *covering* and *suspension* [21, 37] analysis. The domain is parametrized on a set of constant symbols whose meaning is captured, semantically, by means of a concretization function and, operationally, by means of two abstract operations (unification). It is *downwards closed* (see [36]) and therefore adequate for success pattern analysis by bottom-up abstract interpretation [12].

As a matter of fact, abstract substitutions in this domain are exactly the kind of objects handled by the constraint language Toupie. Therefore, implementation of the domain is straightforward within Toupie.

The domain can also be interpreted over ground terms only. Hence it can be used for a simple form of *type* analysis.

This section is organized as follows. Subsection 4.1 recalls basic notions about substitutions and provides notational conventions. Subsections 4.2 and 4.3 present the abstract substitutions and their semantics. Subsection 4.4 defines the abstract unification and subsection 4.5 the other abstract operations. The generic domain is instantiated to two variants of the `Prop` domain in subsection 4.6. Two other instantiations, for type analysis are given in subsection 4.7. Finally, we explain in section 4.8 a method to support automatic instantiation of the generic domain.

4.1 Basic notions

We use the same conventions as in [38, 10, 39] for (concrete) substitutions. Those conventions are convenient for a clear and rigorous definition of the abstract domain.

We assume the existence of sets F_i and P_i ($i \geq 0$) denoting sets of functors and predicate symbols of arity i and of an infinite set PV of *program variables*. Variables in PV are ordered and denoted by the $x_1, x_2, \ldots, x_i, \ldots$.

The motivation behind these definitions is to allow the result of any predicate p/n to be expressed as a set of substitutions on program variables $x_1, \ldots, x_n$.

We assume the existence of another infinite set RV of *renaming (or standard) variables*. We distinguish two kinds of substitutions: *program substitutions* whose domain and codomain are subsets of PV and RV respectively, and *renaming (or standard) substitutions* whose domain and codomain are subsets of RV. In the following, PS denotes the set of program substitutions and RS the set of renaming substitutions. We use $var(o)$ to represent the set of variables in the syntactical object o.

PS_D is the set of θ such that $dom(\theta) = D$ We note CS_D, the set of subsets of PS_D which are *complete*. A set of substitutions is complete if it contains all variants of any of its elements.

Let θ be a substitution and $D \subseteq dom(\theta)$. The *restriction* of θ to D, denoted $\theta_{/_D}$, is the substitution θ' such that $dom(\theta') = D$ and $x\theta = x\theta'$ for all $x \in D$.

The definition of substitution composition is slightly modified to take into account the special role held by program variables. The modification occurs when $\theta \in PS$ and $\sigma \in RS$ for which $\theta\sigma \in PS$ is defined by $dom(\theta\sigma) = dom(\theta)$ and $x(\theta\sigma) = (x\theta)\sigma$ for all $x \in dom(\theta)$.

Unifiers are defined as usual but only belong to RS hereafter. We use $mgu(t_1, t_2)$ to denote the set of most general unifiers of t_1 and t_2.

4.2 Abstract Substitutions

In our generic abstract domain, abstract substitutions can be defined either as syntactic objects (i.e. formulae) or as semantic objects (i.e. functions from assignments to Boolean values). The first approach is more convenient for writing particular abstract substitutions. The second approach is simpler to define the semantics of the abstract domain. Therefore we basically use the semantic presentation in the following but we take the liberty of denoting semantic objects (functions) by formulae when convenient.

Definition 14 [Constants]
The abstract domain is parametrized on a finite set of *constant symbols* or *constants*. We note *Const*, the set of constants. We use the letter c to denote constants.

Definition 15 [Assignments]
Let D be a finite set of program variables. An *assignment* for D is any total function from D to *Const*. As usual, the set of assignments for D is denoted $D \rightarrow Const$.

Definition 16 [Abstract Substitutions]
Let D be a finite set of program variables. An *abstract substitution* over D is a function from $D \rightarrow Const$ to the set $\mathcal{B}$ of truth-values. Truth is denoted 1; falsity is denoted 0. The set of abstract substitutions over D is thus $(D \rightarrow Const) \rightarrow \mathcal{B}$. We use the griek letter β to denote elements of this set. D is called the domain of β and denoted by $dom(\beta)$.

Abstract substitutions over D can be denoted in an obvious way by first order formulae built with free variables from D, constants from *Const*, the equality predicate, logical connectives and quantifiers. Formulae are interpreted on *Const*.

4.3 Semantics of Abstract Substitutions

The meaning of abstract substitutions only depends on the meaning of constants which is given by the following function.

Definition 17 [Concretization of Constants]
The *concretization function* for constants has the following signature:

$$Cc : Const \rightarrow \mathcal{P}(Terms).$$

It is required that

$$\bigcup_{c \in Const} Cc(c) = Term.$$

It can be useful to define Cc in such a way that the family $\{Cc(c)\}_{c \in Const}$ is a partition of *Term* because otherwise the abstract domain entails some redundancy. But, on the other hand, non-redundancy can increase the number of constants and induce a loss of efficiency. Moreover, abstract operations can be simpler to define and to implement. As the partition hypothesis is not strictly necessary we do not assume it in the following.

The meaning of an abstract substitution over D is given by the following definition. The underlying intuition is that a variable assignment must satisfy the (formula representing the) abstract substitution whenever it consistently describes the terms bound to the variables of the domain of a (concrete) substitution described by β.

Definition 18 [Concretization Function]
Let $\beta \in (D \rightarrow Const) \rightarrow \mathcal{B}$. The *meaning* or *concretization* of β is, by definition, the set of program substitutions $Cc(\beta)$ such that

$$Cc(\beta) \;=\; \{\theta : \forall \sigma \in RS : \forall \alpha \in (D \rightarrow Const) : (\forall x \in D : x\theta\sigma \in Cc(\alpha x)) \Rightarrow \beta\alpha = 1\}.$$

Example 19 Assume the set *Const* defined as { `g, ng` }, and let D={ `x, y, z` }, and consider β as

$$(\ \texttt{x=g} \ \wedge \ \texttt{y=g} \ \wedge \ \texttt{z=g}\ ) \vee (\ \texttt{x=g} \ \wedge \ \texttt{y=ng} \ \wedge \ \texttt{z=ng}\ )$$

Then the substitution θ={ `x/f(a), y/g(z)` }, belongs to $Cc(\beta)$

We now define a notion of *non redundant* abstract substitution which will be useful for discarding useless abstract substitutions from the abstract domain. The motivation is that, in general, several abstract substitutions have the same value through the concretization function. The aim is to keep only one of them in the domain. It is natural to choose the least one (wrt implication). For example, 0 will be the only abstract substitution whose concretization is empty.

Definition 20 [Preordering over Assignments]
Let α, α' be two variable assignments for D. By definition,

$$\alpha' \leq \alpha$$

iff

$$\forall \theta \in PS_D : (\forall x \in D : x\theta \in Cc(\alpha x)) \Rightarrow (\exists \sigma \in RS : \forall x \in D : x\theta\sigma \in Cc(\alpha' x)).$$

$\alpha' \leq \alpha$ means that every substitution which satisfies α has an instance which satisfies α'. Therefore, if α satisfies an abstract substitution β but α' does not satisfy β, no concrete substitution corresponding to α belongs to $Cc(\beta)$. So β is in some sense redundant and can be replaced by a stronger abstract substitution. This observation is captured by the following definition.

Definition 21 [Non Redundant Abstract Substitutions]
Let $\beta \in (D \rightarrow Const) \rightarrow \mathcal{B}$. β is non redundant if and only if, by definition,

$$\forall \alpha, \alpha' \in (D \rightarrow Const) : \alpha' \leq \alpha \;\Rightarrow\; (\beta\alpha = 1 \Rightarrow \beta\alpha' = 1).$$

The following example will enlight the previous definitions.

Example 22 Let D = { `x, y` }, $Const$ = { `g, ng` }, α = [`x/g, y/ng`], α' = [`x/g, y/g`]. Then $\alpha' \leq \alpha$.

If one considers β as `(x=g` $\wedge$ `y=ng)` $\vee$ `(x=g` $\wedge$ `y=g)` and β' as `(x=g` $\wedge$ `y=ng)` , then β is non redondant whereas β' is not.

Definition 23 [Abstract Substitutions Revisited]

We will denote AS_D the set of $\beta \in (D \rightarrow Const) \rightarrow \mathcal{B}$ which are non redundant. All abstract substitutions considered in the rest of this section belong to some AS_D.

There are *abstraction* functions which map program substitutions on their "best" description and sets of program substitutions on the "smallest" abstract substitution whose concretization is a superset of the given set.

Definition 24 [Abstraction of a Program Substitution]

Let $\theta \in PS_D$. By definition,

$$Abs(\theta) \quad = \bigvee_{\substack{c_1,\ldots,c_n \in Const\,: \\ (\exists \sigma \in RS\,:\, \forall i\,:\, 1 \leq i \leq n\,:\, x_i\theta\sigma \in Cc(c_i))}} (x_1 = c_1 \wedge \ldots \wedge x_n = c_n).$$

This equality defines a function $Abs : PS_D \rightarrow AS_D$.

Example 25 Assume the set $Const$ defined as { `g, ng` }, and let D={ `x, y, z` }, and consider the substitution θ defined as follows: θ={ `x/f(a), y/g(z)` }. Then $Abs(\theta)$ is

(`x=g` $\wedge$ `y=g` $\wedge$ `z=g`) $\vee$ (`x=g` $\wedge$ `y=ng` $\wedge$ `z=ng`)

Definition 26 [Abstraction of a Set of Program Substitutions]

Let $\Theta \in CS_D$. By definition,

$$Abs(\Theta) \quad = \bigvee_{\theta \in \Theta} Abs(\theta).$$

This equality defines a function $Abs : CS_D \rightarrow AS_D$.

The next theorem expresses that the concrete and abstract domains fulfill all conditions required by the abstract Interpretation framework of the Cousots [40].

Theorem 27 (CS_D, AS_D, Abs, Cc) is a *Galois insertion.*

4.4 Abstract Unification

In this section, we give general formulae which allow the definition of abstract versions of built-in predicates, for any particular instantiation of the abstract domain. We focus on *abstract unification.* Similar formulae are easily found for other built-in predicates such as arithmetic predicates: `is`, `<`,... . Notice that some built-in predicates are not usable in the basic (bottom-up) Abstract Interpretation framework implemented by Toupie. However this aspect is independent of the abstract

domain *in se*. Moreover it is also possible to use Toupie for top-down Abstract Interpretation. In this more elaborate context, test predicates will be considered.

To define an abstract operation, we state two different kind of definitions. The *specification* states the property that the abstract operation must enjoy in order to be a correct (consistent) approximation of the corresponding "concrete" operation. The *implementation* defines the practical way to compute the abstract operation, i.e. an algorithm or the definition of a finitely representable object. Correctness of an operation is ensured if its implementation meets its specification.

In our framework we use *flat* (or normalized) programs. Therefore, unification of terms is achieved by two operations: *unification of two program variables* and *unification of a program variable and a functor*. We define hereunder the abstract versions of those operations.

4.4.1 Unification of Two Program Variables $(x_i = x_j)$

Specification 28

Let $D = \{x_1, \ldots, x_n\}$. The abstract unification of x_i and x_j $(1 \leq i, j \leq n)$, denoted `UNIF-VAR`(i, j, D), returns an abstract substitution $\beta \in AS_D$ for which the following holds:

$$\left.\begin{array}{l} x_i\theta = x_j\theta \\ \theta \in PS_D \end{array}\right\} \Rightarrow \quad \theta \in Cc(\beta).$$

Implementation 29

$$\beta \stackrel{\text{def}}{=} (x_i = x_j).$$

Theorem 30 Implementation of `UNIF-VAR` is consistent.

4.4.2 Unification of a Program Variable and a Functor $(x_i = f(x_{j_1}, \ldots, x_{j_m}))$

Specification 31

Let $D = \{x_1, \ldots, x_n\}$. The abstract unification of x_i and $f(x_{j_1}, \ldots, x_{j_m})$ $(1 \leq i, j_1, \ldots, j_m \leq n)$, denoted `UNIF-FUNC`$(f, i, j_1, \ldots, j_m, D)$, returns an abstract substitution $\beta \in AS_D$ for which the following holds:

$$\left.\begin{array}{l} x_i\theta = f(x_{j_1}, \ldots, x_{j_m})\theta \\ \theta \in PS_D \end{array}\right\} \Rightarrow \quad \theta \in Cc(\beta).$$

Implementation 32

Let R_f be the $(m+1)$-ary relation such that $\forall c, c_1, \ldots, c_m \in Const$:

$$(c, c_1, \ldots, c_m) \in R_f \text{ iff } \exists t_1, \ldots, t_m \in Term : \left\{\begin{array}{l} f(t_1, \ldots, t_m) \in Cc(c) \ \ \& \\ \forall i : 1 \leq i \leq m : t_i \in Cc(c_i) \end{array}\right.$$

$$\beta \stackrel{\text{def}}{=} \left(\bigvee_{(c,c_1,\ldots,c_m)\in R_f} (x_i = c \wedge x_{j_1} = c_1 \wedge \ldots \wedge x_{j_m} = c_m) \right).$$

Theorem 33 Implementation of `UNIF-FUNC` is consistent.

4.5 Other Abstract Operations

Additionally to abstract unification, some other abstract operations are needed for the purpose of bottom-up abstract interpretation: intersection, union, projection, extension and renaming.

4.5.1 Intersection of Abstract Substitutions

This operation is useful to express that an atom $p(x_1,\ldots,x_n)\theta$ is an output pattern whenever there exists a clause $p(x_1,\ldots,x_n) \leftarrow l_1,\ldots,l_m$ such that $l_1\theta,\ldots,l_m\theta$ are all output patterns.

Specification 34 [Intersection of Abstract Substitutions]
Let $(\beta_i)_{i\in I}$ a finite family of abstract substitutions over the same domain D. The intersection of $(\beta_i)_i \in I$, denoted $\bigsqcap_{i\in I}\beta_i$ has to satisfy the following properties.

1. $dom(\bigsqcap_{i\in I}\beta_i) = D$,
2. $\bigcap_{i\in I} Cc(\beta_i) \subseteq Cc(\bigsqcap_{i\in I}\beta_i)$.

Implementation 35

$$\bigsqcap_{i\in I}\beta_i \stackrel{\text{def}}{=} \bigwedge_{i\in I}\beta_i$$

Theorem 36 Operation $\bigsqcap$ is consistent. In fact, there is a stronger result: $\bigsqcap$ is exact in the sense that:

$$\bigcap_{i\in I} Cc(\beta_i) = Cc(\bigsqcap_{i\in I}\beta_i).$$

4.5.2 Union of Abstract Substitutions

This operation is useful to express that an atom $p(x_1,\ldots,x_n)\theta$ is an output pattern for a procedure of name p whenever there exists a clause c of the procedure for which $p(x_1,\ldots,x_n)\theta$ is an output pattern.

Specification 37 [Union of Abstract Substitutions] Let $(\beta_i)_{i\in I}$ a finite family of abstract substitutions over the same domain D. The union of $(\beta_i)_i \in I$, denoted $\bigsqcup_{i\in I}\beta_i$ has to satisfy the following properties.

1. $dom(\bigsqcup_{i\in I}\beta_i) = D$,
2. $\bigcup_{i\in I} Cc(\beta_i) \subseteq Cc(\bigsqcup_{i\in I}\beta_i)$.

Implementation 38

$$\bigsqcup_{i\in I} \beta_i \stackrel{\text{def}}{=} \bigvee_{i\in I} \beta_i.$$

Theorem 39 Operation $\bigsqcup$ is consistent. Once again, there is a stronger result:

$$\bigcup_{i\in I} Cc(\beta_i) = Cc(\bigsqcup_{i\in I} \beta_i).$$

4.5.3 Projection on a Subdomain

The following operation will be used to express that $p(x_1,\ldots,x_n)\theta_{/\{x_1,\ldots,x_n\}}$ is an ouput pattern whenever $p(x_1,\ldots,x_n)\theta$ is.

Specification 40 [Projection of an Abstract Substitution on a Subdomain] Let β be an abstract substitution and D a set of program variables such that $D \subseteq dom(\beta)$. The projection of β on D, denoted $\beta_{/D}$, has to respect the following conditions:

1. $dom(\beta_{/D}) = D$,
2. $\forall\theta \in Cc(\beta) :\ \theta \in Cc(\beta) \ \Rightarrow\ \theta_{/D} \in Cc(\beta_{/D})$

Implementation 41

$\beta_{/D} \stackrel{\text{def}}{=} (\exists x_{i_1} \ldots \exists x_{i_m} : \beta)$
where $\{x_{i_1},\ldots,x_{i_m}\} = dom(\beta) - D$.

Theorem 42
Projection is consistent.

4.5.4 Domain Extension

This operation will be used to express that $p(x_{i_1},\ldots,x_{i_n})\theta$ is an ouput pattern whenever $p(x_{i_1},\ldots,x_{i_n})\theta_{/\{x_{i_1},\ldots,x_{i_n}\}}$ is.

Definition 43 [Domain Extension]
Let β be an abstract substitution and D a set of program variables such that $dom(\beta) \subseteq D$. The extension of β to D, denoted β_D, is the unique abstract substitution such that:

1. $dom(\beta_D) = D$,
2. $\forall\alpha \in D : \beta_D\alpha = \beta[x_{i_1}/\alpha x_{i_1},\ldots,x_{i_n}/\alpha x_{i_n}]$
   where $x_{i_1},\ldots,x_{i_n} = dom(\beta)$.

Theorem 44
Let β be an abstract substitution and D a set of program variables such that $dom(\beta) \subseteq D$. Let θ such that $dom(\theta) = D$. Under those hypotheses,

$\theta_{/dom(\beta)} \in Cc(\beta) \quad\Rightarrow\quad \theta \in Cc(\beta_D)$.

4.5.5 Domain Renaming

The following definitions will be helpful to handle procedure calls.

Definition 45 [Domain Renaming for a Program Substitution]
Let θ be a substitution of the form $\{x_{i_1}/t_1, \ldots, x_{i_n}/t_n\}$. Let $x_{j_1}, \ldots, x_{j_n}$ be distinct program variables. The substitution $\{x_{j_1}/t_1, \ldots, x_{j_n}/t_n\}$ will be denoted by

$$\theta[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}].$$

When $\{x_{i_1}, \ldots, x_{i_n}\} \subseteq dom(\theta)$,

$$\theta[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}]$$

will denote

$$(\theta_{/\{x_{i_1}, \ldots, x_{i_n}\}})[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}].$$

Definition 46 [Domain Renaming for an Abstract Substitution]
Let β be an abstract substitution such that $dom(\beta) = \{x_{i_1}, \ldots, x_{i_n}\}$. Let $x_{j_1}, \ldots, x_{j_n}$ be distinct program variables. The abstract substitution $\beta' : (\{x_{j_1}, \ldots, x_{j_n}\} \rightarrow Const) \rightarrow \mathcal{B}$ such that

$$\beta'\alpha = \beta[x_{j_1}/\alpha x_{i_1}, \ldots, x_{j_n}/\alpha x_{i_n}],$$

will be denoted by

$$\beta[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}].$$

When $\{x_{i_1}, \ldots, x_{i_n}\} \subseteq dom(\beta)$,

$$\beta[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}]$$

will denote

$$(\beta_{/\{x_{i_1}, \ldots, x_{i_n}\}})[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}].$$

Theorem 47 [Consistency of Domain Renaming]
Let β and θ be , respectively, an abstract and a program substitution such that $dom(\beta) = dom(\theta)$. Let $x_{j_1}, \ldots, x_{j_n}$ be distinct program variables. Assume $\{x_{i_1}, \ldots, x_{i_n}\} \subseteq dom(\beta)$. Then,

$$\theta \in Cc(\beta) \quad \Rightarrow \theta[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}] \in Cc(\beta[x_{i_1} \leftarrow x_{j_1}, \ldots, x_{i_n} \leftarrow x_{j_n}]).$$

4.6 Instantiations 1 and 2: two variants of `Prop`

`Prop` The first instantiation is equivalent to the domain `Prop`, first introduced by Marriott and Sondergaard [36], studied theoretically in [1] and experimentally in [5]. The only difference is that we use two constants instead of interpreting program variables as propositional formulae. This domain was shown useful for a large variety of applications in [37]. Its accuracy was shown very high in [5].

Recall that instantiating the abstract domain amounts to defining the set *Const* of constants and the basic concretization function $Cc : Const \rightarrow \mathcal{P}(Terms)$. In this instantiation:

$$\begin{array}{lcl} Const & \stackrel{\text{def}}{=} & \{\mathtt{g}, \mathtt{ng}\} \\ \\ Cc(\mathtt{g}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is ground.}\} \\ Cc(\mathtt{ng}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is not ground.}\} \end{array}$$

For each m-ary functor symbol f, the corresponding relation R_f is defined by:

$$(c, c_1, \ldots, c_m) \in R_f \ \text{ iff } \ (c = c_1 = \ldots = c_m = \mathtt{g}) \ \text{ or } (c = \mathtt{ng} \ \& \ \exists i : 1 \leq i \leq m : c_i = \mathtt{ng}).$$

Recall that the underlying meaning of `Prop` is to compute definite groundness of variables.

`Prop`$^+$ The second instantiation is a slightly more specialized version of `Prop`, first sketched in [26]. Three constants are used and constant `ng` is replaced by `ngv` and `v`. This domain allows for more refined results on many interesting examples such as `quicksort`.

In this instantiation:

$$\begin{array}{lcl} Const & \stackrel{\text{def}}{=} & \{\mathtt{g}, \mathtt{ngv}, \mathtt{v}\} \\ \\ Cc(\mathtt{g}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is ground.}\} \\ Cc(\mathtt{ngv}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is not ground nor a variable.}\} \\ Cc(\mathtt{v}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is a variable.}\} \end{array}$$

For each m-ary functor symbol f, the corresponding relation R_f is defined by:

$$(c, c_1, \ldots, c_m) \in R_f \ \text{ iff } \ (c = c_1 = \ldots = c_m = \mathtt{g}) \ \text{ or } (c = \mathtt{ngv} \ \& \ \exists i : 1 \leq i \leq m : c_i \neq \mathtt{g}).$$

The intended meaning of `Prop`$^+$ is to compute definite nonfreeness information.

4.7 Instantiation 3 and 4: `Type`, Simple Type Analysis

The two next instantiations implement a simple type analysis which can be useful for optimizations such as tag removal in a compiler.

`Ground Types` This third instantiation only deals with ground types. Clearly our framework can be restricted to ground patterns. All theoretical results remain valid and all proofs can be rewritten without any change except restricting to ground terms and patterns. Four basic types are considered: integers, lists, constants and functors (the other terms)

More precisely, the set of constants and the basic concretization function are defined as follows:

$$\begin{array}{lcl} Const & \stackrel{\text{def}}{=} & \{\mathtt{int}, \mathtt{lst}, \mathtt{cst}, \mathtt{fct}\} \\ \\ Cc(\mathtt{int}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is an integer.}\} \\ Cc(\mathtt{lst}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is a list.}\} \\ Cc(\mathtt{cst}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is a constant different from the empty list.}\} \\ Cc(\mathtt{fct}) & \stackrel{\text{def}}{=} & \{t : t \in Term \ \ \& \ \ t \text{ is any other term.}\} \end{array}$$

For each m-ary functor symbol f, the corresponding relation R_f is defined as follows (i denotes an integer, a denotes a constant different from `nil` and f denotes an m-ary functor different from the list constructor ($m > 0$):

$$\begin{array}{lll} (c) \in R_i & \text{iff} & c = \texttt{int}. \\ (c) \in R_{[]} & \text{iff} & c = \texttt{lst}. \\ (c, c_1, c_2) \in R_{[.|.]} & \text{iff} & c = \texttt{lst} \ \& \ \ c_2 = \texttt{lst}. \\ (c) \in R_a & \text{iff} & c = \texttt{cst}. \\ (c, c_1, \ldots, c_m) \in R_f & \text{iff} & c = \texttt{fct}. \end{array}$$

`Types` In this instantiation, non ground terms are also considered. Curiously however, the only modification amounts to adding a new constant `v` to the set *Const*. The relations R_f are unchanged.

4.8 Towards the Automatic Instantiation of 0^+

Instantiating the relations R_f manually can become cumbersone and error-prone for more complex instantiations. It is possible to design a formal language for describing the denotation of constants. This language could be processed to infer automatically the basic relations. A similar approach has already been used for type verification in the *FOLON* system, a programming environment for logic programming (see [41, 42]). It has proven to be useful and efficient in such a context.

In the context of type verification, types can be specified by context-free grammars or any equivalent formalism (see [43]). Deriving the basic information is then almost straightforward. This approach is different from the usual approach of "infering" the types from the program. We believe that it can nevertheless be useful, for example, in a programming environment context.

5 Bottom-up Abstract Interpretation of Prolog with Toupie

In this section, we present our framework for Abstract Interpretation. Bottom-up Abstract Interpretation of logic programs has been studied by numerous researchers. See, for example, [7, 22, 12]. The correctness of our approach mostly relies on the results of [7, 12]. We provide a direct proof of our method, however, in order to propose a self-contained and convincing presentation.

The rest of the section is organized as follows.

Subsection 5.1 describes the syntax of flat logic programs. (Considering flat programs is not a restriction, because any pure logic program can be translated into a flat program in a straightforward way). Subsection 5.2 gives the rules allowing to translate a flat logic programs into a Toupie program whose execution performs the abstract interpretation of the logic program. Subsection 5.3 formally proves the correctness of the method.

5.1 Syntax of Flat Prolog

The abstract syntax of the language can be defined by the following grammar:

$$
\begin{array}{lcl}
P & \in & Programs \\
pr & \in & Procedures \\
c & \in & Clauses \\
g & \in & Goals \\
l & \in & Literals \\
f & \in & Functors \\
p & \in & ProcedureNames \\
x_i & \in & ProgramVariables
\end{array}
\qquad
\begin{array}{lcl}
P & ::= & \{pr_1, \ldots, pr_n\} \\
pr & ::= & c \mid c.pr \\
c & ::= & p(x_1, \ldots, x_n) \leftarrow g \\
g & ::= & <> \mid l.g \\
l & ::= & x_i = x_j \mid x_{i_1} = f(x_{i_2}, \ldots, x_{i_n}) \mid p(x_{i_1}, \ldots, x_{i_n})
\end{array}
$$

5.2 Translation Rules

In order to handle abstract unification of a variable and a functor, a Toupie predicate definition is associated to each functor f/n, as follows:

$$
\begin{array}{lll}
\texttt{bi\_f(X}_0\texttt{,X}_1\texttt{,...,X}_n\texttt{)} & \texttt{+=} & (\texttt{X}_0\texttt{=c}_0^1 \;\&\; \texttt{X}_1\texttt{=c}_1^1 \;\&\; \ldots \;\&\; \texttt{X}_n\texttt{=c}_n^1) \;\; | \\
 & & \quad\ldots \qquad\quad \ldots \qquad\qquad\quad | \\
 & & (\texttt{X}_0\texttt{=c}_0^m \;\&\; \texttt{X}_1\texttt{=c}_1^m \;\&\; \ldots \;\&\; \texttt{X}_n\texttt{=c}_n^m)\texttt{.}
\end{array}
$$

where $\{(c_0^i, c_1^i, \ldots, c_n^i) \;:\; 1 \leq i \leq m\} = R_f$. It is worth noticing that any given Prolog program contains only finitely many functors. Therefore, only a finite number of such predicate definitions will be actually generated for a given analysis. Moreover, in practice, many relations R_f are the same. Hence several simplifications are possible.

It is straightforward that the Toupie formula $\texttt{bi\_f(X}_{i_0}\texttt{,X}_{i_1}\texttt{,...,X}_{i_n}\texttt{)}$ denotes the same abstract substitution as the following "usual" formula:

$$\bigvee_{(c_0,c_1,\ldots,c_n)\in R_f} (x_{i_0} = c_0 \wedge x_{i_1} = c_1 \wedge \ldots \wedge x_{i_n} = c_n).$$

Therefore it correctly implements operation `UNIF_FUNC`.

Several other abstract operations have been used in the experiments. They implement arithmetic predicates such as `is`, `<`, `>`, ... They can be implemented in a similar way that we do not explain for brevity. Finally, other built-in predicates can be "simulated" at the Prolog program level (see section 6).

The translation rules of a flat Prolog program in Toupie are given in figure 9. We take the convention of denoting *constr** the Toupie translation of the syntactic construct *constr*. Figure 7 depicts the translation of the quicksort program from figure 6 wrt `Prop`.

5.3 Correctness

The correctness of our Abstract Interpretation method relies on the next theorem which is a straightforward consequence of well-known results of logic programming.

Theorem 48
let l be a literal of the form $p(x_1, \ldots, x_n)$. Let θ be a program substitution such that $dom(\theta) = \{x_1, \ldots, x_n\}$. Then $l\theta$ is a success pattern wrt a flat logic program *Prog* if and only if there exists a clause c in *Prog* and a program substitution θ' such that:

constr	*constr**
$\{pr_1,\ldots,pr_n\}$	$\{pr_1^*,\ldots,pr_n^*,bi_1,\ldots,bi_m\}$ where $bi_1,\ldots,bi_m$ are the predicate definitions corresponding to all the built-ins used in $\{pr_1,\ldots,pr_n\}$.

pr, of the form
$p(x_1,\ldots,x_n) \leftarrow g_1$
$\ldots$
$p(x_1,\ldots,x_n) \leftarrow g_m$
	`pr_p(X`$_1$`,...,X`$_n$`) +=` `(exist X`$_{n+1}$`,...,X`$_{r_1}$`:`g_1^*`) |`
	$\ldots$ `|`
	`(exist X`$_{n+1}$`,...,X`$_{r_m}$`:`g_m^*`)`
	where `X`$_{n+1}$`,...,X`$_{r_j}$ are the local variables of g_j^* $(1 \leq j \leq m)$.

$l_1,\ldots,l_n$	l_1^* `&` $\ldots$ `&` l_n^*
$x_i = x_j$	`X`$_i$`=X`$_j$
$x_{i_1} = f(x_{i_2},\ldots,x_{i_n})$	`bi_f(X`$_{i_1}$`,X`$_{i_2}$ `, ..., X`$_{i_n}$`)`
$p(x_{i_1},\ldots,x_{i_n})$	`pr_p(X`$_{i_1}$`,...,X`$_{i_n}$`)`

Figure 9: Translation rules

1. $dom(\theta') = \{x_1, \ldots, x_m\}$ where $x_1, \ldots, x_m$ is the set of variables in c;
2. $c ::= p(x_1, \ldots, x_n) \leftarrow l_1, \ldots, l_r$;
3. $l_i\theta'$ is a success pattern wrt P $(1 \leq i \leq r)$;
4. $\theta = \theta'_{/\{x_1,\ldots,x_n\}}$.

The previous theorem motivates the next definition:

Definition 49 [Level of a success pattern]
We define the *level* of a success pattern thanks to the following two rules:

1. A success pattern has level 0 if and only if it is an instance of a built-in predicate.
2. Let $i \geq 1$. A success pattern has level i iff it has not level j for $0 \leq j < i$ and it can be obtained from success patterns of level lower than i according to the previous theorem.

Theorem 50
Any success pattern has a unique level according to the previous definition.

Proof Consequence of well-known results from logic programming. □

The next, fundamental, theorem states the correctness of the proposed Abstract Interpretation method. To understand its meaning it is worth remembering that

1. Given a Toupie program, Toupie computes a set of decision diagrams; each of those decision diagrams is associated to a predicate symbol `p` of the program and encodes the relation $\mathcal{D}[\![\mathtt{p}(\mathtt{X_1}, \ldots, \mathtt{X_n})]\!]$.
2. The computed decision diagrams can then be displayed or used for further calculations.
3. The relations $\mathcal{D}[\![\mathtt{p}(\mathtt{X_1}, \ldots, \mathtt{X_n})]\!]$ are abstract substitutions in the sense of section 4.

Theorem 51 [Correctness of Bottom-up Abstract Interpretation with Toupie]
Let *Prog* be a flat Prolog program. Assume *Prog** to be the underlying Toupie program. Let l be a literal occurring in *Prog* and p the name of a n-ary procedure of *Prog*. Let D be a set of program variables and θ a program substitution such that $D = dom(\theta)$. The following results hold:

1. $\left.\begin{array}{l} var(l) \subseteq D, \\ l\theta \text{ is a success pattern} \end{array}\right\} \Rightarrow \theta \in Cc(\mathcal{D}[\![l^*]\!]_D)$;
2. $\left.\begin{array}{l} dom(\theta) = \{x_1, \ldots, x_n\}, \\ p(x_1, \ldots, x_n)\theta \text{ is a success pattern} \end{array}\right\} \Rightarrow \theta \in Cc(\mathcal{D}[\![p(x_1, \ldots, x_n)^*]\!])$.

Proof
The proof is by induction on the level of success patterns.

- Consider first success patterns of level 0. Then l must be a built-in. Therefore two cases are to be considered.

1. l is of the form $x_i = x_j$. In this case,

   $l\theta$ is a success pattern
   $\Rightarrow$ $x_i\theta = x_j\theta$
   $\Rightarrow$ $\theta \in Cc((x_i = x_j)_D)$ (consistency of `AI_VAR`)
   $\Rightarrow$ $\theta \in Cc(\mathcal{D}[\![\mathtt{X_i} = \mathtt{X_j}]\!]_D)$
   $\Rightarrow$ $\theta \in Cc(\mathcal{D}[\![l^*]\!]_D)$

2. l is of the form $x_{i_0} = f(x_{i_1}, \ldots, x_{i_n})$. In this case,

   $l\theta$ is a success pattern
   $\Rightarrow$ $x_{i_0}\theta = f(x_{i_1}, \ldots, x_{i_n})\theta$
   $\Rightarrow$ $\theta \in Cc(\bigvee_{(c_0,c_1,\ldots,c_n)\in R_f}(x_{i_0} = c_0 \wedge x_{i_1} = c_1 \wedge \ldots \wedge x_{i_n} = c_n)_D)$
   (consistency of `AI_FUNC`)
   $\Rightarrow$ $\theta \in Cc(\mathcal{D}[\![\mathtt{bi\_f(X_{i_0}, X_{i_1}, \ldots, X_{i_n})}]\!]_D)$
   $\Rightarrow$ $\theta \in Cc(\mathcal{D}[\![l^*]\!]_D)$

- Let us consider now patterns of level $i > 0$.

  1. We first prove the second part of the theorem because the first part is based on the second one.

     $p(x_1, \ldots, x_n)\theta$ is a success pattern

     $\Rightarrow$ { there exists a clause $p(x_1, \ldots, x_n) \leftarrow l_1, \ldots, l_r$ and a substitution θ' such that $l_j\theta'$ are success patterns of level $k_j < i$ $(1 \le j \le r)$ and $\theta = \theta'_{/\{x_1,\ldots,x_n\}}$

     $\Rightarrow$ { $\theta' \in Cc(\mathcal{D}[\![l_j^*]\!]_D)$ $(1 \le j \le r)$ where D is the set of all variables in the clause (induction hypothesis)

     $\Rightarrow$ $\theta' \in Cc(\bigwedge_{j=1}^{r} \mathcal{D}[\![l_j^*]\!]_D)$ (consistency of abstract intersection)

     $\Rightarrow$ $\theta' \in Cc(\mathcal{D}[\![l_1^* \ \& \ \ldots \ \& \ l_r^*]\!]_D)$ (definition of $\mathcal{D}$)

     $\Rightarrow$ { $\theta \in Cc(\exists x_{n+1} \ \ldots \ \exists x_m \ : (\mathcal{D}[\![l_1^* \ \& \ \ldots \ \& \ l_r^*]\!]_D))$ where $x_{n+1}, \ldots, x_m$ are the local variables of the clause (consistency of projection)

     $\Rightarrow$ { $\theta \in Cc(\mathcal{D}[\![g_1^* \mid \ldots \mid g_s^*]\!])$ where $g_1, \ldots, g_s$ are the bodies of the clauses of procedure p (consistency of abstract union, definition of $\mathcal{D}$)

     $\Rightarrow$ { $\theta \in Cc(\mathcal{D}[\![p(x_1, \ldots, x_n)^*]\!])$ (definition of $\mathcal{D}$)

  2. Let $l\theta$ be a success pattern of level $i > 0$. l must be of the form $p(x_{i_1}, \ldots, x_{i_n})$. Therefore,

$l\theta$ is a success pattern

$\Rightarrow$ $p(x_{i_1}, \ldots, x_{i_n})\theta$ is a success pattern

$\Rightarrow$ $p(x_1, \ldots, x_n)\theta[x_{i_1} \leftarrow x_1, \ldots, x_{i_n} \leftarrow x_n]$ is a success pattern

$\Rightarrow$ $\theta[x_{i_1} \leftarrow x_1, \ldots, x_{i_n} \leftarrow x_n] \in Cc(\mathcal{D}[\![p(x_1, \ldots, x_n)^*]\!])$ (part 1. above)

$\Rightarrow$ $\begin{cases} \theta_{/\{x_{i_1},\ldots,x_{i_n}\}} \in Cc(\mathcal{D}[\![p(x_{i_1}, \ldots, x_{i_n})^*]\!]) \\ \text{(consistency of renaming, definition of } \mathcal{D}) \end{cases}$

$\Rightarrow$ $\theta \in Cc(\mathcal{D}[\![p(x_{i_1}, \ldots, x_{i_n})^*]\!]_D)$ (consistency of domain extension)

$\Rightarrow$ $\theta \in Cc(\mathcal{D}[\![l^*]\!]_D)$

□

6 Experimental Evaluation

In this section, we apply the four abstract domains presented in subsections 4.6 and 4.7 to the analysis of a significant set of benchmark programs and we give some statistics to demonstrate the space and time efficiency of our approach.

The section is organized as follows. Subsection 6.1 describes the benchmark programs. Subsections 6.2 to 6.6 discuss in great details the time and space efficiency of the system on the benchmarks. Subsection 6.7 provides information related to the accuracy of the analysis.

6.1 Benchmark Programs

We use the benchmark programs proposed by B. Le Charlier and P. Van Hentenryck in [2, 3, 31, 5]. The hereunder description is extracted from [2].

The programs are hopefully representative of "pure" logic programs (i.e. without the use of dynamic predicates such as `assert` and `retract`). They are taken from a number of authors and used for various purposes from compiler writing to equation-solvers, combinatorial problems, and theorem-proving. Hence they should be representative of a large class of programs. In order to accommodate the many built-ins provided in Prolog implementations and not supported in our current implementation, some programs have been extended with some clauses achieving the effect of the built-ins. Examples are the predicates to achieve input/output, meta-predicates such as `setof`, `bagof`, `arg`, and `functor`. The clauses containing `assert` and `retract` have been dropped in the program containing them (i.e. Syntax error handling in the reader program).

The program `kalah` (`KA` for short) is a program which plays the game of kalah. It is taken from [44] and implements an alpha-beta search procedure. The program `Press` is an equation-solver program taken from [44] as well. We use two versions of this interesting program. The first version is the standard version (`P.1` for short) while the second version (`P.2` for short) has a procedure call repeated in the program (i.e. a procedure call is executed twice in a clause). The

two versions illustrate a fact often neglected in abstract interpretation. A more precise domain, although requiring a higher cost for the basic operations, might in fact be much more efficient since fewer elements in the domain are explored. The repetition of some procedure call in the `Press` program allows us to simulate a more precise domain (and hence to gain efficiency). The program `cuttingstock` (`CS` for short) is taken from [45]. It is a program used to generate a number of configurations representing various ways of cutting a wood board into small shelves. The program uses, in various ways, the nondeterminism of Prolog. The program `disjunctive` (`DS` for short) is taken from [46] and is the generate and test equivalent of a constraint program used to solve a disjunctive scheduling problem. This is also a program using the nondeterminism of Prolog. The program `read` (`RE` for short) is the tokeniser and reader written by R. O'keefe and D.H.D. Warren for Prolog. It is mainly a deterministic program, with mutually recursive procedures. The program `projgeom` (`PG` for short) is a program written by W. Older to solve a specific mathematical problem. The program `browse` (`BR` for short) is a program taken from the Gabriel benchmarks [47]. The program `planning` (`PL` for short) is a planning program taken from Sterling & Shapiro. The program `queens` (`QU` for short) is a simple program to solve the n-queens problem. Finally, `peephole` (`PE` for short) is a program written by S.Debray to carry out the *peephole* optimization in the SB-Prolog compiler. It is a deterministic program. We also use three classical small programs: `append`, `reverse` and `quicksort` with difference lists (`AP`, `RV` and `QS` for short). Table 1 contains a summary of the abbreviations.

`AP`	append
`KA`	an alpha beta procedure for kalah
`QU`	a queens program
`P.1`	the press program for symbolic equation solving
`P.2`	the press program with a procedure call repeated
`PE`	peephole optimization in SB-Prolog
`CS`	a nondeterministic cutting stock program
`DS`	a generate and test program for scheduling
`PG`	a program to solve a geometrical problem
`RE`	the reader and tokeniser of Prolog
`BR`	the `browse` program from the Gabriel benchmark
`PL`	a planning program
`RV`	reverse
`QS`	quicksort with difference lists

Table 1: Summary of the Abbreviations for the Prolog Programs

A number of measures are given in the table 2 to evaluate the size of the programs. These include the number of procedures, the number of clauses, the number of literals in the normalized programs, and the number of procedure calls.

6.2 Time Efficiency

In the following table 3, the running times (in seconds) as well as the ratio number of tops of the clock versus size of the program are given for each program and each domain. The size of the program is defined here as the total number of variables occurring in its rules. All the running

	KA	QU	P.1	P.2	PE	CS	DS	PG	RE	BR	PL
Number of Procedures	44	5	52	52	19	32	28	10	42	20	13
Number of Clauses	82	9	158	158	168	55	52	18	163	45	26
Number of Literals	475	38	742	744	808	336	296	93	820	207	94
Number of Procedure calls	84	8	130	131	90	57	60	17	168	37	29
Number of Variables	363	42	355	356	148	294	284	97	328	160	74

Table 2: Sizes of the Programs

times have been obtained on a Sun sparc 1 IPX, with 16MB of RAM.

The results show that Toupie is extremely fast on the benchmark programs and much more faster that any system we are aware of. Similar experiments for the `Prop` domain and the same programs have been reported in [5] for the generic abstract interpretation algorithm *GAIA* (see also [4, 2, 31]). Toupie is at least five time faster than *GAIA*, on the average. Interestingly, the implementation of [5] also uses decision diagrams for the domain and caching techniques. However, caching is used to a lower extent. Finally, it is fair noticing that *GAIA* performs a top-down analysis implying that input patterns have to be recorded, not only success patterns. This also complicates the fixpoint algorithm.

	prop		*prop+*		*small types*		*types*	
name	time	ratio	time	ratio	time	ratio	time	ratio
append	0s01	0.10	0s01	0.10	0s01	0.00	0s03	0.20
cs	0s53	0.13	0s90	0.18	0s20	0.03	0s21	0.05
disj	0s48	0.11	0s68	0.15	0s30	0.06	0s36	0.08
gabriel	0s18	0.08	0s26	0.10	0s18	0.08	0s21	0.09
kalah	0s50	0.09	0s80	0.13	0s35	0.06	0s43	0.07
peep	0s83	0.34	1s30	0.55	0s55	0.22	0s70	0.30
pg	0s11	0.07	0s16	0.11	0s11	0.08	0s15	0.11
plan	0s10	0.08	0s15	0.11	0s06	0.07	0s11	0.08
press1	0s76	0.13	1s20	0.21	0s68	0.12	1s00	0.17
press2	0s75	0.13	1s20	0.21	0s70	0.13	1s00	0.19
qsort	0s03	0.08	0s05	0.15	0s06	0.19	0s08	0.27
queens	0s03	0.07	0s06	0.10	0s03	0.05	0s03	0.05
read	0s83	0.16	1s51	0.29	2s40	0.45	4s05	0.80
reverse	0s00	0.08	0s03	0.15	0s03	0.08	0s05	0.23

Table 3: Running times and Ratios

6.3 Number of iterations

In the following table 4, the total number of iterations required to compute all the fixpoint equations as well as the ratio number of iterations versus size of the program are given for each bench and each domain. The size of the program is defined here as the total number of variables occurring in its rules.

The results show that the number of iterations is approximatively equal to half the size of the

program. This ratio does not depend on the size of the program, but slightly on the domain. The average number of iteration for a procedure ranges between 2 and 3 for recursive procedures, showing that the potentially exponential behaviour of the domain does not occur in practice. The algorithm appears to be basically linear. For most programs, the maximum number of iterations, for a procedure, is equal to 4 or 5 on `Prop` and `Prop`$^+$, and to 3 or 4 on the `type` domains showing that the complexity of the domain does not necessarily increase with the number of constants. The maximum number of iteration is 9 and is reached for `PE` on `Prop`$^+$.

	prop		*prop+*		*small types*		*types*	
name	iter.	ratio	iter.	ratio	iter.	ratio	iter.	ratio
append	8	0.80	8	0.80	8	0.80	8	0.80
cs	96	0.33	96	0.33	67	0.23	67	0.23
disj	91	0.32	91	0.32	71	0.25	71	0.25
gabriel	71	0.44	74	0.46	58	0.36	58	0.36
kalah	121	0.33	121	0.33	97	0.27	97	0.27
peep	71	0.48	77	0.52	58	0.39	59	0.40
pg	37	0.38	37	0.38	34	0.35	34	0.35
plan	41	0.55	41	0.55	39	0.53	39	0.53
press1	158	0.45	161	0.45	135	0.38	135	0.38
press2	158	0.44	161	0.45	135	0.38	135	0.38
qsort	13	0.50	14	0.54	14	0.54	14	0.54
queens	25	0.60	25	0.54	22	0.52	22	0.52
read	135	0.41	135	0.41	135	0.41	138	0.42
reverse	10	0.77	10	0.77	9	0.69	9	0.69

Table 4: Number of iterations

6.4 Size of the (Substitutions) Decision DAGs

In the following table 5, the size of the biggest decision DAG, the average size of decision DAGs and the average ratio size of decision DAG versus number of variables in the DAG are given for each bench and each domain.

Once again the results contradict the potential exponential behaviour of the domain. It can be observed that the size of decision diagrams is almost always lower than two times the number of variables. More detailed results are given at subsection 6.6 showing that even the ratio does not increase with the number of variables.

6.5 Number of Operations

The tables 6, 7, 8, 9 contain, for each domain and each bench,

- The number of operations stored.
- The number of operations reused.
- The ratio number of operations stored versus number of tops of the clock.

	prop			*prop+*			*small types*			*types*		
name	max	ave	rat	max	ave	rat	max	ave	rat	max	ave	rat
append	15	8.88	2.12	23	11.75	2.50	12	7.50	1.88	15	8.75	1.88
cs	321	24.14	2.04	454	28.94	2.27	24	4.61	1.30	24	4.70	1.30
disj	139	24.26	2.00	153	27.11	2.22	28	5.97	1.38	30	6.52	1.45
gabriel	21	8.56	1.23	23	10.26	1.43	19	6.74	1.16	19	7.09	1.19
kalah	112	12.21	1.45	159	13.60	1.54	18	5.46	0.89	20	5.66	0.89
peep	75	16.41	1.99	98	23.16	2.60	28	7.07	0.90	30	7.68	1.00
pg	22	7.92	1.35	32	9.46	1.62	20	6.74	1.26	26	7.53	1.35
plan	29	8.10	1.83	29	8.39	1.88	10	4.74	1.31	11	4.85	1.31
press1	42	11.34	1.41	63	14.19	1.78	25	7.47	1.00	29	8.33	1.15
press2	42	11.39	1.41	63	14.19	1.78	25	7.48	1.00	29	8.35	1.15
qsort	22	7.69	1.69	31	10.71	1.93	20	8.00	1.43	27	9.57	1.43
queens	14	5.76	1.56	14	5.76	1.56	5	3.73	1.27	5	3.73	1.27
read	88	13.62	1.41	134	17.87	1.74	156	22.62	2.02	231	26.85	2.37
reverse	15	8.00	2.00	20	9.50	2.30	13	7.00	1.67	15	7.67	1.78

Table 5: Decision DAGs

- The ratio number of operations stored versus size of the program.
- The number of operations performed when executing the program without cashing.
- The ratio number of operations performed versus number of tops of the clock when executing the program without cashing.
- The ratio number of operations performed versus size of the program when executing the program without cashing.

The results show that the generalized caching technique improves dramatically the efficiency of the system. The fact that operations are cached implies that many operations which are called by "top-level" operations are not reconsidered at all, even for beeing reused.

6.6 Sizes of Dags versus Numbers of Variables

The tables 10, 11, 12, 13 give for each domain and each number of variables the number of dags studied, the average size of these dags and the ratio average size versus number of variables (for all the programs of the benchmarks).

The results show that the ratio size of the decision diagrams versus number of variables does not increase with the number of variables on the benchmarks indicating that programs with a big number of variables can be analyzed as efficiently as small ones.

name	stored	reused	sto vs top	sto vs siz	perf	perf vs top	perf vs siz
append	95	70	95.00	9.50	210	210.00	21.00
cs	5843	3528	149.82	19.87	18136	279.02	61.69
disj	4220	3589	131.88	14.86	15654	274.63	55.12
gabriel	1266	1361	105.50	7.91	3665	215.59	22.91
kalah	4151	3922	129.72	11.44	14779	268.71	40.71
peep	4555	9269	91.10	30.78	32698	268.02	220.93
pg	775	561	110.71	7.99	1822	182.20	18.78
plan	551	620	91.83	7.45	1865	207.22	25.20
press1	4877	7385	103.77	13.74	23316	256.22	65.68
press2	4891	7412	101.90	13.74	23375	251.34	65.66
qsort	238	120	119.00	9.15	433	144.33	16.6
queens	263	202	87.67	6.26	568	189.33	13.52
read	5160	7074	97.36	15.73	32016	266.80	97.61
reverse	116	82	116.00	8.92	278	139.00	21.38

Table 6: Operations for Prop

name	stored	reused	sto vs top	sto vs siz	perf	perf vs top	perf vs siz
append	95	70	95.00	9.50	302	151.00	30.20
cs	7454	5418	140.64	25.35	32273	262.38	109.77
disj	5069	5972	120.69	17.85	52077	313.72	183.37
gabriel	1596	2041	99.75	9.98	5707	219.50	35.67
kalah	4936	7041	102.83	13.60	58630	305.36	161.52
peep	5666	14544	69.10	38.28	53913	247.31	364.28
pg	966	811	87.82	9.96	2560	170.67	26.39
plan	687	1050	85.88	9.28	2940	183.75	39.73
press1	6255	11813	82.30	17.62	45314	248.98	127.65
press2	6291	11882	82.78	17.67	45479	244.51	127.75
qsort	375	231	93.75	14.42	999	166.50	38.42
queens	280	288	70.00	6.67	780	156.00	18.57
read	7665	14568	81.54	23.37	126298	295.09	385.05
reverse	154	126	77.00	11.85	449	149.67	34.54

Table 7: Operations for Prop$^+$

name	stored	reused	sto vs top	sto vs siz	perf	perf vs top	perf vs siz
append	118	58	Inf	11.80	235	235.00	23.50
cs	683	814	75.89	2.32	1656	118.29	5.63
disj	1216	1694	67.56	4.28	4483	172.42	15.79
gabriel	768	1083	64.00	4.80	2649	155.82	16.56
kalah	1103	1838	52.52	3.04	26844	263.18	73.95
peep	1826	3849	55.33	12.34	15999	216.20	108.10
pg	669	447	83.62	6.90	1616	146.91	16.66
plan	257	231	51.40	3.47	569	113.80	7.69
press1	2119	4340	49.28	5.97	8973	154.71	25.28
press2	2136	4383	47.47	6.00	9033	138.97	25.37
qsort	359	150	71.80	13.81	720	144.00	27.69
queens	97	77	48.50	2.31	177	59.00	4.21
read	7372	26609	49.48	22.48	663500	281.86	2022.87
reverse	120	59	120.00	9.23	277	138.50	21.31

Table 8: Operations for Small Types

name	stored	reused	sto vs top	sto vs siz	perf	perf vs top	perf vs siz
append	150	67	75.00	15.00	287	143.50	28.70
cs	736	966	49.07	2.50	1875	117.19	6.38
disj	1479	2131	67.23	5.21	5870	172.65	20.67
gabriel	860	1217	61.43	5.38	2912	145.60	18.20
kalah	1235	2102	47.50	3.40	27400	260.95	75.48
peep	2080	4539	47.27	14.05	20958	209.58	141.61
pg	787	547	71.55	8.11	2036	145.43	20.99
plan	282	264	47.00	3.81	642	80.25	8.68
press1	2548	6013	43.19	7.18	12669	136.23	35.69
press2	2569	6086	38.92	7.22	12763	143.40	35.85
qsort	480	179	68.57	18.46	939	117.38	36.12
queens	102	92	51.00	2.43	198	66.00	4.71
read	10118	43639	38.33	30.85	916403	370.41	2793.91
reverse	147	74	49.00	11.31	366	122.00	28.15

Table 9: Operations for Types

var	1	2	3	4	5	6	7	8	9	10	11	12	13
num	116	70	100	110	64	128	86	54	64	43	18	34	22
ave	2.90	4.00	5.78	6.35	8.78	9.27	11.36	12.33	16.75	15.60	19.83	34.38	13.09
rat	2.90	2.00	1.93	1.59	1.76	1.55	1.62	1.54	1.86	1.56	1.80	2.87	1.01

var	14	15	16	17	18	19	20	22	23	24	25	27	42
num	6	31	23	13	8	4	4	8	4	15	4	2	2
ave	23.17	30.55	27.35	39.38	49.00	3.50	40.25	61.12	97.50	52.60	47.25	32.00	321.00
rat	1.65	2.04	1.71	2.32	2.72	0.18	2.01	2.78	4.24	2.19	1.89	1.19	7.64

Table 10: Sizes of DAGs vs number of Variables for Prop

var	1	2	3	4	5	6	7	8	9	10	11	12	13
num	116	70	100	116	64	131	88	54	67	44	18	35	22
ave	2.90	4.00	5.98	7.00	9.34	11.65	16.23	14.33	20.15	17.57	21.50	52.26	14.27
rat	2.90	2.00	1.99	1.75	1.87	1.94	2.32	1.79	2.24	1.76	1.95	4.35	1.10

var	14	15	16	17	18	19	20	22	23	24	25	27	42
num	6	31	23	13	8	4	4	8	4	15	4	2	2
ave	23.17	41.06	35.87	52.08	61.50	4.00	40.75	71.88	110.50	53.00	47.75	32.00	454.00
rat	1.65	2.74	2.24	3.06	3.42	0.21	2.04	3.27	4.80	2.21	1.91	1.19	10.81

Table 11: Sizes of DAGs vs number of Variables for Prop$^+$

var	1	2	3	4	5	6	7	8	9	10	11	12	13
num	115	70	99	105	57	102	71	41	42	33	13	28	22
ave	2.88	3.74	4.43	4.88	7.42	7.66	9.82	10.68	9.14	8.45	12.54	21.04	11.18
rat	2.88	1.87	1.48	1.22	1.48	1.28	1.40	1.34	1.02	0.85	1.14	1.75	0.86

var	14	15	16	17	18	19	20	22	23	24	25	27	42
num	3	21	18	8	7	3	2	6	3	7	2	1	1
ave	3.00	26.24	49.78	9.75	13.86	2.33	3.00	90.33	21.33	2.71	3.00	1.00	1.00
rat	0.21	1.75	3.11	0.57	0.77	0.12	0.15	4.11	0.93	0.11	0.12	0.04	0.02

Table 12: Sizes of DAGs vs number of Variables for Small Types

var	1	2	3	4	5	6	7	8	9	10	11	12	13
num	115	70	99	106	57	103	71	43	42	33	13	28	22
ave	2.88	3.74	4.60	5.33	8.02	8.52	10.97	12.86	9.81	9.48	13.15	25.04	12.09
rat	2.88	1.87	1.53	1.33	1.60	1.42	1.57	1.61	1.09	0.95	1.20	2.09	0.93

var	14	15	16	17	18	19	20	22	23	24	25	27	42
num	3	21	18	8	7	3	2	6	3	7	2	1	1
ave	3.00	31.14	66.11	10.25	14.43	2.33	3.00	101.00	23.00	2.71	3.00	1.00	1.00
rat	0.21	2.08	4.13	0.60	0.80	0.12	0.15	4.59	1.00	0.11	0.12	0.04	0.02

Table 13: Sizes of DAGs vs number of Variables for Types

6.7 Accuracy

It is difficult to evaluate the accuracy of our analyses with a statistical approach because they provide information about success patterns, not about program points. Information about success patterns can be reused by a post-processing algorithm to collect information about the program points (see section 8), but we have not implemented such an algorithm yet.

However a "visual" examination of the results shows that they are very accurate. Let us consider, for instance, the case of the quicksort program with difference lists which is know to be a very difficult program to analyze accurately. The analysis on `Prop` provides the result $x_1 = x_2$, expressing that the two outputs share exactly the same variables and, hence, will be grounded simultaneously. Analysis on $\texttt{Prop}^+$ returns $(x_1 = x_2) \wedge (x_1 \neq \texttt{v})$ expressing additionally that the outputs cannot be free variables. Finally, the analyses on the `Type` domains derive that both outputs are lists. Similar results are obtained for the other programs.

Nevertheless, we have performed some measurements on the accuracy of the results. In the table 14, the total number of variables occurring in the head of the rules as well as those compelled to take an interesting value (`g` for the `Prop` domains, `int` or `lst` for the `Type` domains) are given for each bench and each domain. To have a better idea of the accuracy, we should unify (abstractly) the abstract ouput patterns with relevant input patterns. Unfortunately this is not easily automatable.

name	vars	prop	prop+	small types	types
append	7	1	1	2	2
cs	104	37	37	70	70
disj	70	10	30	32	32
gabriel	75	17	17	17	17
kalah	140	47	47	57	57
peep	79	12	12	21	21
pg	43	19	19	19	19
plan	46	14	14	13	13
press1	154	37	37	39	39
press2	154	37	37	39	39
qsort	15	6	6	9	9
queens	23	7	7	11	11
read	144	38	38	24	24
reverse	9	1	1	4	4

Table 14: The number of useful variables

7 Another (upwards) Abstract Domain based on $\texttt{O}^+$: `Partition`, a Domain for Possible Sharing Analysis

7.1 Introduction

In this section, we show that $\texttt{O}^+$ can also be used to define other kinds of abstract domains for logic program analysis. We present an exemple of an *upwards closed* abstract domain for *possible sharing analysis* (see [48, 19, 16, 49, 50, 39, 51]). This domain is not as accurate as in most proposals. It

is equivalent to the sharing component of the domain `Mode` described in [39] and extensively used for experimental evaluation in [4, 31, 5, 52, 53]. In these experiments possible sharing analysis was mainly used for improving the accuracy of mode analysis, not for its own information content. In this particular context, it was shown useful enough, particularly when used in conjunction with a *reexecution strategy*. In the Toupie context, it can be used in a similar way in the context of a post processing algorithm (see section 8).

We also show that more complex domains for sharing analysis can be *expressed* by means of 0^+. Unfortunately these domains cannot be easily nor efficiently implemented in Toupie because the corresponding abstract operations cannot be expressed with simple logical operations on 0^+ formulae.

7.2 Abstract Domain

In the context of this new abstract domain, a two valued domain of constants is sufficient. Therefore we could use `Prop` [1] as an alternative notation. As the particular meaning of the constants is irrelevant in this context, we leave them unspecified. It can also be noticed that more elements could be added to the constant set *without* enhancing the expressiveness of the abstract domain. These additional constants should of course have bad consequences from the efficiency point of view. Therefore we ignore those possibilities in the following.

It seems simpler to define abstract substitutions syntactically, in this case. Abstract substitutions over D represents partitions of D, with the meaning that two program variables belong to the same class if and only if they *possibly* share a (standard) variable.

Definition 52 [Abstract Substitutions]
Let D be a finite set of program variables. An *abstract substitution over* D is any function $(D \to Const) \to \mathcal{B}$ which can be represented by the formula

$$\bigwedge_{D' \in \mathcal{P}} \left(\bigwedge_{x,x' \in D'} (x = x') \right)$$

where $\mathcal{P}$ is a partition of D. As usual, we note AS_D the set of abstract substitutions over D.

Theorem 53 [Semantic characterization of Abstract Substitutions]
Let $\beta \in (D \to Const) \to \mathcal{B}$. $\beta \in AS_D$ if and only if there exists a partition $\mathcal{P}$ of D such that for all $\alpha \in (D \to Const)$,

$$\beta\alpha = 1 \quad \text{iff} \quad \left\{ \begin{array}{c} \text{there exists a partition } \{\mathcal{P}_1, \mathcal{P}_2\} \text{ of } \mathcal{P} \text{ such} \\ \text{that } \{\bigcup_{D' \in \mathcal{P}_1} D', \bigcup_{D' \in \mathcal{P}_2} D'\} \ = \ \alpha^{-1}(Const). \end{array} \right.$$

Proof
Let $\mathcal{P}$ be a partition of D. Let $\alpha \in (D \to Const)$. Consider the formula

$$\bigwedge_{D' \in \mathcal{P}} \left(\bigwedge_{x,x' \in D'} (x = x') \right).$$

Clearly α satisfies this formula if and only if it also satisfies the condition of the theorem.
□

Definition 54 [Ordering on AS_D]
Let $\beta, \beta' \in D$. By definition,

$$\beta \leq \beta' \quad \text{iff} \quad \beta' \Rightarrow \beta.$$

Theorem 55 AS_D is a complete lattice.

Proof See [39] for the proof of a similar result for partitions.
The least upper bound and greatest lower bound operations are defined by the conjunction and the disjunction, respectively.
□

7.3 Semantics

Definition 56

Let $\theta \in PS_D$. Let $y \in codom(\theta)$. We note

$$occ(y, \theta) \stackrel{\text{def}}{=} \{x : x \in D \ \ \& \ \ y \in var(x\theta)\}.$$

Definition 57 [Abstraction of a Program Substitution]
Let $\theta \in PS_D$. The abstraction of θ is given by the following equality:

$$Abs(\theta) \ = \ \bigwedge_{y \in codom(\theta)} \left(\bigwedge_{x,x' \in occ(y,\theta)} (x = x') \right).$$

Definition 58 [Abstraction of a Set of Program Substitutions]
Let $\Theta \in CS_D$. The abstraction of Θ is given by the following equality:

$$Abs(\Theta) \ = \ \bigwedge_{\theta \in \Theta} Abs(\theta).$$

It is interesting to compare these definitions with the corresponding definition of section 4.3: symbol "$\bigvee$" has been replaced by "$\bigwedge$". This change is related to the fact that the new domain is upwards closed while the previous one is downwards closed.

Definition 59 [Concretization]
Let $\beta \in AS_D$. The concretization of β is given by the following equality:

$$Cc(\beta) \ = \ \{\theta : \forall x, x' \in D : (\exists \alpha \in (D \rightarrow Const) : \beta\alpha = 1 \ \& \ \alpha x \neq \alpha x') \Rightarrow var(x\theta) \cap var(x'\theta) = \{\} \ \}.$$

Theorem 60 (CS_D, AS_D, Cc, Abs) is a Galois-insertion.

7.4 Abstract Operations

7.4.1 Unification of Two Program Variables ($x_i = x_j$)

Specification 61

Let $D = \{x_1, \ldots, x_n\}$. Let $\beta \in AS_D$. The abstract unification of x_i and x_j $(1 \leq i, j \leq n)$, denoted `UNIF-VAR`(i, j, β), returns an abstract substitution β' for which the following holds:

$$\left.\begin{array}{l} \sigma \in mgu(x_i\theta, x_j\theta) \\ \theta \in Cc(\beta) \end{array}\right\} \Rightarrow \theta\sigma \in Cc(\beta').$$

Implementation 62

$$\beta' \stackrel{\text{def}}{=} (x_i = x_j) \wedge \beta.$$

Theorem 63 Implementation of `UNIF-VAR` is consistent.

7.4.2 Unification of a Program Variable and a Functor ($x_i = f(x_{j_1}, \ldots, x_{j_m})$)

Specification 64

Let $D = \{x_1, \ldots, x_n\}$. Let $\beta \in AS_D$. The abstract unification of x_i and $f(x_{j_1}, \ldots, x_{j_m})$ $(1 \leq i, j_1, \ldots, j_m \leq n)$, denoted `UNIF-FUNC`$(f, i, j_1, \ldots, j_m, \beta)$, returns an abstract substitution β' for which the following holds:

$$\left.\begin{array}{l} \sigma \in mgu(x_i\theta, f(x_{j_1}, \ldots, x_{j_m})\theta) \\ \theta \in Cc(\beta) \end{array}\right\} \Rightarrow \theta\sigma \in Cc(\beta').$$

Implementation 65

$$\beta' \stackrel{\text{def}}{=} (x_i = x_{j_1} \wedge \ldots \wedge x_i = x_{j_m}) \wedge \beta.$$

Theorem 66 Implementation of `UNIF-FUNC` is consistent.

7.4.3 Other operations

It is interesting to note that in this abstract domain, the union of sets of substitutions is represented by a *conjunction* of formulae. Therefore, the translation of a logic program in Toupie will be different for sharing analysis: a procedure will be translated as a *conjunction*, no longer as a disjunction.

7.4.4 Testing Impossible Sharing

As stated before the main *raison d'être* of this domain, is to infer with certainty that some program variables cannot share a free renaming variable or, in other words, that they are *independent*. Hence the question arises to know if impossible sharing will be effectively (and efficiently) checkable with Toupie. The answer to this question lies in the following theorem.

Theorem 67 [Testing Impossible Sharing]
Let $\beta \in AS_D$ and $x, x' \in D$. The two following statements are equivalent.

1. $\forall \theta \in Cc(\beta) : \ var(x\theta) \cap var(x'\theta) = \{\}$.
2. The formula $(\beta \wedge (x \neq x'))$ is satisfiable in the intended interpretation.

Proof
Due to the given definitions, the following statements are equivalent.

1. $\forall \theta \in Cc(\beta) : \ var(x\theta) \cap var(x'\theta) = \{\}$.
2. There exists $\alpha \in (D \rightarrow Const)$ such that $\beta\alpha = 1$ and $\alpha x \neq \alpha x'$.
3. $(\beta \wedge (x \neq x'))$ is true for the assignment α.

□

First, the previous results show, that Toupie can be used to test variable independence by evaluating $(\beta \wedge (x \neq x'))$. If the result is 0, there is a possible sharing. Otherwise the variables are independent. Second, this shows that the μ-calcul is not sufficient for the purpose of Abstract Interpretation: we need a more general language to program this kind of *ask* actions.

7.5 About possibly more precise domains for Sharing Analysis

In this last subsection, we show that the domain $\mathtt{O}^+$ is able to express deeper sharing information. However this potentiality seems only theoretic because the implementation of the corresponding abstract operations should require complicated and costly processing.

In the rest of this section, we discuss an abstract domain equivalent to the domain of [49].

Definition 68 [Representation of a Set of Sets of Program Variables]
Assume that $Const = \{\mathtt{sh}, \mathtt{nosh}\}$. Let $P \subseteq \mathcal{P}(D)$. We call *representation* of P, the function $\beta \in (D \rightarrow Const) \rightarrow \mathcal{B}$ corresponding to the following formula:

$$\bigvee_{D' \in P} \left(\left(\bigwedge_{x \in D'} (x = \mathtt{sh}) \right) \wedge \left(\bigwedge_{x \in D - D'} (x = \mathtt{nosh}) \right) \right).$$

Theorem 69 [Validity of the representation]
Let $P \subseteq \mathcal{P}(D)$. Let β the representation of P. For each $\alpha \in (D \rightarrow Const)$, the following holds:

$$\beta\alpha = 1 \quad \text{iff} \quad \exists D' \in P : (\forall x \in D' : \alpha x = \mathtt{sh}) \ \& \ (\forall x \in D - D' : \alpha x = \mathtt{nosh}).$$

Obviously the given representation defines a bijection between $(D \rightarrow Const) \rightarrow \mathcal{B}$ and $\mathcal{P}(D)$. This bijection can be used to adapt from [49], the definitions of the concretization and abstraction functions. The idea is that $\theta \in Cc(\beta)$ iff for all $y \in codom(\theta)$, there exists $D' \in P$ such that $D' = occ(y, \theta)$.

The Problem with this representation is that abstract unification is not easily nor efficiently implementable. Consider, for example, the simplest case of the built-in $x_i = x_j$. Roughly speaking, for a given abstract substitution β representing the set P (of sets of variables), the abstract operation has to build β' representing P' where all D' in P which can be influenced by the execution of $x_i = x_j$ are put together in all possible ways.

If we try to implement the operation as before, by adding the constraint $x_i = x_j$, we will simply eliminate all sets in P which contain only one of the two variables. This is clearly not the intended effect.

Any correct implementation should probably involve the execution of many satisfiability tests and many constraints accumulations resulting in a very inefficient operation.

8 Conclusions

This section summarizes related work, and addresses some possible extensions of our work.

8.1 Related Work

The abstract interpretation of logic programs by means of constraint logic languages has already been proposed in [54, 55] but only from a theoretical point of view. In this paper we have adressd and demonstrated the practicality of such an approach. In an as yet unpublished work, M. Codish et al use a similar approach to abstract interpretation over the domain Prop. Instead of translating logic programs to Toupie, Codish et al transform them to Datalog, and apply a simple Tp evaluation. Their approach is simpler since they do not use a constraint language. But, we believe that our approach is more powerful, since BDD's encoding permits larger interpretation domains without loss of efficiency.

In [56], P. Dart defines a groundness formula algorithm. There are two main differences between Dart's approach and ours. First, our approach is not restricted to ground formulae, and is thus more general. Second, even when our approach is applied to the domain Prop, it permits a larger class of formulae, in particular those of the form "X or Y".

In [57], J. Gallagher et al describe a method for deriving RUL programs from logic programs, where the derived program approximates the original one. However, their implementation is written in Prolog and cannot (as this writing) deal with large programs; they report in the conclusion that the computation of the fixpoint for a program of 80 Prolog clauses takes several minutes.

8.2 Summary

We have shown in this paper how constraint solving over symbolic finite domains can be used to implement efficiently a significant class of Abstract Interpretations of logic programs. The constraint language Toupie is very adequate to bottom-up Abstract Interpretation of logic programs because it implements an efficient fixpoint algorithm and because a straightforward translation of logic programs in Toupie produces the required Abstract Interpretation. Our method also uses constraints as abstract substitutions which leads to very accurate analyses because constraints memorize dependencies between variables more faithfully than most abstract domains do. The price to pay for this accuracy was shown very reasonable on the proposed benchmarks and domains. Although the size of abstract substitutions can, in theory, become exponential in the number of

variables, it is equal to two times the number of variables on the average, without any significant exception. Of course, this is only true for the proposed benchmarks and domains, but we believe that they are good representatives. As our implementation uses a careful caching technique which ensures that no operation is performed twice, the time efficiency of our system is significantly better than for other systems on this class of Abstract Interpretations.

8.3 Extensions

Some interesting extensions to our work will be addressed in the future.

First, we will address the problem of instantiating automatically the generic domain from a formal description of the meaning of constants. This will allow us to test the practicability of the system on much bigger domains.

Second, we will extend our approach to top-down analysis of logic programs (see, for instance, [8, 9, 10, 11, 13, 14, 15]). Three possible ways will be investigated and compared experimentally.

1. *Magic Sets* transformations [58, 59, 60] allow to rewrite a logic program in such a way that its bottom-up execution (or abstract interpretation) implements a top-down execution of the original one. This approach seems straightforward to adapt in our context.

2. Instead of describing output patterns, it is possible to define a transformation of flat logic procedures into a description of their *input/output relation*. This transformation is more expensive than the one presented in this paper as the number of variable will be doubled. However this approach is probably more accurate than magic sets (see [60]) and provides a complete description of the input/output behaviour of the procedures.

3. The last approach consists in exploiting the results of the bottom-up analysis to derive information about the top-down execution of the program. The principle of the method is to reexecute each clause of the flat logic program once and to use the information about success patterns to "solve" the calls by abstract unification. A similar *post-processing* algorithm is used by the system *GAIA* [2]. However *GAIA* uses input/ouput patterns not merely success patterns. Hence we will also implement a similar algorithm to exploit results produced by the second extension.

References

[1] A. Cortesi, G. Filè, and W. Winsborough. Prop revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991.

[2] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. (to appear).

[3] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. In J. Cordy, editor, *Proceedings of the IEEE fourth International Conference on Programming Languages (ICCL'92)*, Oakland, U.S.A., April 1992. IEEE Press.

[4] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic abstract interpretation algorithms for prolog: Two optimization techniques and their experimental evaluation. Technical report, Institute of Computer Science, University of Namur, Belgium, (also Brown University: Technical Report No. CS-91-67), December 1991. To appear in Software Practice and Experience,(an extended abstract appeared in Proc PLILP'92).

[5] B. Le Charlier and P. Van Hentenryck. Groundness Analysis for Prolog: Implementation and Evaluation of the Domain *prop* (extended abstract), November 1992. to appear in Proc. of the 1993 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (*PEPM'93*).

[6] C.S. Mellish. The automatic generation of mode declarations for Prolog programs. DAI Report 163, Department of Artificial Intelligence, University of Edinburgh, England, 1981.

[7] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. (To appear in ACM Transaction On Programming Languages and Systems).

[8] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[9] N.D. Jones and H. Søndergaard. A semantic-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis Horwood Limited, 1987.

[10] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.

[11] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. In *North American Conference on Logic Programming (NACLP'89)*, Cleveland, Ohio, 1989.

[12] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceeding of Fifth International Conference on Logic Programming (ICLP'88)*, pages 733–748, Seattle, Washington, August 1988. MIT Press.

[13] C.S. Mellish. Abstract interpretation of Prolog programs. In S. Abramski and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 8, pages 181–198. Ellis Horwood Limited, 1987.

[14] U. Nilsson. Systematic semantic approximations of logic programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming (PLILP'90)*, volume 456 of *Lecture Notes in Computer Science*, pages 293–306, Linköping, Sweden, August 1990. Springer-Verlag.

[15] W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(4), February 1992.

[16] M. Bruynooghe and G. Janssens. An instance of abstract interpretation: Integrating type and mode inferencing. In R.A. Kowalsky and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, pages 669–683, Seattle, Washington, August 1988. MIT Press.

[17] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, March 1988.

[18] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In D.H.D. Warren and P. Szeridi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, pages 747–762, Jerusalem, Israel, June 1990. MIT Press.

[19] K. Muthukumar and M. Hermenegildo. Determination of variable dependence information through abstract interpretation. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP'89)*, pages 166–185, Cleveland, Ohio, October 1989. MIT Press.

[20] H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proceedings of European Symposium on Programming (ESOP'86)*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338, Saarbruecken, Germany, 1986. Springer-Verlag.

[21] C. Codognet, P. Codognet, and M-M. Corsini. Abstract interpretation of concurrent logic languages. In S.K. Debray, editor, *Proceedings of the North American Conference on Logic Programming (NACLP'90)*, Austin, Texas, October 1990. MIT Press.

[22] T. Kanamori and T. Kawamura. Analysing success patterns of logic programs by abstract hybrid interpretation. Technical report, ICOT, 1987.

[23] R.A. O'Keefe. Finite fixed-point problems. In J-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP'87)*, pages 729–743, Melbourne, Australia, May 1987. MIT Press.

[24] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, 1992.

[25] R. Warren, M. Hermenegildo, and S.K. Debray. On the practicality of global flow analysis of logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, pages 684–699, Seattle, Washington, August 1988. MIT Press.

[26] M-M. Corsini, K. Musumbu, and A. Rauzy. The μ-calculus over finite domains as an abstract semantics of prolog. In M. Billaud, P. Castéran, M-M. Corsini, K. Musumbu, and A. Rauzy, editors, *WSA'92 Workshop on Static Analysis (Bordeaux)*, volume 81–82 of *Bigre*, pages 51–59. Atelier Irisa, Sept. 23–25 1992.

[27] M-M. Corsini, K. Musumbu, and A. Rauzy. First experiments with toupie, 1993. submitted to IJCAI'93.

[28] A. Rauzy. Toupie: Technical report, 1992.

[29] J.R. Burch, E.M. Clarke, and K.L. Mc Millan. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE, 1990.

[30] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE Design Automation Conference*. IEEE Press, 1990.

[31] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of prolog. In K. Apt, editor, *Proceedings of the Join International Conference and Symposium on Logic Programming (JICSLP'92)*, Washington, U.S.A., November 1992. MIT Press.

[32] D. Park. Finitiness is mu-ineffable. *Theory of Computation*, 3, 1974.

[33] J.R. Burch, E.M. Clarke, K.L. Mc Millan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE design automation conference*, pages 46–51. ACM, 1990.

[34] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE transactions on computer*, 35:677–691, 1987.

[35] B. Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report 92-22, Institute of Computer Science, University of Namur, Belgium, (also Brown University), April 1992.

[36] K. Marriott and H. Søndergaard. Abstract interpretation of logic programs : the denotational approach. In *Proceedings of the Quatro Convegno sulla Programmazione Logica*, 1990.

[37] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.

[38] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. Efficient and accurate algorithms for the abstract interpretation of Prolog programs. Technical Report 37/90, Institute of Computer Science, University of Namur, Belgium, 1990.

[39] K. Musumbu. *Interprétation Abstraite de Programmes Prolog.* PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990. In French.

[40] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

[41] P. De Boeck and B. Le Charlier. Automatic construction of Prolog primitives for type checking analysis. In M. Billaud and al., editors, *JTASPEFL'91*, pages 165–172, Bordeaux, France, 1991. Bigre 74.

[42] J. Henrard and B. Le Charlier. FOLON: An environment for Declarative Construction of Logic Programs (extended abstract). In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, Lecture Notes in Computer Science, Leuven, August 1992. Springer-Verlag.

[43] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. Technical Report CW107, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990. To appear in Journal of Logic Programming.

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

[45] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming. MIT Press, 1990.

[46] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[47] T.P. Dobry. *A High Performance Architecture for Prolog*. Kluwer Academic Publishers, 1990.

[48] P. De Boeck and B. Le Charlier. Static type analysis of Prolog procedures for ensuring correctness. In *Proceedings of Programming Language Implementation and Logic Programming (PLILP'90)*, volume 456 of *Lecture Notes in Computer Science*, pages 222–237, Linköping, Sweden, August 1990. Springer-Velag.

[49] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP'89)*, pages 154–165, Cleveland, Ohio, October 1989. MIT Press.

[50] G. Janssens. *Deriving Run Time Properties of Logic Programs by Means of Abstract Interpretation*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990.

[51] A. Mulkers. *Deriving Live Data Structures in Logic Programs by Means of Abstract Interpretation*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 1991. In preparation.

[52] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. Abstract interpretation of prolog based on oldt-resolution. Technical Report 93-10, Institute of Computer Science, University of Namur, Belgium, (also Brown University), Namur, Belgium, 1993.

[53] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The impact of Granularity in Abstract Interpretation of Prolog. Technical Report 93-11, Institute of Computer Science, University of Namur, Belgium, (also Brown University), Namur, Belgium, 1993.

[54] P. Codognet and G. Filè. Computations, abstractions and constraints in logic programs. In *Proceedings of the IEEE International Conference on Computer Languages, ICCL'92*. IEEE Press, 1992.

[55] R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constaint Logic Programming. Technical report, Dipartimento di Informatica, Università di Pisa, October 1992.

[56] P.W. Dart. On derived dependencies and connected databases. *Journal of Logic Programming*, 11(2):163–188, 1991.

[57] J.P. Gallagher and D.A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, Computer Science Department of Bristol University, March 1992.

[58] F. Bancilhon and al. Magic sets and other strange ways to implement logic programs. In *Proceedings of Fifth ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, 1986.

[59] M. Codish, D. Dams, and E. Yardeni. Bottom–up abstract interpretation of logic programs. Technical Report CS90–24, Weizmann Institute of Science, Dept of appl. maths and comp. sci., 1990. To appear in *Theoretical Computer Science*.

[60] U. Nilsson. Abstract interpretation: A kind of magic. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming (PLILP'91)*, Lecture Notes in Computer Science, Passau, Germany, August 1991. Springer-Verlag.