NORTH-HOLLAND

EVALUATION OF THE DOMAIN PROP

PASCAL VAN HENTENRYCK, AGOSTINO CORTESI, AND BAUDOUIN LE CHARLIER

▷ The domain Prop [11, 30] is a conceptually simple and elegant abstract domain to compute groundness information for Prolog programs, where abstract substitutions are represented by Boolean functions. 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. The purpose of this paper is to study the performance of domain Prop. Its first contribution is to describe an implementation of the domain Prop and to use it to instantiate a generic abstract interpretation algorithm [17, 23, 27]. A key feature of the implementation is the use of ordered binary decision graphs to provide a compact representation of many Boolean functions. Its second contribution is to describe the design and implementation of a new domain, Pat(Prop), combining the domain Prop with structural information about the subterms. This new domain may significantly improve the accuracy of the domain Prop on programs manipulating difference-lists. Both implementations (resp. 6000 and 12,000 lines of C) have been evaluated systematically, and their efficiency and accuracy for groundness inference have been compared with several other abstract domains. The interest of Pat(Prop) and Prop for on-line analysis is also investigated. ◁

This paper is an extended version of [26]. Part of this research was done while A. Cortesi was visiting Brown University.

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

Received February 1993; accepted September 1994.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

0743-1066/95/$9.50
SSDI 0743-1066(94)00029-6

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 realized in this area in terms of the frameworks (e.g., [1, 2, 5, 9, 28, 29, 32, 41]), the algorithms (e.g., [2, 8, 21, 23, 36]), the abstract domains (e.g., [20, 3, 34]), and the implementations (e.g., [17, 19, 39, 27]).

An abstract domain which has raised much interest in recent years is the domain `Prop` proposed by Marriott and Sondergaard [30]. 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 [11] and related to other abstract domains in [12].

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. Note also that the study of `Prop` has a broader interest since many domains (e.g., nonlinearity) can be expressed using Boolean formulas. Hence, performance results on `Prop` may provide us with useful information on the use of Boolean functions to represent abstract substitutions.

The purpose of this paper is to study the performance of domain `Prop`. Its first contribution is to describe an implementation of the domain `Prop` and to use it to instantiate a generic abstract interpretation algorithm [17, 23, 27]. A key feature of the implementation is the use of ordered binary decision graphs to provide a compact representation of many Boolean functions. Its second contribution is to describe the design and implementation of a new domain, `Pat(Prop)`, combining the domain `Prop` with structural information about the subterms. This new domain may significantly improve the accuracy of the domain `Prop` on programs manipulating difference-lists.

Both implementations (resp. 6000 and 12,000 lines of C) have been evaluated systematically, and their efficiency and accuracy for groundness inference have been compared with several other abstract domains: the domain `Mode` (mode, same-value, sharing), the domain `Pattern` (mode, same-value, sharing, pattern), and the domains `Mode` and `Pattern` used inside reexecution algorithm [25] to improve accuracy. These last two algorithms are denoted by `Mode-reex` and by `Pat-reex` in the following. The interest of `Pat(Prop)` and `Prop` for on-line[1] analysis [15] are also investigated.

The rest of the paper is organized as follows. The first section gives an overview of the abstract interpretation framework. The second section describes the concrete

[1] On-line analysis is also known in the logic programming community as goal-independent or condensing analysis [18, 20].

semantics. The third section presents the domain `Prop`, illustrates the analysis on a simple example, and discusses the implementation of Boolean functions. The fourth section presents the new domain `Pat(Prop)` as an instantiation of a generic pattern domain presented in [13]. The fifth section reports experimental results on `Prop` and `Pat(Prop)`. The experimental results include accuracy for groundness inference, efficiency, and various statistics on the use of Boolean functions. It also discusses the use of `Prop` and `Pat(Prop)` for on-line analysis and the impact of caching on the efficiency. The last section draws the conclusions of this research and suggests directions for future work.

2. OVERVIEW OF THE ABSTRACT INTERPRETATION FRAMEWORK

In this section, we briefly review our abstract interpretation framework. A detailed theoretical presentation of the framework can be found in [22] and [33]. This last reference also contains all the correctness proofs. The framework is close to the work of Marriott and Sondergaard [28] and Winsborough [40]. It follows the traditional approach to abstract interpretation [14]. The generic abstract interpretation algorithm `GAIA` is presented in detail in [27] and more formally but more briefly in [23].

CONCRETE SEMANTICS. As is traditional in abstract interpretation, the starting point of the analysis is a collecting semantics for the programming language. Our concrete semantics is a collecting fixpoint semantics which captures the top-down execution of logic programs using a left-to-right computation rule and ignores the clause selection rule. The semantics manipulates sets of substitutions which are of the form $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ for some $n \geq 0$. Two main operations are performed on substitutions: unification and projection. The semantics associates with each of the predicate symbol p in the program a set of tuples of the form $(\Theta_{in}, p, \Theta_{out})$ which can be interpreted as follows:

> "the execution of $p(x_1, \ldots, x_n)\theta$ with $\theta \in \Theta_{in}$ produces a sequence of substitutions $\theta_1, \ldots, \theta_n, \ldots$, all of which belongs to Θ_{out}."

ABSTRACT SEMANTICS. The second step of the methodology is the abstraction of the concrete semantics. Our abstract semantics consists of abstracting a set of substitutions by a single abstract substitution, i.e., an abstract substitution represents a set of substitutions. As a consequence, the abstract semantics associates with each predicate symbol p a set of tuples of the form $(\beta_{in}, p, \beta_{out})$ which can be read informally as follows:

> "the execution of $p(x_1, \ldots, x_n)\theta$ with θ satisfying the property described by β_{in} produces a sequence of substitutions $\theta_1, \ldots, \theta_n, \ldots$, all of which satisfying the property described by β_{out}."

The abstract semantics assumes a number of operations on abstract substitutions, in particular, unification, projection, and upper bound. The first two operations are simply consistent approximations of the corresponding concrete operations. The upper bound operation is a consistent abstraction of the union of sets of substitutions.

THE FIXPOINT ALGORITHM. The last step of the methodology consists of computing the least fixpoint or a postfixpoint of the abstract semantics. The fixpoint algorithm GAIA [27] is a top-down fixpoint algorithm computing a small, but sufficient, subset of least fixpoint (or of a postfixpoint) necessary to answer a user query. The algorithm uses memoization, a dependency graph to avoid redundant computation, the abstract operations of the abstract semantics, and the ordering relation on the abstract domain. It has many similarities with PLAI [35], and can be seen either as an implementation of Bruynooghe's framework [2] or as an instance of a general fixpoint algorithm [24].

3. THE CONCRETE SEMANTICS

The purpose of this section is to present the concrete semantics which is the basis of the analysis. It sets up the terminology necessary to specify the abstract operations, and helps in understanding the experimental results by presenting the concrete transformation which is then abstracted. The concrete semantics is a collecting fixpoint semantics. It is defined on normalized programs [2] which are defined in Section 3.1. The main semantic objects manipulated are sets of substitutions which are defined in Section 3.2. The main operations on sets of substitutions are described informally in Section 3.3. They will be specified formally in Section 4.2, together with their abstractions. The concrete semantics is described in Section 3.4.

3.1. Normalized Programs

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 $x_i = g(x_{j_1}, \ldots, x_{j_n})$ where $i, j_1, \ldots, j_n$ are all distinct indices and $g \in F_n$.

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, as discussed later.

3.2. Concrete Domain

The concrete semantics is defined in terms of sets of concrete substitutions. We provide the necessary notions here.

We assume the existence of another infinite set RV of *renaming variables*. We distinguish two kinds of substitutions: *program substitutions*, denoted by θ, whose domain and codomain are subsets of PV and RV, respectively, and *standard substitutions*, denoted by σ, whose domain and codomain are subsets of RV. The domain of a substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, denoted by $dom(\theta)$, is simply $\{x_1, \ldots, x_n\}$. 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 denotes 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 by $\theta\sigma$, 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 nonstandard to avoid clashes between variables during renaming. A free variable is represented by a binding to a renaming variable that appears nowhere else. As a consequence, the domain of a substitution is invariant under renaming.

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[2] implies that $\theta' \in \Theta$. Let D be a finite subset of PV. $CS_D = \{\Theta : \forall\theta \in \Theta\ dom(\theta) = D$ and Θ is complete$\}$. CS_D is a complete lattice w.r.t. set inclusion $\subseteq$.

3.3. Concrete Operations

We now provide an informal presentation of the concrete operations. They are specified formally together with their abstractions in Section 4.2. The concrete semantics uses the following operations.

- UNION$(\Theta_1, \ldots, \Theta_n)$ where the Θ_i are a set of substitutions on the same domain: this operation returns sets of substitutions which is the union of all Θ_i. It is used to compute the output of a procedure given the outputs for its clauses.
- AI_VAR(Θ) where Θ is a set of substitutions with domain $\{x_1, x_2\}$: this operation returns the set of substitutions obtained by unifying the terms bound to x_1 and x_2 in each substitution of Θ. It is used for literals of the form $x_i = x_j$ in normalized programs.
- AI_FUNC(Θ, g) where Θ is a set of substitutions with domain $\{x_1, \ldots, x_n\}$ and g is a function symbol of arity $n-1$: this operation returns the set of substitutions obtained by unifying in each substitution $\theta \in \Theta$ the terms t_1 and $g(t_2, \ldots, t_n)$ where t_i is the term bound to x_i in θ. It is used for literals $x_{i_1} = g(x_{i_2}, \ldots, x_{i_n})$ in normalized programs.
- EXTC(c, Θ) where Θ is a set of substitutions with domain $\{x_1, \ldots, x_n\}$ and c is a clause containing variables $\{x_1, \ldots, x_m\}$ $(m \geq n)$: this operation returns a set of substitutions obtained by extending each substitution in Θ 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, Θ) where Θ is a set of substitutions on the variables $\{x_1, \ldots, x_m\}$ and $\{x_1, \ldots, x_n\}$ are the head variables of clause c $(n \leq m)$: this operation

[2]This implies that there exist $\sigma, sigma' \in SS$ such that $\theta' = \theta\sigma$ and $\theta = \theta'\sigma'$.

returns the set of substitutions obtained by projecting each substitution in Θ 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, Θ) where Θ is a set of substitutions on domain $D = \{x_1, \ldots, x_n\}$, and l is a literal $p(x_{i_1}, \ldots, x_{i_m})$ (or $x_{i_1} = x_{i_2}$ or $x_{i_1} = g(x_{i_2}, \ldots, x_{i_m})$): this operation returns the set of substitutions obtained by

  1. projecting each substitution $\theta \in \Theta$ on $\{x_{i_1}, \ldots, x_{i_m}\}$ obtaining θ';
  2. expressing θ' 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 substitutions in the resulting set are expressed in terms of $\{x_1, \ldots, x_m\}$, i.e., as substitutions for p/m.
- EXTG(l, Θ, Θ') where Θ is a set of substitutions 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 $x_{i_1} = g(x_{i_2}, \ldots, x_{i_m})$) with $\{x_{i_1}, \ldots, x_{i_m}\} \subseteq D$, and Θ' is a set of substitutions on $\{x_1, \ldots, x_m\}$ representing the result of $p(x_1, \ldots, x_m)$ Θ'' where Θ''=RESTRG(l, Θ): this operation returns the set of substitutions obtained by instantiating each substitution $\theta \in \Theta$ to take into account each resulting substitution $\theta' \in \Theta'$ 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.

3.4. Concrete Semantics

We are now in a position to define the concrete semantics.

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 by $sct(\Theta, p)$. $dom(sct)$ 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 by SCT the set of all *monotonic* sets 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 by $SCTT$ the set of all *total* sets of concrete tuples. $SCTT$ is endowed with a structure of cpo (i.e., complete partial order) by defining

- $\bot = \{(\Theta, p, \emptyset) : (\Theta, p) \in UD\}$;
- $sct \leq sct' \equiv \forall (\Theta, p) \in UD\ sct(\Theta, p) \subseteq sct'(\Theta, p)$.

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

Informally speaking, the first rule of T defines a procedure execution, the second rule defines a clause execution, while the third rule defines a clause suffix execution. A procedure is executed by executing its clauses and taking the union of their results. A clause is executed by extending its substitutions to take into account

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

$T(\Theta, p, sct) = \mathtt{UNION}(\Theta_1, \ldots, \Theta_n)$
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 $\Theta_3 = \mathtt{EXTG}(l, \Theta, \Theta_2)$,

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

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

FIGURE 1. The semantic transformation.

the local variables, executing its body, and projecting its local variables. A suffix is executed by restricting the substitutions to the variables of the first goal, applying the goal, extending the result on all the variables of the clause, and executing the rest of the suffix. The execution of a goal is either a unification or a lookup in the set of results (procedure call).

CONCRETE SEMANTICS. The transformation and functions are monotonic and continuous w.r.t. $SCTT$ and the canonical ordering on the Cartesian product $CS_D \times SCTT$, respectively. Since $SCTT$ is a cpo, the concrete semantics of a program is defined as the least fixpoint of the transformation $TSCT$, denoted $\mu(TSCT)$. This fixpoint can be shown to be consistent w.r.t. SLD-resolution in the following sense:

Theorem 3.1. 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 $\Theta_{in} = \{\theta \in PS : \theta$ and θ_{in} are variant$\}$. The following statement is true (we assume that SLD-refutation uses renaming variables belonging to SS):

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

4. THE DOMAIN PROP

We now show how the concrete semantics can be abstracted using the domain `Prop`. Intuitively, the abstraction consists of replacing the concrete domain (e.g., sets of substitutions) by an abstract domain (e.g., Boolean formula), and of defining abstract operations which are consistent approximations of each concrete operation. Section 4.1 describes the abstract domain. Section 4.2 describes the abstract operations as consistent approximations of the concrete operations. Section 4.3 sketches the abstract semantics. Section 4.5 describes some implementation details. Section 4.4 gives an example of analysis.

4.1. Abstract Domain

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 $Bool = \{false, true\}$. 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 4.1. 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 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. See Section 4.5 for more discussion of OBDD.

Definition 4.2. A truth assignment over D is a function $I : D \rightarrow Bool$. The value of a Boolean function f w.r.t. 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 $\{x_1/y_1, x_2/y_1\}$, $\{x_1/a, x_2/a\}$, but not $\{x_1/a, x_2/y\}$ nor $\{x_1/y_1, x_2/y_2\}$.

Definition 4.3. 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 4.4. The valuation of a function f w.r.t. 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 4.5. The dependence set D_f of a Boolean function f is the set

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

Definition 4.6. The normalization of a function f w.r.t. $[x_{i_1}, \ldots, x_{i_n}]$ 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 4.7. The denormalization of a function f w.r.t. $[x_{i_1}, \ldots, x_{i_n}]$ 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* $f\ [x_{i_1}, \ldots, x_{i_n}]$.

4.2. Abstract Operations

We now describe the abstract operations as consistent approximations of the concrete operations. Recall that if $o_c : (CS_{D_1} \times \cdots \times CS_{D_n}) \to CS_D$ and $o_a : (Prop_{D_1} \times \cdots \times Prop_{D_n}) \to Prop_D$ are corresponding concrete and abstract operations, o_a is a consistent approximation of o_c if and only if

$$\forall f_1 \in Prop_{D_1} : \cdots \forall f_n \in Prop_{D_n} : o_c(Cc(f_1), \ldots, Cc(f_n)) \subseteq Cc(o_a(f_1, \ldots, f_n)).$$

For each operation, we give both its concrete version and its abstract version and overload the names of the operations by dropping the subscripts. The informal presentation of the operation was given in Section 3.3, but we repeat some of them here for clarity.

UNION (UPPER BOUND). Operation `UNION` is used to collect the results of the clauses of a procedure to define the result of the procedure. Its concrete version is specified as follows, assuming that $\Theta_1, \ldots, \Theta_n \in CS_D$:

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

Its abstract version is obtained by taking the disjunction of the Boolean formula:

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

It is important to note that this abstraction is very precise and almost never loses precision in practice. It is not optimal, however, as the following example (adapted from one of the reviews) shows. Let $\theta = \{x_1 \leftarrow y, x_2 \leftarrow z\}$ where $y, z \in RV$ and y and z are distinct. We have

$$\theta \notin Cc(x_1)\ \&\ \theta \notin Cc(x_1 \Rightarrow x_2).$$

Since $x_1 \vee (x_1 \Rightarrow x_2)$ is logically equivalent to *true*, we also have

$$\theta \in Cc(x_1 \vee (x_1 \Rightarrow x_2)) = Cc(true).$$

A practical program leading to the above example is as follows:

```
p(X1,X2) :- X1=g(X3,X2).
p(X1,X2) :- X1=a.
```

The loss of precision can be removed by using sets of Boolean functions or an algorithm based on OLDT-resolution. It remains to see if this would yield a practical analysis.

UNIFICATION OF TWO VARIABLES Operation AI_VAR performs the unification of the terms bound to variables x_1, x_2. Its concrete version is specified as follows, assuming that $D = \{x_1, x_2\}$ and $\Theta \in CS_D$:

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

Its abstract version is defined by adding an equivalence between x_1 and x_2 in the input abstract substitution:

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

UNIFICATION OF A VARIABLE AND A FUNCTOR Operation AI_FUNC unifies the terms t_1 with $g(t_2, \ldots, t_n)$, where t_i are the terms bound x_i in the substitutions. Its concrete version is specified as follows, assuming that $D = \{x_1, \ldots, x_n\}$, $\Theta \in CS_D$, and $g \in F_{n-1}$:

$$\texttt{AI\_FUNC}(\Theta, g) = \{\theta\sigma : \theta \in \Theta \ \&\ \sigma \in SS \ \&\ \sigma \in mgu(x_1\theta, g(x_2, \ldots, x_n)\theta)\}.$$

Its abstract version also adds an equivalence which is slightly more complex than in the previous operation.

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

RESTRICTION OF A CLAUSE SUBSTITUTION. Operation RESTRC restricts a set of substitutions expressed on all the clause variables to a substitution expressed on the head variables. It is used at the end of a clause execution. Its concrete version is specified as follows, assuming that c is a clause, D' is the set of variables in the head, and D is the set of variables of c:

$$\texttt{RESTRC}(c, \Theta) = \{\theta_{/_{D'}} : \theta \in \Theta\}.$$

The abstract version 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

$$\begin{aligned}
&\texttt{elim_all}\ [\,]\ f = f \\
&\texttt{elim_all}\ [x_j, \ldots, x_m]\ f = \\
&\quad \texttt{elim_all}\ [x_{j+1}, \ldots, x_m]\ (f|_{x_j=true} \vee f|_{x_j=false}) \qquad (n < j \leq m).
\end{aligned}$$

Note that this operation is one of the operations where precision can be lost in practice.

EXTENSION OF A CLAUSE SUBSTITUTION. Operation EXTC extends a set of substitutions expressed on variables in the head of a clause to a set of substituions expressed on all variables in the clause. It is used at the beginning of a clause execution. Its concrete version is specified as follows, assuming that c is a clause, D' is the set of variables in the head, and D is the set of variables of c.

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

The abstract version is trivial:

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

RESTRICTION OF A SUBSTITUTION BEFORE A LITERAL. Operation RESTRG is used before executing a literal l. It expresses a set of substitutions Θ in terms of the formal parameters $x_1, \ldots, x_n$ of the literal l by projecting the variables not appearing in l and mapping the remaining variables $x_{i_1}, \ldots, x_{i_n}$ to the formal parameters $x_1, \ldots, x_n$. Its concrete version can be specified as follows:

$$\texttt{RESTRG}(l, \Theta) = \{\theta : dom(\theta) = D' \ \& \ \exists \theta' \in \Theta : x_j\theta = x_{i_j}\theta' \qquad (1 \leq j \leq n)\}.$$

Its abstract version amounts to eliminating from the Boolean function all variables not appearing in the literal and normalizing the resulting function. Let S be the list of variables in $D_f \setminus \{x_{i_1}, \ldots, x_{i_n}\}$:

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

Note that, once again, this operation may lose precision.

EXTENSION OF A SUBSTITUTION AFTER A LITERAL. Operation EXTG is used after the execution of a literal l to extend the result of l (expressed on its variables) to all clause variables. More precisely, EXTG extends a set of substitutions Θ with a set of substitutions Θ' representing the result of executing a literal l on Θ. Its concrete version is specified as follows, assuming that D is the domain of Θ, $D'' = \{x_{i_1}, \ldots, x_{i_n}\}$ is the set of variables appearing in l exactly in that order, and $D' = \{x_1, \ldots, x_n\}$.

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

Its abstract version amounts to denormalizing the substitution and taking its conjunction with the clause substitution.

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

4.3. Abstract Semantics

The abstract semantics can be obtained easily by replacing sets of substitutions by abstract substitutions and each concrete operation by its abstract version to obtain a transformation $TSAT$. The abstract semantics is then defined as $\mu(TSAT)$, i.e., the least fixpoint of $TSAT$. Moreover, the semantics can be proven consistent w.r.t. the concrete semantics.

Theorem 4.1. Let P be a program, $TSAT$ and $TSCT$ be the associated transformations, $sat = \mu(TSAT)$, and $sct = \mu(TSCT)$. Let p be a predicate and $D = \{x_1, \ldots, x_n\}$ where n is the arity of p.

$$\beta \in Prop_D \ \Rightarrow \ sct(Cc(\beta), p) \subseteq Cc(sat(\beta, p)).$$

```
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.

4.4. 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 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 have 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` (line 29) since it returns $\bot$ and is shown during the second iteration (lines 34–40). The Boolean functions are shown in a readable form. This is a slightly edited version of the output of our system which depicts formulas in disjunctive normal form, although the canonical form used by the algorithm is different. Also, we use $A \Leftrightarrow B \Leftrightarrow C$ to abbreviate $(A \Leftrightarrow B) \wedge (B \Leftrightarrow C)$. 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 [17]. 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 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 arguments. This enables the domain `Prop` to achieve maximal precision in this example without keeping track of functors and working only on the clause variables.

```
1  Try clause 1
2      Exit EXTC x1
3      Exit AI-FUNC x3 ∧ x1
4      Call PRO-GOAL qsort/3 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 ∧ x1
19         Exit AI-FUNC (x9 ⇔ x8) ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x3 ∧ x1
20         Call PRO-GOAL qsort/3 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/3 x1
35             Exit PRO-GOAL qsort/3 (x3 ⇔ x2) ∧ x1
36             Exit EXTG (x9 ⇔ x8 ⇔ x2) ∧ x7 ∧ x6 ∧ x5 ∧ x4 ∧ x1
37             Call PRO-GOAL qsort/3 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/3 x3 ∧ x1
46         Exit PRO-GOAL qsort/3 ⊥
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/3 x3 ∧ x1
53         Exit PRO-GOAL qsort/3 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/3 x3 ∧ x2 ∧ x1
59     Exit EXTG x3 ∧ x2 ∧ x1
60     Exit RESTRC x2 ∧ x1
61     Exit UNION x2 ∧ x1
62 Exit clause 1
```

FIGURE 3. Analysis of `qsort/2` using `Prop`.

4.5. Implementation

Our implementation of the domain `Prop` uses ordered binary decision graphs (OBDG) as a canonical form for Boolean functions [6]. 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 $x_1 < x_2 < \cdots < x_n$. The data structure underlying OBDGs is a binary tree with a number of restrictions.

Definition 4.8 [6]. 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 4.9 [6]. 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 4.10 [6]. 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.[3]

Reference [6] describes several algorithms for the 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 two-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.

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

TABLE 1. Complexity results of the basic operations on graphs.

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	$f\|_{x=true} \vee f\|_{x=false}$	$O(\|G\|^2)$

This is to be expected since Boolean satisfiability is an NP-complete problem. An important measure in the experiments will thus be the size of the graphs in practice.

5. THE DOMAIN PAT(PROP)

The domain Prop presented in the previous section may lose accuracy since it only works on the clause variables. In this section, we lift up this limitation and consider an infinite abstract domain integrating Prop with a pattern component preserving structural information about terms. This new domain is interesting for a number of reasons. On the one hand, it is likely to improve the accuracy of the analysis, since even more sophisticated relationships between variables will be maintained. On the other hand, its computational cost is not bounded in the same way as the domain Prop. It is thus particularly important to identify whether the execution of the analysis remains reasonable under these conditions.

The new domain can be obtained by instantiating the generic pattern domain proposed in [13] to Prop. The generic pattern domain upgrades any domain expressed on clause variables, called the $\mathfrak{R}$-domain, into an abstract domain combining the $\mathfrak{R}$-domain and a pattern component.

The presentation of the generic domain and its associated algorithms is outside the scope of this paper, but the reader can refer to [13] for a comprehensive overview of this approach. In the rest of this section, we briefly review the semantic part of the generic domain Pat($\mathfrak{R}$) and the operations it requires from the $\mathfrak{R}$-domain. We also show how to define these operations for the domain Prop to obtain Pat(Prop).

The rest of this section is organized as follows. Section 5.1 gives some basic intuitions about the generic domain. Section 5.2 describes the generic domain Pat($\mathfrak{R}$), including its concretization function. Section 5.3 defines Pat(Prop) as an instantiation of Pat($\mathfrak{R}$). Section 5.4 describes the concrete and abstract versions of the operations needed for the instantiation. Sections 5.2 and 5.3 can be skipped in a first reading.

5.1. Informal Overview

The key concept in the representation of the substitutions in this generic domain is the notion of subterm. With each subterm appearing in a substitution, the generic abstract domain may associate a *pattern* which specifies the main functor, as well as the subterms which are its arguments. In addition, it associates with each subterm its *properties*. These properties (e.g., sharing, groundness, freeness)

are left unspecified and are represented in the $\Re$-domain. Moreover, each variable in the domain of the substitution is associated with one of the subterms. Note that this information enables us to express that two arguments have the *same value* (and hence that two variables are bound together) by associating two arguments with the same subterm. To identify the subterms in an unambiguous way, an index is associated with each of them. If there are n subterms, we make use of indices $1, \ldots, n$. For instance, the substitution

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

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

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

The *pattern component* (possibly) assigns to an index an expression $g(i_1, \ldots, i_n)$ where g is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. If it is omitted, the pattern is said to be undefined. In our example, the pattern component makes the following associations:

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

The *same value* component, in this example, maps x_1 to 1, x_2 to 4, and x_3 to 5.

The properties of each of the subterms are stored by the $\Re$-domain. The $\Re$-domain has no knowledge about the pattern component. This allows the $\Re$-domain to be viewed as working on clause variables. The identification of subterms (and hence the link between the structural component and the $\Re$-domain) is a somewhat arbitrary choice. In the following, we identify the subterms with integer indices, say $1 \ldots n$ if n subterms are considered. The $\Re$-domain thus represents properties of the subterms by using these indices. For instance, when the $\Re$-domain corresponds to `Prop`, the Boolean formula $1 \wedge 2 \wedge 3 \wedge (5 \Leftrightarrow 6) \wedge 7$ can be used to store information on the above substitution.

NOTATION. In the following, we denote by I_p the set of indices $\{1, \ldots, p\}$, by ST_p the set of tuples of terms $\langle t_1, \ldots, t_p \rangle$, and by ST the set of all sets ST_p for some $p \geq 0$.

5.2. *The Generic Domain* `Pattern`

An abstract substitution in `Pat(`$\Re$`)` over the PV variables $x_1, \ldots, x_n$ is a triple (frm, sv, ℓ), where sv (the *same value component*) is a total function, frm (the *pattern component*) is a partial function, and ℓ is an element of the $\Re$-domain. The meaning of the *pattern*, *same value*, and $\Re$-domain components is as follows.

5.2.1. THE PATTERN COMPONENT. The pattern component associates with some of the indices in I_p an expression $g(i_1, \ldots, i_q)$ where g is a function symbol of arity q and $\{i_1, \ldots, i_q\} \subset I_p$. The pattern component is a partial function $frm : I_p \nrightarrow F_p$, where F_p is the set of all patterns on I_p, satisfying the following condition: let G_{frm} be the graph whose nodes belong to I_p and whose arcs are the pairs $\langle i, j \rangle$ such that $frm(i) = g(\ldots, j, \ldots)$. G_{frm} must be an acyclic graph. We take the convention of denoting by $frm(i) = undef$ the fact that no pattern is associated with i. The meaning of the component is given by the concretization

function that specifies that the component represents all p_tuples of terms that satisfy simultaneously all pattern constraints:

$$Cc(frm) = \{(t_1, \ldots, t_p) \mid \forall i, i_1, \ldots, i_q \in I_p : frm(i) = g(i_1, \ldots, i_q) \Rightarrow t_i = g(t_{i_1}, \ldots, t_{i_q})\}.$$

The condition on G_{frm} ensures that $Cc(frm)$ is not empty. In the following, we denote by FRM_p the set of all functions frm for a fixed p and by FRM the union of all FRM_p ($p \geq 0$).

5.2.2. THE SAME VALUE COMPONENT. The second component assigns a subterm to each variable in the substitution. Given a set D of program variables and a set of indices I_m, this component is a surjective function $sv : D \rightarrow I_m$. Its meaning is given by a concretization function that makes sure that two variables assigned to the same index have the same value:

$$Cc(sv) = \{\theta \mid dom(\theta) = D \text{ and } \forall x_i, x_j \in D : sv(x_i) = sv(x_j) \Rightarrow x_i\theta = x_j\theta\}.$$

We denote by $SV_{D,m}$ the set of all same value functions for fixed D and m and by SV the union of all sets $SV_{D,m}$ for any D and m.

5.2.3. THE $\Re$-DOMAIN AND ITS BASIC OPERATIONS. The $\Re$-component of the generic domain is a domain $\Re_p$ that gives information on a set of terms $\langle t_1, \ldots, t_p \rangle$. The domain is assumed to satisfy traditional requirements. For instance, the $\Re_p$ may be a cpo with an order $\leq_{\Re}$, an upper bound operation, and a monotone concretization function w.r.t. $\leq_{\Re}$.[4] In the following, we denote by $\Re$ the set of all $\Re_p$ ($p \geq 0$).

The $\Re$-domain needs a number of basic operations, i.e., `T_UNION`, `T_AI_VAR`, `T_AI_FUNC`, `PROJ`, `INTR`, `JOIN`, `REN`, in terms of which the standard operations are implemented. The implementation of the standard operations in terms of those basic operations is outside the scope of this paper, but the reader may consult [13] for more details. The basic operations will be specified later, together with their abstract versions.

5.2.4. THE GENERIC ABSTRACT DOMAIN. We are now in a position to specify the abstract domain.

Definition 5.1. Let D be a finite set of program variables. The set of abstract substitutions `Pat(`$\Re$`)` is the subset of $FRM \times SV \times \Re$ consisting of elements $\langle frm, sv, l \rangle$ satisfying the following conditions:

i. $\exists m, p \in N,\ p \geq m\ \&\ \ell \in \Re_p\ \&\ sv \in SV_{D,m}\ \&\ frm \in FRM_p$;
ii. $\forall i : m \leq i \leq p : \exists j : 1 \leq j \leq p : frm(j) = g(\ldots, i, \ldots)$.

Formally, the meaning of an abstract substitution $\beta = (frm, sv, \ell)$ is given by the following concretization function:

$$Cc(\beta) = \{\theta : dom(\theta) = D\ \&\ \exists (t_1, \ldots, t_p) \in Cc(\ell) \cap Cc(frm) : \forall x \in D : x\theta = t_{sv(x)}\}.$$

[4]Some of these requirements can be lifted up. See [24] for more details.

5.3. *The Domain* Pat(Prop)

We now consider the domain Pat(Prop) as an instantiation of Pat($\Re$) to Prop. The basic idea is to associate a variable i with each term t_i. The concretization function is easily generalized to tuples of terms as follows.

Definition 5.2. The concretization function for $\mathtt{Prop}_{I_p}$ is a function $Cc : Prop_{I_p} \rightarrow ST_p$ defined as follows:

$$Cc(f) = \{\langle t_1, \ldots, t_p\rangle \mid \forall \sigma \in SS : assign\ (\langle t_1\sigma, \ldots, t_p\sigma\rangle)(f) = true\}$$

where $assign : ST_p \rightarrow I_p \rightarrow Bool$ is defined by $assign\ \langle t_1, \ldots, t_p\rangle\ i = true$ iff t_i is ground.

5.4. *Abstract Operations*

We now specify the abstraction of the $\Re$-domain operations for Pat(Prop). As for Prop, we give the concrete and abstract versions for each operation, the abstract version being a consistent approximation of the concrete version.

The concrete operations are of two kinds. First, there are a certain number of operations which are similar to the traditional operations, but on sets of tuples instead of on sets of substitutions. These are operations T_UNION, T_AI_VAR, T_AI_FUNC. Second, there are a number of operations motivated by the need for introducing, removing, and renaming terms as the computation proceeds. Let us explain informally why some of them are needed. Operation INTR is used each time new terms are being introduced in a substitution. This is the case at clause entry (operation EXTC) as well as during the unification operations (operations AI_VAR, AI_FUNC, EXTG). Operation PROJ is used each time some terms should be removed from a substitution. This occurs in many operations, including clause exit (RESTRC) and procedure entry (RESTRG). Operation JOIN is used to join two tuples in operation EXTG just before calling the general unification algorithm. We now turn to the operations whose implementations are conceptually simple in the case of Prop and are closely related to those of the Prop domain.

UNION. This operation takes the union of two sets of tuples. Its concrete version is specified as follows:

$$\mathtt{T\_UNION}(\Phi_1, \Phi_2) = \Phi_1 \cup \Phi_2.$$

Its abstract version uses disjunction once again:

$$\mathtt{T\_UNION}(f_1, f_2) = f_1 \vee f_2.$$

UNIFICATION OF TWO VARIABLES. This operation is very close to the standard operation. The concrete version of operation T_AI_VAR is as follows:

$$\begin{aligned}\mathtt{T\_AI\_VAR}(\Phi, i, j) \;=\; \{\; &\langle t_1\sigma, \ldots, t_p\sigma\rangle \mid \langle t_1, \ldots, t_p\rangle \in \Phi\ \&\\ &\sigma \in mgu(t_i, t_j)\ \&\ \sigma \in SS\ \}.\end{aligned}$$

Its abstract version is given as follows.

$$\mathtt{T\_AI\_VAR}(f, i, j) = f \ \wedge\ (i \Leftrightarrow j).$$

UNIFICATION OF A VARIABLE AND A FUNCTOR. This operation is very close to the standard operation. The concrete version of operation `T_AI_FUNC` is as follows:

$$\texttt{T\_AI\_FUNC}(\Phi, i, \{j_1, \ldots, j_n\}, g) = \{ \langle t_1\sigma, \ldots, t_p\sigma\rangle \mid \langle t_1, \ldots, t_p\rangle \in \Phi \;\&\; \sigma \in mgu(t_i, g(t_{j_1}, \ldots, t_{j_n})) \;\&\; \sigma \in SS \}.$$

Its abstract version adds an equivalence, as was the case for unification in `Prop`:

$$\texttt{T\_AI\_FUNC}(f, i, \{j_1, \ldots, j_n\}, g) = f \wedge (i \Leftrightarrow j_1 \wedge \cdots \wedge j_n).$$

PROJECTION. This operation projects out of term t_j. Its concrete version is specified as follows:

$$\texttt{PROJ}(\Phi, j) = \{\langle t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_p\rangle \mid \langle t_1, \ldots, t_p\rangle \in \Phi \}.$$

Its abstract version is simply

$$\texttt{PROJ}(f, j) = \texttt{denorm}\,[1, \ldots, j-1, p, j, \ldots, p-1]\; f|_{j=true} \vee f|_{j=false}.$$

INTRODUCTION OF VARIABLES. This operation introduces k variables in locations $m+1, \ldots, m+k$. Its concrete version can be specified as follows:

$$\texttt{INTR}(\Phi, m, k) = \{ \langle t_1, \ldots, t_m, y_1, \ldots, y_k, t_{m+1}, \ldots, t_p\rangle \mid \langle t_1, \ldots, t_p\rangle \in \Phi \;\&\; y_1, \ldots, y_k \text{ are new distinct variables}\}.$$

Its abstract version is obtained by shifting the indices of the last $p-m$ variables by k positions.

$$\texttt{INTR}(f, m, k) = \texttt{denorm}[1, \ldots, m, m+1+k, m+2+k, \ldots, p+k]\; f.$$

JOIN. This operation concatenates tuples of terms coming from two different sets. Its concrete version is given as follows:

$$\texttt{JOIN}(\Phi_1, \Phi_2) = \{ \langle t_1^1, \ldots, t_n^1, t_1^2, \ldots, t_m^2\rangle \mid \langle t_1^1, \ldots, t_n^1\rangle \in \Phi_1 \;\&\; \langle t_1^2, \ldots, t_m^2\rangle \in \Phi_2 \}.$$

Its abstract version is given in terms of conjunction. Let $f_1 \in \texttt{Prop}_{I_p}$, $f_2 \in \texttt{Prop}_{I_q}$:

$$\texttt{JOIN}(f_1, f_2) = f_1 \wedge \texttt{denorm}\,[p+1, \ldots, p+q]\; f_2.$$

RENAMING OF VARIABLES. The $\Re$-domain also needs a renaming operation. Let $r : I_p \to I_p$ be a renaming of indices. The concrete version can be specified as follows:

$$\texttt{REN}(\Phi, r) = \{\langle t_{r(1)}, \ldots, t_{r(p)}\rangle \mid \langle t_1, \ldots, t_p\rangle \in \Phi\}.$$

Its abstract version is implemented by using the `denorm` function previously defined:

$$\texttt{REN}(f, t) = \texttt{denorm}\,[t(1), \ldots, t(p)]\; f.$$

6. EXPERIMENTAL EVALUATION

In this section, we report experimental results about the efficiency and accuracy of `Prop` and `Pat(Prop)` and compare them with other abstract domains. Section 6.1 describes the preliminaries, including a description of the benchmarks and the domains and algorithms used in the experiments. Sections 6.2 and 6.3 describe, respectively, the accuracy and efficiency of `Prop` and `Pat(Prop)`. Section 6.4 discusses the use of `Prop` and `Pat(Prop)` for on-line analysis, while Section 6.5 discusses the impact of caching on this domain.

It is important to stress that the experiments were not chosen to obtain as many ground arguments as possible to improve efficiency. In fact, the on-line (or condensing or goal-independent) analysis makes no assumption on the queries, and hence manipulates mostly nonground substitutions. Hence, the experiments cover well the possible cases that may occur in practice.

6.1. Preliminaries

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 [37] and implements an alpha-beta search procedure. The program `press1` is a symbolic equation-solver program taken from [37] as well. `Press2` is the same program, but one literal is repeated to improve precision.[5] The program `cs` is a cutting-stock program taken from [38]. 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 [16], 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 tokenizer 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 and 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

[5] That is, to simulate the effect of the reexecution strategy [25].

programs, say `Append` (with input modes `(var,var,ground)`) and `Qsort` (difference lists sorting the small elements first).

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 is best viewed as an abstraction of the domain of Bruynooghe and Janssens [3] where a pattern component has been added. The domain is fully described in [33], which also contains the proofs of monotonicity and consistency.

As for the generic domain `Pat(`$\Re$`)` presented before, which is in fact a generalization of `Pattern`, the key concept in the representation of the substitutions is the notion of subterm. Given a substitution on a set of variables, an abstract substitution associates with each subterm the following information:

- its *mode* (e.g., Gro, Var, Ngv (i.e., neither ground nor variable));
- its *pattern* which specifies the main functor as well as the subterms which are its arguments. Note that the *pattern* is optional. If it is omitted, the pattern is said to be undefined;
- its possible *sharing* with other subterms.

The correspondence between each variable in the domain of the substitution and one of the subterms is provided by a function called *same value*, which behaves as in `Pat(`$\Re$`)`.

If we consider again the substitution presented in Section 5.1, the association of indices is the same, giving the pattern representation

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

Each index is associated with a mode taken from

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

In the example, we have the following associations:

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

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.

Note that all components of this domain are not useful for a groundness analysis. If only groundness is important, the mode component could be simplified to contain only two modes: `any` and `ground`. If only pure programs are used, then sharing could be omitted as well. The same-value and structural information are, however, fundamental to obtain a good precision. Hence, the efficiency results given in the following would be better if those components were omitted, but the present results give an idea of how well `Prop` and `Pat(Prop)` compare with other domains.

THE DOMAIN Mode. The domain of [33] 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. Note once again that the mode and sharing components can be simplified if only groundness information would be important.

THE GENERIC ABSTRACT INTERPRETATION ALGORITHM. The algorithm used in the experimental results is the so-called "prefix optimization" algorithm [17]. It is essentially our original algorithm [23, 27] 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 [24] to the abstraction of the semantics depicted in Figure 1.

We also use the reexecution algorithm of [25]. 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 programming 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.

6.2. The Domain `Prop`

6.2.1. ACCURACY. In this section, we compare `Mode`, `Mode-reex`, `Pattern`, and `Prop` 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 procedure heads, 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

TABLE 2. Accuracy of the analysis on inputs: Comparison of Mode and Prop.

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 3. Accuracy of the analysis on outputs: Comparison of Mode and Prop.

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

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 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[6] and Qsort, while it loses precision in 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.

[6]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 [27].

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	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 5. Accuracy of the Analysis on Outputs: 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

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 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 nonlocal literals are not reexecuted inside a clause. Here is an artificial example of a program where Prop will derive groundness of the output,

TABLE 6. Accuracy of the analysis on inputs: Comparison of Prop and Pattern in percentage.

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

but Mode-Reex will not:

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

Mode-reex does not detect groundness since it never considers the reexecution of X1 = f(X2 , X3) during the solving of p/3 and the groundness information is lost by the UNION operation. Note that global reexecution (or propagation) [4, 31] is able to detect groundness in this case as well.

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.,

TABLE 7. Accuracy of the analysis on outputs: Comparison of Prop and Pattern.

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 8. Efficiency results for the domain Prop.

Program	Time	G-Iter	C-Iter	G-Iter/Time	C-Iter/Time
CS	1.34	50	94	37.31	70.15
Disj	1.01	45	88	44.55	87.13
Gabriel	0.47	47	114	100.00	242.55
Kalah	0.93	65	129	69.89	138.71
Peep	1.16	36	249	31.03	214.66
PG	0.16	16	31	100.00	193.75
Plan	0.12	19	41	158.33	341.67
Press1	5.96	287	866	48.15	145.30
Press2	6.03	287	878	47.60	145.61
QSort	0.05	7	15	140.00	300.00
Queens	0.04	9	17	225.00	425.00
Read	1.66	76	311	45.78	187.35
Mean				87.31	207.66

all groundness information is inferred correctly). Loss of precision appears only on the press programs and on read. It also achieves exactly the same precision as the reexecution algorithm on mode on the benchmark programs. This positive result is due to the ability of preserving sophisticated relationships between variables in Prop.

6.2.2. 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 SS10/30 workstation, 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 6.5 s, and most programs are under 1.5 s. The most time-consuming programs are Press1 and Press2, which are also the programs where Prop loses accuracy. Prop performs about 88 goal iterations per second on the average. In contrast, Pattern and Mode perform about 112 and 191 iterations per second, indicating that the abstract operations in Prop are more expensive. This last result should be interpreted with care, however, since, on the one hand, the first iteration of a goal is generally (but not always) more time consuming than the subsequent ones due to the prefix optimization and, on the other hand, Pat(Prop) converges more quickly than the other domains.

We 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 77% of the time of Pattern on the average, is 1.56 as slow as Mode, and requires 122% 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

TABLE 9. Computation times: Comparison of the domains.

Program	Prop:Pr	Pattern:Pa	Mode:Mo	Mode-Reex:Mr	Pr/Pa	Pr/Mo	Pr/Mr
CS	1.34	2.00	1.29	1.67	0.67	1.04	0.80
Disj	1.01	1.12	0.74	1.01	0.90	1.36	1.00
Gabriel	0.47	0.69	0.31	0.40	0.68	1.52	1.18
Kalah	0.93	1.86	0.72	0.81	0.50	1.29	1.15
Peep	1.16	2.14	1.11	1.28	0.54	1.05	0.91
PG	0.16	0.27	0.16	0.13	0.59	1.00	1.23
Plan	0.12	0.20	0.11	0.08	0.60	1.09	1.50
Press1	5.96	8.80	1.51	3.12	0.68	3.95	1.91
Press2	6.03	2.77	1.55	3.09	2.18	3.89	1.95
QSort	0.05	0.06	0.08	0.05	0.83	0.63	1.00
Queens	0.04	0.05	0.06	0.04	0.80	0.67	1.00
Read	1.66	5.29	1.39	1.58	0.31	1.19	1.05
Mean					0.77	1.56	1.22

other information that it needs to compute (i.e., patterns and sharing). Pattern is also about twice as fast as Prop on Press2. Prop is almost always slower than Mode-Reex. 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. Informally speaking, the goal iterations are the number of iterations of the semantic function T used with a procedure as second argument. It indicates that, on 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 fewer iterations than Pattern on all programs but Press2. This result is important, and seems to indicate that Prop converges more quickly than the other domains. Its operations, however, seem to be more expensive, as mentioned previously, although this should be interpreted with care, as stated before.

Table 11 gives some results on the sizes of the abstract substitutions. We collect information each time an abstract operation is executed. 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 table, Op denotes the number of call points, V the summation of the number of variables over all operations, MV the maximum number of variables over all operations, and AV the average number of variables. S is the summation of all sizes of the graph (i.e., the number of nodes in the graph) over all operations, MS the maximal size of a graph, and AS the mean of all sizes. We also give two ratios, MS/MV and AS/AV, the last one giving the number of nodes used per variable. The results indicate that the maximum size of a graph on all programs is 123, while the theoretical maximum is 2^{42}. On the average, a graph uses 1.13 nodes per variable with a maximum of 1.30 over all programs. The ratio MS/MV is also never greater than 8. The results clearly indicate the compactness of the representation and explain the behavior of Prop.

Finally, Table 12 gives the repartition of the time between the various abstract operations. It indicates that 80% of the time is spent in the abstract operations for

TABLE 10. Goal iteration: Comparison of the domains.

Program	Prop:Pr	Pattern:Pa	Mode:Mo	Mode-Reex:Mr	Pr/Pa	Pr/Mo	Pr/Mr
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

this domain. The most consuming operations are RESTRG (about 19%), AI_FUNC (about 16%), while SMALLEREQ, RESTRC, and EXTG are all above 10%.

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). It is useful at this point to mention that the on-line analysis presented in Section 6.4 will show that the efficiency is not too dependent on the fact that the results are ground at the end of the computation in many programs.

TABLE 11. Statistics on the substitutions: Standard analysis.

Program	Op	V	MV	AV	S	MS	AS	MS/MV	AS/AV
CS	2122	16530	42	7.79	17,437	107	8.22	2.55	1.06
Disj	2095	14047	25	6.71	13,443	38	6.42	1.52	0.96
Gabriel	1621	7950	19	4.90	9754	31	6.02	1.63	1.23
Kalah	3446	18,314	19	5.31	18,845	35	5.47	1.84	1.03
Peep	4549	23,984	15	5.27	24,603	29	5.41	1.93	1.03
PG	727	3569	16	4.91	3845	30	5.29	1.88	1.08
Plan	972	3024	8	3.11	3921	13	4.03	1.63	1.30
Press1	20,259	89,201	17	4.40	114,554	123	5.61	7.24	1.28
Press2	20,528	90,601	17	4.41	115,778	123	5.64	7.24	1.28
QSort	360	1474	9	4.09	1588	18	4.41	2.00	1.08
Queens	352	1122	10	3.19	1372	14	3.90	1.4	1.22
Read	6325	34300	22	5.42	34,383	79	5.44	3.59	1.00
Mean	5279.67	25,343	18.25	4.96	29,960	53.33	5.49	2.87	1.13

TABLE 12. Statistics on the time of the operations for Prop.

Program	RG	AIF	RC	EG	AIT	AIV	EC	LEQ	LUB	ToT
CS	27.70	19.85	15.11	9.78	1.33	0.30	0.30	10.81	5.19	90.37
Disj	37.30	16.01	10.72	8.37	0.59	0.29	0.15	10.57	4.55	88.55
Gabriel	14.09	14.85	10.91	12.18	1.27	1.40	0.13	12.44	9.26	76.52
Kalah	17.96	19.40	10.78	10.34	2.73	0.72	0.43	13.94	6.18	82.47
Peep	14.45	27.14	12.68	6.78	0.29	2.51	0.29	11.50	5.16	80.83
PG	19.50	16.74	9.40	9.06	1.26	1.26	0.11	16.40	6.77	80.50
Plan	17.50	13.84	5.09	14.51	0.66	0.22	0.22	18.38	8.19	78.63
Press1	18.26	18.57	14.46	13.39	0.91	0.61	0.30	16.89	6.39	89.80
Press2	18.35	18.05	14.14	13.53	0.90	0.60	0.30	15.94	6.17	87.97
QSort	16.79	6.76	7.67	9.49	0.65	0.65	0.00	11.44	3.77	57.22
Queens	10.50	9.8	5.9	5.5	1.6	0.00	0.3	13.20	19.70	66.50
Read	21.47	14.56	12.35	12.06	1.62	0.88	0.29	12.21	5.59	81.03
Mean	19.49	16.30	10.77	10.42	1.15	0.79	0.24	13.64	7.24	80.03

6.3. The Domain Pat(Prop)

6.3.1. ACCURACY. Tables 13 and 14 compare Prop and Pat(Prop) for the input and output arguments. The results indicate that Pat(Prop) improves on Prop on the press programs as far as inputs are concerned and on the press programs and read for the outputs. The improvement comes from the better handling of difference-lists provided by Pat(Prop). Note also that, the increase in precision is substantial for the press program. Tables 15 and 16 compare Pattern with Pat(Prop). The results indicate that Pat(Prop) improves on Pattern on the program press1, once again due to its better handling of difference-lists. We also compared Pat(Prop) with Pat-reex, i.e., the reexecution algorithm on Pattern. Once again, the results were exactly the same as was the case for Prop and Mode-reex. Note also that, in theory, Pat(Prop) is more accurate than Pat-reex, as the following example demonstrates.

```
test(X):- p(X),q(X).
p(X):- X = g(Y,Z),Y = f(Z).
p(X):- X = g(Y,Z),Z = f(Y).
q(X):- X = g(Y,Z),Y = a.
q(X):- X = g(Y,Z),Z = a.
```

Informally speaking, the key to understanding this example is to notice that p/1 returns the term $g(A, B)$ with the function $A \Leftrightarrow B$, while q/1 gives the term $g(A, B)$ with the function $A \vee B$. The result of Pat(Prop) is thus the term $f(A, B)$ with $A \wedge B$. Pat-reex would not be able to infer groundness in this case since groundness is lost in operation UNION.

In summary, Pat(Prop) and Pat-reex are more accurate than all the other domains and produce improvements on programs with sophisticated handling of difference-lists. On our benchmarks, Pat(Prop) produces optimal results on all programs but read. We were not able to detect if the results were optimal for read since only the source of the program was at our disposal (no specification

TABLE 13. Accuracy of the analysis on inputs: Comparison of `Prop` and `Pat(Prop)`.

Program	Args	G-Pro	G-PPr	B-Pro	B-PPr	Procs	B-Pro-P	B-PPr-P
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	99	0	84	52	0	50
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 14. Accuracy of the analysis on outputs: Comparison of `Prop` and `Pat(Prop)`.

Program	Args	G-Pro	G-PPr	B-Pro	B-PPr	Procs	B-Pro-P	B-PPr-P
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	55	0	0	19	0	0
PG	31	31	31	0	0	10	0	0
Plan	32	31	31	0	0	13	0	0
Press1	143	39	140	0	101	52	0	47
Press2	143	39	140	0	101	52	0	47
QSort	9	7	7	0	0	3	0	0
Queens	11	11	11	0	0	5	0	0
Read	122	70	74	0	4	43	0	4

TABLE 15. Accuracy of the analysis on inputs: Comparison of `Pattern` and `Pat(Prop)`.

Program	Args	G-Pat	G-PPr	B-Pat	B-PPr	Procs	B-Pat-P	B-PPr-P
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	99	0	84	52	0	50
Press2	143	99	99	0	0	52	0	0
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 16. Accuracy of the analysis on outputs: Comparison of Pattern and Pat(Prop).

Program	Args	G-Pat	G-PPr	B-Pat	B-PPr	Procs	B-Pat-P	B-PPr-P
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	53	55	0	2	19	0	2
PG	31	31	31	0	0	10	0	0
Plan	32	31	31	0	0	13	0	0
Press1	143	40	140	0	100	52	0	47
Press2	143	140	140	0	0	52	0	0
QSort	9	6	7	0	1	3	0	1
Queens	11	11	11	0	0	5	0	0
Read	122	74	74	0	0	43	0	0

TABLE 17. Efficiency results for the domain Pat(Prop).

Program	Time	G-Iter	C-Iter	G-Iter/Time	C-Iter/Time
CS	20.95	84	166	4.01	7.92
Disj	9.59	68	134	7.09	13.97
Gabriel	11.98	62	141	5.18	11.77
Kalah	22.52	117	236	5.20	10.48
Peep	15.98	76	410	4.76	25.66
PG	2.42	36	76	14.88	31.40
Plan	2.50	31	67	12.40	26.80
Press1	34.26	190	631	5.55	18.42
Press2	34.85	192	655	5.51	18.79
QSort	0.31	10	22	32.26	70.97
Queens	0.32	15	29	46.88	90.63
Read	182.07	178	804	0.98	4.42
Mean				12.06	27.60

or explanation were available). We also believe that Pat(Prop) produces almost optimal results on almost all Prolog programs, but this remains to be validated experimentally.

6.3.2. EFFICIENCY. Table 17 depicts the efficiency results of Pat(Prop). All but one program are below 35 s, and most of them are below 20 s. The most demanding program is clearly read, which takes about 3 min. The average number of goal iterations per seconds is 12, which is significantly less than the 87 iterations per seconds of Prop. It follows that the cost of the operations in Pat(Prop) is much higher than in Prop.

Table 18 compares the efficiency of Pat(Prop), Pat-reex, Pattern, and Prop. The results indicate that, on the average, Pat(Prop) is, respectively, 6, 11, and 22 times slower than Pat-reex, Pattern, and Prop. Most programs are also close to the average. This indicates that the additional accuracy provided by Pat(Prop) comes at a price since the increase in computation time is significant. Pat(Prop) is

TABLE 18. Computation times: Comparison of the domains with Pat(Prop).

Program	A:Pat(Prop)	B:Pat-reex	C:Pattern	D:Prop	A/B	A/C	A/D
CS	20.95	5.83	2.00	1.34	3.59	10.48	15.63
Disj	9.59	2.56	1.12	1.01	3.75	8.56	9.50
Gabriel	11.98	1.52	0.69	0.47	7.88	17.36	25.49
Kalah	22.52	3.12	1.86	0.93	7.22	12.11	24.22
Peep	15.98	3.57	2.14	1.16	4.48	7.47	13.78
PG	2.42	0.34	0.27	0.16	7.12	8.96	15.13
Plan	2.50	0.24	0.20	0.12	10.42	12.50	20.83
Press1	34.26	4.28	8.80	5.96	8.00	3.89	5.75
Press2	34.85	4.58	2.77	6.03	7.61	12.58	5.78
QSort	0.31	0.14	0.06	0.05	2.21	5.17	6.20
Queens	0.32	0.06	0.05	0.04	5.33	6.40	8.00
Read	182.07	28.29	5.29	1.66	6.44	34.42	109.68
Mean					6.17	11.66	21.66

thus appropriate for a very highly optimizing option or for programs relying heavily on difference-lists, since those programs would not be handled well by Prop.

Table 19 compares the goal iterations for the same programs. Interestingly, they indicate that Pat(Prop) makes only 1.6 more iterations than Prop and makes fewer iterations than Pat-reex and Pattern. This seems to indicate that the cost of the operations in Pat(Prop) is significantly higher. Table 20 gives some information on the number of operations on Boolean formulas performed by Pat(Prop) and the size of the graphs manipulated. The results indicate that the average size of a graph in Pat(Prop) is about 17 nodes on 12 variables, giving an average of 1.35 nodes per variable. The maximal size is 419 on program cs and the maximum number of variables is 81. Table 21 compares these results with those of Prop. They indicate that Pat(Prop) performs about 4.5 more operations than Prop on graphs whose sizes are about three times larger. This clearly explains where the time goes in Pat(Prop). We also measured the time spent in the various operations related to the Boolean expressions. The most interesting result is probably the fact that Pat(Prop) spends about 80% of its time on only these operations. The most costly operations are PROJ and REN, taking, respectively, about 27 and 22% of the computing time.

6.4. On-Line Analysis

We now consider the use of Prop and Pat(Prop) for an on-line analysis [15]. On-line analyses are also called condensing analyses [20] and goal-independent analyses [18] in the logic programming community. The key idea consists of performing the analysis without any assumption on the queries. The result for a given query can then be obtained by specializing the on-line results with the input query. On-line analyses are thus particularly appropriate for compositional or modular analyses. The key benefit of on-line analyses is that a predicate can be analyzed once (in a general fashion), and then specialized for various specific uses. It is important to stress, however, that on-line analyses put additional requirements on the domain to enable an effective specialization.

TABLE 19. Goal iteration: Comparison of the domains with `Pat(Prop)`.

Program	A:Pat(Prop)	B:Pat-reex	C:Pattern	D:Prop	A/B	A/C	A/D
CS	84	152	85	50	0.55	0.99	1.68
Disj	68	115	68	45	0.59	1.00	1.51
Gabriel	62	133	81	47	0.47	0.77	1.32
Kalah	117	153	117	65	0.76	1.00	1.80
Peep	76	122	94	36	0.62	0.81	2.11
PG	36	48	38	16	0.75	0.95	2.25
Plan	31	44	36	19	0.70	0.86	1.63
Press1	190	322	552	287	0.59	0.34	0.66
Press2	192	331	210	287	0.58	0.91	0.67
QSort	10	24	13	7	0.42	0.77	1.43
Queens	15	17	16	9	0.88	1.00	1.67
Read	178	595	209	76	0.30	0.85	2.34
Mean					0.60	0.85	1.59

`Prop` and `Pat(Prop)` are potentially interesting domains 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, in `Prop`, `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. In the case of `Prop` and `Pat(Prop)`, the specialization simply amounts to making the conjunction of the input queries and the result. For instance, if `append` is called with the last argument being ground, the specialization is simply

$$(x_3 \Leftrightarrow x_2 \wedge x_1) \wedge x_3$$

which is equivalent to

$$x_1 \wedge x_2 \wedge x_3.$$

TABLE 20. Statistics on the substitutions for `Pat(Prop)`.

Program	Op	MV	AV	MS	AS	MS/MV	AS/AV
CS	13,143	81	20.92	419	24.01	5.17	1.15
Disj	10,256	41	16.87	67	17.57	1.63	1.04
Gabriel	7046	43	12.05	285	26.47	6.63	2.20
Kalah	20,264	48	16.45	84	17.48	1.75	1.06
Peep	25,460	23	10.14	53	10.84	2.30	1.07
PG	4454	30	11.01	34	12.34	1.13	1.12
Plan	4059	27	9.18	39	11.31	1.44	1.23
Press1	38,146	44	11.95	128	14.05	2.91	1.18
Press2	39,235	44	11.86	128	13.95	2.91	1.18
QSort	791	15	7.63	30	9.16	2.00	1.20
Queens	1048	13	7.41	20	8.28	1.54	1.12
Read	60,080	44	13.24	1601	34.51	36.39	2.61
Mean	18695.17	37.75	12.39	240.67	16.66	5.48	1.35

TABLE 21. Statistics on the substitutions: Ratio Pat(Prop)/Prop.

Program	Op	V	MV	AV	S	MS	AS	MS/MV	AS/AV
CS	6.19	16.64	1.93	2.69	18.09	3.92	2.92	2.03	1.09
Disj	4.90	12.32	1.64	2.51	13.40	1.76	2.74	1.08	1.09
Gabriel	4.35	10.68	2.26	2.46	19.12	9.19	4.40	4.06	1.79
Kalah	5.98	18.53	2.53	3.10	19.13	2.40	3.20	0.95	1.03
Peep	5.60	10.77	1.53	1.92	11.22	1.83	2.00	1.19	1.04
PG	6.13	13.74	1.88	2.24	14.29	1.13	2.33	0.60	1.04
Plan	4.18	12.32	3.38	2.95	11.71	3.00	2.81	0.89	0.95
Press1	1.88	5.11	2.59	2.72	4.68	1.04	2.50	0.40	0.92
Press2	1.91	5.14	2.59	2.69	4.73	1.04	2.47	0.40	0.92
QSort	2.20	4.10	1.67	1.87	4.56	1.67	2.08	1.00	1.11
Queens	2.98	6.92	1.30	2.32	6.32	1.43	2.12	1.10	0.91
Read	9.50	23.20	2.00	2.44	60.30	20.27	6.34	10.13	2.60
Mean	4.65	11.62	2.11	2.49	15.63	4.06	2.99	1.99	1.21

In the rest of this section, we give experimental results on the use of Prop and Pat(Prop) for on-line analysis. All programs have been run without any assumption on the input patterns (and/or the database) and have been specialized afterwards with the input patterns. The execution is exactly similar to the standard analysis, except that the initial input pattern is *true*, as are the results of the database predicates.[7]

THE DOMAIN Prop. Table 22 depicts the efficiency results on the use of Prop for on-line analysis and compares them to the standard analysis. The computation times for the on-line analysis are 1.81 slower than the standard analysis. The peak is reached on program disj, which is about four times slower. On the average, the on-line analysis takes about 1.3 more iterations than the traditional analysis. Table 23 depicts the statistics on the various graphs during the computation. The average size of a graph for the on-line analysis is 7.85 (instead of 5.49 for the standard analysis), while the ratio AS/AV is 1.30 (instead of 1.13). The efficiency of Prop for on-line analysis remains reasonable. It should be clear that the on-line analysis deals with programs with few ground arguments; the only ground arguments come from built-ins or generators of values. Table 24 compares the number of ground arguments in the standard and on-line analyses and the execution times of the analyses. Gro-on and Gro-st give the number of output ground arguments in the on-line and standard analysis, respectively. The results indicate that the number of ground outputs decreases by a factor of about 4.5 on the average in the on-line analysis, while the efficiency only slows down by a factor of 1.81 on the average. This seems to indicate that the previous experimental results were not too dependent on the fact that the results are ground at the end of the computation in many programs. An interesting theoretical issue is to understand why this is indeed the case, and whether static analysis of Prolog has some special properties w.r.t. Boolean formulas.

[7] It is possible to make an on-line analysis for all predicates at the same time, but this requires modifying the fixpoint algorithm slightly. This is outside the scope of this paper.

TABLE 22. On-line analysis: Efficiency results of Prop.

Program	Time-on:TO	Iter-on:IO	Time-st:TS	Iter-st:IS	TO/TS	IO/IS
CS	3.05	61	1.34	50	2.28	1.22
Disj	4.06	64	1.01	45	4.02	1.42
Kalah	0.99	72	0.93	65	1.06	1.11
Peep	2.94	61	1.16	36	2.53	1.69
PG	0.16	17	0.16	16	1.00	1.06
Plan	0.16	27	0.12	19	1.33	1.42
Press1	6.00	287	5.96	287	1.01	1.00
Press2	6.22	287	6.03	287	1.03	1.00
QSort	0.12	12	0.05	7	2.40	1.71
Queens	0.09	15	0.04	9	2.25	1.67
Read	1.66	77	1.66	76	1.00	1.01
Mean	2.31	89.09	1.68	81.55	1.81	1.30

TABLE 23. On-line analysis: Statistics on the substitutions for Prop.

Program	Op	V	MV	AV	S	MS	AS	MS/MV	AS/AV
CS	2390	24,419	42	10.22	33,125	271	13.86	6.45	1.36
Disj	1889	17,751	25	9.40	39,717	223	21.03	8.92	2.24
Kalah	2770	16,933	19	6.11	18,332	53	6.62	2.79	1.08
Peep	5870	38,584	15	6.57	47,199	62	8.04	4.13	1.22
PG	699	3533	16	5.05	3828	30	5.48	1.88	1.09
Plan	1074	3788	8	3.53	4951	23	4.61	2.88	1.31
Press1	20,276	89,360	17	4.41	113,671	123	5.61	7.24	1.27
Press2	20,545	90,760	17	4.42	115,895	123	5.64	7.24	1.28
QSort	667	2861	9	4.29	3127	25	4.69	2.78	1.09
Queens	463	1847	10	3.99	2456	25	5.30	2.50	1.33
Read	6326	34,300	22	5.42	34,385	79	5.43	3.59	1.00
Mean	5724.45	29,466	18.18	5.76	37,880.55	94.27	7.85	4.58	1.30

As far as the accuracy is concerned, the quality of the results was rather surprising. We performed an on-line analysis on the whole program, and specialized the result of the top-level goal with the input query. On all programs, the specialization of the on-line analysis with the input pattern gave the same result for the top-level goal as the traditional analysis with Prop.[8] This indicates that Prop is appropriate for on-line analysis.

It is interesting to compare this result with the domain Pattern for this kind of analysis. The on-line analysis of Pattern, specialized with the input queries, only gives the same result as the traditional analysis on four programs (kalah, peep, pg, qsort), and two of these (i.e., peep, qsort) do not produce optimal results, as shown before. On all other programs, there was a loss of accuracy in the top-level goal, i.e., the analysis would give any and novar instead of ground.

The main reason is that the domain does not keep sophisticated dependencies

[8]Recall, however, that Prop loses precision on programs press1 and press2.

TABLE 24. On-line versus standard analysis: Groundness and efficiency results of Prop.

Program	Gro-on	Gro-st	Gro-st/Gro-on	Time-on	Time-st	Time-on/Time-st
CS	32	94	2.93	3.05	1.34	2.28
Disj	8	60	7.50	4.06	1.01	4.02
Kalah	38	121	3.18	0.99	0.93	1.06
Peep	8	63	7.87	2.94	1.16	2.53
PG	13	31	2.38	0.16	0.16	1.00
Plan	4	31	7.75	0.16	0.12	1.33
Press1	29	39	1.34	6.00	5.96	1.01
Press2	29	39	1.34	6.22	6.03	1.03
QSort	3	7	2.33	0.12	0.05	2.40
Queens	1	11	11.00	0.09	0.04	2.25
Read	32	70	2.18	1.66	1.66	1.00
Mean			4.50	2.31	1.68	1.81

TABLE 25. On-line analysis: Efficiency results of Pat(Prop).

Program	Time-on:TO	Iter-on:IO	Time-st:TS	Iter-st:IS	TO/TS	IO/IS
CS	39.12	99	20.95	84	1.87	1.18
Disj	53.14	74	9.59	68	5.54	1.09
Kalah	34.80	130	22.52	117	1.55	1.11
Peep	36.93	80	15.98	76	2.31	1.05
PG	2.66	37	2.42	36	1.10	1.03
Plan	3.27	40	2.50	31	1.31	1.29
Press1	33.85	190	34.26	190	0.99	1.00
Press2	34.35	192	34.85	192	0.99	1.00
QSort	0.43	11.00	0.31	10.00	1.39	1.10
Queens	0.65	16.00	0.32	15.00	2.03	1.07
Read	182.07	179.00	182.07	178.00	1.00	1.01
Mean	38.30	95.27	29.62	90.64	1.82	1.08

between the variables. Note also that the same result holds for the other domains as well since they essentially contain the same information in the domain.

THE DOMAIN Pat(Prop). Table 25 depicts the efficiency results on the use of Pat(Prop) for on-line analysis and compares them to the standard analysis. Interestingly, the computation times for the on-line analysis are 1.82 slower than the standard analysis, confirming the results on Prop. The peak is reached once again on program disj, which is about 5.5 times slower. On the average, the on-line analysis takes about 1.08 more iterations than the traditional analysis. Table 26 depicts the statistics on the various graphs during the computation. The average size of a graph for the on-line analysis is 22.43 (instead of 16.66 for the standard analysis), while the ratio AS/AV is 1.74 (instead of 1.35). The efficiency of Pat(Prop) for on-line analysis remains reasonable, indicating once again that the previous experimental results were not too dependent on the fact that the results are ground at the end of the computation in many programs.

As far as the accuracy is concerned, the quality of the results was also rather

TABLE 26. On-line analysis: Statistics on the substitutions for `Pat(Prop)`.

Program	Op	MV	AV	MS	AS	MS/MV	AS/AV
CS	17,914	87	21.15	784	30.98	9.01	1.46
Disj	11,202	41	16.88	1485	59.62	36.22	3.53
Kalah	20,658	48	16.45	255	25.09	5.31	1.53
Peep	25,828	23	10.07	214	18.20	9.30	1.81
PG	4478	30	10.98	51	12.61	1.70	1.15
Plan	4589	27	9.07	83	12.47	3.07	1.37
Press1	38,159	44	11.96	128	14.05	2.91	1.17
Press2	39,248	44	11.87	128	13.96	2.91	1.18
QSort	814	15	7.54	47	12.24	3.13	1.62
Queens	1021	13	7.57	73	12.98	5.62	1.71
Read	60,096	44	13.24	1601	34.50	36.39	2.61
Mean	20,364.27	37.82	12.43	440.82	22.43	10.51	1.74

surprising. On all programs, the specialization of the on-line analysis with the input pattern gives the same result for the top-level goal as the traditional analysis with `Pat(Prop)`. This indicates that `Pat(Prop)` is really a domain of choice for on-line analysis.

6.5. The Impact of Caching

In this section, we evaluate the impact of the caching optimization [17] on the performance of `Prop` and `Pat(Prop)`. This is an interesting issue to investigate since the hashing function and the copy of abstract substitution are much more expensive in `Pat(Prop)` and `Prop` than in `Mode` and `Pattern`. Table 27 reports the results of the prefix algorithm augmented with caching on `Prop` and `Pat(Prop)`. Recall that all results given previously in the paper were obtained using the prefix algorithm without caching.

The results indicate that `caching` brings an additional improvement over the prefix optimization for `Pat(Prop)`, although this improvement is small. This indicates that `caching` is even better in this domain than it was for `Pattern`, where caching brought about 30% improvement over the original version, but none over the prefix version. On the order hand, for `Prop`, caching does not bring any improvement.

7. 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 a 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.

TABLE 27. Efficiency: The impact of caching.

Program	C-Prop:CP	Prop:P	CP/P	C-Pat(Prop):CPP	Pat(Prop):PP	CPP/PP
CS	1.61	1.34	1.20	21.29	20.95	1.02
Disj	1.23	1.01	1.22	10.23	9.59	1.07
Gabriel	0.62	0.47	1.32	12.61	11.98	1.05
Kalah	1.10	0.93	1.18	18.99	22.52	0.84
Peep	1.39	1.16	1.20	16.66	15.98	1.04
PG	0.16	0.16	1.00	2.43	2.42	1.00
Plan	0.13	0.12	1.08	2.25	2.50	0.90
Press1	6.51	5.96	1.09	30.94	34.26	0.90
Press2	6.56	6.03	1.09	31.51	34.95	0.90
QSort	0.01	0.05	0.20	0.31	0.31	1.00
Queens	0.01	0.04	0.25	0.31	0.32	0.97
Read	2.10	1.66	1.27	179.64	182.07	0.99
Mean	1.79	1.58	1.01	27.26	28.15	0.97

The purpose of this paper was to study the performance of domain Prop. Its first contribution is to describe an implementation of the domain Prop and to use it to instantiate a generic abstract interpretation algorithm [17, 23, 27]. A key feature of the implementation is the use of ordered binary decision graphs to provide a compact representation of many Boolean functions. Its second contribution is to describe the design and implementation of a new domain, Pat(Prop), combining the domain Prop with structural information about the subterms. This new domain may significantly improve the efficiency of the domain Prop on programs manipulating difference-lists.

Both implementations (resp. 6000 and 12,000 lines of C) have been evaluated systematically, and their efficiency and accuracy for groundness inference have been compared with several other abstract domains: the domain Mode (mode, same-value, sharing), the domain Pattern (mode, same-value, sharing, pattern), and the domains Mode and Pattern used inside a reexecution algorithm [25] to improve accuracy. The interest of Pat(Prop) and Prop for on-line analysis are also investigated.

Various domains have been compared in this paper. As far as accuracy is concerned, the following two orderings summarize the results on our benchmarks:

$$\{\text{Mode}\} < \{\text{Mode-reex}, \text{Prop}\} < \{\text{Pat(Prop)}, \text{Pat-reex}\}$$
$$\{\text{Mode}\} < \{\text{Pattern}\} < \{\text{Pat(Prop)}, \text{Pat-reex}\}.$$

Mode is clearly the least accurate algorithm, while Pat(Prop) and Pat-reex are the most accurate. An interesting result of these experiments is the fact that the reexecution algorithm on Mode and Pattern have the same accuracy as the standard algorithm on Prop and Pat(Prop). An interesting open issue is to find practical programs for which Prop and Pat(Prop) would outperform Mode-reex and Pat-reex. We also believe that, on almost all practical programs, Pat(Prop) should produce close to optimal groundness information.

As far as efficiency is concerned, the results can be summarized by the following

orderings:

```
Mode < Mode-reex < Prop < Pattern < Pat-reex < Pat(Prop).
```

This result indicates that there is a price to pay for the additional accuracy provided by `Prop` and `Pat(Prop)`. This price is very reasonable for `Prop` and less so for `Pat(Prop)`. Note also that, when only groundness information is desired, the domains `Mode` and `Pattern` could be simplified, further improving their efficiencies.

When efficiency and accuracy are considered, it is not clear which approach is best since the choice mainly depends upon the tradeoff between efficiency and accuracy to be achieved. However, it seems tempting to consider that `Mode-reex` and `Pat-reex` are to be preferred to `Prop` and `Pat(Prop)`. This is true on our benchmarks, but this result needs to be interpreted with care for several reasons.

1. The `Prop`-based domains are particularly well suited for on-line analysis, and should outperform the other domains significantly for this application. Our experimental results indicate that `Pat(Prop)` is as precise in on-line mode as in standard analysis, while `Prop` is close to being as accurate. Moreover, the analysis time remains reasonable, and can be factored out between several applications.
2. The `Prop`-based domains are theoretically more precise. In practice, `Prop` should certainly bring additional accuracy over `Mode-reex` for some programs, and hence may be preferred. The case of `Pat-reex` and `Pat(Prop)` is more difficult since the programs seem much more contrived. An interesting issue is to characterize the class of programs for which the additional theoretical expressiveness of the `Prop`-based domains would produce better practical results.
3. The `Prop`-based domains are easier to apply when nonlogical features are taken into account.

It is worth stressing that the implementation techniques of `Prop` and `Pat(Prop)` can be reused in other contexts such as, for instance, nonlinearity and sharing. Hence, our results also give some ideas of the applicability of Boolean formulas for representing abstract substitutions.

Finally, note that, since the submission of this work, two new implementations of `Prop` [7, 10] have emerged, confirming the results of this paper and extending them. Both of these works use a bottom-up framework. Reference [10] uses a generalization of OBDD representation of Boolean formulas to symbolic finite domains inside the constraint language `Toupie`, while [7] compiles the abstract semantics to a datalog program and uses some deductive database technology.

Olivier Degimbe and Laurent Michel helped in implementing the caching version of the algorithms. The comments of the reviewers were very helpful in improving the presentation of the paper. We are especially grateful to reviewer 2 who suggested the example for operation `UNION` in `Prop`, and to reviewer 3 who suggested including the groundness results of all predicates in the on-line analysis to show the impact of ground predicates on performance. This research was partly supported by the National Science Foundation under Grant CCR-9108032 and the National Young Investigator Award, the Office of Naval Research under Grant N00014-91-J-4052 ARPA Order 8225, and the Belgian National Incentive-Program for fundamental Research in Artificial Intelligence.

REFERENCES

1. Barbuti, R., Giacobazzi, R., and Levi, G., A general framework for semantics-based bottom-up abstract interpretation of logic programs, *ACM Transactions on Programming Languages and Systems* 15(1):133–181 (Jan. 1993).
2. Bruynooghe, M., A practical framework for the abstract interpretation of logic programs, *Journal of Logic Programming* 10(2):91–124, (Feb. 1991).
3. Bruynooghe, M. and Janssens, G., An instance of abstract interpretation: integrating type and mode inferencing, in: *Proc. Fifth International Conference on Logic Programming*, Seattle, WA, Aug. 1988, pp. 669–683, MIT Press, Cambridge.
4. Bruynooghe, M. and Janssens, G., Propagation: A new operation in a framework for abstract interpretation of logic programs, in: A. Pettorossi, ed., *Proc. of Meta-Programming in Logic (META'92)*, no. 649 in Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 294–307.
5. Bruynooghe, M., Janssens, G., Callebaut, A., and Demoen, B., Abstract interpretation: Towards the global optimization of Prolog programs, in: *Proc. 1987 Symposium on Logic Programming*, San Francisco, CA, August 1987, pp. 192–204, IEEE, New York.
6. Bryant, R. E., Graph based algorithms for Boolean function manipulation, *IEEE Transactions on Computers* C-35(8):677–691, (1986).
7. Codish, M. and Demoen, B., Analysing logic programs using "Prop"-ositional logic programs and a magic wand, in: *Proc. of the International Symposium on Logic Programming (ILPS'93)*, Vancouver, Canada, (Nov. 1993).
8. Codognet, C., Codognet, P., and Corsini, J. M., Abstract interpretation of concurrent logic languages, in: *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, Oct. 1990, MIT Press, Cambridge.
9. Corsini, A. and Filé, G., A complete framework for the abstract interpretation of logic programs: Theory and applications, Research Report, Department of Computer Science, University of Padova, Italy, 1989.
10. Corsini, M., Musumbu, K., Rauzy, A., and Le Charlier, B., Efficient bottom-up abstract interpretation of Prolog by means of constraint solving over symbolic finite domains, in: *Proc. Fifth International Conference on Programming Language Implementation and Logic Programming*, Tallinn, Estonia, (Aug. 1993).
11. Cortesi, A., Filé, G., and Winsborough, W., Prop revisited: Propositional formulas as abstract domain for groundness analysis, in: *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991, pp. 322–327.
12. Cortesi, A., Filé, G., and Winsborough, W., Comparison of abstract interpretations, in: *Proc. 19th International; Colloquium on Automata, Languages and Programming (ICALP'92)*, 1992.
13. Cortesi, A., Le Charlier, B., and Van Hentenryck, P., Combinations of abstract domains for logic programming, in: *21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, Jan. 1994.
14. Cousot, P. and Cousot, R., Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: New York ACM Press, ed., *Conf. Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, Los Angeles, CA, Jan. 1977 pp. 238–252.
15. Deutsch, A., A storeless model of aliasing and its abstraction using finite representa-

tions of right-regular equivalence relations, in: *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Francisco, CA, (Apr. 1992).

16. Dincbas, M., Simonis, H., and Van Hentenryck, P., Solving large combinatorial problems in logic programming, *Journal of Logic Programming* 8(1-2):75–93, (Jan./Mar. 1990).
17. Englebert, V., Le Charlier, B., Roland, D., and Van Hentenryck, P., Generic abstract interpretation algorithms for Prolog: Two optimization techniques and their experimental evaluation, *Software Practice and Experience* 23(4), (Apr. 1993).
18. Gabbrielli, M., Giacobazzi, R., and Levi, G., Goal independency and call patterns in the analysis of logic programs, Technical Report, Dipartimento di Informatica, Universita di Pisa, 1993.
19. Hermenegildo, M., Warren, R., and Debray, S., Global flow analysis as a practical compilation tool, *Journal of Logic Programming* 13(4):349–367, (Aug. 1992).
20. Jacobs D. and Langen, A., Accurate and efficient approximation of variable aliasing in logic programs, in: *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, OH., Oct. 1989, pp. 154–165, MIT Press, Cambridge.
21. Kanamori, T. and Kawamura, T., Analysing success patterns of logic programs by abstract hybrid interpretation, Technical Report, ICOT, 1987.
22. Le Charlier, B., Musumbu, K., and Van Hentenryck, P., Efficient and accurate algorithms for the abstract interpretation of prolog programs, Research Paper RP-90/9, Department of Computer Science, University of Namur, Aug. 1990.
23. Le Charlier, B., Musumbu, K., and Van Hentenryck, P., A generic abstract interpretation algorithm and its complexity analysis (extended abstract), in: *Eighth International Conference on Logic Programming (ICLP-91)*, Paris, France, June 1991, pp 64–78, MIT Press, Cambridge.
24. Le Charlier, B. and Van Hentenryck, P., A universal top-down fixpoint algorithm, Technical Report CS-92-25, CS Department, Brown University, 1992.
25. Le Charlier, B. and Van Hentenryck, P., Reexecution in abstract interpretation of Prolog, in: *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, Nov. 1992. To appear in *Acta Informatica*.
26. Le Charlier, B. and Van Hentenryck, P., Groundness analysis for Prolog: Implementation and evaluation of the domain `Prop`, in: *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM93)*, Copenhagen, Denmark, June 1993.
27. Le Charlier, B. and Van Hentenryck, P., Experimental evaluation of a generic abstract interpretation algorithm for Prolog, *ACM Transactions on Programming Languages and Systems* 16(1):35–101, (Jan. 1994).
28. Marriott, K. and Sondergaard, H., Notes for a tutorial on abstract interpretation of logic programs, North American Conference on Logic Programming, Cleveland, OH, Oct. 1989.
29. Marriott, K. and Sondergaard, H., Semantics-based dataflow analysis of logic programs, in: *Information Processing-89*, San Francisco, CA, 1989, pp. 601–606.
30. Marriott, K. and Sondergaard, H., Analysis of constraint logic programs, in: *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, Oct. 1990.
31. Marriott, K. and Sondergaard, H., Propagation and reexecution reexamined, in: *ILPS*

Workshop on Global Compilation, Vancouver, Canada, Nov. 1993.

32. Mellish, C., *Abstract Interpretation of Prolog Programs*, Ellis Horwood, Chichester, 1987, pp. 181–198.
33. Musumbu, K., *Interpretation Abstraite de Programmes Prolog*, Ph.D. Dissertation, Department of Computer Science, University of Namur, Belgium, Sept. 1990.
34. Muthukumar, K. and Hermenegildo, M., Determination of variable dependence information through abstract interpretation, in: *Proceedings of the North American Conference on Logic Programming (NACLP-89)*, Cleveland, OH, Oct. 1989, pp. 166–188, MIT Press, Cambridge.
35. Muthukumar, K. and Hermenegildo, M., Compile-time derivation of variable dependency using abstract interpretation, *Journal of Logic Programming* 13(2–3):315–347, (Aug. 1992).
36. O'Keefe, R.A., Finite fixed-point problems, in: J.-L. Lassez, ed., *Fourth International Conference on Logic Programming*, Melbourne, Australia, 1987, pp. 729–743.
37. Sterling, L. and Shapiro, E., *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, MA, 1986.
38. Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
39. Warren, R., Hermedegildo, M., and Debray, S. L., On the practicality of global flow analysis of logic programs, in: *Proc. Fifth International Conference on Logic Programming*, Seattle, WA, Aug. 1988, pp. 684–699, MIT Press, Cambridge.
40. Winsborough, W., Multiple specialization using minimal-function graph semantics, *Journal of Logic Programming* 13(4), (July 1992).
41. Winsborough, W. H., A minimal function graph semantics for logic programs, Technical Report TR-711, Computer Science Department, University of Wisconsin at Madison, Aug. 1987.