TECHNICAL REPORT: CLIP-1/2023.0

A Methodology for Designing and Composing Abstract Domains Using Rewriting Rules *

Daniel Jurjo[1,2], Jose F. Morales[1,2],
Pedro Lopez-Garcia[1,3], and Manuel V. Hermenegildo[1,2]

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid (UPM), Spain
[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 ($a \sqsubseteq a \sqcup b \wedge b \sqsubseteq a \sqcup b$). 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_α. 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 $(D, \alpha, D_\alpha, \gamma)$ 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($\mathcal{R}$) 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_α) is typically built-in, as an abstract domain of *analysis graphs*. The framework is *parametric* on a second abstract domain, D_α, whose elements appear as labels in the nodes of the analysis graph. We refer to such nodes with tuples $\langle p(V_1, \ldots, V_n), \lambda^c, \lambda^s \rangle$, where p is a predicate in the program under analysis, and λ^c, λ^s (both elements of D_α), 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. $\gamma(\texttt{z}) = \{0\}$, `o` (one) s.t. $\gamma(\texttt{o}) = \{1\}$, `b` (bit) s.t.

```
main(Msg,P) :-
    par(Msg,0,P).

par([],P,P).
par([C|Cs],P0,P) :-
    xor(C,P0,P1),
    par(Cs,P1,P).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
```

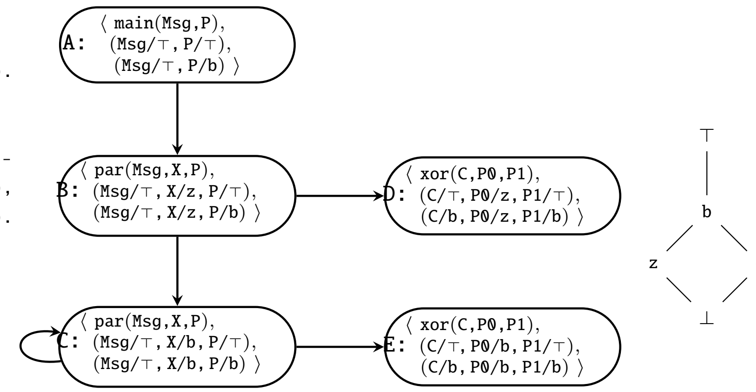

Fig. 1. A set of CHCs for computing parity (left) and a possible analysis graph (right).

$\gamma(\mathtt{b}) = \{0, 1\}$, and $\top$ such that $\gamma(\top)$ is the set of all concrete values. Consider an initial abstract goal $G_\alpha = \langle$ `main(Msg,P)`, `(Msg/⊤, P/⊤)` $\rangle$, representing that the arguments of `main` can be bound to any concrete value (see node `A` in the figure). Node `B` = ($\langle$ `par(Msg,X,P)`, `(Msg/⊤, X/z, P/⊤)`, `(Msg/⊤, X/z, P/b)` $\rangle$) captures the fact that `par` may be called with `X` bound to 0 in $\gamma(\mathtt{z})$ and, if such a call succeeds, the third argument `P` will be bound to any value in $\gamma(\mathtt{b}) = \{0, 1\}$. Note that node `C` captures the fact that, after this call, there are other calls to `par` where `X/b`. Edges in the graph capture calls and paths. For example, two such edges exist from node `B`, denoting that `par` may call `xor` (edge from `B` to `D`) or `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 *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 ∇ operator); the abstract semantics of the primitive constraints (representing the *built-ins*, or basic operations of the source language) via *abstract transfer functions* (f^α); and the following additional instrumental operations over abstract substitutions:

- `Aproj`(λ, Vs): restricts the abstract substitution λ to the set of variables Vs.
- `Aextend`$(A_{k,n}, \lambda^c, \lambda^s)$: propagates the abstract success substitution λ^s, defined over the variables of the n-th literal of clause k ($A_{k,n}$), to λ^c, which is defined over all the variables of clause k (which contains $A_{k,n}$ in its body).
- `Acall`(A, λ, A_k): performs the *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.
- `Aproceed`(A_k, λ_k^s, A): performs the *abstract proceed*, i.e., the reverse operation of `Acall`. It unifies the head of clause k (A_k) and the abstract substitution at the end of clause k (λ_k^s) with the original call A to produce the success substitution over the variables of A.
- `Ageneralize`$(\lambda, \{\lambda^1, \ldots, \lambda^k\})$: joins λ with the set of abstract substitutions $\{\lambda^1, \ldots, \lambda^k\}$, all over the same variables. The result is an abstract substitution greater than or equal to λ. It either returns λ, when no generalization is

needed; performs the least upper bound ($\sqcup$); or performs the widening (∇) of λ together with $\{\lambda^1, \ldots, \lambda^k\}$, 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 $(E, \alpha_i, D_i, \gamma_i)$, $i \in \{1, \ldots, n\}$ Galois insertions. The *direct product domain* is a quadruple $(E, \alpha_x, D_x, \gamma_x)$ where $D_x = D_1 \times \cdots \times D_n$, $\gamma_x : D_x \to E$ such that $\gamma_x((d_1, \ldots, d_n)) = \gamma_1(d_1) \sqcap_E \cdots \sqcap_E \gamma_n(d_n)$ and $\alpha_x : E \to D_x$ where $\alpha_x(e) = (\alpha_1(e), \ldots, \alpha_n(e))$. However the direct product domain is *not* a Galois insertion, as shown in [9]. Consider a direct product $(E, \alpha_x, D_x, \gamma_x)$ and the relation $\equiv \subseteq D_x \times D_x$ defined by $d \equiv d' \Leftrightarrow \gamma_x(d) = \gamma_x(d')$. The *reduced product domain* is a quadruple $(E, \alpha_\equiv, D_\equiv, \gamma_\equiv)$ where $\alpha_\equiv : E \to D_\equiv$ such that $\alpha_\equiv(e) = [\alpha_x(e)]_\equiv$ and $\gamma_\equiv : D_\equiv \to E$ such that $\gamma_\equiv([d]_\equiv) = \gamma_x(d)$. 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 γ_i. The *reduced product function*, $\mu_\equiv : D_\equiv \to D_\equiv$ is defined by $\mu_\equiv([d]_\equiv) = [(\mu_1(d_1)), \ldots, \mu_n(d_n)]_\equiv$ where $(d_1, \ldots, d_n) = \sqcap_{D_x}[d]_\equiv$. 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 "`ground(X), Y > 0`," where `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 anaysis 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

```
function AND-REWRITING(Store, Context, R^)
    R ← ApplicableRule(Store, Context, R^)
    if R = false then
        return Store
    else
        (RmEls, NewEls) ← ApplyRule(R, Store, Context)
        Store' ← (Store \ RmEls) ∪ NewEls
        return AND-REWRITING(Store', Context, R^)
```

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

$$AND\text{-rules}: \; l_1, \ldots, \; l_n \,|\, g_1, \ldots, \; g_l \;\Rightarrow\; r_1, \ldots, \; r_m \;\#\; label \tag{1}$$

$$OR\text{-rules}: \; l_1 \;;\; l_2 \,|\, g_1, \ldots, \; g_l \;\Rightarrow\; r_1, \ldots, \; r_m \;\#\; label \tag{2}$$

where $l_1, \ldots, l_n$, $r_1, \ldots, r_m$ are elements of a set of properties $\mathcal{L}$ and each $g_1, \ldots, g_l$ is an element of $\mathcal{C}_1, \ldots, \mathcal{C}_s$, 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 $\mathcal{L}$, while checking the contents of s stores containing properties respectively from $\mathcal{C}_1, \ldots, \mathcal{C}_s$ (the "context").

Since the language is parameterized by the sets of properties being used, we will use $AND(\mathcal{L}, (\mathcal{C}_1, \ldots, \mathcal{C}_s))$ (resp. $OR(\mathcal{L}, (\mathcal{C}_1, \ldots, \mathcal{C}_s))$) to refer to the language of *AND*-rules (resp. *OR*-rules) where the left and right sides are elements of $\mathcal{L}$ and the guards are elements of $\mathcal{C}_1, \ldots, \mathcal{C}_s$.

We say that an *abstract substitution* λ is in *extended form* iff for each program variable $x \in vars(\lambda)$ there is a unique element of λ 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 $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$.

- The abstract substitution $\{\mathtt{X}/i(1,2), \mathtt{Y}/i(0,1), \mathtt{Z}/i(-\infty,\infty)\}$ 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 not in extended form, but it can be transformed into a (more verbose) extended form `[sh(X, [Y]), sh(Y, [X]), sh(Z, [])]`, where the property `sh(A, ShSet)` expresses that all the variables in `ShSet` share with `A`.

Let $\mathcal{L}, \mathcal{C}_1, \ldots, \mathcal{C}_n$ be lattices and *Context* an element of $\mathcal{P}(\mathcal{C}_1) \times \cdots \times \mathcal{P}(\mathcal{C}_n)$ where $\mathcal{P}(S)$ denotes the powerset of S. We also assume that abstract substitutions are in *extended form* representation.

AND semantics. Let $\mathcal{R}^\wedge \subseteq AND(\mathcal{L}, (\mathcal{C}_1, \ldots, \mathcal{C}_n))$ and $Store \subseteq \mathcal{L}$. The operational meaning of applying the set of rules $\mathcal{R}^\wedge$ over store *Store* in context

Context is given by function AND-REWRITING($Store, Context, \mathcal{R}^\wedge$) defined in Algorithm 1. Function ApplicableRule($Store, Context, \mathcal{R}^\wedge$) (Line 2), returns a rule $R \in \mathcal{R}^\wedge$ of the form *Left* | *Guard* $\Rightarrow$ *Right* such that *Left* unifies with elements *RmEls* in *Store* with *unifier* θ, and *Guard* holds in *Context*, if such a rule exists, otherwise it returns `false`. Then, function ApplyRule($R, Store, Context$) (Line 6) returns the pair ($RmEls, NewEls$), where $NewEls$ are the elements in $(Right)\theta$, i.e., the instance of the right hand side of the unifying rule. Finally, a new store $Store'$ is created by taking out *RmEls* from, and adding *NewEls* to *Store*, and the process is continued.

Example 3. Consider the sets $Store = \{\texttt{leq}(\texttt{X}, +\infty),\ \texttt{leq}(\texttt{Y}, +\infty),\ \texttt{leq}(\texttt{Z}, +\infty)\}$, $Context = \{\texttt{X} \leq \texttt{Y} + \texttt{Z},\ \texttt{Y} \leq \texttt{Z} + 3,\ \texttt{Z} = 0\}$ and $\mathcal{R}^\wedge = \{$

$$\texttt{leq}(\texttt{A}, +\infty) \mid \texttt{A} = \mathit{Val} \Rightarrow \texttt{leq}(\texttt{A}, Val) \# \mathit{eq},$$
$$\texttt{leq}(\texttt{A}, +\infty),\ \texttt{leq}(\texttt{B}, Val1) \mid \texttt{A} \leq \texttt{B} + \mathit{Val2} \Rightarrow$$
$$\texttt{leq}(\texttt{A}, Val1 + Val2),\ \texttt{leq}(\texttt{B}, Val1) \# \mathit{addVInt},$$
$$\texttt{leq}(\texttt{A}, +\infty),\ \texttt{leq}(\texttt{B}, Val1),\ \texttt{leq}(\texttt{C}, Val2) \mid \texttt{A} \leq \texttt{B} + \texttt{C} \Rightarrow$$
$$\texttt{leq}(\texttt{A}, Val1 + Val2),\ \texttt{leq}(\texttt{B}, Val1),\ \texttt{leq}(\texttt{C}, Val2) \# \mathit{addVarVar}\ \}$$

Then, Algorithm 1 (AND-rewriting) proceeds as follows:

- In Line 2, function ApplicableRule($Store, Context, \mathcal{R}^\wedge$) returns in R the rule with *eq* label (if no rule were applicable it would return `false`).
- ApplyRule($R, Store, Context$) returns the pair $(\{\texttt{leq}(\texttt{Z}, +\infty)\},\ \{\texttt{leq}(\texttt{Z}, 0)\})$.
- The new store is (Line 7) $Store' = (\{\texttt{leq}(\texttt{X}, +\infty), \texttt{leq}(\texttt{Y}, +\infty), \texttt{leq}(\texttt{Z}, +\infty)\} \setminus \{\texttt{leq}(\texttt{Z}, \infty)\}) \cup \{\texttt{leq}(\texttt{Z}, 0)\} = \{\texttt{leq}(\texttt{X}, \infty),\ \texttt{leq}(\texttt{Y}, +\infty),\ \texttt{leq}(\texttt{Z}, 0)\}$.
- The recursive call in Line 8 selects the *addVInt* rule and applies it, obtaining $Store' = \{\texttt{leq}(\texttt{X}, +\infty),\ \texttt{leq}(\texttt{Y}, 3),\ \texttt{leq}(\texttt{Z}, 0)\}$.
- The next recursive call in Line 8 selects the *addVarVar* rule, whose application obtains $Store' = \{\texttt{leq}(\texttt{X}, 3),\ \texttt{leq}(\texttt{Y}, 3),\ \texttt{leq}(\texttt{Z}, 0)\}$.
- Finally, since there is no applicable rule in the next recursive call, R is assigned `false` (Line 2) and the process finishes, returning the current store.

OR semantics. Let $\mathcal{R}^\vee \subseteq OR(\mathcal{L}, (\mathcal{C}_1, \ldots, \mathcal{C}_n))$ and $Store_i \subseteq \mathcal{L}$, $1 \leq i \leq m$. The operational meaning of applying the set of rules $\mathcal{R}^\vee$ over the set of m stores $\{Store_1, \ldots, Store_m\}$ in context *Context* is given by function OR-REWRITING($\{Store_1, \ldots, Store_m\}, Context, \mathcal{R}^\vee$), defined in Algorithm 2.

Example 4. Consider the sets $Store_1 = \{\texttt{leq}(\texttt{X}, \texttt{Y}),\ \texttt{leq}(\texttt{Y}, +\infty),\ \texttt{leq}(\texttt{Z}, \texttt{X})\}$, $Store_2 = \{\texttt{leq}(\texttt{X}, 3),\ \texttt{leq}(\texttt{Y}, +\infty),\ \texttt{leq}(\texttt{Z}, \texttt{Y})\}$, $Context = \{\texttt{Y} >= 3\}$, and $\mathcal{R}^\vee = \{$

$$\texttt{leq}(\texttt{A}, Val1)\ ;\ \texttt{leq}(\texttt{A}, Val2) \mid \mathit{Val1} \geq \mathit{Val2} \Rightarrow \texttt{leq}(\texttt{A}, Val1) \# \mathit{grval},$$
$$\texttt{leq}(\texttt{A}, Val1)\ ;\ \texttt{leq}(\texttt{A}, Val1) \Rightarrow \texttt{leq}(\texttt{A}, Val1) \# \mathit{identical}\ \}$$

Then, OR-REWRITING($\{Store_1, Store_2\}, Context, \mathcal{R}^\vee$) proceeds as follows:

- The condition in Line 2 holds, so that, after selecting the two stores (Line 3), the call to OR-REWRITING-PAIR (Line 4) calls function APPLY-OR-RULES in turn (Line 10), which selects the rule with label *grval* (Line 19). [4]

[4] In this context, functions ApplicableRule and ApplyRule are similar to the ones defined for *AND*-rules, but the left hand side of the *OR*-rules is unified with two stores.

Algorithm 2 OR-rewriting algorithm

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

- The condition in Line 20 does not hold, so the rule is applied (Line 23) obtaining: $\mathit{MSt}_1 = \{\mathtt{leq}(\mathtt{X}, \mathtt{Y})\}$, $\mathit{MSt}_2 = \{\mathtt{leq}(\mathtt{X}, 3)\}$, and $\mathit{RElems} = \{\mathtt{leq}(\mathtt{X}, \mathtt{Y})\}$.
- The stores are updated (Lines 24–26), obtaining $\mathit{St}_1 = \{\mathtt{leq}(\mathtt{Y}, +\infty), \mathtt{leq}(\mathtt{Z}, \mathtt{X})\}$, $\mathit{St}_2 = \{\mathtt{leq}(\mathtt{Y}, +\infty), \mathtt{leq}(\mathtt{Z}, \mathtt{Y})\}$, and $\mathit{AccSt} = \{\mathtt{leq}(\mathtt{X}, \mathtt{Y})\}$.
- The recursive call to APPLY-OR-RULES is performed (Line 27). In this new invocation, the rule with label *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: $\mathit{St}_1 = \{\mathtt{leq}(\mathtt{Z}, \mathtt{X})\}$, $\mathit{St}_2 = \{\mathtt{leq}(\mathtt{Z}, \mathtt{Y})\}$, and $\mathit{AccSt} = \{\mathtt{leq}(\mathtt{X}, \mathtt{Y}), \mathtt{leq}(\mathtt{Y}, +\infty)\}$.
- 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 ($R = \mathtt{false}$ in Line 19), so that the APPLY-OR-RULES "loop" finishes in Line 21, returning control to Line 10 with result $\mathit{St}_1 = \{\mathtt{leq}(\mathtt{Z}, \mathtt{X})\}$, $\mathit{St}_2 = \{\mathtt{leq}(\mathtt{Z}, \mathtt{Y})\}$, and $\mathit{RewSt} = \{\mathtt{leq}(\mathtt{X}, \mathtt{Y}), \mathtt{leq}(\mathtt{Y}, +\infty)\}$.
- Now, the updates in Lines 14–16 obtain: $\mathit{Ints} = \emptyset$, $\mathit{Diffs} = \{\mathtt{leq}(\mathtt{Z}, \mathtt{X}), \mathtt{leq}(\mathtt{Z}, \mathtt{Y})\}$, and $\mathit{TopInfo} = \{\mathtt{leq}(\mathtt{Z}, +\infty)\}$. Note that function

Algorithm 3 AND-rewriting fixpoint

1: **function** AND-FIXP(λ, $\langle \mathcal{R}_1^{\wedge}, \ldots, \mathcal{R}_l^{\wedge} \rangle$)
2: $\quad (\lambda_1, \ldots, \lambda_m) \leftarrow \lambda$
3: $\quad$ **for** $i \in \{1, \ldots, l\}$ **do**
4: $\quad\quad Context_i \leftarrow (\lambda_1, \ldots, \lambda_{i-1}, \lambda_{i+1}, \ldots, \lambda_m)$
5: $\quad\quad \lambda_i^{rew} \leftarrow$ AND-REWRITING(λ_i, $Context_i$, $\mathcal{R}_i^{\wedge}$)
6: $\quad \lambda^{rew} \leftarrow (\lambda_1^{rew}, \ldots, \lambda_l^{rew}, \lambda_{l+1}, \ldots, \lambda_m)$
7: $\quad$ **if** areEqual(λ, λ^{rew}) **then**
8: $\quad\quad$ **return** λ^{rew}
9: $\quad$ **else**
10: $\quad\quad$ **return** AND-FIXP(λ^{rew}, $\langle \mathcal{R}_1^{\wedge}, \ldots, \mathcal{R}_l^{\wedge} \rangle$)

sendToTop assigns the `top` ($\top$) value (of the corresponding lattice) to the variables corresponding to the elements in the set *Diffs*: only variable Z in this case, the one on the left hand side of `leq(_,_)`.

– The call to OR-REWRITING-PAIR in Line 4 finishes, returning store $Store = \{\texttt{leq}(\texttt{X}, \texttt{Y}), \texttt{leq}(\texttt{Y}, +\infty), \texttt{leq}(\texttt{Z}, +\infty)\}$, and the recursive call in Line 6 also finishes, returning $\{Store\}$ (since condition in line 2 does not hold and such a result is returned in Line 8).

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 λ_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 s.t. $l \leq m$. For each i in $\{1, \ldots, l\}$, let $RC_i = (D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_m)$, $\mathcal{R}_i^{\wedge} \subseteq AND(D_i, RC_i)$ and $\mathcal{R}_i^{\vee} \subseteq OR(D_i, RC_i)$. Then, the application of the collection of sets of AND-rules $\mathcal{R}_1^{\wedge}, \ldots, \mathcal{R}_l^{\wedge}$ over λ (i.e., applying $\mathcal{R}_i^{\wedge}$ over each λ_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

unifications) and the information will be mainly gathered from the information obtained by the domains $D_{l+1}, \dots, D_m$, after the given operation.
Finally, given a domain operation (for example `Aproj`), we denote the corresponding operation for domain D_i as $\texttt{Aproj}_i$. Now the operations for rule-based domains are:

- $\texttt{Aproj}(\lambda, Vs) = (\texttt{Aproj}_1(\lambda_1, Vs), \dots, \texttt{Aproj}_m(\lambda_m, Vs))$.
- $\texttt{Aextend}(A_{k,n}, \lambda^c, \lambda^s) = \text{AND-FIXP}(\lambda^{ext}, \langle \mathcal{R}_1^\wedge, \dots, \mathcal{R}_l^\wedge \rangle)$ where $\lambda^{ext} = (\lambda_1^{ext}, \dots, \lambda_m^{ext})$ and $\lambda_j^{ext} = \texttt{Aextend}_j(A_{k,n}, \lambda^c{}_j, \lambda^s{}_j)$ for $j \in \{1, \dots, m\}$.
- $\texttt{Acall}(A, \lambda, A_k) = \text{AND-FIXP}(\lambda^{call}, \langle \mathcal{R}_1^\wedge, \dots, \mathcal{R}_l^\wedge \rangle)$ where $\lambda^{call} = (\lambda_1^{call}, \dots, \lambda_m^{call})$, $\lambda_j^{call} = \texttt{Acall}_j(A, \lambda_j, A_k)$ for $j \in \{1, \dots, m\}$.
- $\texttt{Aproceed}(A_k, \lambda_k^s, A) = \text{AND-FIXP}(\lambda^{pri}, \langle \mathcal{R}_1^\wedge, \dots, \mathcal{R}_l^\wedge \rangle)$ where $\lambda^{pri} = (\lambda_1^{pri}, \dots, \lambda_m^{pri})$, $\lambda_j^{pri} = \texttt{Aproceed}_j(A_k, \lambda_{kj}^s, A)$ for $j \in \{1, \dots, m\}$.
- $\texttt{Ageneralize}(\lambda, \{\lambda^1, \dots, \lambda^k\}) = \text{AND-FIXP}(\lambda^{gen}, \langle \mathcal{R}_1^\wedge, \dots, \mathcal{R}_l^\wedge \rangle)$ where $\lambda = (\lambda_1, \dots, \lambda_m)$, $\lambda^{gen} = (\lambda_1^{gen}, \dots, \lambda_m^{gen})$, and:
  - If we are defining domains $D_1, \dots, D_l$, then, for $j \in \{l+1, \dots, m\}$, $\lambda_j^{gen} = \texttt{Ageneralize}_j(\lambda_j, \{\lambda_j^1, \dots, \lambda_j^k\})$, and for $j \in \{1, \dots, l\}$, $\lambda_j^{gen} = \text{OR-REWRITING}(\lambda_j^{tmp}, (\lambda_1^{tmp}, \dots, \lambda_{j-1}^{tmp}, \lambda_{j+1}^{tmp}, \dots, \lambda_m^{tmp}), \mathcal{R}_j^\vee)$, where $\lambda_t^{tmp} = \{\lambda_t\} \cup \{\lambda_t^1, \dots, \lambda_t^k\}$ for $t \in \{1, \dots, l\}$, and $\lambda_t^{tmp} = \lambda_t^{gen}$ for $t \in \{l+1, \dots, m\}$.
  - If we are combining domains $D_1, \dots, D_m$, then, for $j \in \{1, \dots, m\}$, $\lambda_j^{gen} = \texttt{Ageneralize}_j(\lambda_j, \{\lambda_j^1, \dots, \lambda_j^k\})$.

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

- $\texttt{Aproj}_{def}(ASub, Vars) = \{elem \in ASub \mid \exists v \in \mathsf{varset}(elem)\ s.t.\ v \in Vars\}$.
- $\texttt{Aextend}_{def}(A, ASub1, ASub2) = ASub1 \cup ASub2$.
- $\texttt{Acall}_{def}(A, ASub, A_k) = \texttt{Aproj}_{def}(ASub_k, \mathsf{varset}(A_k))$ where $ASub_k$ is $ASub$ after the unification of A and A_k.
- $\texttt{Aproceed}_{def}(A_k, \lambda_k^s, A) = \lambda^{pri}$ where λ^{pri} is λ_k^{pri} after the unification of A_k with A and $\lambda_k^{pri} = \texttt{Aproj}_{def}(\lambda_k^s, \mathsf{varset}(A_k))$.

The *built-ins* are abstracted in each of the domains $D_1, \dots, D_n$ 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: $\texttt{X}/zero$, $\texttt{X}/one$, $\texttt{X}/bit$, $\texttt{X}/\top$, and $\texttt{X}/\bot$. In order to correctly capture this, our analysis should meet the following conditions, encoded in the rules of Fig. 2: (**i**) If a unification $\texttt{X} = 0$ is encountered, then `X` should be abstracted to $\texttt{X}/zero$; this behaviour is captured by Rule 3 (labeled abs_{zero}); (**ii**) if a unification $\texttt{X} = 1$ is encountered, then `X` should be abstracted to $\texttt{X}/one$, as is captured by Rule 4 (labeled abs_{one}); (**iii**) if a variable has been abstracted to *bot* and also to any other element of the lattice, then it has to be kept as

$\mathtt{X}/\top \mid \mathtt{X}=0 \Rightarrow \mathtt{X}/zero \ \#\ abs_{zero}$	(3)	$\mathtt{X}/bit\ , \mathtt{X}/one \Rightarrow \mathtt{X}/one \ \#\ glb_5$	(9)
$\mathtt{X}/\top \mid \mathtt{X}=1 \Rightarrow \mathtt{X}/one \ \#\ abs_{one}$	(4)		
$\mathtt{X}/\bot\ , \mathtt{X}/\mathtt{Y} \Rightarrow \mathtt{X}/\bot \ \#\ glb_1$	(5)	$\mathtt{X}/one\ ;\ \mathtt{X}/zero \Rightarrow \mathtt{X}/bit \ \#\ lub_1$	(10)
$\mathtt{X}/\top\ , \mathtt{X}/\mathtt{Y} \Rightarrow \mathtt{X}/\mathtt{Y} \ \#\ glb_2$	(6)	$\mathtt{X}/\bot\ ;\ \mathtt{X}/\mathtt{Y} \Rightarrow \mathtt{X}/\mathtt{Y} \ \#\ lub_2$	(11)
$\mathtt{X}/one\ , \mathtt{X}/zero \Rightarrow \mathtt{X}/\bot \ \#\ glb_3$	(7)	$\mathtt{X}/\top\ ;\ \mathtt{X}/\mathtt{Y} \Rightarrow \mathtt{X}/\top \ \#\ lub_3$	(12)
$\mathtt{X}/bit\ , \mathtt{X}/zero \Rightarrow \mathtt{X}/zero \ \#\ glb_4$	(8)	$\mathtt{X}/\mathtt{Y}\ ;\ \mathtt{X}/\mathtt{Y} \Rightarrow \mathtt{X}/\mathtt{Y} \ \#\ lub_4$	(13)

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

bot, captured by Rule 5 (labeled 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 glb_2); (**v**) if a variable `X` has been abstracted to $\mathtt{X}/zero$, and to $\mathtt{X}/one$, then it has to be abstracted to $\mathtt{X}/\bot$, captured by Rule 7 (labeled glb_3); (**vi**) if a variable `X` has been abstracted to $\mathtt{X}/\bot$ and to $\mathtt{X}/zero$ then it has to be kept as $\mathtt{X}/zero$, captured by Rule 8 (labeled glb_4); (**vii**) if a variable `X` has been abstracted to $\mathtt{X}/bit$ and to $\mathtt{X}/one$ then it has to be kept as $\mathtt{X}/one$, captured by Rule 9 (labeled 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 $\mathtt{X}/zero$ in one sustitution and to $\mathtt{X}/one$ in other, then the join is $\mathtt{X}/bit$, captured by Rule 10 (labeled lub_1); (**ii**) if a variable `X` has been abstracted to $\mathtt{X}/\bot$ in one sustitution and to anything else in the other, then the *lub* is the latter, captured by Rule 11 (labeled lub_2); (**iii**) if a variable `X` has been abstracted to $\mathtt{X}/\top$ in one sustitution and to anything else in the other, then the *lub* is $\mathtt{X}/\top$, this is captured by Rule 12 (labeled 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 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 $AND(Bit, Depth\text{-}k)$ and $OR(Bit, Depth\text{-}k)$ respectively.

Example 5. Consider the very simple Prolog program with two clauses: `is_a_bit(A) :- A=0.` and `is_a_bit(A) :- A=1.`, and the set of *AND*- and *OR*-rules in Fig. 2 ($\mathcal{R}^{\wedge}_{bit}$ and $\mathcal{R}^{\vee}_{bit}$ 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 `ASub1`$=$ $(\{\mathtt{X}/\top\}, \{\mathtt{X=U}\})$ for a call `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 `Acall` performs the renaming of `X` to `A`. Since no rule is applicable after such renaming, the analysis proceeds, and when the *built-in* `A=0` is processed, the available domain operations are performed.

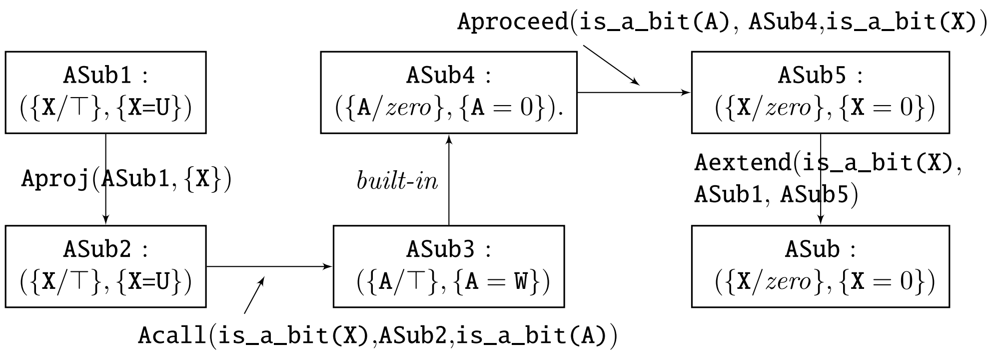

Fig. 3. Analysis flow for the clause `is_a_bit(A):-A=0.` with `Bit` domain.

Domain `depth`-3 has specific operation definitions, while `bit` has none, [5] so nothing is done for the latter domain. In contrast, an updated abstract substitution $\{\mathtt{A} = 0\}$ is obtained for the former, `depth`-3. Now, the execution of AND-FIXP($\{\mathtt{A}/\top\}, \{\mathtt{A} = 0\}, \mathcal{R}^{\wedge}_{bit}$), which applies Rule 3, results in `ASub4`=($\{\mathtt{A}/zero\}, \{\mathtt{A} = 0\}$). `Aproceed` perfoms a back unification (renaming in this case), obtaining `ASub5`= ($\{\mathtt{X}/zero\}, \{\mathtt{X} = 0\}$). Finally, `Aextend` extends the abstract substitution before the `Aproceed` operation (`ASub1`) with `ASub5`. Since there is no specific $\mathtt{Aextend}_{\mathtt{bit}}$ operation for the `bit` domain, the default $\mathtt{Aextend}_{\mathtt{def}}$ is used, obtaining the abstract substitution $\{\mathtt{X}/zero, \mathtt{X}/\top\}$ for such a domain. The application of the corresponding, specific `Aextend` operation for the `depth`-3 domain ($\mathtt{Aextend}_{\mathtt{depth\text{-}3}}$) obtains $\{\mathtt{X} = 0\}$. Now, the AND-FIXP operation applies Rule 6 to obtain a (combined) abstract success substution: AND-FIXP($\{\mathtt{X}/\top, \mathtt{X}/zero\}, \{\mathtt{X} = 0\}, \mathcal{R}^{\wedge}_{bit}$)= ($\{\mathtt{X}/zero, \{\mathtt{X} = 0\}$). Similarly, the anaysis of the second clause, `is_a_bit(A) :- A=1.`, obtains the abstract substitution ($\{\mathtt{X}/one, \{\mathtt{X} = 1\}$). Now, the global analysis applies the `Ageneralize` operation to the abstract substitutions resulting from the analysis of the two clauses, by calling `Ageneralize`(($\{\mathtt{X}/one, \{\mathtt{X} = 1\}$), $\{(\{\mathtt{X}/zero\}, \{\mathtt{X} = 0\})\}$), which, for the *bit* domain, calls OR-REWRITING($\{\{\mathtt{X}/one\}, \{\mathtt{X}/zero\}\}, \{\{\mathtt{X} = 1\}, \{\mathtt{X} = 0\}\}$) in turn, obtaining $\{\mathtt{X}/bit\}$, due to the application of Rule 10. For the `depth`−3 domain, `Ageneralize` applies $\mathtt{Ageneralize}_{\mathtt{depth\text{-}3}}$ to abstract substituions $\{\mathtt{X} = 1\}$ and $\{\mathtt{X} = 0\}$, obtaining $\{\mathtt{X} = \mathtt{V}\}$, where `V` is a fresh free variable. The analysis finishes with the execution of AND-FIXP(($\{\mathtt{X}/bit\}, \{\mathtt{X} = \mathtt{V}\}$), $\mathcal{R}^{\wedge}_{bit}$), which performs the AND-rewriting on the resulting abstract generalization for the combined domain, obtaining the success abstract substitution ($\{\mathtt{X}/bit\}, \{\mathtt{X} = \mathtt{V}\}$) for the initial call `is_a_bit(X)`.

[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.

```
qsort([], []).
qsort([X|L], R) :-
    partition(L, X, L1, L2),
    qsort(L2, R2), qsort(L1, R1), append(R1, [X|R2], R).

partition([], _, [], []).
partition([E|R], C, Left, [E|Right1]) :- E >= C, partition(R, C, Left, Right1).
partition([E|R], C, [E|Left1], Right) :- E < C, partition(R, C, Left1, Right).

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

Fig. 4. A classical implementation of *quick sort* in Prolog (using `append/3`).

$$\mathtt{inf}(\mathtt{L},\top) \mid \mathtt{L=[]} \Rightarrow \mathtt{inf}(\mathtt{L},\infty)\ \#\ \textit{empty} \tag{14}$$
$$\mathtt{inf}(\mathtt{L},\mathtt{X}) \mid \mathtt{L=[H|T]} \Rightarrow \mathtt{inf}(\mathtt{L},\mathtt{X}),\mathtt{inf}(\mathtt{T},\mathtt{X})\ \#\ \textit{prop1} \tag{15}$$
$$\mathtt{inf}(\mathtt{T},\mathtt{X}) \mid \mathtt{L=[H|T]},\mathtt{X}\leq\mathtt{H} \Rightarrow \mathtt{inf}(\mathtt{L},\mathtt{X}),\mathtt{inf}(\mathtt{T},\mathtt{X})\ \#\ \textit{prop2} \tag{16}$$
$$\mathtt{inf}(\mathtt{T},\mathtt{X}) \mid \mathtt{L=[H|T]},\mathtt{H}\leq\mathtt{X} \Rightarrow \mathtt{inf}(\mathtt{L},\mathtt{H}),\mathtt{inf}(\mathtt{T},\mathtt{X})\ \#\ \textit{prop3} \tag{17}$$
$$\mathtt{inf}(\mathtt{L1},\mathtt{X}) \mid \mathtt{L1=L2} \Rightarrow \mathtt{inf}(\mathtt{L1},\mathtt{X}),\mathtt{inf}(\mathtt{L2},\mathtt{X})\ \#\ \textit{unif} \tag{18}$$
$$\mathtt{inf}(\mathtt{L},\mathtt{X}) \mid \mathtt{Y}\leq\mathtt{X} \Rightarrow \mathtt{inf}(\mathtt{L},\mathtt{Y})\ \#\ \textit{reduce} \tag{19}$$
$$\mathtt{inf}(\mathtt{L},\mathtt{X});\mathtt{inf}(\mathtt{L},\mathtt{Y}) \mid \mathtt{X}\leq\mathtt{Y} \Rightarrow \mathtt{inf}(\mathtt{L},X)\ \#\ \textit{lub} \tag{20}$$

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 `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 $\{\top, \textit{sorted}, \textit{unsorted}, \bot\}$ 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 *sorted* and *unsorted* cannot be compared. Now we start defining rules about

$$\mathtt{sup}(\mathtt{L},\top) \mid \mathtt{L=[]} \Rightarrow \mathtt{sup}(\mathtt{L},-\infty)\ \#\ \textit{empty} \tag{21}$$
$$\mathtt{sup}(\mathtt{L},\mathtt{X}) \mid \mathtt{L=[H|T]} \Rightarrow \mathtt{sup}(\mathtt{L},\mathtt{X}),\mathtt{sup}(\mathtt{T},\mathtt{X})\ \#\ \textit{prop1} \tag{22}$$
$$\mathtt{sup}(\mathtt{T},\mathtt{X}) \mid \mathtt{L=[H|T]},\mathtt{H}\leq\mathtt{X} \Rightarrow \mathtt{sup}(\mathtt{L},\mathtt{X}),\mathtt{sup}(\mathtt{T},\mathtt{X})\ \#\ \textit{prop2} \tag{23}$$
$$\mathtt{sup}(\mathtt{T},\mathtt{X}) \mid \mathtt{L=[H|T]},\mathtt{X}\leq\mathtt{H} \Rightarrow \mathtt{sup}(\mathtt{L},\mathtt{H}),\mathtt{sup}(\mathtt{T},\mathtt{X})\ \#\ \textit{prop3} \tag{24}$$
$$\mathtt{sup}(\mathtt{L1},\mathtt{X}) \mid \mathtt{L1=L2} \Rightarrow \mathtt{sup}(\mathtt{L1},\mathtt{X}),\mathtt{sup}(\mathtt{L2},\mathtt{X})\ \#\ \textit{unif} \tag{25}$$
$$\mathtt{sup}(\mathtt{L},\mathtt{X}) \mid \mathtt{X}\leq\mathtt{Y} \Rightarrow \mathtt{sup}(\mathtt{L},\mathtt{Y})\ \#\ \textit{reduce} \tag{26}$$
$$\mathtt{sup}(\mathtt{L},\mathtt{X});\mathtt{sup}(\mathtt{L},\mathtt{Y}) \mid \mathtt{X}\leq\mathtt{Y} \Rightarrow \mathtt{sup}(\mathtt{L},X)\ \#\ \textit{lub} \tag{27}$$

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/*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 *glb*, taking into account the order that we introduced before: if a variable has been abstracted to *sorted* and also to *unsorted* the *glb* is $\bot$; if the variable has been abstracted to $\top$ and to any other element, then the latter is the *glb*; 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, **inf** and **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 **inf**(L, X) (resp. **sup**(L, X)). With the help of the new **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 polyhedra domain based on [2]. The sets of *AND*- and *OR*-rules for the **inf** and **sup** domains are in fact subsets of *AND*(**inf**, (**depth**-k,**polyCLPQ**)) and *OR*(**inf**, (**depth**-k,**polyCLPQ**)), shown in Fig. 5, and *AND*(**sup**, (**depth**-k,**polyCLPQ**)) and *OR*(**sup**, (**depth**-k,**polyCLPQ**)), shown in Fig. 6, respectively. The sets of *AND*- and *OR*-rules for the **sort** domain that are presented in Fig. 7 are subsets of *AND*(**sort**, (**inf**, **depth**-k,**polyCLPQ**)) and *OR*(**sort**, (**inf**, **depth**-k, **polyCLPQ**)) respectively. However the previously defined rules may not be enough to infer sortedness. With the domains **inf** and **sup** the analysis after `partition(L, X, L1, L2)` in the `qsort` algorithm in Fig. 4 would have inferred that **sup**(L1, X) and **inf**(L2, X) (the complete analysis of the `partition/4` predicate with the **inf** domain can be found in Appendix C but even assuming that after the `qsort(L2, R2)`, `qsort(L1, R1)` recursive calls R2/*sorted* and R1/*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 `qsort(A, B)`, `A` is a permutation of `B`. And that if an element `X` satisfies **inf**(A, X), then clearly **inf**(B, X). To deal with this we introduce a new abstract domain, **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 **mset**(A = B) capturing that `B` is a permutation of `A` and **mset**(A = B+C) meaning that the multiset `A` is a permutation of the union of the multisets `B` and `C`.

Consider the goal `append(R1, S, R)` together with the information inferred on call ({R1/*sorted*, R2/*sorted*}, {A=[X|L], S=[X|R2]}, {}, {**inf**(L2, X), **inf**(R2, X)}, {**sup**(L1, X), **sup**(R1, X)}, {**mset**(L = L1+L2), **mset**(L1 = R1), **mset**(L2 = R2)}). In the first case we get the unifications: C_1 = ({R1 = [], S = Z}, {R1/*sorted*, S/*sorted*, Z/*sorted*}, {**inf**(S, X)}, {**sup**(R1, X)})

$$\mathtt{L}/\top \mid \mathtt{L=[]} \Rightarrow \mathtt{L}/sorted \;\#\; absEmpty \tag{28}$$
$$\mathtt{L}/X \mid \mathtt{L=S} \Rightarrow \mathtt{L}/X, \mathtt{S}/X \;\#\; unif \tag{29}$$
$$\mathtt{T}/sorted \mid \mathtt{L=[H|T]}, \mathtt{inf}(\mathtt{T},\mathtt{X}), \mathtt{H} \leq \mathtt{X} \Rightarrow \mathtt{T}/sorted, \mathtt{L}/sorted \;\#\; sortProp \tag{30}$$
$$\mathtt{T}/sorted \mid \mathtt{L=[H|T]}, \mathtt{inf}(\mathtt{T},\mathtt{X}), \mathtt{X} < \mathtt{H} \Rightarrow \mathtt{T}/sorted, \mathtt{L}/unsorted \;\#\; unsortPrp \tag{31}$$
$$\mathtt{L}/sorted \mid \mathtt{L=[H|T]} \Rightarrow \mathtt{L}/sorted, \mathtt{T}/sorted \;\#\; sortInh \tag{32}$$
$$\mathtt{L}/sorted, \mathtt{L}/unsorted \Rightarrow \mathtt{L}/\bot \;\#\; glb1 \tag{33}$$
$$\mathtt{L}/\top, \mathtt{L}/\mathtt{X} \Rightarrow \mathtt{L}/\mathtt{X} \;\#\; glb2 \tag{34}$$
$$\mathtt{L}/\bot, \mathtt{L}/\mathtt{X} \Rightarrow \mathtt{L}/\bot \;\#\; glb3 \tag{35}$$
$$\mathtt{L}/sorted; \mathtt{L}/unsorted \Rightarrow \mathtt{L}/\top \;\#\; lub1 \tag{36}$$
$$\mathtt{L}/\top; \mathtt{L}/\mathtt{X} \Rightarrow \mathtt{L}/\top \;\#\; lub2 \tag{37}$$
$$\mathtt{L}/\bot; \mathtt{L}/\mathtt{X} \Rightarrow \mathtt{L}/\mathtt{X} \;\#\; lub3 \tag{38}$$

Fig. 7. Sets of *AND*-rules (28–35) and *OR*-rules (36–38) for the `sort` domain.

after applying Rules *absEmpty* 28 and *unif* 29. In the second case we have the entry: $C_2 = (\{\mathtt{R1 = [H|L]}, \mathtt{Z = [H|T]}\}, \{\mathtt{R1}/sorted, \mathtt{S}/sorted\}, \{\mathtt{inf}(\mathtt{S},\mathtt{X})\}, \{\mathtt{sup}(\mathtt{R1},\mathtt{X})\})$. However, after `append(X, Y, Z)` we have to apply the *lub* and we get: $(\{\}, \{\mathtt{R1}/sorted, \mathtt{S}/sorted\}, \{\{\mathtt{inf}(\mathtt{S},\mathtt{X})\}, \{\mathtt{sup}(\mathtt{R1},\mathtt{X})\})$ which is not proving that `R` is sorted. This is because if `Z` is sorted and $\mathtt{inf}(\mathtt{Z},\mathtt{X})$ (which is inferred in the recursive call) since `R=[H|Z]`, what we need for `R` to be sorted is that $\mathtt{H}\leq\mathtt{X}$ holds. `polyCLPQ` is not able to prove that, but it can be enhanced to use `sup` by adding a purely combinatorial rule:

$$\mathtt{true} \mid \mathtt{sup}(\mathtt{L},\mathtt{X}), \mathtt{L=[H|T]} \Rightarrow \mathtt{H} \leq \mathtt{X} \;\#\; combinePoly \tag{39}$$

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 $\mathtt{true} \mid \mathtt{L}/sorted, \mathtt{L=[H|T]}, \mathtt{inf}(\mathtt{T},\mathtt{X}) \Rightarrow \mathtt{H} \leq \mathtt{X}$, which exploits both the sortedness and the `inf` property to get better abstractions for `polyCLPQ`. Now, with this new information, we can infer that, since $\mathtt{H}\leq\mathtt{X}$, then $\mathtt{inf}(\mathtt{Z},\mathtt{H})$. With this, since `Z` is sorted and `R = [H|Z]`, then `R` is sorted and $\mathtt{inf}(\mathtt{R},\mathtt{H})$, and therefore we infer that the second argument of `qsort/2` is sorted.

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 `qsort(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:

```
:- true pred qsort(A,B)
   : ( asub([s(A,top),s(B,top)]),
       ↪ asub(([inf(A,top),inf(B,top)],[sup(A,top),sup(B,top)])), true, true )
   => ( asub([s(A,top),s(B,sorted)]),
       ↪ asub(([inf(A,top),inf(B,top)],[sup(A,top),sup(B,top)])), mset([A=B]),
       ↪ true ).
```

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

1. de Angelis, E., Proietti, M., Fioravanti, F., Pettorossi, A.: Verifying catamorphism-based contracts using constrained horn clauses. Theory and Practice of Logic Programming **22**(4), 555–572 (2022). `https://doi.org/10.1017/S1471068422000175`

2. Benoy, F., King, A., Mesnard, F.: Programming pearl: Computing convex hulls with a linear solver. TPLP **5**, 259–271 (01 2005). https://doi.org/10.1017/S1471068404002261
3. Braine, J., Gonnord, L., Monniaux, D.: Data Abstraction: A General Framework to Handle Program Verification of Data Structures. In: SAS 2021 - 28th Static Analysis Symposium. LNCS, vol. 12913, pp. 215–235. Chicago, United States (Oct 2021). https://doi.org/10.1007/978-3-030-88806-0_11, https://inria.hal.science/hal-03321868
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. SIGPLAN Not. **44**(10), 243–262 (October 2009). https://doi.org/10.1145/1639949.1640108
5. Bruynooghe, M., Codish, M.: Freeness, sharing, linearity and correctness–all at once. In: Proc. Third International Workshop on Static Analysis. pp. 153–164. No. 724 in LNCS, Springer (1993)
6. Bueno, F., Cabeza, D., Hermenegildo, M.V., Puebla, G.: Global Analysis of Standard Prolog Programs. In: ESOP (1996). https://doi.org/10.1007/3-540-61055-3_32
7. Campion, M., Preda, M.D., Giacobazzi, R.: Partial (in)completeness in abstract interpretation: limiting the imprecision in program analysis. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498721
8. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE TKDE **1**(1), 146–166 (1989). https://doi.org/10.1109/69.43410
9. Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M., Hermenegildo, M.: Improving Abstract Interpretations by Combining Domains. ACM TOPLAS **17**(1), 28–44 (January 1995)
10. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: ACM Symposium on Principles of Programming Languages (POPL'77). pp. 238–252. ACM Press (1977). https://doi.org/10.1145/512950.512973
11. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: POPL'79. pp. 269–282. ACM (1979)
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: ESOP 2005. pp. 21–30 (2005)
13. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and Transformation of Constrained Horn Clauses for Program Verification. TPLP (2021)
14. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. JLP, Special Issue on CLP **37**(1–3) (October 1998)
15. Garcia-Contreras, I., Morales, J.F., Hermenegildo, M.V.: Incremental and Modular Context-sensitive Analysis. TPLP **21**(2), 196–243 (January 2021)
16. Garcia-Contreras, I., Morales, J., Hermenegildo, M.V.: Incremental Analysis of Logic Programs with Assertions and Open Predicates. In: LOPSTR'19. LNCS, vol. 12042, pp. 36–56. Springer (2020)
17. García de la Banda, M., Hermenegildo, M.V., Bruynooghe, M., Dumortier, V., Janssens, G., Simoens, W.: Global Analysis of Constraint Logic Programs. ACM Trans. on Programming Languages and Systems **18**(5), 564–615 (1996)
18. Giacobazzi, R., Ranzato, F.: History of abstract interpretation. IEEE Ann. Hist. Comput. **44**(2), 33–43 (2022), https://doi.org/10.1109/MAHC.2021.3133136

19. Henriksen, K.S., Gallagher, J.P.: Abstract Interpretation of PIC Programs through Logic Programming. In: SCAM '06. pp. 184–196. IEEE Computer Society (2006). https://doi.org/10.1109/SCAM.2006.1
20. Hermenegildo, M., Puebla, G., Bueno, F., Garcia, P.L.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2), 115–140 (2005)
21. Hermenegildo, M.V., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm: a 25–Year Perspective, pp. 161–192. Springer-Verlag (1999)
22. Hermenegildo, M.V., Puebla, G., Marriott, K., Stuckey, P.: Incremental Analysis of Constraint Logic Programs. ACM TOPLAS **22**(2), 187–223 (March 2000). https://doi.org/10.1145/349214.349216
23. Jacobs, D., Langen, A.: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In: North American Conference on Logic Programming (1989)
24. Kelly, A., Marriott, K., Søndergaard, H., Stuckey, P.: A Practical Object-Oriented Analysis Engine for CLP. Software: Practice and Experience **28**(2), 188–224 (1998)
25. Le Charlier, B., Van Hentenryck, P.: Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. ACM TOPLAS **16**(1), 35–101 (1994)
26. Madsen, M., Yee, M., Lhoták, O.: From Datalog to FLIX: a Declarative Language for Fixed Points on Lattices. In: PLDI, ACM. pp. 194–208 (2016)
27. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: LOPSTR. LNCS, vol. 4915, pp. 154–168. Springer-Verlag (August 2007). https://doi.org/10.1007/978-3-540-78769-3_11
28. Méndez-Lojo, M., Navas, J., Hermenegildo, M.V.: An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In: ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07) (2007)
29. Mesnard, F., Neumerkel, U.: CHR for Protoyping Abstract Interpretation (1997), http://lim.univ-reunion.fr/staff/fred/Publications/00-MesnardN.pdf, unpublished note
30. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: NACLP'89. pp. 166–189. MIT Press (October 1989)
31. Muthukumar, K., Hermenegildo, M.: Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Comp. Tech. Corp. (MCC) (April 1990), http://cliplab.org/papers/mcctr-fixpt.pdf
32. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. JLP **13**(2/3), 315–347 (July 1992)
33. Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (2000). https://doi.org/10.1007/10722311_2
34. Puebla, G., Hermenegildo, M.V.: Optimized Algorithms for the Incremental Analysis of Logic Programs. In: SAS'96. pp. 270–284. Springer LNCS 1145 (1996). https://doi.org/10.1007/3-540-61739-6_47
35. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., Martín-Mateos, F.J.: Termination in ACL2 Using Multiset Relations, pp. 217–245. Springer Netherlands, Dordrecht

(2003). https://doi.org/10.1007/978-94-017-0253-9_9, https://doi.org/10.1007/978-94-017-0253-9_9
36. Sato, T., Tamaki, H.: Enumeration of success patterns in logic programs. In: Diaz, J. (ed.) Automata, Languages and Programming. pp. 640–652. Springer Berlin Heidelberg, Berlin, Heidelberg (1983)
37. Seidl, H., Vogler, R.: Three improvements to the top-down solver. Math. Struct. Comput. Sci. **31**(9), 1090–1134 (2021). https://doi.org/10.1017/S0960129521000499
38. Tilscher, S., Stade, Y., Schwarz, M., Vogler, R., Seidl, H.: The Top-Down Solver—An Exercise in A^{2}I. In: Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (eds.) Challenges of Software Verification, vol. ISRL 238, chap. 9, pp. 157–179. Springer, Singapore (2023). https://doi.org/10.1007/978-981-19-9601-6_9
39. Van Roy, P., Despain, A.M.: The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In: North American Conf. on Logic Programming. pp. 501–515. MIT Press (October 1990)
40. Warren, R., Hermenegildo, M., Debray, S.K.: On the Practicality of Global Flow Analysis of Logic Programs. In: JICSLP. pp. 684–699. MIT Press (August 1988)
41. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI. pp. 131–144. ACM (2004)

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: $t_1 \leq t_2 \Leftrightarrow \texttt{instance}(t_1, t_2)$, i.e., iff there exists a substitution θ 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 $\{\texttt{a,b,c}\}$. Now, given multisets M, N, S over a set A the following properties hold:

- $M \subseteq N \Leftrightarrow M(x) \leq N(x) \forall x \in A$
- $S = M \cup N \Leftrightarrow S(x) = M(x) + N(x) \forall x \in A$
- $S = M \backslash N \Leftrightarrow S(x) = M(x) \dot{-} N(x)\, \forall x \in A$

where $x \dot{-} y$ is $x - y$ if $x \geq y$ and 0 otherwise. These equivalences allow us to

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

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

$$\texttt{true} \mid \texttt{X=Y} \Rightarrow \texttt{mset}(\texttt{X} = \texttt{Y}) \;\#\; \textit{unif} \tag{42}$$

Fig. 8. Sets of *AND*-rules used to define the `mset` domain.

```
:- true pred partition(L,X,L1,L2) => mset([L=L2+L1]).

partition(L,X,L1,L2) :-
    true(true),
    L=[],
    true(mset([L=0])),
    L1=[],
    true(mset([L=0,L1=0])),
    L2=[],
    true(mset([L=L1+L2])).
partition(L,X,L1,L2) :-
    true(true),
    L=[Y|Temp],
    true(mset([L=Y+Temp])),
    Y=<X,
    true(mset([L=Temp+Y])),
    partition(Temp,X,Temp1,L2),
    true(mset([L=Y+Temp,Temp1= L-Y-L2])),
    L1=[Y|Temp1],
    true(mset([L2=L-L1,Temp=L-Y,Temp1=L1-Y])).
partition(L,X,L1,L2) :-
    true(true),
    L=[Y|Temp],
    true(mset([L=Y+Temp])),
    X=<Y,
    true(mset([L=Temp+Y])),
    partition(Temp,X,L1,Temp2),
    true(mset([L1= -Y+L-Temp2,Temp= -Y+L])),
    L2=[Y|Temp2],
    true(mset([L1=L-L2,Temp=L-Y,Temp2=L2-Y])).
```

Fig. 9. Analysis result of applying the `mset` analysis over the `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 `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 `CiaoPP`'s `polyCLPQ` domain (an implementation of polyhedra using CLP(Q) based on [2]) and run its *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 *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 `polyCLPQ` operations. It is also important to note that since we are not defining *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

```
:- module(_1,[partition/4],[assertions]).

:- true pred partition(L,_X,L1,L2)
   : ( asub([inf(L,'$top'),inf(_X,'$top'),inf(L1,'$top'),inf(L2,'$top')]), true )
   => ( asub([inf(L,'$top'),inf(_X,'$top'),inf(L1,'$top'),inf(L2,_X)]), 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,0.Inf),inf(_X,'$top'),inf(L1,'$top'),inf(L2,'$top')]),
        'terms_check:instance'(L,[]),
        true
    )),
    L1=[],
    true((
        asub([inf(L,0.Inf),inf(_X,'$top'),inf(L1,0.Inf),inf(L2,'$top')]),
        'terms_check:instance'(L,[]),
        'terms_check:instance'(L1,[]),
        true
    )),
    L2=[],
    true((
        asub([inf(L,0.Inf),inf(_X,'$top'),inf(L1,0.Inf),inf(L2,0.Inf)]),
        '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,[Y|Temp]),
        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,'$top')]),
        'terms_check:instance'(L,[Y|Temp]),
        'native_props:constraint'([Y=<X])
    )),
    partition(Temp,X,Temp1,L2),
    true((
         asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,X),
               inf(Y,'$top'),inf(Temp,'$top'),inf(Temp1,'$top')]),
        'terms_check:instance'(L,[Y|Temp]),
        'native_props:constraint'([X>=Y])
    )),
```

```
    L1=[Y|Temp1],
    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,[Y|Temp]),
        'terms_check:instance'(L1,[Y|Temp1]),
        'native_props:constraint'([Y=<X])
    )).
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(Temp2,'$top')]),
        true
    )),
    L=[Y|Temp],
    true((
         asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top'),
               inf(Y,'$top'),inf(Temp,'$top'),inf(Temp2,'$top')]),
        'terms_check:instance'(L,[Y|Temp]),
        true
    )),
    X=<Y,
    true((
        asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top'),
              inf(Y,'$top'),inf(Temp,'$top'),inf(Temp2,'$top')]),
        'terms_check:instance'(L,[Y|Temp]),
        'native_props:constraint'([Y>=X])
    )),
    partition(Temp,X,L1,Temp2),
    true((
         asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,'$top'),
               inf(Y,'$top'),inf(Temp,'$top'),inf(Temp2,X)]),
        'terms_check:instance'(L,[Y|Temp]),
        'native_props:constraint'([X=<Y])
    )),
    L2=[Y|Temp2],
    true((
         asub([inf(L,'$top'),inf(X,'$top'),inf(L1,'$top'),inf(L2,X),
               inf(Y,'$top'),inf(Temp,'$top'),inf(Temp2,X)]),
        'terms_check:instance'(L,[Y|Temp]),
        'terms_check:instance'(L2,[Y|Temp2]),
        'native_props:constraint'([Y>=X])
    )).
```

D Inferred info for **qsort/2**

```
:- true pred qsort(X,Y)
   : ( asub([s(X,'$top'), s(Y,'$top')]), asub(([inf(X,'$top'),
       ↪ inf(Y,'$top')],[sup(X,'$top'), sup(Y,'$top')])), true, true )
   => ( asub([s(X,'$top'), s(Y,sorted)]), asub([inf(X,'$top'),
       ↪ inf(Y,'$top')]),asub([sup(X,'$top'), sup(Y,'$top')]), mset([X=Y]),
       ↪ true ).

qsort(X,Y) :-
    true((
        asub([s(X,'$top'), s(Y,'$top')]),
        asub([inf(X,'$top'), inf(Y,'$top')]),
        asub([sup(X,'$top'), sup(Y,'$top')]),
        true,
        true
    )),
    X=[],
    true((
```

```
        asub([s(X,sorted), s(Y,'$top')]),
        asub([inf(X,0.Inf), inf(Y,'$top')]),
        asub([sup(X,-0.Inf), sup(Y,'$top')]),
        mset([X=0]),
        'terms_check:instance'(X,[]),
        true
    )),
    Y=[],
    true((
        asub([s(X,sorted), s(Y,sorted)]),
        asub([inf(X,0.Inf), inf(Y,0.Inf)]),
        asub([sup(X,-0.Inf), sup(Y,-0.Inf)]),
        mset([Y=X]),
        'terms_check:instance'(X,[]),
        'terms_check:instance'(Y,[]),
        true
    )).
qsort(A,R) :-
    true((
        asub([s(A,'$top'), s(R,'$top'), s(X,'$top'), s(L,'$top'), s(L1,'$top'),
            ↪ s(L2,'$top'), s(R2,'$top'), s(R1,'$top'), s(B,'$top')]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'),
               inf(L1,'$top'), inf(L2,'$top'), inf(R2,'$top'), inf(R1,'$top'),
                   ↪ inf(B,'$top')]),
        asub([sup(A,'$top'), sup(R,'$top'), sup(X,'$top'), sup(L,'$top'),
            ↪ sup(L1,'$top'), sup(L2,'$top'),
               sup(R2,'$top'), sup(R1,'$top'), sup(B,'$top')]),
        true,
        true
    )),
    A=[X|L],
    true((
        asub([s(A,'$top'), s(R,'$top'), s(X,'$top'), s(L,'$top'), s(L1,'$top'),
            ↪ s(L2,'$top'),
               s(R2,'$top'), s(R1,'$top'), s(B,'$top')]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'),
            ↪ inf(L1,'$top'), inf(L2,'$top'),
               inf(R2,'$top'), inf(R1,'$top'), inf(B,'$top')]),
        asub([sup(A,'$top'), sup(R,'$top'), sup(X,'$top'), sup(L,'$top'),
            ↪ sup(L1,'$top'), sup(L2,'$top'),
               sup(R2,'$top'), sup(R1,'$top'), sup(B,'$top')]),
        mset([A=X+L]),
        'terms_check:instance'(A,[X|L]),
        true
    )),
    partition(L,X,L1,L2),
    true((
        asub([s(A,'$top'), s(R,'$top'), s(X,'$top'), s(L,'$top'), s(L1,'$top'),
            ↪ s(L2,'$top'),
               s(R2,'$top'), s(R1,'$top'), s(B,'$top')]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'),
            ↪ inf(L1,'$top'), inf(L2,X),
               inf(R2,'$top'), inf(R1,'$top'), inf(B,'$top')]),
        asub([sup(A,'$top'), sup(R,'$top'), sup(X,'$top'), sup(L,'$top'),
            ↪ sup(L1,X), sup(L2,'$top'),
               sup(R2,'$top'), sup(R1,'$top'), sup(B,'$top')]),
        mset([L= -X+A,L1= -X+A-L2]),
        'terms_check:instance'(A,[X|L]),
        true
    )),
    qsort(L2,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,'$top'), s(B,'$top')]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'),
            ↪ 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'), 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,'$top'), s(L2,'$top'),
            s(R2,sorted), s(R1,sorted), s(B,'$top')]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'), inf(L1,'$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'), 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'), s(L2,'$top'),
            s(R2,sorted), s(R1,sorted), s(B,sorted)]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'), inf(L1,'$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,'$top'), sup(L1,X), sup(L2,'$top'),
            sup(R2,'$top'), sup(R1,X), sup(B,'$top')]),
        mset([L=A-X,L1= -B+A,L2=B-X,R2=B-X,R1= -B+A]),
        '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'), s(L2,'$top'),
           s(R2,sorted), s(R1,sorted), s(B,sorted)]),
        asub([inf(A,'$top'), inf(R,'$top'), inf(X,'$top'), inf(L,'$top'), inf(L1,'$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,'$top'), sup(L1,X), sup(L2,'$top'),
           sup(R2,'$top'), sup(R1,X), sup(B,'$top')]),
        mset([A=L+X,R=L+X,L1=L-R2,L2=R2,R1=L-R2,B=R2+X]),
        'terms_check:instance'(A,[X|L]),
        true
    )).
```