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

Baudouin Le Charlier[1]
Pascal Van Hentenryck[2]

Technical Report No. CS-92-49
October 1992

[1]University of Namur, 21 rue Grandgagnage, B-5000 Namur (Belgium)
[2]Brown University Box 1910, Providence, RI 02912

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

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

Pascal Van Hentenryck
Brown University,
Box 1910, Providence, RI 02912 (USA)
Email: pvh@cs.brown.edu

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. 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 [16, 12, 18]. 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 by an order of magnitude, 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.

1 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. Considerable progress has been realised in this area in terms of the frameworks (e.g. [1, 3, 2, 7, 20, 21, 23, 30]), the algorithms (e.g. [2, 6, 15, 16, 26]), the abstract domains (e.g. [4, 14, 25]) and the implementations (e.g. [13, 18, 12, 29]).

An abstract domain which has raised much interest in recent years is the domain `Prop` proposed by Marriott and Sondergaard [22]. The domain is intended to compute groundness information in Prolog programs. It is conceptually simple and elegant since abstract substitutions are represented by Boolean functions built using the logical connectives $\Leftrightarrow, \vee, \wedge$. The domain has been further investigated in [8] and related to other abstract domains in [9].

Although the domain is properly understood from a theoretical standpoint, many practical questions regarding its efficiency and accuracy remain to be answered. In particular, the efficiency of `Prop` has been subject to much debate. On the one hand, it requires the solving of a co-NP-Complete problem (i.e. equivalence of two Boolean functions). On the other hand, in many frameworks, `Prop` would only deal with the variables appearing in the clauses whose number should be, in general, reasonably small. The accuracy of `Prop` is also an interesting problem since sophisticated dependencies between the variables can compensate the fact that `Prop` does not keep track of functors.

In this paper, we present an implementation of the domain `Prop` and we use it to instantiate a fast generic abstract interpretation algorithm for Prolog [16, 18, 12]. The key feature of our implementation of `Prop` is the use of ordered binary decision graphs [5] to represent Boolean functions in a compact way. The overall implementation (about 6000 lines of C) has been systematically evaluated and its efficiency and accuracy have 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 over the domain `mode`, denoted `Mode-Reex`, is also given. The benchmark programs used include symbolic equation-solvers, peephole optimizers, cutting-stock programs, and parsers.

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` by an order of magnitude, and is exactly as accurate that `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 and indicate that the efficiency of `Prop` remains competitive for this class of analyses. .

The rest of the paper is organized as follows. The first section describes the concrete semantics and the specification of the concrete operations. The second section describes the abstraction of the concrete semantics using 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.

2 Concrete Semantics

This section briefly reviews our semantic framemork including the concrete substitutions, the specification of the concrete operations, and the concrete semantics.

2.1 Normalized Programs and Concrete 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 $x_1, x_2, \ldots, x_i, \ldots$.

Normalized programs contain clauses with heads of the form $p(x_1, \ldots, x_n)$ where $n \geq 0$ and $p \in P_n$. Normalized clauses also contain bodies of the form $l_1, \ldots, l_n$ ($n \geq 0$) where the l_i are either procedure calls of the form $p(x_{i_1}, \ldots, x_{i_n})$ where $x_{i_1}, \ldots, x_{i_n}$ are all distinct variables and $p \in P_n$ or built-in predicates of one of the forms $x_i = x_j \;\; (i \neq j)$ or $x_i = f(x_{j_1}, \ldots, x_{j_n})$ where $i, j_1, \ldots, j_n$ are all distinct indices and $f \in F_n$.

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 $x_1, \ldots, x_n$. Normalization may induce some loss of precision in abstract domains which are sensitive to the syntactical form of the programs.

We assume the existence of another infinite set RV of *renaming variables*. We distinguish two kinds of substitutions: *program substitutions*, denoted θ, whose domain and codomain are subsets of PV and RV respectively, and *standard substitutions*, denoted σ, whose domain and codomain are subsets of RV. In the following, PS denotes the set of program substitutions, PS_D denotes the set of program substitutions the domain of which is D and SS the set of standard substitutions.

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

The definitions of *substitution composition* and *most general unifier* are the usual ones but are only used for standard substitutions. Program substitutions and standard substitutions can only be combined by *applying* a standard substitution σ *to* a program substitution θ. The result, denoted $\theta\sigma$, is defined by $dom(\theta\sigma) = dom(\theta)$ and $x(\theta\sigma) = (x\theta)\sigma$ for all $x \in dom(\theta)$. For program substitutions, the notion of *free variable* is non-standard to avoid clashes between variables during renaming. A free variable is represented by a binding to a renaming variable that appears nowhere else. We say that a substitution θ grounds a syntactic object o when $var(o\theta)$ is empty, where $var(t)$ is the set of variables in t. Let Θ be a subset of PS. Θ is *complete* if and only if, for all $\theta \in \Theta$, θ and θ' are variant implies that $\theta' \in \Theta$.

Let D be a finite subset of PV. $CS_D = \{\Theta : \forall \theta \in \Theta \; dom(\theta) = D \text{ and } \Theta \text{ is complete}\}$. CS_D is a complete lattice wrt set inclusion $\subseteq$.

2.2 Concrete Operations

Union of Sets of Substitutions Let $\Theta_1, \ldots, \Theta_n \in CS_D$.

$$\texttt{UNION}(\Theta_1, \ldots, \Theta_n) \;=\; \Theta_1 \cup \ldots \cup \Theta_n.$$

Unification of two Variables Let $D = \{x_1, x_2\}$ and $\Theta \in CS_D$.

$$\texttt{AI\_VAR}(\Theta) \;=\; \{\theta\sigma : \theta \in \Theta, \sigma \in SS, \text{ and } \sigma \in mgu(x_1\theta, x_2\theta)\}$$

Unification of a Variable and a Functor This operation unifies x_1 with $f(x_2, \ldots, x_n)$. Let $D = \{x_1, \ldots, x_n\}$, $\Theta \in CS_D$, and $f \in F_{n-1}$.

$$\texttt{AI\_FUNC}(\Theta, f) \;=\; \{\theta\sigma : \theta \in \Theta, \sigma \in SS, \text{ and } \sigma \in mgu(x_1\theta, f(x_2, \ldots, x_n)\theta)\}$$

Restriction and Extension of a Set of Substitutions for a Clause The `RESTRC` operation restricts a set of substitutions on all variables in a clause to the variables in the head. The `EXTC` operation extends a set of substitutions on variables in the head of a clause to all variables in the clause. Let c a clause, D' be the set of variables in the head and D be the set of variables of c.

$$
\begin{array}{lcl}
\texttt{RESTRC}(c,\Theta) & = & \{\theta_{/_{D'}} : \theta \in \Theta\}. \\
\texttt{EXTC}(c,\Theta) & = & \{\theta : dom(\theta) = D,\ \theta_{/_{D'}} \in \Theta, \text{ and } \forall x \in D \setminus D',\ x \text{ is free in } \theta\}.
\end{array}
$$

Restriction and Extension of a Set of Substitutions for a Literal The `RESTRG` operation expresses a set of substitutions Θ in terms of the formal parameters $x_1,\ldots,x_n$ corresponding to a literal l. The `EXTG` operation extends a set of substitutions Θ with a set of substitutions Θ' representing the result of executing a literal l on Θ. Let D be the domain of Θ, $D'' = \{x_{i_1},\ldots,x_{i_n}\}$ the set of variables appearing in l exactly in that order, and $D' = \{x_1,\ldots,x_n\}$.

$$
\begin{array}{lcl}
\texttt{RESTRG}(l,\Theta) & = & \{\theta : dom(\theta) = D', \exists \theta' \in \Theta : x_j\theta = x_{i_j}\theta' \ (1 \leq j \leq n)\}. \\
\texttt{EXTG}(l,\Theta,\Theta') & = & \{\theta\sigma :\ \theta \in \Theta,\ \sigma \in SS\ ,\ \theta'\sigma \in \Theta'\ ,\ dom(\sigma) \subseteq codom(\theta'), \\
 & & \quad (codom(\theta) \setminus codom(\theta')) \cap codom(\sigma) = \emptyset, \\
 & & \quad dom(\theta') = D'\ ,\ x_j\theta' = x_{i_j}\theta \ (1 \leq j \leq n)\}.
\end{array}
$$

2.3 Sets of Concrete Tuples

We assume in the following an underlying program P. The semantics of P is captured by a set of concrete tuples of the form $(\Theta_{in}, p, \Theta_{out})$ where Θ_{out} is intended to represent the set of output substitutions obtained by executing $p(x_1,\ldots,x_n)$ on the set of input substitutions Θ_{in} and $\Theta_{in}, \Theta_{out} \in CS_D$ with $D = \{x_1,\ldots,x_n\}$. We only consider *functional* sets *sct* of concrete tuples, implying that for all (Θ, p), there exists at most one set Θ' such that $(\Theta, p, \Theta') \in sct$. This set is denoted $sct(\Theta, p)$. $dom(sct)$ is the set of pairs (Θ, p) for which there exists an Θ' such that $(\Theta, p, \Theta') \in sct$. We call *underlying domain* UD the set of pairs (Θ, p) where p is a predicate symbol of arity n in P, $D = \{x_1,\ldots,x_n\}$ and $\Theta \in CS_D$. We denote SCT the set of all *monotonic* set of concrete tuples, i.e. those satisfying $\Theta_1 \subseteq \Theta_2 \Rightarrow sct(\Theta_1, p) \subseteq sct(\Theta_2, p)$, each time $sct(\Theta_1, p)$ and $sct(\Theta_2, p)$ are defined. We denote $SCTT$ the set of all *total*, and *continuous* sets of concrete tuples, i.e. those for which any non-decreasing chain $\Theta_1 \subseteq \Theta_2 \subseteq \ldots \subseteq \Theta_n \subseteq \ldots$ satisfies $sct(\bigcup_{i=1}^{\infty} \Theta_i, p) = \bigcup_{i=1}^{\infty}\{sct(\Theta_i, p)\}$. $SCTT$ is endowed with a structure of cpo (i.e. complete partial order) by defining (1) $\perp = \{(\Theta, p, \emptyset) : (\Theta, p) \in UD\}$ and (2) $sct \leq sct' \equiv \forall (\Theta, p) \in UD\ sct(\Theta, p) \subseteq sct'(\Theta, p)$.

2.4 A Fixpoint Concrete Semantics

The concrete semantics is defined in terms of three functions and one transformation given in Figure 1. We assume an underlying program P. p, c, g and l denotes respectively a procedure name, a clause, a sequence of literals and a literal, using only predicate symbols from P.

The transformation and functions are monotonic and continuous wrt $SCCT$ and the canonical ordering on the cartesian product $CS_D \times SCTT$ respectively. Since $SCTT$ is a cpo, the semantics of logic programs is the least fixpoint of the transformation $TSCT$, denoted $\mu(TSCT)$. This fixpoint can be shown to be consistent wrt SLD-resolution in the following sense:

$TSCT(sct) = \{(\Theta, p, \Theta') : (\Theta, p) \in UD \text{ and } \Theta' = T(\Theta, p, sct)\}.$

$T(\Theta, p, sct) = \mathtt{UNION}(\Theta_1, \ldots, \Theta_n)$
where $\Theta_i = T(\Theta, c_i, sct)$,
$c_1, \ldots, c_n$ are the clauses of p.

$T(\Theta, c, sct) = \mathtt{RESTRC}(c, \Theta')$
where $\Theta' = T(\mathtt{EXTC}(c, \Theta), g, sct)$,
g is the body of c.

$T(\Theta, <>, sct) = \Theta.$
$T(\Theta, l.g, sct) = T(\Theta_3, g, sct)$
where $\Theta_3 = \mathtt{EXTG}(l, \Theta, \Theta_2)$,

$$\Theta_2 = \begin{array}{ll} sct(\Theta_1, p) & \text{if } l \text{ is } p(\ldots) \\ \mathtt{AI\_VAR}(\Theta_1) & \text{if } l \text{ is } x_i = x_j \\ \mathtt{AI\_FUNC}(\Theta_1, f) & \text{if } l \text{ is } x_i = f(\ldots), \end{array}$$

$\Theta_1 = \mathtt{RESTRG}(l, \Theta)$.

Figure 1: The Semantic Transformation

Theorem 1 Let P be a program, $l = p(x_1, \ldots, x_n)$ be a literal, θ_{in} be a program substitution with $dom(\theta_{in}) = \{x_1, \ldots, x_n\}$, sct be $\mu(TSCT)$ and $\Theta_{in} = \{\theta \in PS : \theta \text{ and } \theta_{in} \text{ are variant}\}$. The following statement is true (we assume that SLD-Resolution uses renaming variables belonging to SS):

> if σ is an answer-substitution of SLD-resolution applied to $P \cup \{\leftarrow l\theta_{in}\}$, then there exists a substitution $\theta_{out} \in sct(\Theta_{in}, p)$ such that $\theta_{out} = \theta_{in}\sigma$.

3 The domain `Prop`

In this section, we give an overview of the domain `Prop` and present the definition of its abstract operations. The abstract semantics is simply obtained by replacing sets of concrete substitutions and concrete operations by abstract substitutions and abstract operations.

3.1 Abstract Substitutions

In `Prop`, a set of concrete substitutions 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 $\perp$ to denote the abstract substitution $false$.

Definition 2 The domain `Prop` over $D = \{x_1, \ldots, x_n\}$, denoted $\mathtt{Prop}_D$, 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 $\mathtt{Prop}_D$ 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 3 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 $\mathtt{Prop}$ 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 $\{x_1/y_1, x_2/y_1\}$, $\{x_1/a, x_2/a\}$, but not $\{x_1/a, x_2/y\}$ nor $\{x_1/y_1, x_2/y_2\}$.

Definition 4 The concretization function for $\mathtt{Prop}_D$ 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 5 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 6 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 7 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 8 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 $denorm\ f\ [x_{i_1}, \ldots, x_{i_n}]$.

3.2 Abstract Operations

We now define the abstract operations as safe approximations of the concrete operations. Recall that if $o_C : (CS_{D_1} \times \ldots \times CS_{D_n}) \rightarrow CS_D$ and $o_A : (Prop_{D_1} \times \ldots \times Prop_{D_n}) \rightarrow Prop_D$ are corresponding concrete and abstract operations, o_A safely approximate o_C if and only if

$$\forall f_1 \in Prop_{D_1} : \ldots \forall f_n \in Prop_{D_n} : o_C(Cc(f_1), \ldots, Cc(f_n)) \subseteq Cc(o_A(f_1, \ldots, f_n)).$$

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

`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.

`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.

`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.

`RESTRC`(c, f) = `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

`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 $D_f \setminus \{x_{i_1}, \ldots, x_{i_n}\}$.

`RESTRG`(l, f) = `norm` $[x_{i_1}, \ldots, x_{i_n}]$ (`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.

`EXTG`(l, f, f') = f $\wedge$ `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 2: Quicksort on Difference Lists in Normalized Form

3.3 An Example

Figure 3 depicts the analysis of a quicksort algorithm using difference lists, whose normalized form is shown in Figure 2. Note that the first recursive call is performed with an open-ended list which makes the program difficult to analyse (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 $\perp$ and is shown during the second iteration (line 29). The Boolean functions are shown in a readable form. This is an slighly 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 [12]. 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 $\perp$ for the first iteration (line 46) and $x_3 \wedge x_2 \wedge x_1$ for 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.

```
Try clause 1
   Exit EXTC x1
   Exit AI-FUNC x3 ∧ x1
   Call PRO-GOAL x3 ∧ x1
   Try clause 1
      Exit EXTC x3 ∧ x1
      Exit AI-FUNC x3 ∧ x1
      Exit AI-VAR x3 ∧ x2 ∧ x1
      Exit RESTRC x3 ∧ x2 ∧ x1
      Exit UNION x3 ∧ x2 ∧ x1
   Exit clause 1
   Try clause 2
      Exit EXTC x3 ∧ x1
      Exit AI-FUNC x5 ∧ x4 ∧ x3 ∧ x1
      Call PRO-GOAL partition x2 ∧ x1
      ...
      Exit PRO-GOAL partition x4 ∧ x3 ∧ x2 ∧ x1
      Exit EXTG x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x
      Exit AI-FUNC (x9 ⇔ x8) ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x1
      Call PRO-GOAL qsort x1
      Try clause 1
         Exit EXTC x1
         Exit AI-FUNC x1
         Exit AI-VAR (x3 ⇔ x2) ∧ x1
         Exit RESTRC (x3 ⇔ x2) ∧ x1
         Exit UNION (x3 ⇔ x2) ∧ x1
      Exit clause 1
      Try clause 2
         ...
         Exit RESTRC ⊥
         Exit UNION (x3 ⇔ x2) ∧ x1
      Exit clause 2
      Try clause 2
         Call PRO-GOAL qsort x1
         Exit PRO-GOAL qsort (x3 ⇔ x2) ∧ x1
         Exit EXTG (x9 ⇔ x8 ⇔ x2) ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x1
         Call PRO-GOAL qsort x1
         Exit PRO-GOAL (x3 ⇔ x2) ∧ x1
         Exit EXTG (x9 ⇔ x8 ⇔ x3 ⇔ x2) ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x1
         Exit RESTRC (x3 ⇔ x2) ∧ x1
         Exit UNION (x3 ⇔ x2) ∧ x1
      Exit clause 2
      Exit PRO-GOAL (x3 ⇔ x2) ∧ x1
      Exit EXTG (x9 ⇔ x8 ⇔ x2) ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x1
      Call PRO-GOAL qsort x3 ∧ x1
      Exit PRO-GOAL qsort ⊥
      Exit EXTG ⊥
      Exit RESTRC ⊥
      Exit UNION x3 ∧ x2 ∧ x1
   Exit clause 2
   Try clause 2
      Call PRO-GOAL qsort x3 ∧ x1
      Exit PRO-GOAL qsort x3 ∧ x2 ∧ x1
      Exit EXTG x9 ∧ x8 ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x2 ∧ x1
      Exit RESTRC x3 ∧ x2 ∧ x1
      Exit UNION x3 ∧ x2 ∧ x1
   Exit clause 2
   Exit PRO-GOAL qsort x3 ∧ x2 ∧ x1
   Exit EXTG x3 ∧ x2 ∧ x1
   Exit RESTRC x2 ∧ x1
   Exit UNION x2 ∧ x1
Exit clause 1
```

Figure 3: Analysis of `qsort/2` using `Prop`

4 Implementation

Our implementation of the domain `Prop` uses ordered binary decision graphs (OBDG) as a canonical form for Booleans 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 9 [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 $value(v) \in \{false, true\}$. Futhermore, 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 correspondance between function graphs and Boolean functions is given by the following definitions.

Definition 10 [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

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

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

Definition 11 [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[1].

[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.

Figure 4 gives the Pascal code of the eliminate operation for illustration purposes. The main idea behind procedure `ELIMINATE` is to perform the disjunction and the valuations simultaneaously instead of performing first the two valuations and then the disjunction. The main task

[1] Informally, two graphs are isomorphic if their structures and attributes match.

```
function ELIMINATE(f : OBDG; i : integer):OBDG;

   function ELIMINATE_STEP(f,g: OBDG; i: integer) : OBDG;
   var
      found : boolean;
      entry, idx : integer;
      u, flow, fhigh, glow, ghigh : OBDG;
   begin
      if f↑.index = i then f := f↑.low;
      if g↑.index = i then g := g↑.high;
      LOOKUP_ELIMINATE(f,g,found,entry);
      if found then ELIMINATE_STEP := hash_eliminate[entry].f
      else
         case OR(f↑.value,g↑.value) of
         UNKNOWN:
            begin
            idx := MAX(f↑.index,g↑.index);
            if f↑.index < idx then
            begin flow := f; high := f end
            else begin flow := f↑.low; fhigh := f↑.high end;
            if g↑.index < idx then
            begin glow := g; ghigh := g end
            else begin glow := g↑.low; ghigh := g↑.high; end;
            u := CREATE_NODE(UNKNOWN,idx,nil,nil);
            hash_eliminate[entry].f := u;
            u↑.low := ELIMINATE_STEP(flow,glow,i);
            u↑.high := ELIMINATE_STEP(fhigh,ghigh,i);
            ELIMINATE_STEP := u
            end;
         FALSE:
            begin
            hash_eliminate[entry].f := FALSE_NODE;
            ELIMINATE_STEP := hash_eliminate[entry].f
            end;
         TRUE:
            begin
            hash_eliminate[entry].f := TRUE_NODE;
            ELIMINATE_STEP := hash_eliminate[entry].f
            end
         end
      end;

begin
   ELIMINATE := REDUCE(ELIMINATE_STEP(f,f,i));
end;
```

Figure 4: The Implementation of the `Eliminate` Operation

procedure	result	time complexity
Reduce	G reduced in canonical form	$O(\|G\|)$
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

is achieved by procedure `ELIMINATE_STEP` whose result (a graph not normalized) is then normalized. `ELIMINATE_STEP(f,g,i)` returns a graph representing $f|_{x_i=false} \vee g|_{x_i=true}$. Procedure `ELIMINATE_STEP` carries out the valuation in its first lines by replacing `f` by `f↑.low` and `g` by `g↑.high` when the index of the node is `i`. The rest of the procedure simply performs the disjunction of the two graphs recursively. The general handling is in the `UNKNOWN` case of the `case` instruction which calls `ELIMINATE_STEP` recursively twice after some preprocessing to ensure that the indices correspond to each other. Two optimizations are used. First, a hashtable keeps the results of the elimination operations performed so that subsequent calls to `ELIMINATE_STEP` uses the results instead of recomputing. Second, a 3-value version of the logical or is applied at each step on the value fields of the nodes. This makes sure that the computation terminates early, e.g. as soon as one of the nodes is terminal and has the value 1.

5 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 5.1 describes the benchmarks used in the experiments, Section 5.2 briefly reviews the domains used in our comparison, Section 5.3 provides some background about the algorithm used, Section 5.4 discusses the accuracy results and Section 5.4.1 discusses the efficiency results.

5.1 The Programs Tested

The programs we use 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 one program containing them (i.e. Syntax error handling in the reader program).

The program `kalah` is a program which plays the game of kalah. It is taken from [27] and implements an alpha-beta search procedure. The program `press1` is a symbolic equation-solver program taken from [27] as well. `Press2` is the same program but one literal is repeated to improve precision[2]. The program `cs` is a cutting-stock program taken from [28]. 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 `Disj` is taken from [11] 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` 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 `PG` is a program written by W. Older to solve a specific mathematical problem. The program `Gabriel` is the `Browse` program taken from Gabriel benchmark. The program `Plan` (PL for short) is a planning program taken from Sterling & Shapiro. The program `Queens` is a simple program to solve the n-queens problem. `Peep` 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 the traditional concatenation and quicksort programs, say `Append` (with input modes `(var,var,ground)`) and `Qsort` (difference lists sorting the small elements first).

5.2 Overview of the Abstract Domains

5.2.1 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. The domain is more sophisticated than the sharing domains of, for instance, [14, 25] and than the mode domains of, for instance, [29, 13]. It is best viewed as an abstraction of the domain of Bruynooghe and Janssens [4] where a pattern component has been added. The domain is fully described in [24] which contains also the proofs of monotonicity and consistency.

The key concept in the representation of the substitutions in this domain is the notion of a subterm. Given a substitution on a set of variables, an abstract substitution associates with each subterm the following information:

- its *mode* (Gro, Var, Ngv (i.e. neither ground nor variable) and all combinations of them);
- its *pattern* which specifies the main functor as well as the subterms which are its arguments;
- its possible *sharing* with other subterms.

Note that the *pattern* is optional. If it is omitted, the pattern is said to be undefined.
In addition to the above information, each variable in the domain of the substitution is associated to one of the subterms. Note that this information enables to express that two arguments have the same value (and hence that two variables are bound together). To identify the subterms in an unambiguous way, an index is associated to each of them. If there are n subterms, we make use of indices $1, \ldots, n$. For instance, the substitution

$$\{X_1 \leftarrow t * v \ , \ X_2 \leftarrow v \ , \ X_3 \leftarrow Y_1 \setminus [\,]\}$$

will have 7 subterms. The association of indices to them could be for instance

$$\{(1, t * v), (2, t), (3, v), (4, v), (5, Y_1 \setminus [\,]), (6, Y_1), (7, [\,])\}.$$

[2] i.e. to simulate the effect of the reexecution strategy[19].

As mentioned previously, each index is associated with a mode taken from

$$\{\perp\ ,\ Gro\ ,\ Var\ ,\ Ngv\ ,\ Novar\ ,\ Gv\ ,\ Nogro\ ,\ Any\}.$$

In the above example, we have the following associations

$$\{(1, Gro)\ ,\ (2, Gro)\ ,\ (3, Gro)\ ,\ (4, Gro)\ ,\ (5, Ngv)\ ,\ (6, Var)\ ,\ (7, Gro)\}.$$

The pattern component (possibly) assigns to an index an expression $f(i_1, \ldots, i_n)$ where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. In our example, the pattern component will make the following associations

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

Finally the sharing component specifies which indices, not associated with a pattern, may possibly share variables. We only restrict our attention to indices with no pattern since the other patterns already express some sharing information and we do not want to introduce inconsistencies between the components. The actual sharing relation can be derived from these two components. In our particular example, the only sharing is the couple $(6, 6)$ which expresses that variable Y_1 shares a variable with itself.

5.2.2 The Domain `Mode`

The domain of [24] is a reformulation of the domain of [2]. The domain could be viewed as a simplification of the elaborate domain 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.

5.3 The Generic Abstract Interpretation Algorithm

The algorithm used in the experimental results is the so-called "prefix optimization" algorithm [12]. It is essentially our original algorithm [16, 18] 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 small subset of the least fixpoint necessary 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 [17]. 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. 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.

Program	Args	G-Mod	G-Pro	B-Mod	B-Pro	Procs	B-Mod-P	B-Pro-P
Append	3	1	1	0	0	1	0	0
CS	94	19	56	0	37	34	0	20
Disj	60	11	38	0	27	30	0	17
Gabriel	59	18	18	0	0	20	0	0
Kalah	123	35	79	0	44	44	0	36
Peep	63	22	39	0	17	19	0	9
PG	31	8	20	0	12	10	0	6
Plan	32	5	20	0	15	13	0	9
Press1	143	9	15	0	6	52	0	4
Press2	143	9	15	0	6	52	0	4
QSort	9	1	4	0	3	3	0	2
Queens	11	2	7	0	5	5	0	4
Read	122	34	34	0	0	43	0	0

Table 2: Accuracy of the Analysis on Inputs: Comparison of `Mode` and `Prop`

5.4 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 `Mode` and `Prop` 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. The last columns compare the results at the level of the procedures (instead of at the level of arguments). These two domains were compared since they both work on the variables of the clauses and do not keep track of functors in the abstract domain. The results indicate that `Prop` is more precise than `Mode`. `Mode` never infers more information than `Prop` and loses precision compared to `Prop` in almost all programs.

Tables 4 and 5 report the same comparison for `Prop` and `Pattern`. 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 are particularly interesting and indicate that `Prop` and `Pattern` are very close in accuracy to compute groundness information in the benchmark programs. `Pattern` is slighly 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 are not enough in this case. The results on the output modes indicate than `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

[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 [18].

Program	Args	G-Mod	G-Pro	B-Mod	B-Pro	Procs	B-Mod-P	B-Pro-P
Append	3	2	3	0	1	1	0	1
CS	94	28	94	0	66	34	0	30
Disj	60	24	60	0	36	30	0	20
Gabriel	59	22	22	0	0	20	0	0
Kalah	123	55	121	0	66	44	0	36
Peep	63	30	55	0	25	19	0	13
PG	31	8	31	0	23	10	0	10
Plan	32	7	31	0	24	13	0	10
Press1	143	26	39	0	13	52	0	8
Press2	143	26	39	0	13	52	0	8
QSort	9	1	7	0	6	3	0	3
Queens	11	2	11	0	9	5	0	5
Read	122	68	70	0	2	43	0	2

Table 3: Accuracy of the Analysis on Outputs: Comparison of `Mode` and `Prop`

Program	Args	G-Pro	G-Pat	B-Pro	B-Pat	Procs	B-Pro-P	B-Pat-P
Append	3	1	1	0	0	1	0	0
CS	94	56	56	0	0	34	0	0
Disj	60	38	38	0	0	30	0	0
Gabriel	59	18	18	0	0	20	0	0
Kalah	123	79	79	0	0	44	0	0
Peep	63	39	39	0	0	19	0	0
PG	31	20	20	0	0	10	0	0
Plan	32	20	20	0	0	13	0	0
Press1	143	15	15	0	0	52	0	0
Press2	143	15	99	0	84	52	0	50
QSort	9	4	4	0	0	3	0	0
Queens	11	7	7	0	0	5	0	0
Read	122	34	34	0	0	43	0	0

Table 4: Accuracy of the Analysis on Inputs: Comparison of `Prop` and `Pattern`

Program	`Args`	`G-Pro`	`G-Pat`	`B-Pro`	`B-Pat`	`Procs`	`B-Pro-P`	`B-Pat-P`
`Append`	3	3	3	0	0	1	0	0
`CS`	94	94	94	0	0	34	0	0
`Disj`	60	60	60	0	0	30	0	0
`Gabriel`	59	22	22	0	0	20	0	0
`Kalah`	123	121	121	0	0	44	0	0
`Peep`	63	55	53	2	0	19	2	0
`PG`	31	31	31	0	0	10	0	0
`Plan`	32	31	31	0	0	13	0	0
`Press1`	143	39	40	0	1	52	0	0
`Press2`	143	39	140	0	101	52	0	47
`QSort`	9	7	6	1	0	3	1	0
`Queens`	11	11	11	0	0	5	0	0
`Read`	122	70	74	0	4	43	0	4

Table 5: Accuracy of the Analysis on Outputs: Comparison of `Prop` and `Pattern`

Program	`Args`	`G-Pro`	`G-Pat`	`B-Pro`	`B-Pat`
`Append`	3	33.33	33.33	0.00	0.00
`CS`	94	59.57	59.57	0.00	0.00
`Disj`	60	63.33	63.33	0.00	0.00
`Gabriel`	59	30.50	30.50	0.00	0.00
`Kalah`	123	64.22	64.22	0.00	0.00
`Peep`	63	61.90	61.90	0.00	0.00
`PG`	31	64.51	64.51	0.00	0.00
`Plan`	32	62.50	62.50	0.00	0.00
`Press1`	143	10.48	10.48	0.00	0.00
`Press2`	143	10.48	69.23	0.00	58.74
`QSort`	9	44.44	44.44	0.00	0.00
`Queens`	11	63.63	63.63	0.00	0.00
`Read`	122	27.86	27.86	0.00	0.00

Table 6: Accuracy of the Analysis on Inputs: Comparison of `Prop` and `Pattern` in Percentage

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.

Tables 6 and 7 report the same results in percentage. They indicate that 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 surprising 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. Here is an artificial example of a program where `Prop` will derive groundness of the output, but `Mode-Reex` will not:

Program	Args	G-Pro	G-Pat	B-Pro	B-Pat
Append	3	100.00	100.00	0.00	0.00
CS	94	100.00	100.00	0.00	0.00
Disj	60	100.00	100.00	0.00	0.00
Gabriel	59	37.28	37.28	0.00	0.00
Kalah	123	98.37	98.37	0.00	0.00
Peep	63	87.30	84.12	3.17	0.00
PG	31	100.00	100.00	0.00	0.00
Plan	32	96.87	96.87	0.00	0.00
Press1	143	27.27	27.97	0.00	0.60
Press2	143	27.27	97.90	0.00	70.62
QSort	9	0.77	0.66	0.11	0.00
Queens	11	100.00	100.00	0.00	0.00
Read	122	57.37	60.65	0.00	3.27

Table 7: Accuracy of the Analysis on Outputs: Comparison of Prop and Pattern

```
q(X1) :- X1 = f( X2 , X3) , p( X1 , X2 , X3 ).
p( X1 , X2 , X3 ) :- X1 = a .
p( X1 , X2 , X3 ) :- X2 = b , X3 = c.
```

In conclusion, the experimental results indicate that Prop has a remarkable accuracy although it does not keep track of functors. It outperforms Mode and compares well with Pattern. In many cases, the results are optimal or close to optimal (i.e. all groundness information is inferred correctly). It also achieves exactly the same precision as the reexecution algorithm on mode. This positive result is due to the ability of preserving sophisticated relationships between variables in Prop.

5.4.1 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 8. We report the computation times in seconds on a Sun Sparc 1 workstation (Sun 4/60), the number of procedure iterations and the number of clause iterations, and a number of ratios. 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. Prop performs 29.29 goal iterations per second in the average. In contrast, Pattern and Mode perform 35.11 and 86.37 iterations per second indicating that the abstract operations in Prop are more expensive. This last result should be interpreted with care however since the first iteration of a goal is generally (but not always) more time consuming than the other due to the prefix optimization.

Program	Time	G-Iter	C-Iter	G-Iter/Time	C-Iter/Time
Append	0.00	2	4		
CS	3.88	50	94	12.88	24.22
Disj	3.00	45	88	15.00	29.33
Gabriel	1.32	47	114	35.60	86.36
Kalah	2.65	65	124	24.52	46.79
Peep	3.33	36	249	10.81	77.77
PG	0.46	16	31	37.78	67.38
Plan	0.34	19	41	55.88	120.58
Press1	16.91	287	866	16.97	51.21
Press2	17.48	287	878	16.41	50.22
QSort	0.16	7	15	43.75	93.75
Queens	0.12	9	17	75.00	141.66
Read	5.00	76	311	15.20	62.20
Mean				29.29	70.95

Table 8: Efficiency Results For the domain `Prop`

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

Table 9: Computation Times: Comparison of the Domains

Program	Prop	Pattern	Mode	Mode-Reex	Prop/Pattern	Prop/Mode	Prop/Mode-Reex
Append	2	3	4	4	0.66	0.50	0.50
CS	50	85	81	64	0.58	0.61	0.78
Disj	45	68	62	53	0.66	0.72	0.84
Gabriel	47	81	80	84	0.58	0.58	0.55
Kalah	65	117	91	80	0.55	0.71	0.81
Peep	36	94	75	59	0.38	0.48	0.61
PG	16	38	34	20	0.42	0.47	0.80
Plan	19	36	46	29	0.52	0.41	0.65
Press1	287	552	238	350	0.51	1.20	0.82
Press2	287	210	238	350	1.36	1.20	0.82
QSort	7	13	26	12	0.53	0.30	0.58
Queens	9	15	23	11	0.60	0.39	0.81
Read	76	209	119	115	0.36	0.63	0.66
Mean					0.59	0.63	0.76

Table 10: Goal Iteration: Comparison of the Domains

Program	Time	P-Iter	C-Iter	P-Iter/Time	C-Iter/Time
Append	0.00	3	6		
CS	10.20	55	98	5.39	9.60
Disj	11.47	63	112	5.49	9.76
Gabriel	1.32	47	114	35.60	86.36
Kalah	3.12	73	139	23.39	44.55
Peep	8.37	60	418	7.16	49.94
PG	0.46	16	31	34.78	67.38
Plan	0.51	26	49	50.98	96.07
Press1	8.43	140	422	16.60	50.05
Press2	8.96	140	428	15.62	47.76
QSort	0.36	11	26	30.55	72.22
Queens	0.26	14	27	53.84	103.84
Read	5.19	76	311	14.64	59.92
Mean				24.50	58.12

Table 11: Efficiency Results: On-line Analysis with the domain `Prop`

Program	Calls	Var	MaxV	MeanV	Size	MaxS	MeanS	MaxS/MaxV	MeanS/MeanV
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 12: Statistics on the Substitutions: Standard Analysis

We now compare the efficiency results of `Prop` with `Pattern`, `Mode`, and `Mode-Reex`. Table 9 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.

Table 10 compares the goal iterations of `Prop`, `Pattern`, `Mode` and `Mode-Reex`. It indicates that, in the average, `Prop` makes about 60% of the iterations of `Pattern`, 63% of the iterations of `Mode`, and 76% of the iterations of `Mode-Reex`. `Prop` makes less iterations than `Pattern` on all programs but `Press2`.

Table 11 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 (and/or the database). 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, $\mathtt{append(x_1,x_2,x_3)}$ returns $x_3 \Leftrightarrow x_2 \wedge x_1$, and $\mathtt{qsort(x_1,x_2)}$ 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

[4]See [10] for a definition of on-line analysis and a comparison with usual *global* analysis approaches.

Program	Calls	Var	MaxV	MeanV	Size	MaxS	MeanS	MaxS/MaxV	MeanS/MeanV
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 13: Statistics on the Substitutions: On-line Analysis

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 loss of precision happens in 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. The ratios are also interesting and indicates that the abstract operations require more time in the on-line analysis; on all programs but `Press2`, the on-line analysis has lower ratios.

Finally, Tables 12 and 13 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, `MaxV` the maximum number of variables over all points, and `MeanV` 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, `MaxS` the maximal size of a graph, and `MeanS` the mean of all sizes. We also give two ratios, `MaxS/MaxV` 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 `MaxS/MaxV` 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 `MaxS/MaxV` 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 result is rather positive since `Prop` has roughly the same precision as `Pattern` for groundness analysis. On our benchmarks, `Mode-Reex` and `Prop` are really close in accuracy and efficiency (with an advantage in efficiency for `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.

6 Conclusion

Prop is an elegant and conceptually simple abstract domain proposed by Marriott and Sondergaard to compute groundness information in Prolog programs. In particular, abstract substitutions in Prop are represented by Boolean functions using the logical connectives $\Leftrightarrow, \vee, \wedge$ only. Although Prop was well understood from a theoretical standpoint, many open practical issues remained to be answered. In particular, the efficiency of Prop has been subject to much debate since, on the one hand, it requires the solving of an co-NP-Complete problem (i.e. equivalence of two Boolean functions) but, on the other hand, many frameworks only deal with the variables appearing in the clauses whose number should be in general reasonably small.

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 [12] 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

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 Transaction 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 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.

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

[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. File. A Complete Framework for the Abstract Interpretation of Logic Programs: Theory and Applications. Research report, University of Padova, Italy, 1989.

[8] A. Cortesi, G. File, 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. Cortesi, G. File, and W. Winsborough. Comparison of abstract interpretations. In *Proc. 19th International; Colloquium on Automata, Languages and Programming (ICALP'92)*, 1992.

[10] 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 Fransisco, CA, April 1992.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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 Fransisco, CA, April 1992.

[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 Fransisco, 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*, volume Abstract Interpretation of Declarative Languages, 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] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Ma, 1986.

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

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

[30] 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.