A Methodology for Designing and Composing Abstract Domains Using Rewriting Rules

Daniel Jurjo\textsuperscript{1,2}, Jose F. Morales\textsuperscript{1,2},
Pedro Lopez-Garcia\textsuperscript{1,3}, and Manuel V. Hermenegildo\textsuperscript{1,2}

\textsuperscript{1} IMDEA Software Institute, Spain
\textsuperscript{2} Universidad Politécnica de Madrid (UPM), Spain
\textsuperscript{3} Spanish Council for Scientific Research (CSIC), Spain

Abstract. Abstract interpretation allows constructing sound static analysis tools by safely approximating program semantics. Frameworks for abstract interpretation typically provide an implementation of a specialized iteration strategy to compute an abstract fixpoint, as well as a number of abstract domains in order to approximate different program properties. However, the design and implementation of additional domains, as well as their combinations, is eventually necessary to successfully prove arbitrary program properties. We propose a rule-based methodology for rapid design and prototyping of new domains and combining existing ones, with a focus on the analysis of logic programs. We provide several examples for domains combining numerical properties and data types and apply them to proving complex program properties.

Keywords: Abstract Domain Development, Abstract Domain Combination, Abstract Interpretation, Static Analysis, Logic Programming, Prolog.

1 Introduction

The technique of Abstract Interpretation [10] allows constructing sound program analysis tools which can extract properties of a program by safely approximating its semantics. Abstract interpretation proved practical and effective in the context of (Constraint) Logic Programming ((C)LP) [17,24,25,31,32,39,40], which was one of its first application areas [18], and the techniques developed in this context have also been applied to the analysis and verification of other programming languages by using semantic translation into (Constraint) Horn Clauses (CHCs) [13,19,27]. Frameworks for abstract interpretation (such as

* Partially funded by MICINN projects PID2019-108528RB-C21 ProCode, TED2021-132464B-I00 PRODIGY, and FJC2021-047102-I, and by the Tezos foundation. We also thank the anonymous reviewers for their very useful feedback.
PLAI/CiaoPP [20] or Astrée [12]) provide efficient implementations of algorithms for computing abstract fixpoints as well as several abstract domains, which approximate different program properties. Moreover, due to undecidability [7,10], 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 order to facilitate this task, we propose a rule-based approach for the design and rapid prototyping of new domains, as well as composing and combining existing ones. Our techniques are partially inspired in logic-based languages for implementing constraint domains [14]. We provide several examples for domains combining numerical properties and data types, and apply them to proving complex properties of Prolog programs.

Related work. 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., [4,8,41]) 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 herein, 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 implementation of analyses, such as FLIX [26], a form of Datalog with lattices, which has been used to define several types of analyses for imperative programs, and other work generalizes Datalog with constraints (see again [13]). However, these approaches do not provide per se a specific formalism for defining the abstract domains as in our work. Another promising approach to performing static analysis of complex programs involving algebraic data types is [1], which introduces a transformation technique to convert programs to deal only with basic data types (integers, booleans) that can be handled with other CHC solvers. This method can analyze CHC programs with data types and infer properties like sortedness, but the approach is very different from ours (which works directly on the original program using abstract interpretation). A previous rule-based approach to defining abstract domains was proposed in [29], using Constraint Handling Rules (CHR) [14] but this work only handled conjunctions of constraints in a simple dependency domain, and did not address other fundamental operations such as the least upper bound, nor the application for combining domains. Finally, rewriting systems have also been used to prove the correctness of abstract unifications [5].

2 Preliminaries

Lattices. A partial order on a set \( X \) is a binary relation \( \sqsubseteq \) that is reflexive, transitive and anti-symmetric. The least upper bound (lub) 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 (\( a \sqsubseteq a \sqcup b \land b \sqsubseteq a \sqcup b \)). If it exists, it is unique. Similarly, the greatest lower bound (glb) 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 \( \sqsubseteq \)
is a partial order relation on $X$. $X$ 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 (which are unique). The maximum element of a complete lattice is called **top** and the minimum **bottom** (denoted by $\top$ and $\bot$ resp.).

**Abstract interpretation.** The standard, collecting semantics of a program can be described in terms of a **concrete domain** that contains sets of execution states, 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 denote the concrete domain as $D$ and the abstract domain as $D_\alpha$. We denote the functions that relate sets of concrete substitutions with abstract substitutions as the abstraction function $\alpha : D \to D_\alpha$ and the concretization function $\gamma : D_\alpha \to D$. The concrete domain is typically a complete lattice with the set inclusion order which induces an ordering relation in the abstract domain that we represent by $\sqsubseteq$. Under this relation the abstract domain is usually a complete lattice and $(D_\alpha, \alpha, D, \gamma)$ is a Galois insertion/connection [10].

**The Top-down Algorithm.** Top-down analyses build an **analysis graph** starting from a series of program **entry points**. This approach was first used in analyzers such as MA3 and Ms [40], and matured in the PLAI analyzer [31,32], now also referred to as the **top-down algorithm** or **solver**, using an optimized fixpoint algorithm. It was later applied to the analysis of CLP/CHCs [17] and imperative programs [13,19,27,28], and used in analyzers such as GAIA [25], the CLP(∅) analyzer [24], or Goblint [37,38]. The graph inferred by PLAI 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 concrete trees (the paths through the program) from the abstraction of the **substitutions** at the nodes in those concrete trees (the program states in those paths). The first abstraction ($T_\alpha$) is typically built-in, as an abstract domain of **analysis graphs**. The framework is parametric on a second abstract domain, $D_\alpha$, whose elements appear as labels in the nodes of the analysis 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 over the variables $V_1, \ldots, V_n$. Such tuples represent the set of (concrete) call and success substitutions of the nodes in the concrete AND-trees. A more detailed recent discussion can be found in [13]. Many other PLAI extensions have been proposed, such as incremental and modular versions [15,16,22,34].

**Example 1.** Fig. 1 (from [16]) shows a possible analysis graph (center) for a set of CHCs (left) that encode the computation of the parity of a binary message using the exclusive or, denoted xor. E.g., the parity of message $[1,0,1]$ is $0$. We consider an abstract domain (right) with the following abstract values: $\bot$ s.t. $\gamma(\bot) = \emptyset$, $z$ (zero) s.t. $\gamma(z) = \{0\}$, $o$ (one) s.t. $\gamma(o) = \{1\}$, $b$ (bit) s.t.
\[ \gamma(b) = \{0, 1\}, \] and \( \top \) such that \( \gamma(\top) \) is the set of all concrete values. Consider an initial abstract goal \( G_\alpha = \langle \text{main}(\text{Msg}, P), (\text{Msg}/\top, P/\top) \rangle \), representing 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 \( \gamma(0) \) 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 capture calls and paths. 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).

As mentioned before, the abstract interpretation-based algorithms that we are considering are \textit{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 [15,22]): a number of basic lattice operations \((\sqsubseteq, \sqcap, \sqcup\) and, optionally, the widening \( \nabla \) operator); the abstract semantics of the primitive constraints (representing the \textit{built-ins}, or basic operations of the source language) via \textit{abstract transfer functions} \( f^\alpha \); and the following additional instrumental operations over abstract substitutions:

\begin{itemize}
  \item \text{Aproj}(\lambda, Vs): restricts the abstract substitution \( \lambda \) to the set of variables \( Vs \).
  \item \text{Aextend}(A_{k,n}, \lambda^*, \lambda^s): propagates the abstract success substitution \( \lambda^s \), defined over the variables of the \( n \)-th literal of clause \( k \) \( (A_{k,n}) \), to \( \lambda^c \), which is defined over all the variables of clause \( k \) (which contains \( A_{k,n} \) in its body).
  \item \text{Acall}(A, \lambda, A_k): performs the \textit{abstract call}, i.e., the unification (conjunction) of a literal call \( \langle A, \lambda \rangle \) with the head \( A_k \) of a clause \( k \) defining the predicate of \( A \). The result is a substitution in terms of the variables of clause \( k \).
  \item \text{Aproceed}(A_k, \lambda^s_k, A): performs the \textit{abstract proceed}, i.e., the reverse operation of \text{Acall}. It unifies the head of clause \( k \) \( (A_k) \) and the abstract substitution at the end of clause \( k \) \( (\lambda^s_k) \) with the original call \( A \) to produce the success substitution over the variables of \( A \).
  \item \text{Ageneralize}(\lambda, \{\lambda^1, \ldots, \lambda^s\}): joins \( \lambda \) with the set of abstract substitutions \( \{\lambda^1, \ldots, \lambda^s\} \), all over the same variables. The result is an abstract substitution greater than or equal to \( \lambda \). It either returns \( \lambda \), when no generalization is
needed; performs the least upper bound (⊔); or performs the widening (⊓) of λ together with \{λ₁, ..., λₖ\}, depending on termination, precision, and performance needs.

Note that this general approach and operations are not specific to logic programs, applying in general to a set of blocks that call each other, possibly recursively.

**Combining Abstract Domains.** The idea of combining abstract domains to gain precision is already present in [11], showing that precision can be gained by removing redundancies and introducing new basic operations. Let \( E \) be a concrete domain and \((E, α_i, D_i, γ_i), i \in \{1, ..., n\}\) Galois insertions. The direct product domain is a quadruple \((E, α_x, D_x, γ_x)\) where \( D_x = D_1 \times \cdots \times D_n \), \( γ_x : D_x \to E \) such that \( γ_x((d_1, ..., d_n)) = γ_1(d_1) \cap_E \cdots \cap_E γ_n(d_n) \) and \( α_x : E \to D_x \) where \( α_x(e) = (α_1(e), ..., α_n(e)) \). However the direct product domain is not a Galois insertion, as shown in [9]. Consider a direct product \((E, α_x, D_x, γ_x)\) and the relation \( ≡ Ω D_x \times D_x \) defined by \( d ≡ d' ⇔ γ_x(d) = γ_x(d') \). The reduced product domain is a quadruple \((E, α≡, D≡, γ≡)\) where \( α≡ : E \to D≡ \) such that \( α≡(e) = [α_x(e)]≡ \) and \( γ≡ : D≡ \to E \) such that \( γ≡([d]≡) = gγ≡(d) \). Let \( μ : E \to E \) be a concrete function and \( μ_i : D_i \to D_i \), for \( i \in \{1, ..., n\} \), its approximation via \( γ_i \). The reduced product function, \( μ≡ : D≡ \to D≡ \) is defined by \( μ≡([d]≡) = ([μ_1(d_1)], ..., μ_n(d_n)]≡ \) where \([d]≡ = \cap_i [d_i]≡ \). In [9] a practical approach to such domain combinations is presented, which simplifies proofs and domain implementation reuse. It also shows that it is possible in practice to benefit from such combinations, obtaining a high degree of precision. Many domain combinations are used in the context of logic programs: groundness and sharing, modes and types, sharing and freeness, etc.

### 3 The Approach

**Using property literals/constraints.** The CiaoPP framework includes, in addition to the PLAI analysis algorithm, an assertion language [6,21,33] that is used for multiple purposes, including reporting static analysis results to the user. These results are expressed as assertions which contain conjunctions of literals of special predicates that are labeled as properties, which we also refer sometimes as constraints. An example of such a conjunction is “\( \text{ground}(X) \land Y > 0 \)” where \( \text{ground}/1 \) and \( >/2 \) are examples of properties/constraints. This allows representing the analysis information inferred by the different domains available in the system syntactically as terms, independently of the internal representations used by the domains. Often, the same properties are reused to represent analysis results from domains that infer similar types of information. Also, every abstract domain defines operations for translating from the internal representation of abstract elements into these properties, which thus constitute in practice a common language among domains. A first key component of our approach is to make use of such properties while defining the domain operations.

**Abstract-Domain Rules.** A second component of our approach is a specialized language, which we call Abstract-Domain Rules (ADRs), aimed at easing the
Algorithm 1 AND-rewriting algorithm

1: function AND-rewriting(Store, Context, \(R^\wedge\))
2: \(R \leftarrow \text{ApplicableRule}(Store, Context, R^\wedge)\)
3: if \(R = \text{false}\) then
4: \(\text{return } \text{Store}\)
5: else
6: \((RmEls, NewEls) \leftarrow \text{ApplyRule}(R, \text{Store}, \text{Context})\)
7: \(\text{Store}' \leftarrow (\text{Store} \setminus RmEls) \cup \text{NewEls}\)
8: \(\text{return } \text{AND-rewriting}(\text{Store}', \text{Context}, R^\wedge)\)

process of defining domain operations. It consists of AND- and OR-rules with the following syntax:

1. **AND-rules**:
   
   \[l_1, \ldots, l_n \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m \# \text{label} \]  

2. **OR-rules**:
   
   \[l_1; l_2 \mid g_1, \ldots, g_l \Rightarrow r_1, \ldots, r_m \# \text{label} \]  

where \(l_1, \ldots, l_n, r_1, \ldots, r_m\) are elements of a set of properties \(L\) and each \(g_1, \ldots, g_l\) is an element of \(C_1, \ldots, C_s\), which are also sets of properties. 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*. Intuitively, rules operate on a "store," which contains a subset of properties from \(L\), while checking the contents of \(s\) stores containing properties respectively from \(C_1, \ldots, C_s\) (the "context").

Since the language is parameterized by the sets of properties being used, we will use \(\text{AND}(L, (C_1, \ldots, C_s))\) (resp. \(\text{OR}(L, (C_1, \ldots, C_s))\)) to refer to the language of AND-rules (resp. OR-rules) where the left and right sides are elements of \(L\) and the guards are elements of \(C_1, \ldots, C_s\).

We say that an abstract substitution \(\lambda\) is in extended form iff for each program variable \(x \in \text{vars}(\lambda)\) there is a unique element of \(\lambda\) which captures all the information related to \(x\). The extended form is frequently used in non-relational domains (although usually elements for which there is no information may be not shown).

Example 2. Consider a program with variables \(\{X, Y, Z\}\).

- The abstract substitution \(\{X/i(1, 2), Y/i(0, 1), Z/i(-\infty, \infty)\}\) of the non-relational domain of intervals is in extended form (as usual for this domain).
- The abstract substitutions of the bit domain in Fig. 1 are in extended form.
- The set representation \(([X, Y], [Z])\) of the sharing property \([23, 30]\) is in extended form, but it can be transformed into a (more verbose) extended form \([\text{sh}(X, [Y]), \text{sh}(Y, [X]), \text{sh}(Z, [\ ])]\), where the property \(\text{sh}(A, \text{ShSet})\) expresses that all the variables in \(\text{ShSet}\) share with \(A\).

Let \(L, C_1, \ldots, C_n\) be lattices and \(\text{Context}\) an element of \(\mathcal{P}(C_1) \times \cdots \times \mathcal{P}(C_n)\) where \(\mathcal{P}(S)\) denotes the powerset of \(S\). We also assume that abstract substitutions are in extended form representation.

AND semantics. Let \(\mathcal{R}^\wedge \subseteq \text{AND}(L, (C_1, \ldots, C_n))\) and \(\text{Store} \subseteq L\). The operational meaning of applying the set of rules \(\mathcal{R}^\wedge\) over store \(\text{Store}\) in context
Context is given by function \texttt{AND-rewriting}($\text{Store}, \text{Context}, \mathcal{R}^\lor$) defined in Algorithm 1. Function \texttt{ApplicableRule}($\text{Store}, \text{Context}, \mathcal{R}^\lor$) (Line 2), returns a rule \( R \in \mathcal{R}^\lor \) of the form \( \text{Left} \mid \text{Guard} \Rightarrow \text{Right} \) such that \text{Left} unifies with elements \( \text{RmEls} \) in \text{Store} with unifier \( \theta \), and \text{Guard} holds in \text{Context}, if such a rule exists, otherwise it returns \texttt{false}. Then, function \texttt{ApplyRule($R$, \text{Store}, \text{Context}$) returns the pair \( (\text{RmEls}, \text{NewEls}) \), where \text{NewEls} are the elements in \( \text{(Right)}\theta \), i.e., the instance of the right hand side of the unifying rule. Finally, a new store \( \text{Store}' \) is created by taking out \( \text{RmEls} \) from, and adding \( \text{NewEls} \) to \text{Store}, and the process is continued.

\textbf{Example 3.} Consider the sets \( \text{Store} = \{ \text{leq}(X,+\infty), \text{leq}(Y,+\infty), \text{leq}(Z,+\infty) \} \), 
\( \text{Context} = \{ X \leq Y + Z, Y \leq Z + 3, Z = 0 \} \) and \( \mathcal{R}^\lor = \{ \)
\[ \begin{align*}
\text{leq}(A, +\infty) & \mid A = \text{Val} \Rightarrow \text{leq}(A, \text{Val}) \neq eq, \\
\text{leq}(A, +\infty), \text{leq}(B, \text{Val1}) & \mid A < B + \text{Val2} \Rightarrow \\
\text{leq}(A, \text{Val1} + \text{Val2}), \text{leq}(B, \text{Val1}) & \neq \text{addVInt}, \\
\text{leq}(A, +\infty), \text{leq}(B, \text{Val1}), \text{leq}(C, \text{Val2}) & \mid A < B + C \Rightarrow \\
\text{leq}(A, \text{Val1} + \text{Val2}), \text{leq}(B, \text{Val1}), \text{leq}(C, \text{Val2}) & \neq \text{addVarVar} \\
\end{align*} \]

Then, Algorithm 1 (\texttt{AND-rewriting}) proceeds as follows:

- In Line 2, function \texttt{ApplicableRule($\text{Store}, \text{Context}, \mathcal{R}^\lor$)} returns in \( R \) the rule with \texttt{eq} label (if no rule were applicable it would return \texttt{false}).
- \texttt{ApplyRule($R$, \text{Store}, \text{Context}$) returns the pair \( \{ \text{leq}(Z, +\infty) \} \).
- The new store is \( \{ \text{leq}(X, +\infty), \text{leq}(Y, +\infty), \text{leq}(Z, +\infty) \} \setminus \{ \text{leq}(Z, +\infty) \} \) \( \cup \{ \text{leq}(Z, 0) \} = \{ \text{leq}(X, +\infty), \text{leq}(Y, +\infty), \text{leq}(Z, 0) \} \).
- The recursive call in Line 8 selects the \texttt{addVInt} rule and applies it, obtaining \( \text{Store}' = \{ \text{leq}(X, +\infty), \text{leq}(Y, 3), \text{leq}(Z, 0) \} \).
- The next recursive call in Line 8 selects the \texttt{addVarVar} rule, whose application obtains \( \text{Store}' = \{ \text{leq}(X, 3), \text{leq}(Y, 3), \text{leq}(Z, 0) \} \).
- Finally, since there is no applicable rule in the next recursive call, \( R \) is assigned \texttt{false} (Line 2) and the process finishes, returning the current store.

\textbf{OR semantics.} Let \( \mathcal{R}^\lor \subseteq \mathcal{OR}((C_1, \ldots, C_n)) \) and \( \text{Store}_i \subseteq \mathcal{L}, 1 \leq i \leq m \). The operational meaning of applying the set of rules \( \mathcal{R}^\lor \) over the set of \( m \) stores \( \{ \text{Store}_1, \ldots, \text{Store}_m \} \) in context \( \text{Context} \) is given by function \texttt{OR-rewriting}($\{ \text{Store}_1, \ldots, \text{Store}_m \}, \text{Context}, \mathcal{R}^\lor$), defined in Algorithm 2.

\textbf{Example 4.} Consider the sets \( \text{Store}_1 = \{ \text{leq}(X, Y), \text{leq}(Y, +\infty), \text{leq}(Z, X) \} \), 
\( \text{Store}_2 = \{ \text{leq}(X, 3), \text{leq}(Y, +\infty), \text{leq}(Z, Y) \} \), \( \text{Context} = \{ Y > = 3 \} \), and \( \mathcal{R}^\lor = \{ \)
\[ \begin{align*}
\text{leq}(A, \text{Val1}), \text{leq}(A, \text{Val2}) & \mid \text{Val1} \geq \text{Val2} \Rightarrow \text{leq}(A, \text{Val1}) \neq \text{grval}, \\
\text{leq}(A, \text{Val1}), \text{leq}(A, \text{Val1}) & \Rightarrow \text{leq}(A, \text{Val1}) \neq \text{id} \}
\end{align*} \]

Then, \texttt{OR-rewriting}($\{ \text{Store}_1, \text{Store}_2 \}, \text{Context}, \mathcal{R}^\lor$) proceeds as follows:

- The condition in Line 2 holds, so that, after selecting the two stores (Line 3), the call to \texttt{OR-rewriting-pair} (Line 4) calls function \texttt{Apply-OR-rules} in turn (Line 10), which selects the rule with label \texttt{grval} (Line 19). \footnote{In this context, functions \texttt{ApplicableRule} and \texttt{ApplyRule} are similar to the ones defined for \texttt{AND-rules}, but the left hand side of the \texttt{OR-rules} is unified with two stores.}
Algorithm 2 OR-rewriting algorithm

1: function OR-REWIRING(Stores, Context, \( R' \))
2: \[ \text{if } |\text{Stores}| > 1 \text{ then} \]
3: \( (\text{Store}_1, \text{Store}_2) \leftarrow \text{takeTwo}(\text{Stores}) \)
4: \( \text{Store} \leftarrow \text{OR-REWIRING-PAIR}(\text{Store}_1, \text{Store}_2, \text{Context}, \text{\( R' \)}) \)
5: \( \text{Stores}^{' } \leftarrow (\text{Stores} \setminus \{\text{Store}_1, \text{Store}_2\}) \cup \{\text{Store}\} \)
6: return OR-REWIRING(\( \text{Stores}^{' } \), Context, \( R' \))
7: else
8: return \( \text{Stores} \)
9: function OR-REWIRING-PAIR(\( \text{Store}_1 \), \( \text{Store}_2 \), Context, \( \text{\( R' \)} \))
10: \( (\text{St}_1, \text{St}_2, \text{RewSt}) \leftarrow \text{APPLY-OR-RULES}(\text{Store}_1, \text{Store}_2, \text{Context}, \text{\( R' \)}, \emptyset) \)
11: if \( \text{St}_1 = \text{St}_2 \) then
12: return \( \text{St}_1 \cup \text{RewSt} \)
13: else
14: \( \text{Ints} \leftarrow \text{St}_1 \cap \text{St}_2 \)
15: \( \text{Diffs} \leftarrow (\text{St}_1 \setminus \text{Ints}) \cup (\text{St}_2 \setminus \text{Ints}) \)
16: \( \text{TopInfo} \leftarrow \text{sendToTop}(\text{Diffs}) \)
17: return \( \text{RewSt} \cup \text{Ints} \cup \text{TopInfo} \)
18: function APPLY-OR-RULES(\( \text{Store}_1 \), \( \text{Store}_2 \), Context, \( \text{\( R' \)} \), \( \text{AccStore} \))
19: \( \text{R} \leftarrow \text{ApplicableRule}(\text{Store}_1, \text{Store}_2, \text{Context}, \text{\( R' \)}) \)
20: if \( \text{Store}_1 = \text{Store}_2 \lor \text{R} = \text{false} \) then
21: return \( (\text{Store}_1, \text{Store}_2, \text{AccStore}) \)
22: else
23: \( (\text{MSt}_1, \text{MSt}_2, \text{RElems}) \leftarrow \text{ApplyRule}(\text{R}, \text{Store}_1, \text{Store}_2, \text{Context}) \)
24: \( \text{St}_1 \leftarrow \text{Store}_1 \setminus \text{MSt}_1 \)
25: \( \text{St}_2 \leftarrow \text{Store}_2 \setminus \text{MSt}_2 \)
26: \( \text{AccSt} \leftarrow \text{AccStore} \cup \text{RElems} \)
27: return APPLY-OR-RULES(\( \text{St}_1 \), \( \text{St}_2 \), Context, \( \text{\( R' \)}, \text{AccSt} \))

- The condition in Line 20 does not hold, so the rule is applied (Line 23) obtaining: \( \text{MSt}_1 = \{\text{leq}(X, Y)\} \), \( \text{MSt}_2 = \{\text{leq}(X, 3)\} \), and \( \text{RElems} = \{\text{leq}(X, Y)\} \).
- The stores are updated (Lines 24–26), obtaining \( \text{St}_1 = \{\text{leq}(Y, +\infty), \text{leq}(Z, X)\} \), \( \text{St}_2 = \{\text{leq}(Y, +\infty), \text{leq}(Z, Y)\} \), and \( \text{AccSt} = \{\text{leq}(X, Y)\} \).
- The recursive call to APPLY-OR-RULES is performed (Line 27). In this new invocation, the rule with label \text{identical} is selected (Line 19), and since condition in Line 20 does not hold either, such a rule is applied (Line 23).
- The stores are updated (Lines 24–26), obtaining: \( \text{St}_1 = \{\text{leq}(Z, X)\} \), \( \text{St}_2 = \{\text{leq}(Z, Y)\} \), and \( \text{AccSt} = \{\text{leq}(X, Y), \text{leq}(Y, +\infty)\} \).
- A new recursive invocation of APPLY-OR-RULES is performed (Line 27). Now, the condition in Line 20 does not hold because there is no applicable rule \( (\text{R} = \text{false} \text{ in Line 19}) \), so that the APPLY-OR-RULES “loop” finishes in Line 21, returning control to Line 10 with result \( \text{St}_1 = \{\text{leq}(Z, X)\} \), \( \text{St}_2 = \{\text{leq}(Z, Y)\} \), and \( \text{RewSt} = \{\text{leq}(X, Y), \text{leq}(Y, +\infty)\} \).
- Now, the updates in Lines 14–16 obtain: \( \text{Ints} = \emptyset \), \( \text{Diffs} = \{\text{leq}(Z, X), \text{leq}(Z, Y)\} \), and \( \text{TopInfo} = \{\text{leq}(Z, +\infty)\} \). Note that function
Connecting rule-based domains to PLAI. We now sketch how the previously defined AND-rules and OR-rules are connected to the abstract domain operations introduced in Section 2. Let $D_1, \ldots, D_m$ be a collection of $m$ abstract domains with lattices $\mathcal{L}_1, \ldots, \mathcal{L}_m$ and $\lambda = (\lambda_1, \ldots, \lambda_m)$ an abstract substitution of the combined analysis with abstract domains $D_1, \ldots, D_m$, where each $\lambda_i$ is an abstract substitution of domain $D_i$, i.e., $\lambda_i \subseteq \mathcal{L}_i$. We want to perform a rewriting over a collection of domains $D_1, \ldots, D_l$, for some $l \leq m$. For each $i$ in \{1, \ldots, l\}, let $RC_i = (D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_m)$, $\mathcal{R}^\wedge_i \subseteq \text{AND}(D_i, RC_i)$ and $\mathcal{R}^\vee_i \subseteq \text{OR}(D_i, RC_i)$. Then, the application of the collection of sets of AND-rules $\mathcal{R}^\wedge_1, \ldots, \mathcal{R}^\wedge_l$ over $\lambda$ (i.e., applying $\mathcal{R}^\wedge_i$ over each $\lambda_i$ in the corresponding context) is a fixpoint computation, defined in Algorithm 3.

Termination of this computation is in principle the responsibility of the rule writer, i.e., the rules provided should be confluent. It would be interesting to introduce heuristics for ensuring termination, but this is beyond of the scope of this paper, and for simplicity we assume that the algorithm always terminates. Note that a limit on the number of iterations can always be set up in practice to ensure termination or to improve performance, possibly at the prize of accuracy loss (but of course always ensuring the correctness of the results).

Note that the rules may have different objectives. We may just be interested in combining some existing domains. In this case each domain has its own predefined operations and the objective of the rules is only to propagate the information in the abstract substitutions among domains in order to improve precision. We may instead want the rules to define a collection of domains $D_1, \ldots, D_l$ exploiting the information inferred by domains $D_{l+1}, \ldots, D_m$. In this case the rules will implement general domain operations (usually reduced to set operations and

**Algorithm 3** AND-rewriting fixpoint

\begin{algorithm}
\begin{algorithmic}[1]
\Function{AND-FIXP}{$\lambda$, $(\mathcal{R}^\wedge_1, \ldots, \mathcal{R}^\wedge_l)$}
\State $(\lambda_1, \ldots, \lambda_m) \leftarrow \lambda$
\For{$i \in \{1, \ldots, l\}$}
\State $\text{Context}_i \leftarrow (\lambda_1, \ldots, \lambda_{i-1}, \lambda_{i+1}, \ldots, \lambda_m)$
\State $\lambda_i^{\text{rew}} \leftarrow \text{AND-rewriting}(\lambda_i, \text{Context}_i, \mathcal{R}^\wedge_i)$
\EndFor
\State $\lambda^{\text{rew}} \leftarrow (\lambda_1^{\text{rew}}, \ldots, \lambda_l^{\text{rew}}, \lambda_{l+1}, \ldots, \lambda_m)$
\If{$\text{areEqual}(\lambda, \lambda^{\text{rew}})$}
\State return $\lambda^{\text{rew}}$
\Else
\State return AND-FIXP($\lambda^{\text{rew}}$, $(\mathcal{R}^\wedge_1, \ldots, \mathcal{R}^\wedge_l)$)
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}
domains are: corresponding operation for domain $D_i$ as $\text{Aproj}_i$. Now the operations for rule-based domains are:

- $\text{Aproj}(\lambda, V_s) = (\text{Aproj}_1(\lambda_1, V_s), \ldots, \text{Aproj}_m(\lambda_m, V_s))$.
- $\text{Aextend}(A_{k,n}, \lambda^c, \lambda^s) = \text{AND-FIXP}(\lambda^{ext}, \langle R^1_i, \ldots, R^n_i \rangle)$ where $\lambda^{ext} = (\lambda_1^{ext}, \ldots, \lambda_m^{ext})$ and $\lambda_j^{ext} = \text{Aextend}(A_{k,n}, \lambda^c_j, \lambda^s_j)$ for $j \in \{1, \ldots, m\}$.
- $\text{Acall}(A, \lambda, A_k) = \text{AND-FIXP}(\lambda^{call}, \langle R^1_i, \ldots, R^n_i \rangle)$ where $\lambda^{call} = (\lambda_1^{call}, \ldots, \lambda_m^{call})$, $\lambda_j^{call} = \text{Acall}(A, \lambda_j, A_k)$ for $j \in \{1, \ldots, m\}$.
- $\text{Aproceed}(A_k, \lambda^*_k, A) = \text{AND-FIXP}(\lambda^{pri}, \langle R^1_i, \ldots, R^n_i \rangle)$ where $\lambda^{pri} = (\lambda_1^{pri}, \ldots, \lambda_m^{pri})$, $\lambda_j^{pri} = \text{Aproceed}(A_k, \lambda_j^*, A)$ for $j \in \{1, \ldots, m\}$.
- $\text{Ageneralize}(\lambda, \{\lambda_1^*, \ldots, \lambda_k^*\}) = \text{AND-FIXP}(\lambda^{gen}, \langle R^1_i, \ldots, R^n_i \rangle)$ where $\lambda = (\lambda_1, \ldots, \lambda_m)$, $\lambda^{gen} = (\lambda_1^{gen}, \ldots, \lambda_m^{gen})$, and:
  - If we are defining domains $D_1, \ldots, D_l$, then, for $j \in \{l + 1, \ldots, m\}$, $\lambda_j^{gen} = \text{Ageneralize}(\lambda_j, \{\lambda_1^*, \ldots, \lambda_k^*\})$, and for $j \in \{1, \ldots, l\}$, $\lambda_j^{gen} = \text{OR-REWWRITING}(\lambda_j^{tmp}, (\lambda_1^{tmp}, \ldots, \lambda_j^{tmp}, \lambda_{j+1}^{tmp}, \ldots, \lambda_m^{tmp}), \langle R^1_i, \ldots, R^n_i \rangle)$, where $\lambda_j^{tmp} = \{\lambda_j\} \cup \{\lambda_1^*, \ldots, \lambda_k^*\}$ for $t \in \{1, \ldots, l\}$, and $\lambda_j^{tmp} = \lambda_j^{gen}$ for $t \in \{l + 1, \ldots, m\}$.
  - If we are combining domains $D_1, \ldots, D_m$, then, for $j \in \{1, \ldots, m\}$, $\lambda_j^{gen} = \text{Ageneralize}(\lambda_j, \{\lambda_1^*, \ldots, \lambda_k^*\})$.

When referring to some operation of a domain $D_j$ with $j \in \{1, \ldots, l\}$ (for example $\text{Aproj}_j$), if there is no given definition, a default one ($\text{Aproj}_j^{def}$) is used. These default operations (marked as $\text{def}$) are defined as follows:

- $\text{Aproj}_j^{def}(A \text{Sub}, \text{Vars}) = \{\text{elem} \in A \text{Sub} | \exists v \in \text{varset}(\text{elem}) \text{ s.t. } v \in \text{Vars}\}$.
- Aextend\_def(A, ASub1, ASub2) = ASub1 $\cup$ ASub2.
- Acall\_def(A, ASub, A_k) = Aproj\_def(ASub_k, varset(A_k)) where ASub_k is ASub after the unification of A and A_k.
- Aproceed\_def(A_k, A^*_k, A) = $\lambda^{pri}$ where $\lambda^{pri}$ is $\lambda_k^{pri}$ after the unification of $A_k$ with A and $\lambda_k^{pri} = \text{Aproj}_j^{def}(\lambda_k, \text{varset}(A_k))$.

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 the corresponding AND-rules. It is possible to define the abstraction predicates of given built-ins if needed.

**A motivating example: the Bit domain** We return to the domain of Example 1. We will use here the following abstract values (for a given variable $X$) for additional readability: $X/$ zero, $X/$ one, $X/$ bit, $X/$ $\top$, and $X/$ $\bot$. In order to correctly capture this, our analysis should meet the following conditions, encoded in the rules of Fig. 2: (i) If a unification $X = 0$ is encountered, then $X$ should be abstracted to $X/$ zero; this behaviour is captured by Rule 3 (labeled $\text{abs}_0$); (ii) if a unification $X = 1$ is encountered, then $X$ should be abstracted to $X/$ one, as is captured by Rule 4 (labeled $\text{abs}_1$); (iii) if a variable has been abstracted to $\text{bot}$ and also to any other element of the lattice, then it has to be kept as
\[ X/\top \mid X = 0 \Rightarrow X/0 \# \text{abs}_0 \quad (3) \quad X/\text{bit}, X/\text{one} \Rightarrow X/\text{one} \# \text{glb}_5 \quad (9) \]
\[ X/\bot \mid X = 1 \Rightarrow X/\text{one} \# \text{abs}_1 \quad (4) \]
\[ X/\bot, X/Y \Rightarrow X/\bot \# \text{glb}_1 \quad (5) \quad X/\text{one} ; X/\text{zero} \Rightarrow X/\text{bit} \# \text{lub}_1 \quad (10) \]
\[ X/\top, X/Y \Rightarrow X/Y \# \text{glb}_2 \quad (6) \quad X/\bot ; X/Y \Rightarrow X/Y \# \text{lub}_2 \quad (11) \]
\[ X/\text{one}, X/\text{zero} \Rightarrow X/\bot \# \text{glb}_3 \quad (7) \quad X/\top ; X/Y \Rightarrow X/\top \# \text{lub}_3 \quad (12) \]
\[ X/\text{bit}, X/\text{zero} \Rightarrow X/\text{zero} \# \text{glb}_4 \quad (8) \quad X/Y ; X/Y \Rightarrow X/Y \# \text{lub}_4 \quad (13) \]

**Fig. 2.** Sets of AND-rules and OR-rules capturing the behaviour of the bit domain.

*bot*, captured by Rule 5 (labeled \text{glb}_1); (iv) if a variable has been abstracted to \( \top \) and to any other element of the lattice then it must be kept as the other non-top element, captured by Rule 6 (labeled \text{glb}_2); (v) if a variable \( X \) has been abstracted to \( X/\text{zero} \), and to \( X/\text{one} \), then it has to be abstracted to \( X/\bot \), captured by Rule 7 (labeled \text{glb}_3); (vi) if a variable \( X \) has been abstracted to \( X/\bot \) and to \( X/\text{zero} \) then it has to be kept as \( X/\text{zero} \), captured by Rule 8 (labeled \text{glb}_4); (vii) if a variable \( X \) has been abstracted to \( X/\text{bit} \) and to \( X/\text{one} \) then it has to be kept as \( X/\text{one} \), captured by Rule 9 (labeled \text{glb}_5).

In a similar fashion we need to describe the behaviour of the lub or join of a pair of abstract substitutions. We do it as follows: (i) if a variable \( X \) has been abstracted to \( X/\text{zero} \) in one substitution and to \( X/\text{one} \) in other, then the join is \( X/\text{bit} \), captured by Rule 10 (labeled \text{lub}_1); (ii) if a variable \( X \) has been abstracted to \( X/\bot \) in one substitution and to anything else in the other, then the lub is the latter, captured by Rule 11 (labeled \text{lub}_2); (iii) if a variable \( X \) has been abstracted to \( X/\top \) in one substitution and to anything else in the other, then the lub is \( X/\top \), this is captured by Rule 12 (labeled \text{lub}_3); (iv) if a variable \( X \) has been abstracted to the same value in both substitutions then the lub is such a value, captured by Rule 13 (labeled \text{lub}_4).

To also keep track of the structures that variables are bound to we will use the depth-\( k \) domain, presented in Appendix A. So the rules presented in Fig 2 are subsets of AND(Bit, Depth-\( k \)) and OR(Bit, Depth-\( k \)) respectively.

**Example 5.** Consider the very simple Prolog program with two clauses: \text{is\_a\_bit(A)} :- A=0, and \text{is\_a\_bit(A)} :- A=1, and the set of AND- and OR-rules in Fig. 2 (\( R_{\text{bit}}^\text{\_and} \) and \( R_{\text{bit}}^\text{\_or} \) resp.). Fig. 3 shows the analysis flow of the first clause for the combination of the bit and depth-3 domains. The analysis starts with a top entry \text{ASub1} = ((X/\top), \{X=U\}) for a call \text{is\_a\_bit(X)}, where \( X \) is abstracted to \( \top \) in the bit domain and unified to a fresh free variable \( U \) in the depth-3 domain (which means that no information about the structure of \( X \) is given). Since we are carrying a substitution with information about just one variable, projecting over that variable results in the same abstract substitution. The execution of \text{Acall} performs the renaming of \( X \) to \( A \). Since no rule is applicable after such renaming, the analysis proceeds, and when the built-in \text{A=0} is processed, the available domain operations are performed.
Aproceed(is_a_bit(A), ASub4.is_a_bit(X))

Aproceed(is_a_bit(A), ASub4.is_a_bit(X))

Fig. 3. Analysis flow for the clause is_a_bit(A):-A=0. with Bit domain.

Domain depth-3 has specific operation definitions, while bit has none, so nothing is done for the latter domain. In contrast, an updated abstract substitution \( \{A = 0\} \) is obtained for the former, depth-3. Now, the execution of AND-FIXP(\( \{A/T\}, \{A = 0\}, R_{bit}^\delta \)), which applies Rule 3, results in \( \text{ASub4=}\{\{A/zero\}, \{A = 0\}\} \). Aproceed performs a back unification (renaming in this case), obtaining \( \text{ASub5=}\{\{X/zero\}, \{X = 0\}\} \). Finally, Aextend extends the abstract substitution before the Aproceed operation (ASub1) with ASub5. Since there is no specific Aextend operation for the bit domain, the default Aextenddef is used, obtaining the abstract substitution \( \{X/zero,X/T\} \) for such a domain. The application of the corresponding, specific Aextend operation for the depth-3 domain (Aextenddepth-3) obtains \( \{X = 0\} \). Now, the AND-FIXP operation applies Rule 6 to obtain a (combined) abstract success substitution: AND-FIXP(\( X/T, X/zero \), \( \{X = 0\}, R_{bit}^\delta \)) = \( \{X/zero, \{X = 0\}\} \). Similarly, the analysis of the second clause, is_a_bit(A) :- A=1., obtains the abstract substitution \( \{X/one,X = 1\} \). Now, the global analysis applies the Ageneralize operation to the abstract substitutions resulting from the analysis of the two clauses, by calling Ageneralize(\( \{X/one,X = 1\} \), \( \{X/zero,X = 0\}\))

5 As mentioned before, it is possible to give concrete operations for some built-ins, but in this case we let the rules deal with unifications.
1. \( \text{qsort}([], []). \)
2. \( \text{qsort}([X|L], R) :- \)
3. \( \text{partition}(L, X, L1, L2), \quad \text{qsort}(L2, R2), \quad \text{qsort}(L1, R1), \quad \text{append}(R1, [X|R2], R). \)
4. \( \text{partition}([], [], []). \)
5. \( \text{partition}([E|R], C, [E|Right1]) :- E >= C, \text{partition}(R, C, Left, Right). \)
6. \( \text{partition}([E|R], C, Left, [E|Left1]). \)
7. \( \text{append}([], [], []). \)
8. \( \text{append}(X, S, S). \)
9. \( \text{append}([X|T], S, [X|Z]) :- \text{append}(T, S, Z). \)

**Fig. 4.** A classical implementation of quick sort in Prolog (using \text{append}/3).

\[
\begin{align*}
\text{inf}(L, T) \mid L=[] & \Rightarrow \text{inf}(L, \infty) \quad \# \text{empty} \quad (14) \\
\text{inf}(L, X) \mid L=[H|T] & \Rightarrow \text{inf}(L, X), \text{inf}(T, X) \quad \# \text{prop1} \quad (15) \\
\text{inf}(T, X) \mid L=[H|T], X \leq H & \Rightarrow \text{inf}(L, X), \text{inf}(T, X) \quad \# \text{prop2} \quad (16) \\
\text{inf}(T, X) \mid L=[H|T], X \leq H & \Rightarrow \text{inf}(L, H), \text{inf}(T, X) \quad \# \text{prop3} \quad (17) \\
\text{inf}(L1, X) \mid L1=L2 & \Rightarrow \text{inf}(L1, X), \text{inf}(L2, X) \quad \# \text{unif} \quad (18) \\
\text{inf}(L, X) \mid X \leq Y & \Rightarrow \text{inf}(L, Y) \quad \# \text{reduce} \quad (19) \\
\text{inf}(L, Y) \mid X \leq Y & \Rightarrow \text{inf}(L, X) \quad \# \text{lub} \quad (20)
\end{align*}
\]

**Fig. 5.** Sets of AND-rules (14–19) and OR-rules (20) for the inf domain.

**4 Inferring program properties using rule-based combined domains**

We now show how the technique presented in the previous section can be used to define abstract domains for non-trivial properties. Concretely, we present a rule-based domain to infer sortedness. As an aid in defining this domain, consider the classical implementation in Prolog of the quick sort algorithm in Fig. 4. Given a query \( \text{qsort}(A, B) \) we aim at inferring that \( B \) is a sorted list. To do so we first need to consider the lattice over which to abstract the sortedness property. Let that lattice be \( \{\top, \text{sorted}, \text{unsorted}, \bot\} \) where \( \top \) captures that is it unknown whether the element is sorted or not, and \( \bot \) that some error has been found. The structure of the lattice is given by the order relation such that \( \top \) is greater or equal than every other element, \( \bot \) is smaller or equal than every other element, and \( \text{sorted} \) and \( \text{unsorted} \) cannot be compared. Now we start defining rules about

\[
\begin{align*}
\text{sup}(L, T) \mid L=[] & \Rightarrow \text{sup}(L, -\infty) \quad \# \text{empty} \quad (21) \\
\text{sup}(L, X) \mid L=[H|T] & \Rightarrow \text{sup}(L, X), \text{sup}(T, X) \quad \# \text{prop1} \quad (22) \\
\text{sup}(T, X) \mid L=[H|T], H \leq X & \Rightarrow \text{sup}(L, X), \text{sup}(T, X) \quad \# \text{prop2} \quad (23) \\
\text{sup}(T, X) \mid L=[H|T], X \leq H & \Rightarrow \text{sup}(L, H), \text{sup}(T, X) \quad \# \text{prop3} \quad (24) \\
\text{sup}(L1, X) \mid L1=L2 & \Rightarrow \text{sup}(L1, X), \text{sup}(L2, X) \quad \# \text{unif} \quad (25) \\
\text{sup}(L, X) \mid X \leq Y & \Rightarrow \text{sup}(L, Y) \quad \# \text{reduce} \quad (26) \\
\text{sup}(L, Y) \mid X \leq Y & \Rightarrow \text{sup}(L, X) \quad \# \text{lub} \quad (27)
\end{align*}
\]

**Fig. 6.** Sets of AND-rules (21–26) and OR-rules (27) for the sup domain.
the behaviour that we expect from our domain. Clearly, given an empty list we can abstract that element as sorted (which we represent as \(L_{/\text{sorted}}\); this property is captured by Rule 28 (labeled absEmpty) in Fig. 7. We can also define rules to capture the expected behaviour of the *glob*, taking into account the order that we introduced before: if a variable has been abstracted to *sorted* and also to *unsorted* the *glob* is \(\bot\); if the variable has been abstracted to \(\top\) and to any other element, then the latter is the *glob*; finally, if it has been abstracted to \(\bot\) and to something else the result is \(\bot\). These behaviors are captured by Rules 33, 34, and 35 respectively. If a list \(L\) is known to be sorted, and \(L=[H|T]\), then clearly \(T\) must be sorted, which is represented by Rule 32. Conversely, if we know that \(T\) is a sorted list, and \(H\) is smaller or equal than any element in \(T\) then we can infer that \(L\) is sorted. To this end, it is clear the need to introduce (in Figs. 5 and 6) two new abstract domains, \(\text{inf}\) and \(\text{sup}\), which abstract the property of an element \(X\) being lower or equal (resp., greater or equal) than any element in a given list \(L\). This is represented by property \(\text{inf}(L,X)\) (resp. \(\text{sup}(L,X)\)). With the help of the new \(\text{inf}\) domain, we can now represent the previous inference reasoning with Rule 30. In these domains we not only need to capture the unifications occurring during the program execution/analysis but also the arithmetic properties involved. To this end, we use CiaoPP’s polyCLPQ, a polyhedral domain based on [2]. The sets of \(\text{AND}\)- and \(\text{OR}\)-rules for the \(\text{inf}\) and \(\text{sup}\) domains are in fact subsets of \(\text{AND}(\text{inf},(\text{depth}-k,\text{polyCLPQ}))\) and \(\text{OR}(\text{inf},(\text{depth}-k,\text{polyCLPQ}))\), shown in Fig. 5, and \(\text{AND}(\text{sup},(\text{depth}-k,\text{polyCLPQ}))\) and \(\text{OR}(\text{sup},(\text{depth}-k,\text{polyCLPQ}))\), shown in Fig. 6, respectively. The sets of \(\text{AND}\)- and \(\text{OR}\)-rules for the \(\text{sort}\) domain that are presented in Fig. 7 are subsets of \(\text{AND}(\text{sort},(\text{inf},\text{depth}-k,\text{polyCLPQ}))\) and \(\text{OR}(\text{sort},(\text{inf},\text{depth}-k,\text{polyCLPQ}))\) respectively. However the previously defined rules may not be enough to infer sortedness. With the domains \(\text{inf}\) and \(\text{sup}\) the analysis after \(\text{partition}(L,X,L1,L2)\) in the \(\text{qsort}\) algorithm in Fig. 4 would have inferred that \(\text{sup}(L1,X)\) and \(\text{inf}(L2,X)\) (the complete analysis of the \(\text{partition}/4\) predicate with the \(\text{inf}\) domain can be found in Appendix C but even assuming that after the \(\text{qsort}(L2,R2)\), \(\text{qsort}(L1,R1)\) recursive calls \(R2_{/\text{sorted}}\) and \(R1_{/\text{sorted}}\) are inferred, we would not be able to propagate that the element \([X|R2]\) is a sorted list; nor that \(X\) is greater than all elements in \(R1\), which would be key to obtain the sortedness of \(R\). What we are missing is the fact that, given a query \(\text{qsort}(A,B)\), \(A\) is a permutation of \(B\). And that if an element \(X\) satisfies \(\text{inf}(A,X)\), then clearly \(\text{inf}(B,X)\). To deal with this we introduce a new abstract domain, \(\text{mset}\), to abstract properties between multisets of the lists in the program. This domain is discussed in Appendix B. The abstraction is represented by the property \(\text{mset}(A=B)\) capturing that \(B\) is a permutation of \(A\) and \(\text{mset}(A=B+C)\) meaning that the multiset \(A\) is a permutation of the union of the multisets \(B\) and \(C\).

Consider the goal \(\text{append}(R1,S,R)\) together with the information inferred on call ((\(R1_{/\text{sorted}}, R2_{/\text{sorted}}\)), \(\{A=[X|L], S=[X|R2]\}\), \(\}\), \(\{\text{inf}(L2,X), \text{inf}(R2,X)\}\), \(\{\text{sup}(L1,X), \text{sup}(R1,X)\}\), \(\{\text{mset}(L=L1+L2), \text{mset}(L1=R1), \text{mset}(L2=R2)\}\)). In the first case we get the unifications: \(C_1 = (\{R1 = [], S = Z\}, \{R1_{/\text{sorted}}, S_{/\text{sorted}}, Z_{/\text{sorted}}\}, \{\text{inf}(S,X)\}, \{\text{sup}(R1,X)\})\).
apply the sortedness after applying Rules absEmpty and we have the entry: unsorted \( \Rightarrow \) (which is inferred in the recursive call) since enhanced to use as for example polyCLPQ CiaoPP the sortedness and the enhanced to use sup by adding a purely combinatorial rule: \[
\text{true} \mid \text{sup}(L,X), L=H \Rightarrow H \leq X \# \text{combinePoly}
\]

Note that this rule is enhancing the precision of a previously defined domain in CiaoPP, by exploiting properties that have been defined using rules. In this sense a number of other rules could be introduced to enhance the precision of polyCLPQ as for example true \mid L/\text{sorted}, L=[H|T], \text{inf}(T,X) \Rightarrow H \leq X, which exploits both the sortedness and the \text{inf} property to get better abstractions for polyCLPQ.

With this, since \( Z \) is sorted and \( \text{inf}(Z,H) \) (which is inferred in the recursive call) since \( R=[H|Z] \), what we need for \( R \) to be sorted is that \( H \leq X \) holds. polyCLPQ is not able to prove that, but it can be enhanced to use \text{sup} by adding a purely combinatorial rule:

Thus, we have shown how the rule language presented in Section 3 can be used to define a number of new domains complementing each other and enhancing the precision of some predefined domains as polyCLPQ. Moreover they are powerful enough to prove that given a query \text{qsor}(A, B) of the quicksort implementation presented in Fig. 4, B is a sorted list and a permutation of A. CiaoPP outputs the analysis result as the following assertion:
The complete program-point analysis information for the analysis, as produced by CiaoPP, can be found in Appendix D.

**Variable Scope.** It is important to point out that we have to take into account the fact 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 not to 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 that holds a given property” (for example the `inf` property). This kind of issues are common when trying to capture properties of data structures, and they are more involved 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 for simplicity on syntactic transformations that include "extra" arguments to the required predicates (see the call to `append/4` in Appendix D, 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 losses of precision.

5 Conclusions

We have presented a rule-based methodology for rapid design and prototyping of new domains, as well as for combining existing ones. We have demonstrated the power of our approach by showing how several domains and their combinations can be defined with reduced effort. We have also shown how these techniques can be used to develop domains for interesting properties, using list *sortedness* (a property not supported by the previously existing CiaoPP domains) as an example. We have also shown how our prototype implementation infers this property for the classical Prolog implementation of the quick sort algorithm. From our experience using this implementation, the proposed approach seems promising for prototyping and experimenting with new domains, adding domain combinations, and enhancing precision for particular programs, without the need for a deep understanding of the analysis framework internals. Our current implementation is focused on the feasibility and usefulness of the approach, and lacks many possible optimizations. However, given the promising results so far, we plan to optimize the implementation and to use it to define new domains, exploring the trade-offs between rule-based and native, hand-tuned domains. Other avenues for future work are exploring 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.

References


A Depth-k Abstraction

We present here some basics of the Depth-k domain [36]: 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 \( t \). Given a term \( t \) and an integer \( k \), the depth-\( k \) abstraction of \( t \) is the result of replacing every level \( k \) subterm of \( t \) by a newly created variable. E.g., given \( t = f(g(x, y), z) \) and \( u, v \) new variables, the depth-0 abstraction of \( t \) is \( u \); depth-1 is \( f(u, v) \); depth-2 is \( f(g(u, v), z) \); and depth-3 is \( t \). The order relation over the depth-\( k \) lattice is defined as follows: \( t_1 \leq t_2 \iff \text{instance}(t_1, t_2) \), i.e., iff there exists a substitution \( \theta \) such that \( t_1 = t_2 \theta \). For example, the lub (most specific generalization) of \( f(a, b) \) and \( f(X, b) \) is \( f(Y, b) \), where \( Y \) is a free variable.

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. However, for many programs, relatively small values of \( k \) can often produce very useful results in practice, compared to not keeping any structural information.

B Set properties

It is very usual in many verification tasks to verify properties of multisets. An also 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. Here we show how to derive a domain which abstracts lists to multisets of their elements.

A 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 multisets. For example given 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 given a list we can naturally abstract its elements to a multiset. For example given the list \([a, b, c]\) its multiset is \([a, b, c]\). Now, given multisets \( M, N, S \) over a set \( A \) the following properties hold:

- \( M \subseteq N \iff M(x) \leq N(x) \forall x \in A \)
- \( S = M \cup N \iff S(x) = M(x) + N(x) \forall x \in A \)
- \( S = M \setminus N \iff S(x) = M(x) - N(x) \forall x \in A \)

where \( x \div y \) is \( x - y \) if \( x \geq y \) and 0 otherwise. These equivalences allow us to

\[
\text{true} \mid L=[] \Rightarrow \text{mset}(X=0) \; \# \; \text{emptyset} \tag{40}
\]

\[
\text{true} \mid L=[H|T] \Rightarrow \text{mset}(X=H+T) \; \# \; \text{listConst} \tag{41}
\]

\[
\text{true} \mid X=Y \Rightarrow \text{mset}(X=Y) \; \# \; \text{unif} \tag{42}
\]

Fig. 8. Sets of AND-rules used to define the mset domain.
Fig. 9. Analysis result of applying the \textit{mset} analysis over the \textit{partition/4} predicate.

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 [35]). 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 properties will be encapsulated inside an \textit{mset} term to denote that they are abstracting multiset operations and avoid confusing them with the polyhedra properties. 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 \textsc{CiaoPP}'s \textsc{polyCLPQ} domain (an implementation of polyhedra using CLP(Q) based on [2]) and run its \textit{lub} obtaining a very precise operation with almost no cost on the implementation side. only having to worry in the abstraction side which will be carried by the set of \textit{AND}-rules presented in Figure 8. Notice that in this case we are not using the default operations presented in Section 3 but the corresponding \textsc{polyCLPQ} operations. It is also important to note that since we are not defining \textit{OR}-rules we can relax the constraints added over the structure of the abstract substitution and the representation in extended form is not needed. This domain however has some limitations due to its simplicity. We are considering always that we are abstracting plain lists (i.e., lists containing atoms, variables, or numbers but no more complex structures as other lists). Fig. 9 shows the result of
analyzing the `partition/4` predicate with the `mset` analysis derived before (the analysis results are contained in the `true/1` program-point assertions).

C Inferred info for `partition/4` using `inf`

```prolog
:- module(_1,[partition/4,[assertions]]).
:- true pred partition(L,X,L1,L2)
  : ( asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,0)]), true )
=> ( asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,L1)], true ).
partition(L,X,L1,L2) :-
  true(
    asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top')],
      true
    ),
  L=[],
  true(
    asub([inf(L,0Inf),inf(X,'$top'),inf(L1,0Inf),inf(L2,'$top')],
      'terms_check:instance'(L,[[]]),
      true
    ),
  L1=[],
  true(
    asub([inf(L,0Inf),inf(X,'$top'),inf(L1,0Inf),inf(L2,'$top')],
      'terms_check:instance'(L1,[[]]),
      true
    ),
  L2=[],
  true(
    asub([inf(L,0Inf),inf(X,'$top'),inf(L1,0Inf),inf(L2,0Inf)],
      'terms_check:instance'(L,[[]]),
      'terms_check:instance'(L1,[[]]),
      'terms_check:instance'(L2,[[]]),
      true
    ).
partition(L,X,L1,L2) :-
  true(
    asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top'),
      inf(Y,'$top'),inf(Temp,'$top'),inf(Temp1,'$top')],
      true
    ),
  L=[Y|Temp],
  true(
    asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top'),
      inf(Y,'$top'),inf(Temp,'$top'),inf(Temp1,'$top')],
      'terms_check:instance'(L,[[]]),
      'terms_check:instance'(L1,[[]]),
      'terms_check:instance'(L2,[[]]),
      true
    ),
  X>=Y,
  true(
    asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top'),
      inf(Y,'$top'),inf(Temp,'$top'),inf(Temp1,'$top')],
      'terms_check:instance'(L,[[]]),
      'terms_check:instance'(L1,[[]]),
      'terms_check:instance'(L2,[[]]),
      'native_props:constraint'([Y<X])
    ),
  partition(Temp,X,Temp1,L2),
  true(
    asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,X),
      inf(Y,'$top'),inf(Temp,'$top'),inf(Temp1,'$top')],
      'terms_check:instance'(L,[[]]),
      'terms_check:instance'(L1,[[]]),
      'terms_check:instance'(L2,[[]]),
      'native_props:constraint'([X<Y])
    ),
).```
\[ L_1 = \{ Y | Temp1 \} \]

true(
  \[ \text{asub(}[\text{inf}(L,'$top'),\text{inf}(X,'$top'),\text{inf}(L_1,'$top'),\text{inf}(L_2,X),\text{inf}(Y,'$top'),\text{inf}(Temp,'$top'),\text{inf}(Temp_1,'$top')]\],
  \text{terms_check:instance}(L,[Y|Temp]),
  \text{native_props:constraint}(['Y'=X])
).)

\[ \text{partition}(L,X,L_1,L_2) :- \]

true(
  \[ \text{asub(}[\text{inf}(L,'$top'),\text{inf}(X,'$top'),\text{inf}(L_1,'$top'),\text{inf}(L_2,'$top'),\text{inf}(Y,'$top'),\text{inf}(Temp,'$top'),\text{inf}(Temp_1,'$top')]\],
  \text{terms_check:instance}(L,[Y|Temp]),
  \text{native_props:constraint}(['Y'=X])
).)

\[ \text{partition}(Temp,X,L_1,Temp_2), \]

true(
  \[ \text{asub(}[\text{inf}(L,'$top'),\text{inf}(X,'$top'),\text{inf}(L_1,'$top'),\text{inf}(L_2,'$top'),\text{inf}(Y,'$top'),\text{inf}(Temp,'$top'),\text{inf}(Temp_2,'$top')]\],
  \text{terms_check:instance}(L,[Y|Temp]),
  \text{native_props:constraint}(['Y'=X])
).)

\[ L_2 = \{ Y | Temp_2 \}

true(
  \[ \text{asub(}[\text{inf}(L,'$top'),\text{inf}(X,'$top'),\text{inf}(L_1,'$top'),\text{inf}(L_2,X),\text{inf}(Y,'$top'),\text{inf}(Temp,'$top'),\text{inf}(Temp_2,X)]\],
  \text{terms_check:instance}(L,[Y|Temp]),
  \text{native_props:constraint}(['Y'=X])
).)

\[ \text{D Inferred info for qsort/2} \]

\[ - \text{ true pred qsort(X,Y)} \]

\[ : ( \text{asub(}[s(X,'$top'),s(Y,'$top')]\), \text{asub(}[\text{inf}(X,'$top')]\),
  \text{asub(}[\text{inf}(Y,'$top')]\),[\text{sup}(X,'$top'),\text{sup}(Y,'$top')]\), \text{true}, \text{true} ) \]

\[ \Rightarrow ( \text{asub(}[s(X,'$top'), s(Y,\text{sorted}])]\), \text{asub(}[\text{inf}(X,'$top')]\),
  \text{asub(}[\text{inf}(Y,'$top')]\),[\text{sup}(X,'$top'),\text{sup}(Y,'$top')]\), \text{mset(}[X=Y]\),
  \text{true} \).)

\[ \text{qsort}(X,Y) :- \]

true(
  \[ \text{asub(}[s(X,'$top'), s(Y,'$top')]\), \text{asub(}[\text{inf}(X,'$top')]\),
  \text{asub(}[\text{inf}(Y,'$top')]\),[\text{sup}(X,'$top'),\text{sup}(Y,'$top')]\), \text{true}, \text{true} )
).

\[ X=[]\]

true(()
asub([s(X,sorted), s(Y, 'Stop')]),
asub([inf(X,0,Inf), inf(Y, 'Stop')]),
asub([sup(X,-Inf), sup(Y, 'Stop')]),
mset([X=®]),
'terms_check:instance'(X,[]),
true,

Y=[],
true((
asub([s(X,sorted), s(Y,sorted)]),
asub([inf(X,0,Inf), inf(Y,0,Inf)]),
mset([Y=X]),
'terms_check:instance'(X,[]),
'terms_check:instance'(Y,[]),
true),
qsort(A,R):-
true((
asub([s(A,'Stop'), s(R,'Stop'), s(X,'Stop'), s(L,'Stop'), s(L1,'Stop'),
→ s(L2,'Stop'), s(R2,'Stop'), s(R1,'Stop'), s(B,'Stop')]),
asub([inf(A,'Stop'), inf(R,'Stop'), inf(X,'Stop'), inf(L,'Stop'),
→ inf(L1,'Stop'), inf(L2,'Stop'), inf(R2,'Stop'), inf(R1,'Stop'), inf(B,'Stop')]),
asub([sup(A,'Stop'), sup(R,'Stop'), sup(X,'Stop'), sup(L,'Stop'),
→ sup(L1,'Stop'), sup(L2,'Stop'),
sup(R2,'Stop'), sup(R1,'Stop'), sup(B,'Stop')]),
mset([A=X+L]),
'A=X[L],
'terms_check:instance'(A,[X[L]],
true,

true(,

A=[X[L]],
true((
asub([s(A,'Stop'), s(R,'Stop'), s(X,'Stop'), s(L,'Stop'), s(L1,'Stop'),
→ s(L2,'Stop'),
s(R2,'Stop'), s(R1,'Stop'), s(B,'Stop')]),
asub([inf(A,'Stop'), inf(R,'Stop'), inf(X,'Stop'), inf(L,'Stop'),
→ inf(L1,'Stop'), inf(L2,X),
sup(R2,'Stop'), sup(R1,'Stop'), sup(B,'Stop')]),
asub([sup(A,'Stop'), sup(R,'Stop'), sup(X,'Stop'), sup(L,'Stop'),
→ sup(L1,'Stop'), sup(L2,'Stop'),
sup(R2,'Stop'), sup(R1,'Stop'), sup(B,'Stop')]),
mset([L= -X+A,L1= -X+A-L]),
'terms_check:instance'(A,[X[L]],
true,

true(,

qsort(L2,R),
true((
asub([s(A,'Stop'), s(R,'Stop'), s(X,'Stop'), s(L,'Stop'), s(L1,'Stop'),
→ s(L2,'Stop'),
s(R2,sorted), s(R1,'Stop'), s(B,'Stop')]),
asub([inf(A,'Stop'), inf(R,'Stop'), inf(X,'Stop'), inf(L,'Stop'),
→ inf(L1,'Stop'), inf(L2,X),
sup(R2,X), sup(R1,'Stop'), sup(B,'Stop')]),

partition(L,X,L1,L2),
true((
asub([s(A,'Stop'), s(R,'Stop'), s(X,'Stop'), s(L,'Stop'), s(L1,'Stop'),
→ s(L2,'Stop'),
s(R2,'Stop'), s(R1,'Stop'), s(B,'Stop')]),
asub([inf(A,'Stop'), inf(R,'Stop'), inf(X,'Stop'), inf(L,'Stop'),
→ inf(L1,'Stop'), inf(L2,X),
sup(R2,'Stop'), sup(R1,'Stop'), sup(B,'Stop')]),
mset([L= -X+A,L1= -X+A-L]),
'terms_check:instance'(A,[X[L]],
true,

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