Abstract interpretation is a widely recognized technique that allows constructing static analysis tools by safely approximating program semantics. Frameworks for abstract interpretation (such as CiaoPP) involve the implementation of a chaotic iteration strategy to compute an abstract fixpoint (the “fixpoint” for short) and the implementation of several abstract domains, which approximate different program properties. The implementation of (sound, precise, efficient) abstract domains usually requires coding from scratch a large number of operations. Moreover, due to undecidability, a loss of precision is inevitable, which makes the design (and implementation) of more domains, as well as their combinations, eventually necessary to successfully prove arbitrary program properties. In this paper we focus on the latter problem by proposing a rule-based methodology for designing and rapid prototyping new domains for logic programs, as well as composing and combining existing ones. Our techniques are inspired in logic-based languages for implementing constraint domains at different abstraction levels. We provide several examples for domains combining numerical properties and data types, and apply them to proving complex properties of Prolog programs.

Keywords: Prolog, Abstract Interpretation, Logic Programming, Static Analysis.

1 Introduction

Abstract interpretation was developed by Radhia and Patrick Cousot in the late 1970s (9). This technique allows constructing sound program analysis tools which can extract properties of a program by safely approximating its semantics. Static analysis tools are a crucial component of the development environments for many programming languages. Abstract interpretation proved practical and effective in the context of (Constraint) Logic Programming ((C)LP) (31; 24; 30; 25; 1; 21; 20), which was one of its first application areas (see (16)), and the techniques developed in this context have also been applied to the analysis and verification of other programming paradigms by using semantic translation to Horn Clauses (see the recent survey (12)). Frameworks for abstract interpretation (such as CiaoPP (18) or Astrée (11)) involve the implementation of a chaotic iteration strategy to compute an abstract fixpoint (the “fixpoint” for short) and the implementation of several abstract domains, which approximate different program properties. Unfortunately, the implementation of (sound, precise, efficient) abstract domains usually requires coding from scratch a large number of domain-related operations. Moreover, due to undecidability, a loss of precision is inevitable, which makes the design (and implementation) of more domains, as well as their combinations, eventually necessary to successfully prove arbitrary program properties. In this paper we focus on the latter problem by proposing a rule-based methodology for designing and rapid prototyping new domains for logic programs, as well as composing and combining existing ones. Our techniques are inspired in logic-based languages for implementing constraint domains at different abstraction levels. We provide several examples for domains combining numerical properties and data types, and apply them to proving complex properties of Prolog programs.

The challenges of designing sound, precise, and efficient analyses have made static analysis designers search for ways to simplify these tasks, and logic programming-related technologies such as Datalog (see, e.g., (7; 4; 32)) have in fact become quite popular recently in this context. However, these approaches are quite different from the abstract interpretation framework-based approaches that we address in this work, where significant parts of the analysis (such as abstracting execution paths) are taken care of by the framework. In addition, the lack of data structures makes the Datalog approach less natural for defining domains in our context. Some more general, Datalog-derived languages have been proposed that
are more specifically oriented to the definition and implementation of analyses, such as FLIX (22), a form of Datalog with lattices, which has been used to define several types of analyses for imperative programs, and other works that generalize Datalog with constraints (see again (12)). However, these approaches do not provide per se a specific formalism for defining the abstract domains as in our work. We are also aware of a partial attempt to use Constraint Handling Rules (CHR) (13), normally employed for defining constraint domains, for prototyping abstract interpretations, by Frederic Mesnard (23), but it seems that this potentially interesting work did not go beyond this initial draft. CHR is used to define a dependency domain, but only applying conjunction of constraints, which is not well suited for implementing fundamental operations that we address, such as the Least Upper Bound (LUB). Finally, rewriting systems have also been used to define abstract unifications and prove their correctness (5), but to the best of our knowledge not to derive domains for new properties.

The rest of the paper proceeds as follows: In Section 2 we present all the background required for presenting our technique. In Section 3 we explain the approach introducing the high-level language that we propose and use to derive the abstract operations. In Section 4 we show how different domains can be implemented with this technique including proving the sortedness property. In Section 5 we briefly review the implementation and show some experimental results. Finally, Section 6 presents our conclusions.

2 Background

2.1 Abstract Interpretation

Lattices. A partial order on a set $X$ is a binary relation $\subseteq$ that is reflexive, transitive and anti-symmetric. The least upper bound or join of two elements of a set $a, b \in X$, denoted by $a \sqcup b$ is the smallest element in $X$ greater than both of them ($a \subseteq a \sqcup b \land b \subseteq a \sqcup b$). If it exists, it is unique. Similarly, the greatest lower bound or meet is defined as the greatest element less than both. A partially ordered set or poset is a pair $(X, \sqsubseteq)$ where $X$ is a set and $\subseteq$ is a partial order relation on $X$. We say that $X$ is a lattice if $(X, \sqsubseteq)$ is a poset and for every two elements of $X$ there exist a meet and a join. A lattice is complete if every subset $S \subseteq X$ has both a supremum and an infimum. The maximum element of a complete lattice is called top and the minimum is called bottom (denoted by $\top$ and $\bot$ resp.).

Galois connections. Let $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ be two posets and $f : L_1 \rightarrow L_2$ and $g : L_2 \rightarrow L_1$ two applications such that:

$$\forall x \in L_1, y \in L_2 : f(x) \sqsubseteq_2 y \Leftrightarrow x \sqsubseteq_1 g(y)$$

Then we say that the quadruple $(L_1, f, L_2, g)$ is a Galois connection. If $f \circ g = id$ then the quadruple is called a Galois insertion.

Abstract interpretation. The semantics of a program can be described in terms of the concrete domain, e.g., in the case of Logic Programming it typically consists of sets of variable substitutions that may occur at run-time. The main idea behind abstract interpretation is to interpret the program over a special abstract domain whose elements are finite representations of possibly infinite sets of actual substitutions in the concrete domain. We will denote the concrete domain as $D$ and the abstract domain as $D_\alpha$. We will denote the functions that relate the sets of concrete substitutions with the abstract substitutions as the abstraction function $\alpha : D \rightarrow D_\alpha$ and the concretization function $\gamma : D_\alpha \rightarrow D$. The concrete domain is a complete lattice with the typically set inclusion order which induces an ordering relation in the abstract domain usually represented by $\sqsubseteq$. Under this relation the abstract domain is usually a complete lattice and $(D, \alpha, D_\alpha, \gamma)$ is a Galois insertion.

2.2 Abstract Interpretation for (Constraint) Logic Programs

(Constraint) Logic Programs. A Constraint Logic Program (CLP) is a set of clauses of the form

$H : A_1, \ldots, A_n$ where $A_1, \ldots, A_n$ are literals that form the body and $H$ is an atom said to be the head of the clause. When CLP programs are used for verification purposes only, the term Constrained Horn Clauses (CHCs) is often used.
Resolution basics. A substitution is a set \( \theta = \{ V_1/t_1, \ldots, V_n/t_n \} \) with \( V_i \) distinct variables and \( t_i \) terms. We usually denote (concrete) substitutions by \( \theta \) or \( \sigma \). We say that \( t_i \) is the value of \( V_i \) in \( \theta \). The set \( \{ V_1, \ldots, V_n \} \) is the domain of \( \theta \); the range is the set of variables appearing in \( t_1, \ldots, t_n \).

The composition of two substitutions \( \sigma \) and \( \theta \) is denoted by \( \sigma \theta \). A resolvent is represented by \( \langle (A_1, \ldots, A_n), \sigma_i \rangle \) where \( \sigma_i = \theta_1 \cdots \theta_i \) is the composition of the substitutions applied so far (the accumulated substitution). For the initial resolvent (the query) we have \( \sigma_0 = \epsilon \), the empty substitution. To perform a logical inference, the computation rule selects the leftmost literal \( A_1\sigma_i \). The search rule then selects a clause \( H \vdash (B_1, \ldots, B_m) \), renames it, and unifies the head \( H \) with \( A_1\sigma_i \). If the unification is successful with most general unifier \( \theta_{i+1} \), then the resolvent \( \langle (B_1, \ldots, B_m), A_2, \ldots, A_n\rangle\sigma_{i+1} \) is derived with \( \sigma_{i+1} = \sigma_i\theta_{i+1} \). The new state is the immediate successor of the previous one. Because a literal can match different clause heads, a resolvent can have several immediate successors.

Top-down semantics-based analysis. Top-down analyses represent a class of program analyses for (C)LP that are based on computing an analysis graph. This approach was first used in analyzers such as MA3 and Ms (31), and matured in the PLAI analyzer (24; 25). It was later also used in GAIA (21), or the CLP(R) analyzer (20). This style of analysis was extended early on to CLP/CHCs (1). We concentrate here for concreteness on the PLAI framework. The graph inferred is a finite, abstract object whose concretization approximates the (possibly infinite) set of (possibly infinite) maximal AND-trees of the concrete semantics. The PLAI approach separates the abstraction of the structure of the trees from the abstraction of the constraints at the nodes in the concrete trees. The first abstraction (\( T_0 \)) is typically built-in, and is the abstract domain of the analysis graph, which finitely approximates the shapes of the concrete AND-trees, independently of the content of the nodes. In this regard, \( T_0 \) is parametric on a second abstraction domain called \( D_\alpha \). Elements of \( D_\alpha \) are used as labels in the nodes of the analysis (abstract) graph. We refer to such nodes with tuples \((p(V_1, \ldots, V_n), \lambda^c, \lambda^s)\), where \( p \) is a predicate in the program under analysis, and \( \lambda^c, \lambda^s \) (both elements of \( D_\alpha \)), are respectively, the (abstract) call and success substitutions/constraints over the variables \( V_1, \ldots, V_n \). Such tuples represent the set of (concrete) call and success substitutions/constraints of the nodes in the concrete AND-trees. A more detailed recent discussion can be found in (12). Many PLAI extensions have been proposed, such as incremental and modular versions (27; 19; 15; 14).

Example 1. Figure 1 (from (15)) shows a possible analysis graph (center of figure) for a set of CHCs (left of figure) that encode the computation of the parity of a binary message using the exclusive or, denoted \( \text{xor} \). For instance, the parity of message \( [1,0,1] \) is \( 0 \). We take the abstract domain (right of figure) with the following abstract values: (i) \( \bot \) such that \( \gamma(\bot) = \emptyset \), (ii) \( z \) (for zero) such that \( \gamma(z) = \{ 0 \} \), (iii) \( o \) (for one) such that \( \gamma(o) = \{ 1 \} \), (iv) \( b \) (for bit) such that \( \gamma(b) = \{ 0, 1 \} \), and (v) \( \top \) such that \( \gamma(\top) = \{ \} \) is the set of all concrete values. Consider an initial abstract goal \( G_0 = (\text{main}(\text{Msg,P}), (\text{Msg}/\top, P/\top)) \), which represents that the arguments of \( \text{main} \) can be bound to any concrete value (see node A in the figure). Node B = \( (\langle \text{par}(\text{Msg}, X, P), (\text{Msg}/\top, X/z, P/\top), (\text{Msg}/\top, X/z, P/b) \rangle) \) captures the fact that \( \text{par} \) may be called with \( X \) bound to 0 in \( \gamma(z) \) and, if such a call succeeds, the third argument \( P \) will be bound to any value in \( \gamma(b) = \{ 0, 1 \} \). Note that node C captures the fact that, after this call, there are other calls to \( \text{par} \) where \( X/b \). Edges in the graph stem from the \( (A, \lambda^c, \lambda^s) \rightarrow (B, \mu^c, \mu^s) \) relation. For example, two such edges exist from node B, denoting that \( \text{par} \) may call \( \text{xor} \) (edge from B to D) or \( \text{par} \) itself with a different call description (edge from B to C). In this example we have used a simple, non-relational abstract domain.

Abstract domain operations As mentioned before, the abstract interpretation-based algorithms that we are considering are parametric on the data-related abstract domain, i.e., they are independent of the data abstractions used. Each such abstract domain is then defined by providing (we follow the description in (14; 19)): a number of basic domain operations \((\subseteq, \cap, \cup, \bot)\), \(\cap \), and, optionally, the widening \( \nabla \) operator; the abstract semantics of the primitive constraints (representing the built-ins, or basic operations of the source language) via transfer functions \( (f^o) \); and the following additional instrumental operations over abstract substitutions:

- \( \text{Aproj}(\lambda, V) \): restricts \( \lambda \) to the set of variables \( V \).
- \( \text{Aextend}(A_{k,n}, \lambda^p, \lambda^s) \): propagates the success \( \lambda^s \), defined over the variables of the literal \( A_{k,n} \), to \( \lambda^p \), that includes all the variables of the clause \( k \) of \( A \).
\[ \text{main}(\text{Msg}, P) :\]
\[ \text{par}(\text{Msg}, 0, P). \]
\[ \text{par}([1], P). \]
\[ \text{par}([\text{C}[\text{Cs}], 0], P) :\]
\[ \text{xor}(\text{Cs}, P, 1). \]
\[ \text{par}(\text{Cs}, P, 1). \]
\[ \text{xor}(0, 0, 0). \]
\[ \text{xor}(0, 1, 1). \]
\[ \text{xor}(1, 0, 1). \]
\[ \text{xor}(1, 1, 0). \]

Fig. 1. A set of CHCs for computing parity (left) and a possible analysis graph (right).

- **Acall** \((A, \lambda, A_k)\): performs the abstract call. That is, the unification (conjunction) of a call in a literal \((A, \Lambda)\) with the head of a clause, \(A_k\). The result is a new substitution in terms of the variables of clause \(k\) of \(A\).

- **Aproceed** \((A_k, \lambda^1_k, A)\): performs the abstract proceed. That is, the reverse operation of **Acall**. It unifies the head of the clause \((A_k)\) and the abstract substitution at the end of the clause \((\lambda^1_k)\) with the original call \(A\) to produce the success substitution over the variables of \(A\).

- **Ageneralize** \((\lambda, \{\lambda_i\})\): joins \(\lambda\) with the set of abstract substitution \(\{\lambda_i\}\), all of them over the same variables. The result is an abstract substitution that is greater than or equal to \(\lambda\). It either returns \(\lambda\) when no generalization is needed, performs the least upper bound \((\sqcup)\), or performs the widening \((\nabla)\) of \(\lambda\) together with \(\{\lambda_i\}\), depending on termination or performance needs.

### 2.3 Combined Abstract Domains

A common practice in program analysis is performing a combination of more basic types of information. The idea of combining abstract domains to gain precision is present in (10) where the authors show that precision can be gained by removing redundancies and introducing new basic operations. In the context of abstract domains for logic programs (8) already showed how it is possible to benefit from combinations of previously defined analyses, obtaining a high degree of precision. A large number of examples can be found in this context: groundness and sharing, modes and types, sharing and freeness, etc.

**Direct product.** Let \(E\) be a concrete domain and \((E, \alpha_i, D_i, \gamma_i), i \in \{1, \ldots, n\}\) be Galois insertions. The direct product domain is a quadruple \((E, \alpha, D, \gamma)\) where \(D = \prod_{i=1}^n D_i\) and \(\gamma : E \to D\) such that \(\gamma(d_1, \ldots, d_n) = \gamma_1(d_1) \cap \cdots \cap D_n \gamma_n(d_n)\) and \(\alpha : E \to D\) where \(\alpha(e) = (\alpha_1(e), \ldots, \alpha_n(e))\).

However the direct product domain is not a Galois insertion as shown in (8).

**Reduced product analysis.** Let \(E\) be a concrete domain, \((E, \alpha_i, D_i, \gamma_i), i \in \{1, \ldots, n\}\) Galois insertions, and \((E, \alpha_x, D_x, \gamma_x)\) their corresponding direct product. Consider the relation \(\equiv \subseteq D_x \times D_x\) defined by \(d \equiv d' \iff \gamma_x(d) = \gamma_x(d')\). The reduced product domain is a quadruple \((E, \alpha, \Delta, \gamma)\) where \(\alpha : E \to \Delta\) is such that \(\alpha(e) = [\alpha(e)]_\equiv\) and \(\gamma : \Delta \to E\) such that \(\gamma([d]_\equiv) = \gamma_x(d)\). Let \(\mu : E \to E\) be a concrete function and \(\mu_i : D_i \to D_i\) for \(i \in \{1, \ldots, n\}\), its approximation via \(\gamma_i\). The reduced product function, \(\mu : \Delta \to \Delta\) is defined by \(\mu([d]_\equiv) = ([\mu_1(d_1)], \ldots, [\mu_n(d_n)]_\equiv\) where \((d_1, \ldots, d_n) = \sqcap D_i[d]_\equiv\).

### 3 The Approach

The CiaoPP analysis framework provides a rich assertion language (6; 17; 26) that is used for multiple purposes, including reporting static analysis results. In this regard, every abstract domain defines operations for translating the internal representation of abstract substitutions, used during program analysis, into assertions, which express analysis information as conjunctions of (a wide range of) properties. Such properties are predicates, typically written in (subsets of) the source language. To ease the definition of
new domains, we propose defining the domain operations by using elements of this higher-level assertion language, closer to the developer. However, we need to extend such a language for this purpose, by developing what we call abstract-domain rules language.

3.1 Abstract-domain Rules

Given $s + 1$ sets of constraints, $\mathcal{L}, \mathcal{C}_1, \ldots, \mathcal{C}_s$, we define $\text{AND}(\mathcal{L}, \mathcal{C}_1, \ldots, \mathcal{C}_s)$ as the set of rules of the form:

\[
l_1, \ldots, l_n \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m \neq \text{label}
\]  

(1)

where $s$, $n$, $m$ and $l$ are arbitrary positive integers, and the rule meets the following condition:

$$\forall i, j, k \text{ s.t. } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \text{ and } k \in \{1, \ldots, l\} : (l_i, r_j \in \mathcal{L} \text{ and } \exists u \in \{1, \ldots, s\} \text{ s.t. } g_k \in \mathcal{C}_u)$$

The elements $l_1, \ldots, l_n$ constitute the left side of the rule, $r_1, \ldots, r_m$, the right side, and $g_1, \ldots, g_l$ the guards. For simplicity we usually refer to $\text{AND}(\mathcal{L}, \mathcal{C}_1, \ldots, \mathcal{C}_s)$ as $\text{AND}_\mathcal{L}$, since generally we are more interested in the $\mathcal{L}$ set and the other sets are implicit in the guards. The application of the set of rules $\text{AND}_\mathcal{L}$ over the set $\mathcal{L}$ is a mapping:

\[
\text{AND}_\mathcal{L} : \mathcal{L} \times \mathcal{C}_1 \times \cdots \times \mathcal{C}_n \rightarrow \mathcal{L} \times \mathcal{C}_1 \times \cdots \times \mathcal{C}_n
\]  

(2)

Given $s + t$ sets of constraints $\mathcal{L}_1, \ldots, \mathcal{L}_t, \mathcal{C}_1, \ldots, \mathcal{C}_s$ such that $\forall v \in \{1, \ldots, t\} : \mathcal{L}_v \subseteq \mathcal{L}$, we define $\text{OR}(\mathcal{L}, \mathcal{C}_1, \ldots, \mathcal{C}_n)$ as set of rules of the form:

\[
l_1; \ldots; l_n \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m \neq \text{label}
\]  

(3)

where $s$, $t$, $n$, $m$ and $l$ are arbitrary positive integers, and the rule meets the following condition:

$$\forall i, j, k \text{ s.t. } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \text{ and } k \in \{1, \ldots, l\} :$$

$$\exists v \in \{1, \ldots, t\} \exists u \in \{1, \ldots, s\} \text{ s.t. } (l_i \in \mathcal{L}_v, r_j \in \mathcal{L} \text{ and } g_k \in \mathcal{C}_u)$$

As in the previous case, we usually refer to this rule language as $\text{OR}_\mathcal{L}$. The application of the set of rules $\text{OR}_\mathcal{L}$ over the set $\mathcal{L}$ is a mapping:

\[
\text{OR}_\mathcal{L} : \mathcal{L}^t \times \mathcal{C}_1 \times \cdots \times \mathcal{C}_n \rightarrow \mathcal{L} \times \mathcal{C}_1 \times \cdots \times \mathcal{C}_n
\]  

(4)

The operational meaning of these rules is similar to that of rewriting rules. If the left side holds in the set where the rule is being applied to, and the guards also hold, then the left side elements are replaced by the right side elements.

In the context of abstract interpretation and for the rest of this paper the sets of constraints that we have mentioned have to be seen as abstract domains being the rules applied then over abstract substitutions/constraints. AND-rules are intended to capture the behaviour of operations over one abstract substitution with the knowledge that can be inferred from other substitutions that meet the guards. This allows depicting the greatest lower bound, for instance. Moreover, these rules can also be used to refine such abstract substitution, capture the abstractions and combine different abstract domains, as we will show later. On the other hand, OR-rules are intended to capture the behaviour of operations applied over multiple abstract substitutions of an abstract domain, such as the least upper bound or the widening.

Since these rules are defined to work over sets of elements in a lattice, we consider that if a variable is not present in the set this variable is abstracted to the top element in the lattice. Additionally, since the guards can refer to different elements in various lattices, they will be encapsulated to explicitly indicate the lattice to which they are referring. For example, a polyhedra domain may abstract properties like $X=A$ where $A$ is an arithmetic formula while, other may abstract the same property with $X=\text{any term}$.  

3.2 Rule-based Combination of Abstract Domains

We now sketch how the AND-rules described in the previous section can be used to implement a combination of domains. The same technique can also be used to define abstract domains in a rule-based manner.

As we saw in Section 2.3, the direct product of two analysis is not a Galois insertion and a reduction is needed. However if the reduction exists, it is possible to obtain a reduced product analysis computing at
each analysis step the direct product and then applying the reduction predicate. Of course more complex combinations can be done but let us keep it simple.

In this case the abstract domain operations that we introduced in Section 2.2 for the combination of abstract domains $D_1, \ldots, D_n$ would be defined as follows, where $\lambda = (\lambda_1, \ldots, \lambda_m)$, and each $\lambda_i$ is the abstract substitution of a domain $D_i$ whose lattice is $L_i$. In general the internal representation of the abstract substitution is not a conjunction of properties but there always exists a transfer function from the internal representation to a conjunction representation. In the rest of the paper, when referring to an abstract substitution we will always consider it in its conjunctive representation.

$\text{AReduce}(\lambda_1, \ldots, \lambda_m)$ is the reduction function, which reduces each abstract substitution $\lambda_i$, $i \in \{1, \ldots, m\}$ by using the information in the rest of substitutions $\lambda_1, \ldots, \lambda_i-1, \lambda_i+1, \ldots, \lambda_m$.

In our context, if each abstraction $\lambda_i$ belongs to a lattice $L_i$ there is a set of $\text{AND}$-rules describing the operator $\text{AND}(L_1, E_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m)$ then:

$$\text{AReduce}(\lambda_1, \ldots, \lambda_m) = (\text{AND}(L_1, \text{AReduce}(\lambda_1, \lambda_2, \ldots, \lambda_m), \ldots, \text{AND}(L_m, \lambda_{m-1}, \lambda_m))$$

Using this operation, we redefine/generalize the abstract domain operations in Section 2.2 as follows:

- $\text{Aproj}(\lambda, Vs) = (\text{Aproj}_{D_1}(\lambda_1, Vs), \ldots, \text{Aproj}_{D_m}(\lambda_m, Vs))$
- $\text{Extend}(A_{k,n}, \lambda_p^k, \lambda^k) = \text{AReduce}(\text{Extend}_{D_1}(A_{k,n}, \lambda_1^k, \lambda_1^k), \ldots, \text{Extend}_{D_m}(A_{k,n}, \lambda_m^k, \lambda_m^k))$
- $\text{Acall}(A, \lambda, A_k) = \text{AReduce}(\text{Acall}_{D_1}(A_{k,1}, \lambda_1, \lambda_k), \ldots, \text{Acall}_{D_m}(A_{k,1}, \lambda_m, \lambda_k))$
- $\text{Aproceed}(A_{k,1}, \lambda^k, A) = \text{AReduce}(\text{Aproceed}_{D_1}(A_{k,1}, \lambda_1^k, \lambda_k), \ldots, \text{Aproceed}_{D_m}(A_{k,1}, \lambda_m^k, \lambda_k))$
- $\text{Ageneralize}(\lambda, \{\lambda_i\}) = \text{AReduce}(\text{Ageneralize}_{D_1}(\lambda_{1,1}, \{\lambda_{1,1}\}), \ldots, \text{Ageneralize}_{D_m}(\lambda_{m,1}, \{\lambda_{m,1}\}))$

The built-ins are abstracted in each of the domains $D_1, \ldots, D_n$ and then captured in the combined abstract substitution with the application of $\text{AReduce}$. It is possible to define the abstraction predicates of given built-ins if needed. Unification is abstracted in the same way as it is done with the built-ins.

3.3 Rule-based Abstract Domains in PLAI

The approach proposed to derive abstract domains using rewriting rules is based on the combination of abstract domains presented in the previous section. The generation of a domain $D_1$ with lattice $L$ can be actually seen as a domain combination of that domain with other domains $D_2, \ldots, D_n$ with two major differences: (i) the abstraction of $D_1$ is obtained directly from the information inferred from the other domains and (ii) after some operations as the least upper bound or the widen $\text{OR}$-rules are applied over an "or-abstraction". In these cases the domain operations are defined as:

- $\text{Aproj}(\lambda, Vs) = (\text{Aproj}_{D_1}(\lambda_1, Vs), \ldots, \text{Aproj}_{D_m}(\lambda_m, Vs))$
- $\text{Extend}(A_{k,n}, \lambda_p^k, \lambda^k) = \text{AND}_L(\text{Extend}_{D_1}(A_{k,n}, \lambda_1^k, \lambda_1^k), \ldots, \text{Extend}_{D_m}(A_{k,n}, \lambda_m^k, \lambda_m^k))$

where $\text{Extend}_{D_j}$ for $j$ in $\{2, \ldots, n\}$ is the corresponding $\text{Extend}$ operation for the domain $D_j$ and $\text{Extend}_{D_1}(A_{k,n}, \lambda_1^k, \lambda_1^k) = \lambda_1^k \cup \lambda_1^k$.
- $\text{Acall}(A, \lambda, A_k) = \text{AND}_L(\text{Acall}_{D_1}(A_{k,1}, \lambda_1, \lambda_k), \ldots, \text{Acall}_{D_m}(A_{k,1}, \lambda_m, \lambda_k))$ where $\text{Acall}_{D_j}$ for $j$ in $\{2, \ldots, n\}$ is the corresponding $\text{Acall}$ operation for the domain $D_j$ and $\text{Acall}_{D_1}(A_{k,1}, \lambda_1, \lambda_k)$ performs the unification of $\langle A, \lambda \rangle$ and $A_k$. It projects the corresponding variables in the head, send the free variables to top and merge the substitutions when $A$ and $A_k$.
- $\text{Aproceed}(A_{k,1}, \lambda_1^k, A) = \text{AND}_L(\text{Aproceed}_{D_1}(A_{k,1}, \lambda_1^k, A), \ldots, \text{Aproceed}_{D_m}(A_{k,1}, \lambda_m^k, A))$

where $\text{Aproceed}_{D_j}$ for $j$ in $\{2, \ldots, n\}$ is the corresponding $\text{Aproceed}$ operation for the domain $D_j$ and $\text{Aproceed}_{D_1}(A_{k,1}, \lambda_1^k, A)$ project over the variables in the head (of $A_k$) and unifies the head at the end of $\lambda_1^k$ with $A$.
- $\text{Ageneralize}(\lambda, \{\lambda_i\}) = \text{AND}_L(\text{OR}_L(\lambda_{1,1}, \{\lambda_{1,1}\}), \text{Ageneralize}_{D_2}(\lambda_{2,1}, \{\lambda_{2,1}\}), \ldots, \text{Ageneralize}_{D_m}(\lambda_{m,1}, \{\lambda_{m,1}\}))$

3.4 Revisiting the Depth-k Abstraction

Depth-k abstract terms were introduced by Sato and Tamaki in (29). We present here the basics of this domain; full details can be found in the cited paper.

Definition 1 (level and subterm).
Given a term $t$, $t$ is a level 0 subterm of $t$.

If $f(t_1, \ldots, t_n)$ is a subterm of $t$ with level $k$ then each $t_i$, $i \in \{1, \ldots, n\}$ has level $k + 1$ and its said to be a level $k + 1$ subterm of $t$.

**Definition 2 (depth-$k$ abstraction).** Given a term $t$ and an integer $k$, the result of replacing every level $k$ subterm of $t$ by a newly created variable is called the $k$-term abstraction of $t$. Given an expression $E$ the result of replacing every term in $E$ with its $k$-term abstraction is called the $k$-term abstraction of $E$.

**Example 2.** Given a term $t = f(g(x, y), z)$ the $k$-term abstractions or depth-$k$ abstractions for $k \in \{0, 1, 2, 3\}$ are:

- For $k = 0$ the 0-term of $t$ is $u$ where $u$ is a new variable.
- For $k = 1$ the 1-term of $t$ is $f(u, v)$ where $u, v$ are new variables.
- For $k = 2$ the 2-term of $t$ is $f(g(u, v), z)$ where $u, v$ are new variables.
- For $k = 3$ the 3-term of $t$ is $f(g(x, y), z)$ which is $t$.

Note that a depth-$\infty$ abstraction is always equal to the element being abstracted.

In depth-$k$ analyses, the value of $k$ can be chosen for each execution of the domain. The appropriate $k$-depth that allows obtaining the desired structural information at a given program point is in general program-dependent. Furthermore, there are programs for which no finite $k$ depth can capture their complete meaning (thus the usefulness of, e.g., regular term abstractions). However, for many programs, relatively small values of $k$ (e.g., depth-3) can often produce useful results in practice, when compared to not keeping any structural information. For simplicity in the following we will consider that the most precise $k$ is being used.

**Definition 3 (Lattice operations).** The order relation over the depth-$k$ lattice is defined as follows:

$$t_1 \leq t_2 \Leftrightarrow \text{instance}(t_1, t_2)$$

The least upper bound is defined as the most specific generalization of two terms.

### 3.5 Motivating example: the Bit Domain

As we saw in Example 1 the formal definition of an abstract domain can be relatively straightforward. However actually implementing an abstract domain is often more challenging. In this case, we aim to derive an abstract domain that can capture whether an element (in the concrete domain) is zero, one, a bit, or anything, and also capture errors related with this property. Furthermore, we will also aim to increase the precision by combining the above with a depth-$k$ analysis. In order to correctly capture this our analysis should meet the following conditions:

- If a unification $X = 0$ is encountered, then $X$ should be abstracted as zero($X$).
- If a unification $X = 1$ is encountered, then $X$ should be abstracted as one($X$).
- If a variable has been abstracted to bot and to any other element of the lattice, then it has to be kept as bot.
- If a variable has been abstracted to top and to any other element of the lattice, different from bot, then it has to be kept as top.
- If a variable $X$ has been abstracted to zero($X$) and to one($X$), then it has to be abstracted to bot($X$).
- If a variable $X$ has been abstracted to bit($X$) and to one($X$) or zero($X$), then it has to be kept as bit($X$).

The existence of unifications would be given by a depth-$k$ abstraction. That means that this domain would be obtained by combining with depth-$k$. The rules in Figure 2 encode the zero/one/bit domain by using the previously defined rule language (we assume the same lattice as the one defined in Example 1).

The abs_z and abs_o rules are capturing the behavior of the abstraction function while bit1, bit2 and bot are propagating the information of the abstract substitution. The fail in the bot rule sends the abstract substitution to $\bot$.

The guards of abs_z and abs_o try to capture whether a variable is being unified to zero or one in the depth-$k$ abstract substitution. Now our lattice would be the product between the lattice \{bit, zero, one, top, bot\} and the lattice of the depth-$k$ terms. The order in this lattice is $(a, b) \leq (x, y) \Leftrightarrow a \leq x \land b \leq \text{depth}_k y$. Then the abstraction function $\alpha$ is obtained by applying the set of rules previously defined to the abstract substitution obtained applying $\alpha_{\text{depth}_k}$.
Let \( R_{\text{bit}} \) be the set of rules defined in Figure 2. Then \( AND_{\text{bit}}([top(A), \, top(B), \, top(C)]), \{A = 0, \, B = 0, \, C = 1\} \) = \( \{\{zero(A), \, zero(B), \, one(C)\}\} \) by applying the rules \( \text{abs}_z \) over \( (\text{top}(A), \, A=0) \) and \( (\text{top}(B), \, B=0) \) resp. and \( \text{abs}_o \) over \( (\text{top}(C), \, C=1) \). In a similar fashion we get \( AND_{\text{bit}}([top(A), \, top(B), \, top(C)]), \{A = 1, \, B = 0, \, C = 1\} \) = \( \{\{one(A), \, zero(B), \, one(C)\}\} \) which abstract the cases \( \text{xor}(0, \, 0, \, 1) \) and \( \text{xor}(1, \, 0, \, 1) \) of the predicate \( \text{xor}/3 \) presented in Figure 1.

We have been able to obtain two abstractions for two possible cases. The following step would be to merge the information obtained in those two cases and obtain a more generic abstract substitution capturing that information. We need then to define some rules for the least upper bound (lub) of the bit domain. The behavior of the least upper bound is:

- If a variable can be zero in one abstraction and one in the other then it must be abstracted to bit.
- If a variable is abstracted to bot in any abstraction it must be kept as bot.
- If a variable is abstracted to top in one abstraction and to another element in the other abstraction it must be abstracted to the non-top abstraction.
- If a variable is abstracted to the same value in both abstractions it must be kept the same.

These rules are captured by \( \text{lub}_1 \), \( \text{lub}_2 \), \( \text{lub}_3 \), and \( \text{lub}_4 \) in Figure 2.

Example 3. The easy one. Is clear that \( \text{qs}(\text{[]} \, , \, \text{[]}) \) make free the sorting list property for \( \text{[]} \) and that and empty list is a permutation of itself.

Example 4. The least upper bound of the abstract substitutions obtained before is computed as follows:

1. Compute the lub of the depth-k abstractions \( \{A = 0, \, B = 0, \, C = 1\} \cup \{A = 1, \, B = 0, \, C = 1\} = \{A = A1, \, B = 0, \, C = 1\} \)
2. Compute \( \text{OR}_{\text{bit}}\{\{\text{zero}(A), \, \text{zero}(B), \, \text{one}(C)\}\}, \{\text{one}(A), \, \text{zero}(B), \, \text{one}(C)\}\}, \{A = A1, \, B = 0, \, C = 1\} \) = \( \{\text{bit}(A), \, \text{zero}(B), \, \text{one}(C)\}\}, \{A = A1, \, B = 0, \, C = 1\} \) by applying \( \text{lub}_1 \) over \( \{\text{zero}(A), \, \text{one}(A)\}\) and \( \text{lub}_2 \) over \( \{\text{zer}(B), \, \text{zero}(B)\}\) and \( \{\text{one}(C), \, \text{one}(C)\}\) resp.

4 Inferring program properties using rule-based combined domains

In the previous section we presented the overall approach that we followed to easily encode abstract domains using an abstract substitution representation based on a conjunction of properties. We also illustrated the approach with the encoding of a very simple bit domain, combined with a depth-k abstraction. We now turn our attention to our main overall objective: easing the process of deriving domains for new, possibly program-specific, high-level properties, using rule-based combined domains.

To this point, in this section we will show how the techniques presented before can be used to derive a combined domain that allows the analyzer to infer and verify some complex properties of a classical implementation of the quick sort algorithm in Prolog (Figure 3). To do so we will define some relational domains capturing whether a variable is or not the infimum or the supremum of a list, some basic multiset relations, and (of course) the sorted property.

More concretely, given a query \( \text{qs}(\text{A} \, , \, \text{B}) \) we aim to prove that:

- \( \text{B} \) is a sorted list.
- The elements in \( \text{B} \) are exactly the elements in \( \text{A} \) (with the same repetitions). This is, \( \text{A} \) is a permutation of \( \text{B} \).

To prove these properties let us first depict the steps expected to follow in order to prove them:

i) The easy one. Is clear that \( \text{qs}(\{\}, \, \{\}) \) make free the sorting list property for \( \{\} \) and that an empty list is a permutation of itself.
367 \text{qsort}([], []). \\
368 \text{qsort}([X|L], R) :- \\
369 \text{partition}(L, X, L1, L2), \\
370 \text{qsort}(L2, R2), \\
371 \text{qsort}(L1, R1), \\
372 \text{append}(R1, [X|R2], R).

\textbf{Fig. 3.} Prolog implementation of the quick sort algorithm.

\begin{verbatim}
qsort([], []). qsort([X|L], R) :- partition(L, X, L1, L2), qsort(L2, R2), qsort(L1, R1), append(R1, [X|R2], R).
\end{verbatim}

\section{4.1 Infimum and supremum domain}

Here we not only need to capture the unifications occurring during the program but also the arithmetical constraints. To derive this property, we add to the combination a polyhedra domain based on (2).

Consider the following set of rules for the infimum domain:

\begin{verbatim}
inf(X, top) | X = []  => inf(X, 0.Inf). # empty
inf(L, X) | L = [H|T]  => inf(T, X). # list_const1
inf(T, X) | L = [H|T], H < X  => inf(L, H). # list_const2
inf(L, X) | L = S      => inf(S, X). # unif_prop
inf(L, X) | Y <= X     => inf(L, Y). # reduction
inf(X, A) ; inf(X, B) | A <= B  => inf(X, A). # lub_1
inf(X, A) ; inf(X, B) | A > B   => inf(X, B). # lub_2
\end{verbatim}

And the equivalent ones for the supremum domain:

\begin{verbatim}
sup(X, top) | X = []  => sup(X, +0.Inf). # empty
sup(L, X) | L = [H|T]  => sup(T, X). # list_const1
sup(T, X) | L = [H|T], H >= X => sup(L, T). # list_const2
sup(L, X) | L = S      => sup(S, X). # unif_prop
sup(L, X) | Y > X     => sup(L, Y). # reduction
sup(X, A) ; sup(X, B) | A <= B  => sup(X, A). # lub_1
sup(X, A) ; sup(X, B) | A > B   => sup(X, B). # lub_2
\end{verbatim}

Where $+0.\inf$ is an abstraction of $\pm \infty$ for the integer numbers. Notice that these rules require both knowledge about the structure and about arithmetical relations that we obtain via depth-$k$ and polyhedra respectively.

With these substitutions only one rule can be applied to $U_1$ and none to $U_2$.

\begin{equation}
U_1 \longrightarrow_{\text{empty}} \{(\text{inf}(T, 0.\text{Inf}))\}, \{T = []\}, \{\}\end{equation}
Applying now the \texttt{lub} rules over the two previous equations gives us:

\[
\frac{((\inf(T, 0.\text{Inf})), \{T = [], \}) \cup U_2 =
\frac{((\inf(T, 0.\text{Inf})), \{T = [], L = [H|T], \} \rightarrow \text{list-const2}}
\frac{((\inf(T, 0.\text{Inf})), \{T = [], L = [H|T], \} \rightarrow \text{reduction}}
\frac{((\inf(T, 0.\text{Inf})), \{T = [], L = [H|T], \} \rightarrow \text{reduction}}
\frac{((\inf(T, 0.\text{Inf})), \{T = [], L = [H|T], \} \rightarrow \text{reduction}}
\text{sup}(L_1, X) \quad \text{inf}(L_2, X). \quad \text{The complete analysis of}
\text{the \texttt{partition/4} predicate can be found in Appendix A.}

4.2 Set properties

It is very usual in many verification tasks to find the need for talking about properties of multisets.
An also a very common need is, given a function (or a predicate in the context of logic programming)
operating over sets or lists, to be able to verify that a new list shares all its elements with the older one.
In this section we show how to derive a domain which abstracts lists to multisets of their elements.

\textbf{Definition 4.} A \textit{multiset} $M$ over a set $A$ is a function from $A$ to the set of natural numbers. This is, a
set with repeated elements. Given an element $x \in A$ we say that $M(x)$ is the number of copies of $x$ in $M$.

With some abuse of notation we will use the usual set operations to define the operations over multiset.
For example given the multisets $M = \{a, b, b\}$ and $N = \{a, c\}$ the union will take into account the
multiple occurrences of each element $M \cup N = \{a, a, b, b, c\}$. Notice that a list can be seen as a multiset.
Now, given multisets $M$, $N$, $S$ over a set $A$ the following properties hold:

\begin{align*}
- M \subseteq N \iff M(x) \leq N(x) \forall x \in A \\
- S = M \cup N \iff S(x) = M(x) + N(x) \forall x \in A \\
- S = M \setminus N \iff S(x) = M(x) - N(x) \forall x \in A
\end{align*}

where $x - y$ is $x - y$ if $x \geq y$ and 0 otherwise. These equivalences allow us to describe multiset properties
as constraint relations with few changes. This approach has been used to prove properties of multisets in
different contexts (see for example (28)). With this approach we try to derive an abstract domain that
abstracts the lists in a program as multisets in order to capture their relations. In the following, with
some abuse of notation, we will use the standard mathematical operations, and $M = N + S$ will denote
that $M(x) = N(x) + S(x) \forall x \in A$, where $A$ will be the set of all the elements contained in the union of
$M$, $N$, and $S$ (with no repetitions). These constraints will be encapsulated inside an \texttt{mset} term to denote
that they are abstracting multiset operations and avoid confusing them with the polyhedra constraints.
The empty multiset corresponds to zero because $\forall x \in A \: M(x) = 0$ for any set $A$.

The main difference with the derivation of this domain is that this time we will take advantage of \texttt{CiaoPP}'s
polyhedra\texttt{CLPQ} domain (an implementation of polyhedra using \texttt{CLP(Q)}) and run its least upper bound
obtaining a very precise operation with almost no cost on the implementation side only having to worry
in the abstraction side. We also use the polyhedra projection as it is slightly more precise. Figure 4 shows
the result of analyzing the \texttt{partition/4} predicate with the \texttt{mset} analysis derived before (the analysis
results are contained in the \texttt{true/1} program-point assertions).

4.3 The sorting property

As mentioned before, in order for the analyzer to prove sortedness it needs to prove first certain auxiliary
properties. Specifically, to define the rules of the sorting property we need to use the previously defined
\textit{infimum} domain together with the \textit{depth-k} and \textit{polyhedra} domains.
Consider the abstract substitution obtained from the analysis of the first clause of \texttt{qsort/2}.

$$U_1 = \{\text{sorted}(A), \text{sorted}(B), \{A = [], B = []\}, \{\inf(A, 0.\text{Inf}), \inf(B, 0.\text{Inf})\}, \{\sup(A, -0.\text{Inf}), \sup(B, -0.\text{Inf})\}, \{\text{mset}(A = B)\}$$

Let us now try to analyze the second clause:

$$\begin{align*}
\text{sorted}(A) & \Rightarrow \text{sorted}(A) \quad \text{# abs\_empty} \\
\text{sorted}(B) & \Rightarrow \text{sorted}(B) \quad \text{# abs\_unif} \\
\text{sorted}(T) & \Rightarrow \text{sorted}(T) \quad \text{# sort\_prop} \\
\text{sorted}(L) & \Rightarrow \text{sorted}(L) \quad \text{# sort\_const}
\end{align*}$$

To prove the desired properties of \texttt{qsort/2} we first need to be able to prove that given a query \texttt{append(A, B, C)} if \texttt{sorted(A)}, \texttt{sorted(B)} then \texttt{sorted(C)} and that \texttt{mset(A+B=C)}. Proving it requires to infer that the elements in A are smaller or equal than the elements in B, which can be done using the infimum and supremum domains previously defined. So far we have inferred that \texttt{inf(L2, X)} and \texttt{sup(L1, X)}, and we need some extra rules to obtain \texttt{inf(R2, X)} and \texttt{sup(R1, X)}:
Consider the goal `append(R1, S, R)` together with the information inferred on call `\{\text{sorted}(R1), \text{sorted}(R2)\}`, `\{A = [X|L], S = [X|R2]\}`, `\{\text{inf}(L, X), \text{inf}(R2, X)\}`, `\{\text{sup}(L1, X), \text{sup}(R1, X)\}`, `\{\text{mset}(L = L1 + L2), \text{mset}(L1 = R1), \text{mset}(L2 = R2)\}`

In the first case we get the unifications: 

```
C1 = \{(R1 = [\], S = Z), \{\text{sorted}(R1), \text{sorted}(S), \text{sorted}(Z)), \{\text{inf}(S, X)), \{\text{sup}(R1, X)\}\}
```

In the second clause if we make a recursive call with the first case we would know that:

```
\{\text{sorted}(R1), \text{sorted}(S), \{\text{inf}(S, X)), \{\text{sup}(R1, X)\}\}
```

However, after `append(X, Y, Z)` we have to apply the least upper bound and we get:

```
\{(\), \{\text{sorted}(R1), \text{sorted}(S), \{\text{inf}(S, X)), \{\text{sup}(R1, X)\}\}
```

This is not proving the property. This is because we need to infer that `R` is sorted. Consider the again `append/3`:

```
append([], Z, Z).
append([T|T], S, [B|Z]) :-
append(T, S, Z).
```

In the second clause if we make a recursive call with the first case we would know that:

1. \(T, S\) and \(Z\) are sorted lists;
2. \(\text{inf}(S, X)\) and \(\text{inf}(Z, X)\);
3. \(\text{sup}(R1, X)\) and \(\text{sup}(T, X)\);
4. \(R1 = [H|T], R = [H|Z]\).

However, to prove that \(R\) is sorted we also need to capture that \(H \leq X\). We infer this with a new rule purely combinatorial rule to gain precision in PolyhedraCLPQ:

```
true | \text{sup}(L, X), L = [H|T] \Rightarrow H \ll X . # \text{combine}_\text{poly}
```

Now with this new information we can infer that since \(H \leq X\) then \(\text{inf}(Z, H)\). With this since \(Z\) is sorted and \(R = [H|Z]\) then \(R\) is sorted and \(\text{inf}(R, H)\).

Thus, the domain described by the rules above are enough to proves that if \(\text{sorted}(A), \text{sorted}(B)\) then \(\text{sorted}(C)\) and therefore that the second argument of \(\text{qsort/2}\) is sorted and the elements of both lists are the same (the program point info for \(\text{qsort/2}\) can be found in Appendix B):

```
\text{true pred} \text{qsort}(A,B) = (\text{asub}([s(A,\text{top}),s(B,\text{top})]), \text{asub}([[\text{inf}(A,\text{top}),\text{inf}(B,\text{top})],[\text{sup}(A,\text{top}),\text{sup}(B,\text{top})]]), \text{true}, \text{true}) \Rightarrow (\text{asub}([s(A,\text{top}),s(B,\text{sorted})]), \text{asub}([[\text{inf}(A,\text{top}),\text{inf}(B,\text{top})],[\text{sup}(A,\text{top}),\text{sup}(B,\text{top})]]), \text{mset}([A=B]), \text{true}).
```

**Variable Scope.** It is important to point out that we have to take into account that the scope of the variable \(X\) that we carry around in the abstract substitutions goes beyond the argument and local variables of `append/3` In order to not lose precision, the domain projection operation must preserve the relation between program variables and \(X\) in the form of “existential” variables. That is, to capture that “there exists a variable such that hold a given property” (for example the \(\text{inf}\) property). These kind of problems are common when trying to capture properties of data structures, and they are particularly problematic when combining domains, since a projection for one domain must be aware of the relevant variables for the others.

In our current implementation, we rely on syntactic transformations that includes "extra" arguments to the required predicates (see the call to `append/4` in Appendix B), similar to cell morphing proposed in (3). We are working on fixing this limitation and extending the combination framework to share the information about “existential” variables among the combined domains without syntactic transformations nor loses of precision.

## 5 Implementation and Experimental Evaluation

In this section we present an overview of the implementation and some evaluation results for the domains and code presented in the previous sections.
5.1 Implementation

The current prototype is built on top of the standard CiaoPP features for plugging-in new abstract domains. It consists in two main programs. One of around 250 lines of Prolog that implements a bridge between the rule-based definitions and the operations described in Section 3.3 generalizing some of them when needed. The second consists in around 500 lines of Prolog implementing an interpreter for the rule language presented in Section 3. There also exists a third file defining a general combination domain where the generalized operations are plugged-in.

Bridge operations. The domain operations bridging between the PLAI operations and the rule-based definitions are defined in a very generic way. In the most of cases, they just generate “top” substitutions, project over a subset of variables or merge some abstract substitutions being all the analysis work in the rules side. The only predicates where this is different are the least upper bound and widen. Which are the ones that made the OR-rules required and also required a different approach. Let us consider the least upper bound which corresponds to generalize in Section 3.3. Given \( n \) abstract substitutions \( \text{ASub}_1, \ldots, \text{ASub}_n \), this operation would generate an instrumental “OR-substitution” represented as an \( n \)-tuple separated by semicolons (\( \text{ASub}_1; \text{ASub}_2; \ldots; \text{ASub}_n \)). This structure would easily let to the rule interpreter know which kind of rules have to be executed.

Rule execution. The AND-rule execution implements a simple fixpoint algorithm that applies rules until a fixed point is reached. As usual in rewrite systems, the user must ensure rule convergence. Both for precision and efficiency, the implementation also supports delegating some operations to ad-hoc operations that directly manipulate the lattice, as well as the combination with non rule-based domains.

Algorithm 1 Or-rewriting algorithm

```
1: procedure OR-REWRITING(Store_1, Store_2, OrRuleSet)
2:   R ← takeApplicableRule(Store_1, Store_2, OrRuleSet)
3:   LoopCond ← isRule(R)
4:   RewStore ← ∅
5:   while LoopCond do
6:       (MatchElemsSt_1, MatchElemsSt_2, RewElems) ← applyRule(R, Store_1, Store_2)
7:       Store_1 ← Store_1 \ MatchElemsSt_1
8:       Store_2 ← Store_2 \ MatchElemsSt_2
9:       RewStore ← RewStore ∪ RewElems
10:      if Store_1 = Store_2 then
11:          LoopCond ← false
12:      else
13:          R ← takeApplicableRule(Store_1, Store_2, OrRuleSet)
14:          LoopCond ← isRule(R)
15:      Ints ← Store_1 \ Store_2
16:      Diffs ← (Store_1 \ Ints) ∪ (Store_2 \ Ints)
17:      TopInfo ← sendToTop(Diffs)
18:      RewStore ← Ints ∪ TopInfo
```

Definition 5. Given an Abstract Substitution we say that the abstract substitution is in its extended form if and only if for each program variable in the abstract substitution there is only one different element of the Abstract Substitution which captures all the information related to that variable.

Example 5.
- A Depth-k abstract substitution is in its extended form for all k. For example the abstraction for the program variables \([X, Y, Z]\) \([X = A, Y = f(u, A), Z = g(u, A)]\) is extended.
- The set representation of the sharing property is not extended ([\([X, Y], [Z]\)] but can be transformed to a more verbose extended representation \([\text{sh}(X, [Y]), \text{sh}(Y, [X]), \text{sh}(Z, [\])\]) where the property \(\text{sh}(A, \text{ShSet})\) captures all the variables ShSet sharing with A.
Theorem 1. Let $A_{Sub1}, A_{Sub2}$ two abstract substitutions and let $R_{or}$ a set of OR-rules such that:

i) Both abstract substitutions are expressed in its extended form

ii) For all rule $r \in R_{or}$ and all elements $Elems1 \subseteq A_{Sub1}$, $Elems2 \subseteq A_{Sub2}$ such that they hold in $r$

the resulting set of elements $Rew_{Elems}$ is such that $Elems1 \leq Rew_{Elems}$, $Elems2 \leq Rew_{Elems}$

with $\leq$ the order relation in the corresponding lattice.

Then the Abstract Substitution $Rew_{A_{Sub}}$ obtained after applying Algorithm 1 over the Abstract Substitutions $A_{sub1}, A_{sub2}$ with the set of rules $R_{or}$ verify $A_{Sub1} \leq Rew_{A_{Sub}}$ and $A_{Sub2} \leq Rew_{A_{Sub}}$.

Experimental evaluation. To gain confidence in our approach we developed an ad-hoc pilot implementation of the rule-based domains proposed in Section 4 and the bit domain, and bench-marked them analyzing different predicates, as shown in Table 1. None of these domains were previously implemented in the CiaoPP framework. For reference and comparison we also present the times for other standard domains, like polyhedraCLP, PPL, sharing, the sharing-soondergard combination, and eternas. All the benchmarks have been executed on a MacBook Pro machine with Intel Core i5 and 8GB RAM.

While comparing the performance of domains inferring completely different properties may seem meaningless in principle, the point made is that the analysis times indicate that our prototype implementation of the rule-based approach is not prohibitively inefficient. Nevertheless, our implementation efforts for the rule interpreter at this stage have focused on providing correct and easier to derive analysis, lacking many optimization required for scalability of writing systems. For example, given the rule $p(X), p(Y), p(Z) \Rightarrow q(X), q(Y), q(Z)$, the abstract substitution would consist of 5 elements and the size of the search space would be 35. If the abstract substitution consists in 6 elements, the size of the search space would be 56, 84 if 7, etc. (Given by $\binom{m}{n}$ where $m$ is the number of elements of the abstract substitution and $n$ the number of elements on the left side of the rule). Notice that given an abstract substitution of size 6 and a set of 4 rules with 2 elements on the left size in each rule, we will require in the worst case $4 \cdot \binom{6}{2} = 4 \cdot 21 = 84$ evaluations of rules.

For domains that are the result of combining others domainsthe analysis times actually show that the combination overhead is small so we can argue that most of the bottleneck we found is related with the evaluation of the rules.

We believe that the results are promising and encourage us to optimize our approach and the combination framework, define new domains, as well as explore the conciseness/efficiency trade-off of hand-tuned vs. rule-based domains.

6 Conclusions

We have presented a framework for simplifying the development of abstract domains for logic programs in the context of abstract interpretation frameworks, and concretely that of CiaoPP. Our approach leverages
on the encoding and semantics of program properties in the Ciao assertion language and the bidirectional
mapping of such properties as elements of the abstract domains used in CiaoPP, which we have used to
propose logical abstract domains where those operations are defined by means of logic-based rewriting
rules. Moreover, rules can be composed in a stratified way in order to combine and compose several ab-
stract domains. We have shown how these domains are plugged into the PLAI fixpoint, and demonstrated
the power of our approach by defining, with little effort, a collection of domains and their combinations
that (efficiently) infer interesting properties that could not be inferred before by CiaoPP, such as sorted
lists for a sorting algorithm implemented in Prolog.

While some domains are easier to specify with a rule-based language, keeping a constraint-based
representation for abstract substitutions may not be efficient compared with specialized representations
and operations. In this respect, we plan to explore as future work the use of rules both as an input language
for abstract domain compilation and as a specification language for debugging or verifying properties of
hand-written domains.

In our experience, the proposed approach seems promising for prototyping and experimenting with
new domains, enhancing the precision for particular programs, and adding domain combination rules,
without the need for understanding the analysis framework internals.


A Inferred info for partition/4

:- module(_, [partition/4, [assertions]].

:- true pred partition(L, X, L1, L2).  
  ( asub([inf(L, 0, inf, X, X), inf(L1, 1, stop), inf(L2, 1, stop)]), true ) 
  => ( asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]), true ).

partition(L, X, L1, L2) :-
  true((
    asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
    true
  )).

L=[].

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

L1=[].

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

L2=[].

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

X=X.

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).

true((
  asub([inf(L, 1, stop), inf(X, 1, stop), inf(L1, 1, stop), inf(L2, 1, stop)]),
  true
  )).
\[ X = Y, \]
\[ \text{true}(\text{asub}(\text{inf}(L,'\$top'),\text{inf}(X,'\$top'),\text{inf}(L1,'\$top'),\text{inf}(L2,'\$top'),\text{inf}(Y,'\$top'),\text{inf}(\text{Temp}1,'\$top'),\text{inf}(\text{Temp}2,'\$top'))), \]
\[ \text{'terms_check': instance'(L, [Y|\text{Temp}]),} \]
\[ \text{'native_props:constraint'([(Y=X)])} \).
\[ \text{partition}(\text{Temp}, X, L1, L2).} \)
\[ \text{true}(\text{asub}(\text{inf}(L,'\$top'),\text{inf}(X,'\$top'),\text{inf}(L1,'\$top'),\text{inf}(L2,'\$top'),\text{inf}(Y,'\$top'),\text{inf}(\text{Temp}1,'\$top'),\text{inf}(\text{Temp}2,'\$top'))), \]
\[ \text{'terms_check:instance'(L1, [Y|\text{Temp}]),} \]
\[ \text{'native_props:constraint'([(X=X)])} \).
\[ \text{L1=[Y|\text{Temp}2],} \]
\[ \text{true}(\text{asub}(\text{inf}(L,'\$top'),\text{inf}(X,'\$top'),\text{inf}(L1,'\$top'),\text{inf}(L2,X),\text{inf}(Y,'\$top'),\text{inf}(\text{Temp}1,'\$top'),\text{inf}(\text{Temp}2,X)), \]
\[ \text{'terms_check:instance'(L, [Y|\text{Temp}]),} \]
\[ \text{'native_props:constraint'([(Y=X)])} \).
\[ \text{L2=[Y|\text{Temp}2].} \)
\[ \text{:- redefining(partition/4).} \]
\[ \text{B Inferred info for qsort/2} \]

\[ \text{true} \]
qsort(L1,R1),
true(
  asub([s(A,'$top'),s(R,'$top'),s(X,'$top'),s(L,'$top'),s(L1,'$top'),s(L2,'$top'),
s(R2,sorted),s(R1,sorted),s(B,'$top'))],
inf(A,'$top'),inf(R,'$top'),inf(X,'$top'),inf(L,'$top'),inf(L2,X),
sup(A,'$top'),sup(R,'$top'),sup(X,'$top'),sup(L,'$top'),sup(L1,X),
sup(R2,'$top'),sup(R1,X),sup(B,'$top'))],
mset([A=L-X,L1=L-R2,L2=R2,R1=L-R2]),
'terms_check:instance'(A,[X|L]),
true
).
B=[X|R2],
true(
  asub([s(A,'$top'),s(R,'$top'),s(X,'$top'),s(L,'$top'),s(L1,'$top'),s(L2,'$top'),
s(R2,sorted),s(R1,sorted),s(B,sorted))],
inf(A,'$top'),inf(R,'$top'),inf(X,'$top'),inf(L,'$top'),inf(R2,X),inf(R1,'$top'),inf(B,X))],
'terms_check:instance'(B,[X|R2]),
true
).
append(R1,B,R,X),
true(
  asub([s(A,'$top'),s(R.sorted),s(X,'$top'),s(L,'$top'),s(L1,'$top'),s(L2,'$top'),
s(R2,sorted),s(R1,sorted),s(B,sorted))],
inf(A,'$top'),inf(R,'$top'),inf(X,'$top'),inf(L,'$top'),inf(R2,X),inf(R1,'$top'),inf(B,X))],
mset([A=L-X,L1=L-R2,L2=R2,R1=L-R2,B=R2+X]),
'terms_check:instance'(A,[X|L]),
true
))
)