Groundness Analysis for Prolog: Implementation and Evaluation of the Domain Prop

(Extended Abstract)

Baudouin Le Charlier

Institut d'informatique
University of Namur
21, rue Grandgagnage
B-5000 Namur (Belgium)

Pascal Van Hentenryck

Department of Computer Science
Brown University
Box 1910
Providence RI 02912

Abstract

The domain Prop [22, 8] is a conceptually simple and elegant abstract domain to compute groundness information for Prolog programs. In particular, abstract substitutions are represented by Boolean functions built using the logical connectives $\Leftrightarrow, \vee, \wedge$. Prop has raised much theoretical interest recently but little is known about the practical accuracy and efficiency of this domain.

In this paper, we describe an implementation of Prop and we use it to instantiate a generic abstract interpretation algorithm [14, 10, 17, 15]. A key feature of the implementation is the use of ordered binary decision graphs. The implementation has been compared systematically to two other abstract domains, Mode and Pattern, from the point of view of groundness analysis.

The experimental results indicate that (1) Prop is very accurate to infer groundness information; (2) this domain is quite practical in terms of efficiency, although it is theoretically exponential (in the number of clause variables).

Introduction

Abstract interpretation of Prolog has attracted many researchers in recent years. This effort is motivated by the need of optimization in Prolog compilers to be competitive with procedural languages and the declarative nature of the language which makes it more amenable to static analysis.

ACM-PEPM'93-6/93/Copenhagen, DK

Considerable progress has been realised in this area in terms of the frameworks (e.g. [1, 2, 4, 7, 20, 21, 23, 28]), the algorithms (e.g. [2, 6, 13, 14, 26]), the abstract domains (e.g. [3, 12, 25]) and the implementations (e.g. [10, 11, 17, 15, 27]).

The domain Prop [8, 22] is a conceptually simple and elegant abstract domain to compute groundness information for Prolog programs. In particular, abstract substitutions are represented by Boolean functions built using the logical connectives $\Leftrightarrow, \vee, \wedge$. Prop has raised much theoretical interest recently but little is known about the practical accuracy and efficiency of this domain. Experimental evaluation of Prop is particularly important since Prop theoretically needs to solve a co-NP-Complete problem. However, this complexity issue may not matter much in practice because the size of the abstract substitutions is bounded since Prop would only work on the clause variables in many frameworks.

In this paper, we describe an implementation of the domain Prop and we use it to instantiate a generic abstract interpretation algorithm [10, 14, 17, 15]. A key feature of the implementation is the use of ordered binary decision graphs to provide a compact representation of many Boolean functions. The implementation (about 6000 lines of C) has been evaluated systematically and its efficiency and accuracy has been compared with two other abstract domains: the domain Mode (mode, same-value, sharing) and the domain Pattern (mode, same-value, sharing, pattern). A comparison with a reexecution algorithm [19] over the domain Mode, denoted Mode-Reex, is also given. The benchmark programs include symbolic equation-solvers, peephole optimizers, cutting-stock programs, and parsers. This is, to our knowledge, the first implementation and evaluation of Prop.

The experimental results are particularly interesting. The accuracy results for groundness inference indicate

that, on our benchmark programs, Prop compares well in accuracy with Pattern, outperforms Mode, and is exactly as accurate as Mode-Reex. The efficiency results are even more surprising. They indicate that Prop is only twice as slow as Mode in the average, takes 70% of the time taken by Pattern, and 138% of the time taken by Mode-Reex. Results on on-line analysis are also reported.

The rest of the paper is organized as follows. The first section recalls our framework for abstract interpretation of Prolog programs. The second section describes the domain Prop and illustrates the analysis on a simple example. The third section describes the main ideas behind the implementation of Prop while the last section reports and discusses the experimental results.

1 The Abstract Interpretation Framework[1]

Normalized programs The abstract semantics and the generic algorithms are defined on normalized logic programs. The use of normalized logic programs, suggested first in [4], greatly simplifies the semantics, the algorithm, and its implementation. Figure 1 presents a normalized version of the classical quicksort program using difference lists, as generated by our implementation.

Normalized programs are built from an ordered set of variables $\{x_1, \ldots, x_n, \ldots\}$. The variables are called *program variables*. A normalized program is a set of clauses

$$p(x_1, \ldots, x_n) : -l_1, \ldots, l_r$$

where $p(x_1, \ldots, x_n)$ is called the head, and $l_1, \ldots, l_r$ the body. If a clause contains m variables, these variables are necessarily $x_1, \ldots, x_m$. The literals in the body of the clause are of the form:

- $q(x_{i_1}, \ldots, x_{i_n})$ where $x_{i_1}, \ldots, x_{i_n}$ are distinct variables;
- $x_{i_1} = x_{i_2}$ where x_{i_1} and x_{i_2} are distinct variables;
- $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$ where f is a function of arity $n-1$ and $x_{i_1}, \ldots, x_{i_n}$ are distinct variables.

The second and third forms enable us to achieve unification of terms. It is not a difficult matter to translate any Prolog program into its normalized version. The advantages of normalized programs come from the fact that an (input or output) substitution for a procedure p/n is always expressed in terms of variables $x_1, \ldots, x_n$. This greatly simplifies all the traditional problems encountered with renaming.

[1] We only recall some needed aspects of the framework here. The reader is referred to [10, 14, 17, 15, 19] for a complete account.

(Concrete) Substitutions Our framework uses two kinds of substitutions. We assume another infinite set of variables, called *renaming variables* distinct from the program variables.

A *program substitution*, denoted θ, is a substitution of the form $\{x_{i_1} \rightarrow t_1, \ldots, x_{i_n} \rightarrow t_n\}$, where $x_{i_1} \ldots, x_{i_n}$ are distinct program variables and $t_1, \ldots, t_n$ are terms containing renaming variables only. $D = \{x_{i_1} \ldots, x_{i_n}\}$ is the *domain*, denoted $dom(\theta)$, of θ. The set of θ such that $dom(\theta) = D$ is denoted by PS_D and CS_D is $\mathcal{P}(PS_D)$. In our framework, *abstract substitutions* represent elements of CS_D by means of the *concretization function*.

A *standard substitution*, denoted σ, is a usual substitution using renaming variables only. The set of standard substitutions is denoted by SS. The application of a standard substitution σ to the program substitution θ, denoted $\theta\sigma$, is, by definition, $\{x_{i_1} \rightarrow t_1\sigma, \ldots, x_{i_n} \rightarrow t_n\sigma\}$.

The abstract semantics is close to the works of Winsborough [28] and Marriott and Sondergaard [21]. It is defined in terms of abstract tuples of the form $(\beta_{in}, p, \beta_{out})$ where p is a predicate symbol of arity n and β_{in}, β_{out} are abstract substitutions on variables $(x_1, \ldots, x_n)$. Informally speaking, an abstract tuple can be read as follows:

> "the execution of $p(x_1, \ldots, x_n)\theta$, where θ is a substitution satisfying the property expressed by β_{in}, will produce substitutions $\theta_1, \ldots, \theta_n$, all of which satisfy the property expressed by β_{out}."

The operations necessary to define the abstract semantics are informally defined as follows:

- UNION$(\beta_1, \ldots, \beta_n)$ where the β_i are abstract substitutions on the same domain: this operation returns an abstract substitution representing all the substitutions satisfying at least one β_i. It is used to compute the output of a procedure given the outputs for its clauses.
- AI_VAR(β) where β is an abstract substitution on $\{x_1, x_2\}$: this operation returns the abstract substitution obtained from β by unifying variables x_1, x_2. It is used for literals of the form $x_i = x_j$ in normalized programs.
- AI_FUNC(β, f) where β is an abstract substitution on $\{x_1, \ldots, x_n\}$ and f is a function symbol of arity $n-1$: this operation returns the abstract substitution obtained from β by unifying x_1 and $f(x_2, \ldots, x_n)$. It is used for literals $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$ in normalized programs.

- EXTC (c, β) where β is an abstract substitution on $\{x_1, \ldots, x_n\}$ and c is a clause containing variables $\{x_1, \ldots, x_m\}$ $(m \geq n)$: this operation returns the abstract substitution obtained by extending β to accommodate the new free variables of the clause. It is used at the entry of a clause to include the variables in the body not present in the head.

- RESTRC (c, β) where β is an abstract substitution on the clause variables $\{x_1, \ldots, x_m\}$ and $\{x_1, \ldots, x_n\}$ are the head variables of clause c $(n \leq m)$: this operation returns the abstract substitution obtained by projecting β on variables $\{x_1, \ldots, x_n\}$. It is used at the exit of a clause to restrict the substitution to the head variables only.

- RESTRG (l, β) where β is an abstract substitution on $D = \{x_1, \ldots, x_n\}$, and l is a literal $p(x_{i_1}, \ldots, x_{i_m})$ (or $x_{i_1} = x_{i_2}$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_m})$): this operation returns the abstract substitution obtained by

  1. projecting β on $\{x_{i_1}, \ldots, x_{i_m}\}$ obtaining β';
  2. expressing β' in terms of $\{x_1, \ldots, x_m\}$ by mapping x_{i_k} to x_k.

  It is used before the execution of a literal in the body of a clause. The resulting abstract substitution is expressed in terms of $\{x_1, \ldots, x_m\}$, i.e. as an input abstract substitution for p/m.

- EXTG (l, β, β') where β is an abstract substitution on $D = \{x_1, \ldots, x_n\}$, the variables of the clause where l appears, l is a literal $p(x_{i_1}, \ldots, x_{i_m})$ (or $x_{i_1} = x_{i_2}$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_m})$) with $\{x_{i_1}, \ldots, x_{i_m}\} \subseteq D$ and β' is an abstract substitution on $\{x_1, \ldots, x_m\}$ representing the result of $p(x_1, \ldots, x_m)$ β'' where β''=RESTRG (l, β): this operation returns the abstract substitution obtained by instantiating (abstractly) β to take into account the result β' of the literal l. It is used after the execution of a literal to propagate the results of the literal to all variables of the clause.

Note that normalization can induce a loss of information for abstract domains that do not include structural information.

The algorithm used in the experimental results is the so-called "prefix optimization" algorithm [10]. It is essentially our original algorithm [14, 17, 15] augmented with an advanced dependency graph to avoid recomputing clauses or prefixes of clauses that would not bring additional information. The original algorithm is a top-down algorithm computing a subset of the least fixpoint, small but sufficient to answer the query. It works at a fine granularity, i.e. it keeps multiple input/output patterns for each predicate. Both algorithms can be seen as particular implementations of Bruynooghe's operational framework [2] or, alternatively, as instantiations of a universal top-down fixpoint algorithm [16]. The generic abstract interpretation algorithm is, to our knowledge, the most efficient generic algorithm available for abstract interpretation of Prolog programs.

We also use the reexecution algorithm of [19]. This algorithm is essentially similar to the previous one, except that procedure calls and built-ins are systematically reexecuted to gain precision, exploiting the referential transparency of logic languages. This algorithm only deals with Prolog programs not using side-effects (e.g. `assert`). The reexecution is also local to a clause. Reexecution turns out to be a versatile tool to keep the domain simple and increase precision substantially.

2 The Domain `Prop`

In this section, we give an overview of the domain `Prop` and of its abstract operations. Theoretical results about this domain (in particular, consistency results) can be found in [8].

2.1 Abstract Substitutions

In `Prop`, an abstract substitution over $D = \{x_1, \ldots, x_n\}$ is represented by a Boolean function using variables from D, that is an element of $(D \rightarrow Bool) \rightarrow Bool$, where $Bool = \{false, true\}$. We only consider Boolean functions that can be represented by propositional formulas using variables from D, the truth values, and the logical connectives $\vee, \wedge, \Leftrightarrow$. $x_1 \wedge x_2$ and $x_1 \Leftrightarrow x_2 \wedge x_3$ are such formulas. In the following we denote a Boolean function by any of the propositional formulas which represent it. We also use $\bot$ to denote the abstract substitution $false$.

Definition 1 The domain `Prop` over $D = \{x_1, \ldots, x_n\}$, denoted $\texttt{Prop}_D$, is the poset of Boolean functions that can be represented by propositional formulas constructed from D, the Boolean truth values, and the logical connectives $\vee, \wedge, \Leftrightarrow$ and ordered by implication.

It is easy to see that $\texttt{Prop}_D$ is a finite lattice where the greatest lower bound is given by conjunction and the least upper bound by disjunction. Our implementation uses ordered binary decision graphs (OBDG) to represent Boolean functions since they allow many Boolean functions to have compact representations.

Definition 2 A truth assignment over D is a function $I : D \rightarrow Bool$. The value of a Boolean function f wrt a truth assignment I is denoted $I(f)$. When $I(f) = true$, we say that I satisfies f.

The basic intuition behind the domain `Prop` is that a substitution θ is abstracted by a Boolean function f over D iff, for all instances θ' of θ, the truth assignment I defined by

$$I(x_i) = \text{true iff } \theta' \text{ grounds } x_i \ (1 \leq i \leq n)$$

satisfies f. For instance, $x_1 \Leftrightarrow x_2$ abstracts the substitutions $\{x_1/y_1, x_2/y_1\}, \{x_1/a, x_2/b\}$, but not $\{x_1/a, x_2/y\}$ nor $\{x_1/y_1, x_2/y_2\}$.

Definition 3 The concretization function for $\texttt{Prop}_D$ is a function $Cc : Prop_D \rightarrow CS_D$ defined as follows:

$$Cc(f) = \{\theta \in PS_D \mid \forall \sigma \in SS : (assign\ (\theta\sigma))(f) = true\}$$

where $assign : PS_D \rightarrow D \rightarrow Bool$ is defined by $assign\ \theta\ x_i = true$ iff θ grounds x_i.

The following definitions will be used later.

Definition 4 The valuation of a function f wrt a variable x_i and a truth value b, denoted $f|_{x_i=b}$, is the function obtained by replacing x_i by b in f.

Definition 5 The dependence set D_f of a Boolean function f is the set

$$D_f = \{x_i \mid f|_{x_i=true} \neq f|_{x_i=false}\}$$

Definition 6 The normalization of a function f wrt $[x_{i_1}, \ldots, x_{i_n}]$ is the boolean function obtained by replacing simultaneously $x_{i_1}, \ldots, x_{i_n}$ by $x_1, \ldots, x_n$ in f. This normalization is denoted $norm\ f\ [x_{i_1}, \ldots, x_{i_n}]$.

Definition 7 The denormalization of a function f wrt $[x_{i_1}, \ldots, x_{i_n}]$ is the boolean function obtained by replacing simultaneously $x_1, \ldots, x_n$ by $x_{i_1}, \ldots, x_{i_n}$ in f. This denormalization is denoted $denorm\ f\ [x_{i_1}, \ldots, x_{i_n}]$.

2.2 Abstract Operations

We now define the abstract operations for the domain (see [8] for a discussion of their consistency).

Union of abstract substitutions: the union of substitutions is simply implemented using disjunction.

$$\texttt{UNION}(f_1, \ldots, f_n) \ = \ f_1 \vee \ldots \vee f_n .$$

Unification of two variables: unification of two variables adds an equivalence to the existing substitution.

$$\texttt{AI\_VAR}(f) \ = \ f \wedge (x_1 \Leftrightarrow x_2).$$

Unification of a variable and a functor: unification of a variable x_1 and a functor $t(x_2, \ldots, x_n)$ also uses equivalence.

$$\texttt{AI\_FUNC}(f, t) \ = \ f \wedge (x_1 \Leftrightarrow x_2 \wedge \ldots \wedge x_n).$$

Restriction of a clause substitution: the restriction of a clause substitution simply restricts the Boolean function to the variables appearing in the head. Let $\{x_{n+1}, \ldots, x_m\}$ be the variables appearing only in the body of c.

$$\texttt{RESTRC}(c, f) \ = \ \texttt{elim\_all}\ [x_{n+1}, \ldots, x_m]\ f$$

where

elim_all $[]\ f = f$
elim_all $[x_j, \ldots, x_m]\ f =$
  elim_all $[x_{j+1}, \ldots, x_m]\ (f|_{x_j=true} \vee f|_{x_j=false})$
  $(n < j \leq m)$

Extension of a clause substitution: the extension of a clause substitution is trivial

$$\texttt{EXTC}(c, f) \ = \ f$$

Restriction of a substitution before a literal: the restriction of a substitution for a literal amounts to eliminating all variables not appearing in the literal and normalizing the resulting function. Let $[x_{i_1}, \ldots, x_{i_n}]$ be the sequence of variable in a literal l and S the list of variables in $D_f \setminus \{x_{i_1}, \ldots, x_{i_n}\}$.

$$\texttt{RESTRG}(l, f) \ = \ \texttt{norm}\ [x_{i_1}, \ldots, x_{i_n}]\ (\texttt{elim\_all}\ S\ f).$$

Extension of a substitution after a literal: the extension of a substitution after a literal amounts to denormalizing the substitution and taking its conjunction with the clause substitution. Let $[x_{i_1}, \ldots, x_{i_n}]$ be the sequence of variable in a literal l.

$$\texttt{EXTG}(l, f, f') \ = \ f \ \wedge \ \texttt{denorm}\ [x_{i_1}, \ldots, x_{i_n}]\ f'.$$

Note that `Prop` has some interesting properties. Contrary to many domains, the `UNION` operation is optimal, i.e. it represents exactly the union of the concrete substitutions. `Prop` loses precision only in the restriction operations.

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

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

Figure 1: Quicksort on Difference Lists in Normalized Form

2.3 An Example

Figure 2 depicts the analysis of a quicksort algorithm using difference lists, whose normalized form is shown in Figure 1. Note that the first recursive call is performed with an open-ended list which makes the program difficult to analyze (i.e. many domains would lose precision). The trace of the execution shows the various abstract operations and their associated substitutions. Parts of the trace has been removed for clarity. In particular, the trace for the call to `partition` is omitted (line 16) as well as part of the first iteration of the second clause for one of the recursive calls to `qsort/3` since it returns $\bot$ and is shown during the second iteration (line 29). The Boolean functions are shown in a readable form. This is an slightly edited version of the output of our system which depicts substitutions in disjunctive normal form although the canonical form used by the algorithm is different. The abstract interpretation algorithm used to obtain the trace is the so-called prefix optimization algorithm which avoids reconsidering clauses and prefixes of clauses by keeping an advanced dependency graph [10]. The initial query has a first argument which is ground and a second argument which is a variable. This is abstracted by the formula x_1 in the trace.

`qsort/2` simply calls `qsort/3` (line 4) whose first clause returns the substitution $x_3 \wedge x_2 \wedge x_1$, indicating that all its arguments are ground (line 9). The second clause calls `qsort/3` with a substitution x_1 (line 20) and this call restarts a new subcomputation. The result of this subcomputation is $x_1 \wedge (x_2 \Leftrightarrow x_3)$ (line 43). This means that x_1 and x_2 will be ground as soon as x_3 will be ground, and reciprocally. The second recursive call simply returns $\bot$ for the first iteration (line 46) and $x_3 \wedge x_2 \wedge x_1$ for

```
1  Try clause 1
2      Exit EXTC x1
3      Exit AI-FUNC x3 ∧ x1
4      Call PRO-GOAL x3 ∧ x1
5      Try clause 1
6         Exit EXTC x3 ∧ x1
7         Exit AI-FUNC x3 ∧ x1
8         Exit AI-VAR x3 ∧ x2 ∧ x1
9         Exit RESTRC x3 ∧ x2 ∧ x1
10        Exit UNION x3 ∧ x2 ∧ x1
11     Exit clause 1
12     Try clause 2
13        Exit EXTC x3 ∧ x1
14        Exit AI-FUNC x5 ∧ x4 ∧ x3 ∧ x1
15        Call PRO-GOAL partition x2 ∧ x1
16        ...
17        Exit PRO-GOAL partition x4 ∧ x3 ∧ x2 ∧ x1
18        Exit EXTG x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x
19        Exit AI-FUNC (x9 ⇔ x8) ∧ x7 ∧ x6 ∧ x5 ∧
                    x4 ∧ x3 ∧ x1
20        Call PRO-GOAL qsort x1
21        Try clause 1
22           Exit EXTC x1
23           Exit AI-FUNC x1
24           Exit AI-VAR (x3 ⇔ x2) ∧ x1
25           Exit RESTRC (x3 ⇔ x2) ∧ x1
26           Exit UNION (x3 ⇔ x2) ∧ x1
27        Exit clause 1
28        Try clause 2
29           ...
30           Exit RESTRC ⊥
31           Exit UNION (x3 ⇔ x2) ∧ x1
32        Exit clause 2
33        Try clause 2
34           Call PRO-GOAL qsort x1
35           Exit PRO-GOAL qsort (x3 ⇔ x2) ∧ x1
36           Exit EXTG (x9 ⇔ x8 ⇔ x2) ∧ x7 ∧ x6 ∧
                       x5 ∧ x4 ∧ x3 ∧ x1
37           Call PRO-GOAL qsort x1
38           Exit PRO-GOAL (x3 ⇔ x2) ∧ x1
39           Exit EXTG (x9 ⇔ x8 ⇔ x3 ⇔ x2) ∧ x7 ∧
                       x6 ∧ x5 ∧ x4 ∧ x1
40           Exit RESTRC (x3 ⇔ x2) ∧ x1
41           Exit UNION (x3 ⇔ x2) ∧ x1
42        Exit clause 2
43        Exit PRO-GOAL (x3 ⇔ x2) ∧ x1
44        Exit EXTG (x9 ⇔ x8 ⇔ x2) ∧ x7 ∧ x6 ∧
                    x5 ∧ x4 ∧ x3 ∧ x1
45        Call PRO-GOAL qsort x3 ∧ x1
46        Exit PRO-GOAL qsort ⊥
47        Exit EXTG ⊥
48        Exit RESTRC ⊥
49        Exit UNION x3 ∧ x2 ∧ x1
50     Exit clause 2
51     Try clause 2
52        Call PRO-GOAL qsort x3 ∧ x1
53        Exit PRO-GOAL qsort x3 ∧ x2 ∧ x1
54        Exit EXTG x9 ∧ x8 ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧
                    x3 ∧ x2 ∧ x1
55        Exit RESTRC x3 ∧ x2 ∧ x1
56        Exit UNION x3 ∧ x2 ∧ x1
57     Exit clause 2
58     Exit PRO-GOAL qsort x3 ∧ x2 ∧ x1
59     Exit EXTG x3 ∧ x2 ∧ x1
60     Exit RESTRC x2 ∧ x1
61     Exit UNION x2 ∧ x1
62 Exit clause 1
```

Figure 2: Analysis of `qsort/2` using `Prop`

the second iteration (line 53). As a consequence, all arguments of `qsort/3` are ground at the exit of the clause (line 58) and `qsort/2` returns a ground argument for its second argument.

The really interesting point in this example is the substitution returned by the nested call to `qsort/3` which preserves an equivalence between the second and third argument. This enables the domain `Prop` to achieve maximal precision in this example without keeping track of functors and working only on the clause variables.

3 Implementation

Our implementation of the domain `Prop` uses ordered binary decision graphs (OBDG) as a canonical form for Boolean functions [5]. We review the main concepts here.

OBDGs require a total ordering on the variables. The ordering can have a significant impact on the size of Boolean functions. Since there is no obvious good ordering for abstract interpretation, our implementation simply uses $x_1 < x_2 < \ldots < x_n$.

The data structure underlying OBDGs is a binary tree with a number of restrictions.

Definition 8 [5] A function graph is a rooted, directed graph with vertex set V containing two types of vertices. A *nonterminal* vertex v has as attributes an index $index(v) \in \{x_1, \ldots, x_n\}$ and two children $low(v)$ and $high(v)$ from V. A *terminal* vertex v has as attribute a value $value(v) \in \{false, true\}$. Furthermore, for any nonterminal vertex v, if $low(v)$ is also nonterminal, then $index(v) > index(low(v))$. Similarly, if $high(v)$ is nonterminal, then $index(v) > index(high(v))$.

The correspondence between function graphs and Boolean functions is given by the following definitions.

Definition 9 [5] A function graph G having root vertex v denotes a function f_v defined recursively as

1. if v is a terminal vertex, then $f_v = true$ if $value(v) = true$. $f_v = false$ otherwise.
2. if v is a nonterminal vertex with $index(v) = x_i$, then f_v is the function $f_v(x_1, \ldots, x_n)$ defined as

$$x_i \wedge f_{low(v)}(x_1, \ldots, x_n) \vee \neg x_i \wedge f_{high(v)}(x_1, \ldots, x_n)$$

OBDGs are simply function graphs where redundant vertices and duplicated subgraphs have been removed.

Definition 10 [5] A function graph G is an ordered binary decision graph iff it contains no vertex v with $low(v) = high(v)$ nor does it contain distinct vertices v and v' such that the subgraph rooted by v and v' are isomorphic[2].

procedure	result	time complexity
Reduce	G is normalized	$O(\|G\|)$
Apply	$f_1 \langle op \rangle f_2$	$O(\|G_1\|\|G_2\|)$
Valuate	$f\|_{x_i=b}$	$O(\|G\|)$
Compose	$f_1\|_{x_i=f_2}$	$O(\|G_1\|^2\|G_2\|)$
Compare	true iff $f_1 = f_2$	$O(min(\|G_1\|, \|G_2\|))$
Eliminate	$f\|_{x=true} \vee f\|_{x=false}$	$O(\|G\|^2)$

Table 1: Complexity Results of the Basic Operations on Graphs

[5] describes several algorithms for reduction, restriction, and composition of OBDGs. Other algorithms (e.g. elimination, comparison) can be designed along the same principles. The main complexity results are given in Table 1. Contrary to the implementation of Bryant, our implementation uses hashtables instead of 2-dimensional arrays and avoids the sorting step of the `reduce` operation further reducing the complexity. In the complexity results, we assume that hashing takes constant time. We also note G_i the OBDG associated with a Boolean function f_i and note $|G|$ the number of vertices in the graph G. Although each operation is polynomial, it is important to realize that the size of the resulting graph can be significantly larger than the inputs of the operation. A sequence of operations can thus lead to a graph whose size is exponential in terms of the inputs. This is to be expected since Boolean satisfiability is an NP-complete problem.

4 Experimental Evaluation

In this section, we report experimental results about the efficiency and accuracy of `Prop` and compare it with two other abstract domains. Section 4.1 describes the benchmarks used in the experiments, Section 4.2 briefly reviews the domains used in our comparison, Section 4.3 discusses the accuracy results and Section 4.4 discusses the efficiency results. For space reasons, only a summary is given here. See the technical report version of this paper [18] for more complete descriptions and tables.

4.1 The Programs Tested

The programs tested were an alpha-beta procedure `kalah`, an equation-solver `Press`, a cutting-stock program `cs`, the generate and test version of a disjunctive scheduling program `Disj`, the tokeniser and reader

[2] Informally, two graphs are isomorphic if their structures and attributes match with the same order of children.

Read of R. O'Keefe and D.H.D. Warren, a program PG by W. Older to solve a specific mathematical problem, the Browse program Gabriel taken from Gabriel benchmark, a planning program Plan (PL for short), an n-queens program Queens, the peephole optimizer Peep from SB-Prolog, and the traditional concatenation and quicksort programs, say Append (with input modes (var,var,ground)) and Qsort (difference lists).

4.2 Overview of the Abstract Domains

The domain Pattern The abstract domain Pattern contains patterns (i.e. for each subterm, the main functor and a reference to its arguments are stored), sharing, same-value, and mode components. It should be related to the *depth-k abstraction* of [13], but no bound is imposed a priori to the terms depth. Program normalization does not lose precision on this domain. However, since the domain is infinite, widening operations must be used by the algorithms. The domain is fully described in [24] which contains also the proofs of monotonicity and safeness.

The domain Mode The domain Mode of [24] is a reformulation of the domain of [2]. The domain could be viewed as a simplification of the domain Pattern where the pattern information has been omitted and the sharing has been simplified to an equivalence relation. Only three modes are considered: ground, var and any. Equality constraints can only hold between program variables (and not between subterms of the terms bound to them). The same restriction applies to sharing constraints. Moreover algorithms for primitive operations are significantly different. They are much simpler and the loss of accuracy is significant.

4.3 Accuracy

In this section, we compare the three domains with respect to their precision in computing groundness information. All domains allow to compute other interesting information: *freeness* and *sharing* information is computed by Mode and Pattern as well as *pattern* information for Pattern. *Covering* information can be computed by Prop and Pattern. We only concentrate on the groundness information here.

Tables 2 and 3 compare Prop to Mode and Pattern for the input and output modes of all predicates. The first column reports the total number of arguments in the programs, the next two columns, G-Mod and G-Pro, the number of arguments inferred ground by Mode and Prop, the fourth column, B-Mod, reports the number of cases where Mode infers ground for an argument while Prop does not infer groundness, and the fifth column is just the opposite measure. Column G-Pat gives the number of arguments inferred ground by Pattern, the seventh column B-Pro the number of cases where Prop infers ground for an argument while Pattern does not infer groundness, and the last column is the opposite.

The results indicate first that Mode never infers more information than Prop and loses precision compared to Prop in almost all programs.

Contrary to Prop, Pattern keeps track of the functors and works at the level of subterms. As a consequence, the size of its substitutions is not bounded a priori. The experimental results indicate that Prop and Pattern are very close in accuracy to compute groundness information in the benchmark programs. Pattern is slightly better on the input modes since it infers more groundness on Press2, all other results being the same. The loss of precision in Prop comes from the fact that it loses track of the functors. Boolean functions on the clause variables are not enough in this case. The results on the output modes indicate that Prop is more accurate in some programs, Peep[3] and Qsort, while it loses precision on other programs, Read, Press1 and Press2. All other programs give the same results. The gain of precision in Qsort comes from the inherent loss of precision in Pattern when different clauses defining a predicate return results with different patterns. Prop avoids the drawback in this example by keeping dependencies between the variables, as explained previously in the trace. The loss of precision in Prop is always due to the fact that it only works on the clause variables and not on subterms of the terms bound to them.

Both domains infer a high percentage of ground arguments on the benchmarks. On many programs, they infer more than 80% of ground arguments.

No table is given for the comparison of Prop and Mode-Reex since all results are exactly the same. There is no way to distinguish the precision of the algorithms on our benchmark. This result is explained by the fact that reexecution in fact locally "simulates" Prop since Mode-Reex implicitly keeps all equations and propagates groundness using them. Nevertheless, Prop is better than Mode-Reex, in theory, because non local literals are not reexecuted inside a clause. It is easy to show an artificial example of a program where Prop will derive groundness of the output, but Mode-Reex will not. (See [18].)

[3]The gain in accuracy is Peep is somewhat unreal since it is due to an imprecision in one of the operations of Pattern which can be corrected easily [17, 15].

Program	Args	G-Mod	G-Pro	B-Mod	B-Pro	G-Pat	B-Pro	B-Pat
Append	3	1	1	0	0	1	0	0
CS	94	19	56	0	37	56	0	0
Disj	60	11	38	0	27	38	0	0
Gabriel	59	18	18	0	0	18	0	0
Kalah	123	35	79	0	44	79	0	0
Peep	63	22	39	0	17	39	0	0
PG	31	8	20	0	12	20	0	0
Plan	32	5	20	0	15	20	0	0
Press1	143	9	15	0	6	15	0	0
Press2	143	9	15	0	6	99	0	84
QSort	9	1	4	0	3	4	0	0
Queens	11	2	7	0	5	7	0	0
Read	122	34	34	0	0	34	0	0

Table 2: Accuracy of the Analysis on Inputs

Program	Args	G-Mod	G-Pro	B-Mod	B-Pro	G-Pat	B-Pro	B-Pat
Append	3	2	3	0	1	3	0	0
CS	94	28	94	0	66	94	0	0
Disj	60	24	60	0	36	60	0	0
Gabriel	59	22	22	0	0	22	0	0
Kalah	123	55	121	0	66	121	0	0
Peep	63	30	55	0	25	53	2	0
PG	31	8	31	0	23	31	0	0
Plan	32	7	31	0	24	31	0	0
Press1	143	26	39	0	13	40	0	1
Press2	143	26	39	0	13	140	0	101
QSort	9	1	7	0	6	6	1	0
Queens	11	2	11	0	9	11	0	0
Read	122	68	70	0	2	74	0	4

Table 3: Accuracy of the Analysis on Outputs

Program	Prop	Pattern	Mode	Mo-Re	Pr/Pa	Pr/Mo	Pr/Mo-Re	on-line
Append	0.00	0.00	0.00	0.00				0.00
CS	3.88	6.13	3.24	4.33	0.63	1.19	0.89	10.20
Disj	3.00	3.25	1.67	2.58	0.92	1.79	1.16	11.47
Gabriel	1.32	2.11	0.76	1.00	0.62	1.73	1.32	1.32
Kalah	2.65	6.73	1.48	2.11	0.39	1.79	1.25	3.12
Peep	3.33	6.17	2.49	3.21	0.53	1.33	1.03	8.37
PG	0.46	0.88	0.34	0.31	0.52	1.35	1.48	0.46
Plan	0.34	0.64	0.25	0.20	0.53	1.36	1.70	0.51
Press1	16.91	30.98	3.50	8.15	0.54	4.83	2.07	8.43
Press2	17.48	9.30	3.59	8.23	1.87	4.86	2.12	8.96
QSort	0.16	0.17	0.20	0.12	0.94	0.80	1.33	0.36
Queens	0.12	0.17	0.12	0.11	0.70	1.00	1.09	0.26
Read	5.00	16.00	3.15	4.29	0.31	1.58	1.16	5.19
Mean					0.70	1.96	1.38	

Table 4: Computation Times: Comparison of the Domains

4.4 Efficiency

We now turn to the efficiency of Prop. Efficiency results about Prop were important to obtain since, on the one hand, equivalence of Boolean functions (i.e. determining if two Boolean expressions define the same function) is a co-NP-complete problem and, on the other hand, the complexity of Prop is bounded because our algorithm only works on the variables in the clauses.

Experimental results on Prop are given in Table 4. We report the computation times in seconds on a Sun Sparc 1 workstation (Sun 4/60). The results indicate that the computation times are very reasonable. No program takes more than 18 seconds and most programs are under 5 seconds. The most time-consuming programs are Press1 and Press2 which are also the programs where Prop loses accuracy.

We now compare the efficiency results of Prop with Pattern, Mode, and Mode-Reex. Table 4 compares the efficiency of Prop, Pattern, Mode, and Mode-Reex. It indicates that Prop takes 70% of the time of Pattern in the average, is twice as slow as Mode, and requires 138% of the time of Mode-Reex. Prop is faster than Pattern on all programs but Press2 where Prop loses precision compared to Pattern. On many programs, Prop is twice as fast as Pattern and three times as fast on Read. The last result is explained by the fact that no argument is ground in the second part of the program and hence Pattern makes many more iterations due to other information that it needs to compute (i.e. patterns and sharing). Pattern is also almost twice as fast as Prop on Press2. Prop is always slower than Mode-Reex except on program CS. In general, the differences between the two programs are small; Prop is however twice as slow as Mode-Reex on the Press programs. The case of CS can easily be explained by the fact that it contains very many unifications and that Prop abstracts the information in a better way.

Column on-line of Table 4 also reports some results to demonstrate that the good behaviour of Prop is not too strongly related to the fact that many arguments are or become ground during the execution. All programs have been run without any assumption on the input patterns. The results so obtained are useful for on-line analysis[4] where a general analysis of some components is performed once and specialized for the input patterns encountered during subsequent analysis. Prop is potentially an interesting domain for on-line analysis since it is possible to obtain a specialized output pattern by taking the conjunction of the input pattern and the general output pattern. For instance, append(x_1, x_2, x_3) returns $x_3 \Leftrightarrow x_2 \wedge x_1$, and qsort$(x_1, x_2)$ returns $x_1 \Leftrightarrow x_2$ which can both be specialized optimally. The experimental results indicate that Prop behaves very well once again. A number of programs (e.g. CS, Disj) are significantly slower but the computation times remain acceptable (i.e. less than 12 seconds). Some other programs (such as the Press programs) are twice as fast than the previous version due mainly to a reduction in the number of iterations. This is because there is a loss of precision in the standard analysis of those programs. Therefore the initial abstract calls trigger more general ones which are the same as the initial calls in the on-line analysis.

Finally, Tables 5 and 6 give some results on the sizes of the abstract substitutions. We collect information at call points, i.e. before each execution of a built-in or of a procedure call. The information collected concerns the variables that may occur in the clause substitution and the size of the graph at a call point. In the tables, Calls denotes the number of call points, Var the summation of the number of variables over all call points, MaV the maximum number of variables over all points, and MeV the average number of variables. Size is the summation of all sizes of the graph (i.e. the number of nodes in the graph) over all call points, MaS the maximal size of a graph, and MeS the mean of all sizes. We also give two ratios, MaS/MaV and Size/Var, the last one giving the number of nodes used per variable. The results on the standard queries indicates that the maximum size of a graph on all programs is 123 while the theoretical maximum is 2^{42}. In the average, a graph uses 1.11 nodes per variable with a maximum of 1.32 over all programs. The ratio MaS/MaV is also never greater than 8. A couple of programs use less than 1 node per variable. The results of the on-line analysis indicate that the maximum size of a graph is 580 while, in the average, a graph uses 1.41 nodes per variable. The highest average value is 2.35 nodes per variable and the ratio MaS/MaV is also smaller than 14. The results clearly indicate the compactness of the representation and explain the behaviour of Prop.

In summary, the efficiency of Prop is somewhat intermediary between Mode and Pattern but less efficient than Mode-Reex. The results also indicate that on-line analysis is possible for Prop since the computation times remain reasonable. The representation of the graphs is also very compact and explains the good behaviour of the domain.

The above results seem to indicate that Mode-Reex is superior in practice to Prop. This conclusion should be quantified carefully for several reasons:

1. in theory, Prop is more precise than Mode-Reex;

[4] See [9] for a definition of on-line analysis and a comparison with usual *global* analysis approaches.

Program	Calls	Var	MaV	MeV	Size	MaS	MeS	MaS/MaV	MeS/MeV
Append	6	30	6	5.00	38	10	6.33	1.67	1.27
CS	307	4977	42	16.21	5226	107	17.02	2.55	1.05
Disj	269	4275	25	15.89	3514	38	13.06	1.52	0.82
Gabriel	226	1704	19	7.54	2242	31	9.92	1.63	1.32
Kalah	452	4662	19	10.31	4283	35	9.48	1.84	0.92
Peep	698	6121	15	8.77	5840	29	8.37	1.93	0.95
PG	82	751	16	9.16	820	30	10.00	1.88	1.09
Plan	79	387	8	4.90	506	13	6.41	1.62	1.31
Press1	2196	17344	17	7.90	22455	123	10.23	7.24	1.29
Press2	2248	17734	17	7.89	23097	123	10.27	7.24	1.30
QSort	31	208	9	6.71	236	18	7.61	2.00	1.13
Queens	33	217	10	6.58	235	14	7.12	1.40	1.08
Read	888	7611	22	8.57	7412	45	8.35	2.05	0.97
Mean									1.11

Table 5: Statistics on the Substitutions: Standard Analysis with Prop

Program	Calls	Var	MaV	MeV	Size	MaS	MeS	MaS/MaV	MeS/MeV
Append	7	36	6	5.14	53	14	7.57	2.33	1.47
CS	392	7125	42	18.18	14375	580	36.67	13.81	2.02
Disj	330	5838	25	17.69	13693	223	41.49	8.92	2.35
Gabriel	226	1704	19	7.54	2242	31	9.92	1.63	1.32
Kalah	449	4846	19	10.79	5093	47	11.34	2.47	1.05
Peep	1204	10937	15	9.08	13528	62	11.24	4.13	1.24
PG	82	751	16	9.16	837	30	10.21	1.88	1.11
Plan	98	520	8	5.31	668	20	6.82	2.50	1.28
Press1	1048	8804	17	8.40	13370	123	12.76	7.24	1.52
Press2	1070	8948	17	8.36	13580	123	12.69	7.24	1.52
QSort	62	409	9	6.60	525	25	8.47	2.78	1.28
Queens	62	458	10	7.39	559	25	9.02	2.50	1.22
Read	888	7611	22	8.57	7412	45	8.35	2.05	0.97
Mean									1.41

Table 6: Statistics on the Substitutions: On-line Analysis with Prop

2. `Prop` is particularly well appropriate for on-line analysis, since its results are close to optimality in this context as well;

3. `Prop` is easier to apply whenever non-logical features are used.

Note also that Boolean formulas can be used for many other domains, e.g. nonlinearity. The above results also give an idea of the practicability of using Boolean formulas for abstract domains.

5 Conclusion

In this paper, we have described an implementation and experimental evaluation of `Prop`. The implementation has resulted, at least for us, in a number of surprising results. First, the efficiency of `Prop` on our benchmarks is remarkable. `Prop` is only twice as slow as the simple abstract domain `Mode`; it takes, in the average, 70% of the time of the domain `Pattern` and 138% of the time of `Mode-Reex`. Second, the accuracy of `Prop` to compute groundness information is competitive with `Pattern`, outperforms `Mode`, and is exactly the same as `Mode-Reex`. The fact that `Prop` is competitive with `Pattern` although it does not keep track of functors is an interesting property of the expressiveness of the domain which can represent sophisticated dependencies between the variables. We have also shown that on-line analysis with `Prop` is feasible and that our implementation provides very compact representations of the graphs.

Future work will be devoted to two main issues. First abstract operations in `Prop` seem to take longer than in `Pattern`. Hence an optimization such the caching of the operations [10] should be even more interesting in this case. Second it would be interesting to combine `Prop` with a pattern component. The combination would probably achieve maximal precision on almost all practical programs but may result in a much higher computation cost since the size of the Boolean functions is no longer bounded. Conversely, as `Pattern` allows to express sophisticated equality constraints, it can reduce the number of variables in the `Prop` component. This could reduce the complexity in some cases. Therefore this issue seems worth investigating.

Acknowledgements

The comments of the reviewers helped improve the presentation. This research was partly supported by the Belgian National Incentive-Program for fundamental Research in Artificial Intelligence (Baudouin Le Charlier) and by the National Science Foundation under grant number CCR-9108032 and the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225 (Pascal Van Hentenryck).

References

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. (To appear in *ACM Transactions on Programming Languages and Systems*).

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

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

[4] Bruynooghe, M. et al. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc. 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, CA, August 1987.

[5] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] C. Codognet, P. Codognet, and J.M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

[7] A. Corsini and G. Filé. A Complete Framework for the Abstract Interpretation of Logic Programs: Theory and Applications. Research report, University of Padova, Italy, 1989.

[8] A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: Propositional formulas as abstract domain for groundness analysis. In *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, pages 322–327, 1991.

[9] A. Deutsch. A Storeless Model of Aliasing and its Abstraction Using Finite Representations of Right-Regular Equivalence Relations. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Francisco, CA, April 1992.

[10] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization

Techniques and Their Experimental Evaluation. In *Fourth International Symposium on Programming Language Implementation and Logic Programming (PLILP-92)*, Leuven (Belgium), August 1992. To Appear in *Software Practice and Experience*.

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

[12] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

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

[14] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and Its Complexity Analysis (Extended Abstract). In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[15] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*. (To Appear).

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

[17] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Francisco, CA, April 1992.

[18] B. Le Charlier and P. Van Hentenryck. Groundness Analysis for Prolog: Implementation and Evaluation of the Domain `Prop`. Technical Report CS-92-49, CS Department, Brown University, October 1992. (25 pages).

[19] B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. In *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, November 1992.

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

[21] K. Marriott and H. Sondergaard. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, pages 601–606, San Francisco, CA, 1989.

[22] K. Marriott and H. Sondergaard. Analysis of Constraint Logic Programs. In *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

[23] C. Mellish. *Abstract Interpretation of Prolog Programs*, pages 181–198. Ellis Horwood, 1987.

[24] K. Musumbu. *Interpretation Abstraite de Programmes Prolog*. PhD thesis, University of Namur (Belgium), September 1990.

[25] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information Through Abstract Interpretation. In *Proceedings of the North American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[26] R.A. O'Keefe. Finite Fixed-Point Problems. In J-L. Lassez, editor, *Fourth International Conference on Logic Programming*, pages 729–743, Melbourne, Australia, 1987.

[27] R. Warren, M. Hermedegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proc. Fifth International Conference on Logic Programming*, pages 684–699, Seattle, WA, August 1988.

[28] W.H. Winsborough. A Minimal Function Graph Semantics for Logic Programs. Technical Report TR-711, Computer Science Department, University of Wisconsin at Madison, August 1987.