Relational Abstractions Based on Labeled Union-Find (with appendices)

Dorian Lesbre, Matthieu Lemerre, Hichem Rami Ait-El-Hara, François Bobot

Relational Abstractions Based on Labeled Union-Find (with appendices)

DORIAN LESBRE, Université Paris-Saclay, CEA, List, France
MATTHIEU LEMERRE, Université Paris-Saclay, CEA, List, France
HICHEM RAMI AIT-EL-HARA, OCamlPro, France and Université Paris-Saclay, CEA, List, France
FRANÇOIS BOBOT, Université Paris-Saclay, CEA, List, France

We introduce a new family of abstractions based on a data structure that we call *labeled union-find*, an extension of the classic efficient union-find data structure where edges carry labels. These labels have a composition operation that obey the group axioms. Like union-find, the labeled version can efficiently compute the transitive closure of a relation, but it is not limited to equivalence relations; it can represent any injective transformation between equivalence classes, which includes two-variables per equality (TVPE) constraints of the form $y = a \times x + b$. Using abstract interpretation theory, we study the properties deriving from the use of abstract relations as labels, and the combination of labeled union-find with other representations of constraints, allowing both improvements in precision and simplification of existing constraints. Due to its efficiency, the labeled union-find abstractions could find many uses; we use it in two use cases, program analysis based on abstract interpretation and constraint solving for SMT, with encouraging preliminary results.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Abstraction**; **Equational logic and rewriting**.

Additional Key Words and Phrases: Relational abstract domain, Labeled union-find, Abstract interpretation

1 Introduction

In abstract interpretation, it is common to classify numerical domains into two extremes: the *relational* polyhedra abstract domain [Cousot and Halbwachs 1978], expressing conjunction of linear equalities, which is slow and precise; and the *non-relational* box abstract domain [Cousot and Cousot 1977], expressing the range of values of each variable independently, fast but imprecise. In the middle, one can find *weakly-relational* abstract domains, of intermediate expressivity (they only represent binary relations between variables) and cost, such as octagons [Miné 2006]. These weakly relational domains are already quite costly, requiring $O(|\mathbb{X}|^2)$ storage cost per abstract state, and $O(|\mathbb{X}|^3)$ for the transitive closure computation (where $|\mathbb{X}|$ is the number of program variables). Therefore, using them makes an analysis several orders of magnitude slower [Nazaré et al. 2014]. A long line of research was thus devoted to mitigating this cost. The general emerging strategy is to split a monolithic relational domain into independent clusters, either by exploiting existing loose coupling between variables [Gange et al. 2021; Jourdan 2016; Singh et al. 2015, 2018] or by forcibly decoupling the clusters [Blanchet et al. 2003; Heo et al. 2016; Venet and Brat 2004]. Note that even with these mitigations, many analyzers do not enable these domains by default due to their cost.

One approach that was seldom explored (except by Cox et al. [2015]; Halbwachs et al. [2006]) is, on the contrary, to exploit tight coupling between variables to make relational domains more efficient. For instance, maintaining equality (the tightest possible relation) between variables is cheaply done using the highly efficient union-find data structure. Furthermore, it allows for possible speed-up by grouping equal variables using and performing a quotient on other constraints.

We generalize this technique to relations other than equivalence relations, by describing a *new family of very cheap (weakly-)relational abstract domains* based on an extension

Authors' Contact Information: Dorian Lesbre, Université Paris-Saclay, CEA, List, Palaiseau, France, dorian.lesbre@cea.fr; Matthieu Lemerre, Université Paris-Saclay, CEA, List, Palaiseau, France, matthieu.lemerre@cea.fr; Hichem Rami Ait-El-Hara, OCamlPro, Paris, France, hichem.ait-el-hara@ocamlpro.com and Université Paris-Saclay, CEA, List, Palaiseau, France; François Bobot, Université Paris-Saclay, CEA, List, Palaiseau, France, francois.bobot@cea.fr.

of the union-find data structure that we call *labeled union-find*. Starting from a general formalization of weakly-relational domains (Section 2), we derive our key ingredients for efficient weakly-relational domains: *constraint elimination* (finding constraints that can be removed without changing the concretization, and re-computing them dynamically), and a *unique label* hypothesis that allows *efficient computation of transitive closure* (which is the performance bottleneck).

To efficiently implement this new family of abstract domains, we use the *labeled union-find* data structure (Section 3). We rediscovered this data structure, which was previously introduced by Frühwirth [2007]. Labels in a labeled union-find can be composed following the *group* laws (associativity, existence of a neutral element and inverse), and we argue that group laws naturally derive from the unique label hypothesis. We then provide and prove correct an implementation of the main labeled union-find operations, and study an extension where information is attached to each class of related elements. This structure may well find other applications outside our use case.

For program analysis, we use labeled union-find domains to represent relations between variables. The labels (abstract relations) must thus simultaneously be sound and obey the group axioms. By studying the conjunction of these two constraints (Section 4), we discover that our labeled union-find data structure represents *injective transformations between equivalence classes*. This allows not only to dismiss unsuitable abstract relations, but also to easily find new ones. We propose different examples throughout the paper, including:

The two-values per equality abstract domain (TVPE): encoding affine relations $y = a \times x + b$ between two integer, rational or real variables x and y;

The modular two-values per equality abstract domain: encoding affine relations $y = a \times x + b$ between two bitvectors using modulo arithmetic, but only in the case where a is odd;

Xor and rotations: of the form $y = (x \texttt{ xor } c) \texttt{ rot } n$ between two bitvectors, which encodes bitwise negation and many shifts;

Invertible matrix multiplications: relations $y = a \times x + b$ where y and x represent vector of values in a field, a an invertible constant matrix, and b a constant vector;

Equivalence between sequences modulo relocation: [Ait-El-Hara et al. 2024b] are relations $y =_{\texttt{reloc}(c)} x$ where two sequences have equal contents but their indices are shifted by c;

Conversions between bitvectors and their signed or unsigned representation;

Next, we examine the many interesting combinations between labeled union-find and other abstractions, such as the non-relational abstraction (Section 5), general weakly-relational abstractions, equalities (Section 6.1) and linear equalities (Section 6.2). We study two kinds of interactions: standard reduced product, but also *constraint factorization*, where we "quotient" existing relations using the *relational class* of related variables in the labeled union-find.

Throughout this paper, we use abstract interpretation [Cousot and Cousot 1977] as an invaluable theory to reason about precision gaps between the representations of operations in data structures (the abstract), and how they affect the set of known facts (the concrete). This theory is at the heart of all our theorems and proofs (Appendix B). Our labeled union-find domain can be used as a standard flow-sensitive abstract domain, and we provide a join operation for it in Appendix A. However, this does not mean that labeled union-find is restricted to program analysis: abstract interpretation is a general theory of program approximation, and indeed labeled union-find abstractions could also be used in decision procedures, datalog engines [Nappa et al. 2019; Sahebolamri et al. 2023], constraint solvers, model checkers, etc. In Section 7, we present early implementation results in both a non-relational abstract interpreter, where it has a low performance impact and offers some precision gains, and in a constraint solver, where it allows to easily add new capabilities to the solver without compromising performance across a large set of benchmarks.

This is the technical report accompanying the published paper [Lesbre et al. 2025b]. The implementation is available in the software artifact [Lesbre et al. 2025a].

2 Faster Relational Abstract Domains by Assuming Unique Labels

This section provides a high-level formal presentation of weakly relational domains [Miné 2004; Schwarz et al. 2023], which represent conjunctions of binary abstract relations between pairs of variables. Then, we introduce and motivate the unique-label hypothesis, that improves performance by enabling fast transitive closure computation in a union-find like structure.

2.1 Weakly Relational Domains

2.1.1 Abstract Relations. We consider abstract relations, which are (generally) a finite and computable representation of a *concrete* mathematical relation. Formally, suppose we are given a set of *values* $v \in \mathbb{V}$ (like integers $\mathbb{Z}$ or word-sized bitvectors $\mathbb{BV}_{64}$). Then, an *abstract relation* $\mathcal{R}^\sharp \in \mathbb{R}^\sharp$ represents a concrete relation via a *concretization* operator $\gamma_{\mathbb{R}^\sharp} \in \mathbb{R}^\sharp \to \mathcal{P}(\mathbb{V} \times \mathbb{V})$.

Example 2.1 (Constant difference abstract relation). Suppose $\mathbb{V} = \mathbb{Z}$, the set of integers. We can use a number k to represent a relation stating that the difference between two values is k. Formally, we have $\mathbb{R}^\sharp \triangleq \mathbb{Z}$ and $\gamma_{\mathbb{R}^\sharp}(k) \triangleq \left\{(z_1, z_2) \in \mathbb{Z}^2 \;\middle|\; z_2 - z_1 = k\right\}$.

Example 2.2 (Interval difference abstract relation). We can extend Example 2.1 by constraining the difference to be in an interval, instead of being in a single value. In that case, we use $\mathbb{R}^\sharp \triangleq \mathbb{Z} \times \mathbb{Z}$ and (denoting intervals as $[a; b]$): $\gamma_{\mathbb{R}^\sharp}([a; b]) \triangleq \{(z_1, z_2) \in \mathbb{Z} \mid a \leq z_2 - z_1 \wedge z_2 - z_1 \leq b\}$.

Miné [2002] provides more examples of abstract relations of the form $z_2 - z_1 \in S$. But there are also weakly relational domains outside this class; for instance, using pairs of intervals $([a; b], [c; d])$ with $\gamma_{\mathbb{R}^\sharp}([a; b], [c; d]) = \{(z_1, z_2) \in \mathbb{Z} \mid a \leq z_2 - z_1 \leq b \wedge c \leq z_2 + z_1 \leq d\}$, we can represent octagons [Miné 2006]. Another example is the following one:

Example 2.3 (Bitvector comparison abstract relation). Suppose $\mathbb{V} = \mathbb{BV}_n$ (the set of bitvectors of length n). We can use a *tristate number* [Vishwanathan et al. 2022] (a kind of bitvector whose bits are numbers in the set $\{0, 1, ?\}$, where $?$ represents an unknown value; often used to track known bits in static analyses [Miné 2012; Regehr et al. 2005] or constraint solvers [Chihani et al. 2017; Michel and Hentenryck 2012]) to represent whether the bits at the same index in a bitvector are equal, are different, or if we don't know their relation. Formally, we have $\mathbb{R}^\sharp \triangleq \{0, 1, ?\}^n$, and

$$\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \triangleq \left\{(v, w) \;\middle|\; \forall\, i,\ 0 \leq i < n \implies \quad (\mathcal{R}^\sharp_i = 0 \implies v_i = w_i) \quad \wedge \quad (\mathcal{R}^\sharp_i = 1 \implies v_i = \neg w_i)\right\}$$

2.1.2 Operations on Abstract Relations. We will need some operations on abstract relations, which perform sound over-approximation of the composition ($\fatsemi$), inversion ($\cdot^{-1}$), and identity relation. Formally, we require these operations to fulfill the following soundness rules:

$$\texttt{id}^\sharp \in \mathbb{R}^\sharp \qquad \texttt{inv}^\sharp \in \mathbb{R}^\sharp \to \mathbb{R}^\sharp \qquad \cdot \fatsemi^\sharp \cdot \in \mathbb{R}^\sharp \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$$

$$\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \supseteq \{(v, v) \mid v \in \mathbb{V}\} \tag{HIdentitySound}$$

$$\gamma_{\mathbb{R}^\sharp}(\texttt{inv}^\sharp(\mathcal{R}^\sharp)) \supseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)^{-1} \triangleq \{(v_2, v_1) \mid (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} \tag{HInverseSound}$$

$$\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp) \supseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \triangleq \left\{(v_1, v_3) \;\middle|\; \exists\, v_2,\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \wedge (v_2, v_3) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)\right\} \tag{HComposeSound}$$

When we can replace the $\supseteq$ sign in the above rules with an equality, the operation is said to be both sound and *exact* (i.e., γ-complete [Giacobazzi and Quintarelli 2001; Ranzato 2013]).

Example 2.4. In the constant and interval difference abstract relations, we have $\texttt{id}^\sharp \triangleq 0$ (or $(0, 0)$); $\texttt{inv}^\sharp(\mathcal{R}^\sharp) = -\mathcal{R}^\sharp$; and $\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp \triangleq \mathcal{R}_1^\sharp + \mathcal{R}_2^\sharp$. In the bitvector comparison relation, we have $\texttt{id}^\sharp \triangleq 0$; $\texttt{inv}^\sharp(\mathcal{R}^\sharp) = \mathcal{R}^\sharp$; and $\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp \triangleq \mathcal{R}_1^\sharp \texttt{ xor } \mathcal{R}_2^\sharp$. All of these operations are both sound and exact.

Another operation that we will need over abstract relations is precision ordering $\sqsubseteq_{\mathbb{R}^\sharp}$. Intuitively, $\mathcal{R}_1^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathcal{R}_2^\sharp$ means that $\mathcal{R}_1^\sharp$ represents a tighter constraint than $\mathcal{R}_2^\sharp$, which is formalized as follows:

$$\cdot \sqsubseteq_{\mathbb{R}^\sharp} \cdot \in \mathcal{P}\left(\mathbb{R}^\sharp \times \mathbb{R}^\sharp\right) \qquad \mathcal{R}_1^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathcal{R}_2^\sharp \implies \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \qquad \text{(HOrderSound)}$$

Example 2.5. The constant different abstract relations form a flat lattice: different elements cannot be compared in terms of precision. For the interval difference abstract relation, we have $(a_1, a_2) \sqsubseteq_{\mathbb{R}^\sharp} (b_1, b_2)$ if and only if $b_1 \leq a_1$ and $a_2 \leq b_2$. The precision ordering between bitvector comparison relations is the component-wise ordering of the three-valued lattice ($0 \sqsubseteq ?$ and $1 \sqsubseteq ?$); i.e., $\mathcal{R}_1^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathcal{R}_2^\sharp$ means that at each index, $\mathcal{R}_2^\sharp$ either contains the same "bit" as $\mathcal{R}_1^\sharp$, or contains ?.

From the order $\sqsubseteq_{\mathbb{R}^\sharp}$, we can define the meet $\sqcap_{\mathbb{R}^\sharp} \in \mathbb{R}^\sharp \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$ as the smallest over-approximation of intersection of relations, and the join $\sqcup_{\mathbb{R}^\sharp}$ as the smallest over-approximation of the union. Note that the meet $\sqcap_{\mathbb{R}^\sharp}$ is both sound and exact in all our examples.

2.1.3 Weakly Relational Abstract Domains. Suppose that we want to maintain binary relations between *variables* $x, y \in \mathbb{X}$. A natural way to do so is to use a labeled directed graph, whose nodes are variables. An edge from x to y (if present) is labeled with the abstract relation between x and y. If there is no edge from x to y, then the relation between x and y is not constrained.

Formally, we define elements of a *weakly-relational abstract domain* $\mathcal{W} \in \mathbb{W}^\sharp \triangleq (\mathbb{X} \times \mathbb{X}) \rightharpoonup \mathbb{R}^\sharp$ as a partial map from variable pairs to their possible abstract relation. We do not allow for multiple edges between a pair of variables, as they can be replaced with a single edge carrying the meet of all labels; especially when the meet operation is exact. The meaning of a weakly relational domain is the set of all the *valuations* (mapping from variables to values, noted σ) that fulfill its constraints:

$$\gamma_{\mathbb{W}^\sharp} \in \mathbb{W}^\sharp \to \mathcal{P}(\mathbb{X} \to \mathbb{V})$$
$$\gamma_{\mathbb{W}^\sharp}(\mathcal{W}) \triangleq \left\{\sigma \in \mathbb{X} \to \mathbb{V} \;\middle|\; \forall (x, y) \in \operatorname{dom} \mathcal{W},\ (\sigma[x], \sigma[y]) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{W}[x, y])\right\}$$

We can also define a precision ordering $\cdot \sqsubseteq_{\mathbb{W}^\sharp} \cdot \in \mathcal{P}\left(\mathbb{W}^\sharp \times \mathbb{W}^\sharp\right)$ on weakly relational domains. Intuitively, $\mathcal{W}$ is more precise than $\mathcal{W}'$ if $\mathcal{W}$ has more edges than $\mathcal{W}'$ (i.e., more constraints between variables), and if the labels on the edges of $\mathcal{W}$ are more precise (i.e., stricter constraints):

$$\mathcal{W} \sqsubseteq_{\mathbb{W}^\sharp} \mathcal{W}' \triangleq \forall (x, y) \in \operatorname{dom} \mathcal{W}',\ (x, y) \in \operatorname{dom} \mathcal{W} \wedge \mathcal{W}[x, y] \sqsubseteq_{\mathbb{R}^\sharp} \mathcal{W}'[x, y]$$

This operation is sound, in that more precise elements concretize to smaller sets of valuations.

2.1.4 Constraint Propagation (Reduction). We can now describe one of the central operations in a weakly relational abstract domain: constraint propagation. Its goal is to find more precise constraints on $\mathcal{W}$ (i.e., to find $\mathcal{W}'$ such that $\mathcal{W}' \sqsubseteq_{\mathbb{W}^\sharp} \mathcal{W}$). Constraint propagation must not invent arbitrary constraints, but must deduce them from the existing ones; i.e. $\mathcal{W}'$ and $\mathcal{W}$ must represent the same set of values: $\gamma_{\mathbb{W}^\sharp}(\mathcal{W}') = \gamma_{\mathbb{W}^\sharp}(\mathcal{W})$. In abstract interpretation, such an operation, that improves abstract precision but does not change the concrete set of values, is called a *reduction*.

We write $\mathcal{W} \vDash x\ \mathcal{R}^\sharp\ y$ the predicate "$\mathcal{W}$ implies that $\mathcal{R}^\sharp$ holds between x and y". Here are its core rules. Their soundness derives from the definition.

$$\frac{}{\mathcal{W} \vDash x\ \mathcal{W}[x, y]\ y}\ \textsc{Base} \qquad \frac{}{\mathcal{W} \vDash x\ \mathtt{id}^\sharp\ x}\ \textsc{Refl} \qquad \frac{\mathcal{W} \vDash x\ \mathcal{R}^\sharp\ y}{\mathcal{W} \vDash y\ (\mathrm{inv}^\sharp(\mathcal{R}^\sharp))\ x}\ \textsc{Symm}$$

$$\frac{\mathcal{W} \vDash x\ \mathcal{R}_1^\sharp\ y \qquad \mathcal{W} \vDash y\ \mathcal{R}_2^\sharp\ z}{\mathcal{W} \vDash x\ (\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp)\ z}\ \textsc{Trans} \qquad \frac{\mathcal{W} \vDash x\ \mathcal{R}_1^\sharp\ y \qquad \mathcal{W} \vDash x\ \mathcal{R}_2^\sharp\ y}{\mathcal{W} \vDash x\ (\mathcal{R}_1^\sharp \sqcap_{\mathbb{R}^\sharp} \mathcal{R}_2^\sharp)\ y}\ \textsc{Meet}$$

They state that in addition to constraints already in $\mathcal{W}$, (x, x) is constrained by the concrete equality; that we can use symmetry and transitivity to improve constraints; and that when two constraints exist between the same variables, they can be combined using the meet $\sqcap_{\mathbb{R}^\sharp}$. Such rules are at the heart of abstract domains like DBM [Miné 2001] and octagon [Miné 2006].

A weakly-relational element $\mathcal{W}$ can be improved by changing its values to more precise deductions obtained by these rules. When it can no longer be improved, it is *saturated*. Starting from an arbitrary element $\mathcal{W}$ and applying constraint propagation rules until saturation leads to a unique[1] element, that we denote $\mathcal{W}^\star$ following Miné [2004]. In practice, $\mathcal{W}^\star$ can be built using the Floyd-Warshall transitive closure algorithm [Schwarz and Seidl 2023]. Figure 1 presents an example of saturation on a small graph.

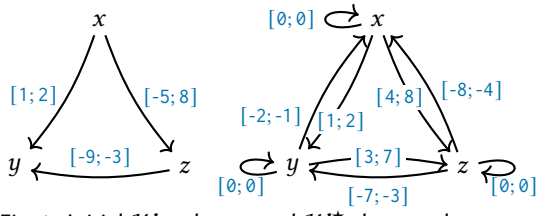


Fig. 1. Initial $\mathcal{W}$ and saturated $\mathcal{W}^\star$ abstract elements.

The problem with this operation is that after saturation, if $|\mathbb{X}|$ represents the number of variables, then the storage cost is $O(|\mathbb{X}|^2)$; furthermore, the computational cost to compute the transitive closure is $O(|\mathbb{X}|^3)$. These supralinear costs prevent the use of weakly relational domains with large number of variables, and requires mitigation. Note that not performing saturation is possible [Logozzo and Fähndrich 2008], but at the expense of precision; for best precision, it is in particular required to saturate abstract elements before they are joined [Miné 2006].

2.1.5 Constraint Elimination. One approach to reduce storage cost is to "reverse" constraint propagation: refrain from storing any constraints that can already be deduced from existing constraints. Instead, lazily compute $\mathcal{W}^\star[x, y]$ as needed when the constraint between x and y is queried. We refer to this method as *constraint elimination*. Eliminating constraints requires a careful implementation of abstract domain operations to avoid losing precision (as removing constraints means being less precise following the definition of $\sqsubseteq_{\mathbb{W}^\sharp}$), in particular when joining abstract elements.

Some constraints are easy to eliminate. For example, with reflexive constraints, we do not need to store $\mathcal{W}[x, x]$ unless it is more precise than $\mathtt{id}^\sharp$. Then when $\mathcal{W}[x, x]$ is queried, we return it if present, or $\mathtt{id}^\sharp$ otherwise. In the case of the difference or bitvector comparison abstract relations, $\mathtt{id}^\sharp$ is maximally precise, so we can eliminate every reflexive constraint. Similarly, for symmetric constraints, we can remove $\mathcal{W}[x, y]$ when $\mathrm{inv}^\sharp(\mathcal{W}[y, x]) \sqsubseteq_{\mathbb{R}^\sharp} \mathcal{W}[x, y]$. For example, with the difference and bitvector inequality relations, $\mathrm{inv}^\sharp$ is exact so $\mathcal{W}[x, y]$ and $\mathcal{W}[y, x]$ carry the same information. Thus, only one has to be stored (as done, for instance, in DBMs [Miné 2001]).

The difficulty comes from eliminating constraints that are recovered by transitivity. We can remove any constraint $\mathcal{W}[x, z]$ such that $\mathcal{W}[x, z] = \mathcal{W}[x, y] \fatsemi^\sharp \mathcal{W}[y, z]$ without affecting the concretization. However, recomputing $\mathcal{W}[x, y]$ dynamically when transitive constraints have been eliminated would be costly, as we basically have to recompute the full transitive closure.

2.2 An Assumption for Efficient Transitive Closure

To remedy this problem, we introduce the central assumption of this article. We call it the unique-label property, but note that this is close to the notion of commutative diagram in category theory.

Assumption 1. We consider only the $\mathcal{W} \in \mathbb{W}^\sharp$ such that, for all $x, y \in \mathbb{X}$, there exists an $\mathcal{R}^\sharp \in \mathbb{R}^\sharp$, such that for all paths $\langle x, x_0, \ldots, x_n, y\rangle$ in $\mathcal{W}$: $\mathcal{W}[x, x_0] \fatsemi^\sharp \mathcal{W}[x_1, x_2] \fatsemi^\sharp \ldots \fatsemi^\sharp \mathcal{W}[x_n, y] = \mathcal{R}^\sharp$.

Introducing Assumption 1 solves our problem: instead of performing a costly transitive closure to compute $\mathcal{W}^\star[x, y]$, it suffices to compose the relations on any path. We can massively eliminate constraints: instead of storing a complete graph, one can instead store a spanning tree, i.e. keep only constraints necessary for the graph to be connected (Figure 2).

[1] If there were two distinct saturated elements $\mathcal{W}_1$ and $\mathcal{W}_2$, we could build $\mathcal{W}' = \mathcal{W}_1 \sqcap \mathcal{W}_2$ such that $\mathcal{W}' \sqsubseteq_{\mathbb{W}^\sharp} \mathcal{W}_1$ and $\mathcal{W}' \sqsubseteq_{\mathbb{W}^\sharp} \mathcal{W}_2$, which contradicts either that $\mathcal{W}_1$ and $\mathcal{W}_2$ are distinct, or that they are both fully saturated.

Implementations of our weakly-relational abstract domain must maintain a spanning tree whose edges carry an abstract relation, while new edges are added and the relation between pairs of variables are queried. This is similar to the dynamic connectivity problem on graphs, except that we also maintain information on the edge. Hence, the data structure that we use, the labeled union-find (Section 3), derives from the classical union-find data structure, which efficiently solves the dynamic connectivity problem [Tarjan 1975]. That is why we call this new kind of abstract domains *labeled union-find abstract domains*.

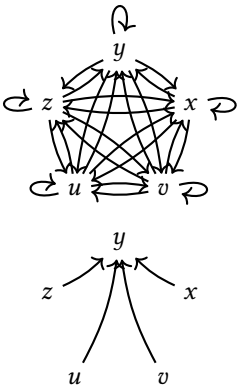


Fig. 2. From saturated to minimal constraints.

Note that not all $\mathcal{W}$ satisfy Assumption 1. For instance in Figure 1, there are two paths from x to y but $[-5;8] \fatsemi^\sharp [-9;3] \neq [1;2]$. In addition, there is no way to turn this example into a $\mathcal{W}$ that would meet Assumption 1 without losing precision. We can find similar examples with the bitvector comparison abstract relations. However, if we restrict $\mathbb{R}^\sharp$ to the constant difference abstract relation, or to the subset of bitvector comparisons which are constant (do not contain any ?), then any $\mathcal{W} \in \mathbb{W}^\sharp$ satisfies the assumption.

In Section 3 we show that if all $\mathcal{W} \in \mathbb{W}^\sharp$ satisfy the assumption, then $\langle \mathbb{R}^\sharp, \fatsemi^\sharp \rangle$ must obey the group axioms; in Section 4 we study the properties of sound abstract relations that obey the group axioms, and show that such abstract relations must represent concrete injective functions (Theorem 4.3)[2]. For example, the constant difference abstract relation corresponds to $[z \in \mathbb{Z} \mapsto z + c]$; and the constant bitvector comparison abstract relation corresponds to $[b \in \mathbb{BV}_n \mapsto b \texttt{ xor } c]$. Actually, any invertible function and their composition can be represented as abstract relations in our domains, e.g. bitvector rotations (or any permutation), integer multiplication, integer exponentiation, etc. Note that the abstraction may not be exact; if it is, then the function is bijective (Theorem 4.5).

Finally, labeled union-find abstract domains can be used in conjunction with other abstract domains for mutual improvements (i.e., as a reduced product). There are some technical difficulties, notably when computing the join operation (Appendix A), as we do not want to perform a full reduction. But another interesting use is *constraint factorization* (Section 5.2). Similarly to the union-find data structure, connected elements in a labeled union-find structure belong to a *relational class* (the set of variables where each pair is related by one of the relation in $\mathbb{R}^\sharp$) and also have a representative element (the root of the tree). The idea of constraint factorization is to eliminate redundant constraints in other domains, by only keeping constraints on the representative of each class. The constraints on the other elements in the class become implicit, which reduces both space and computation time needed for constraint propagation.

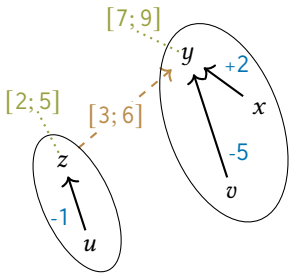


Fig. 3. Factorizing constraints.

Figure 3 describes an example of constraint factorization: we have partitioned the 5 variables into two relational classes ($\{z, u\}$ and $\{y, x, v\}$); and we have a transitive closure of the weakly-relational relation between representative elements (dashed line, using interval difference as abstract relation), and non-relational abstraction on the representative element of the relational class (dotted line). This representation carries the same information (i.e., has the same concretization) as the weakly-relational/non-relational product on variables, but requires significantly less storage. In Section 5

[2]Actually: an injective function between equivalence classes where the equivalence relation is $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$, which is the equality in the examples given in this section.

we study the interaction between labeled union-find and other domains, and detail in particular when constraint factorization is possible without losing precision.

3 Union-Find with a Group of Edges Labels

In this section, we discuss the labeled union-find data structure, which may find uses outside of program verification. For this reason, we generalize our settings to arbitrary nodes and labels, which may represent something else than variables constrained by abstract relations. We introduce, however, an additional hypothesis that will assume in the rest of the paper, which is that labels have a group structure; we justify this hypothesis in the next subsection.

3.1 Directed Graphs with Uniquely Labeled Paths

Let us consider (possibly infinite) directed graphs whose edges are labeled. Formally, we have a set of *nodes* $\mathbb{N}$, a set of *edge labels* $\mathbb{L}$, and an *edge predicate* $n_1 \xrightarrow{\ell} n_2 \in \mathbb{L} \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$. Now, we suppose that labels are equipped with an infix *composition operator* $\hat{\S} \in \mathbb{L} \times \mathbb{L} \to \mathbb{L}$. This allows defining the *label of a path*, which is the composition of labels on its traversed edges. Formally, we define a path predicate $\overset{\ell}{\twoheadrightarrow} \in \mathbb{L} \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$ as the transitive closure of the edge predicate:

$$n_1 \xrightarrow{\ell} n_2 \implies n_1 \overset{\ell}{\twoheadrightarrow} n_2 \ (\textsc{EdgePath}) \qquad n_1 \overset{\ell_1}{\twoheadrightarrow} n_2 \wedge n_2 \overset{\ell_2}{\twoheadrightarrow} n_3 \implies n_1 \overset{\ell_1 \hat{\S} \ell_2}{\twoheadrightarrow} n_3 \ (\textsc{PathTrans})$$

Following Assumption 1, we require that the label on every path between two nodes is unique:

$$\forall\, n_1\ n_2\ \ell\ \ell', \quad n_1 \overset{\ell}{\twoheadrightarrow} n_2 \wedge n_1 \overset{\ell'}{\twoheadrightarrow} n_2 \implies \ell = \ell' \qquad (\textsc{HUniqueLabel})$$

This implies that:

- **Label composition on every path is associative:** for a path $n_1 \overset{\ell_1}{\twoheadrightarrow} n_2 \overset{\ell_2}{\twoheadrightarrow} n_3 \overset{\ell_3}{\twoheadrightarrow} n_4$, we have both $n_1 \overset{\ell_1 \hat{\S} (\ell_2 \hat{\S} \ell_3)}{\twoheadrightarrow} n_4$ and $n_1 \overset{(\ell_1 \hat{\S} \ell_2) \hat{\S} \ell_3}{\twoheadrightarrow} n_4$, thus $\ell_1 \mathbin{\hat{\S}} (\ell_2 \mathbin{\hat{\S}} \ell_3) = (\ell_1 \mathbin{\hat{\S}} \ell_2) \mathbin{\hat{\S}} \ell_3$.
- **Labels of directed cycles are locally neutral:** $n_1 \overset{\ell_1}{\twoheadrightarrow} m \overset{\ell_2}{\twoheadrightarrow} m \overset{\ell_3}{\twoheadrightarrow} n_2$, implies that we have both $n_1 \overset{\ell_1}{\twoheadrightarrow} m$ and $n_1 \overset{\ell_1 \hat{\S} \ell_2}{\twoheadrightarrow} m$; therefore, $\ell_1 \mathbin{\hat{\S}} \ell_2 = \ell_1$. Similarly, we have $\ell_2 \mathbin{\hat{\S}} \ell_3 = \ell_3$.

Note that HUniqueLabel does not imply that composition is fully associative. It only shows associativity for label composition along paths that exist in the graph. Similarly, a graph with cycles does not imply the existence of a neutral element, only that the label of cycles is neutral for composition relative to the labels of any paths that start or end in the cycle.

However, if we want the label uniqueness property to be true on *any graph* labeled with $\mathbb{L}$, then associativity of composition becomes mandatory (we can extend the original graph with all possible paths). Furthermore, the label on every cycle must now truly be a neutral element, and **all labels must be invertible** (since every path can be turned into a cycle, labeled by the neutral element: if $n_1 \overset{\ell_1}{\twoheadrightarrow} n_2$ and $n_2 \overset{\ell_2}{\twoheadrightarrow} n_1$, then $\ell_1 \mathbin{\hat{\S}} \ell_2 = \texttt{id} = \ell_2 \mathbin{\hat{\S}} \ell_1$). Associativity, existence of a neutral element and of an inverse are the core axioms of group theory, and thus from now on we also assume:

Assumption 2. $\langle \mathbb{L}, \hat{\S} \rangle$ is a *group*, i.e. $\hat{\S}$ is associative; there is a unique neutral (identity) element $\texttt{id} \in \mathbb{L}$; and each element ℓ is invertible (and we write $\texttt{inv}(\ell) \in \mathbb{L} \to \mathbb{L}$ the inverse of ℓ).

We can now extend our path predicate using these elements to make it reflexive and symmetric:

$$\forall\, n,\ n \overset{\texttt{id}}{\twoheadrightarrow} n \ (\textsc{PathRefl}) \qquad \forall\, n_1\ n_2\ \ell,\ n_1 \overset{\ell}{\twoheadrightarrow} n_2 \implies n_2 \overset{\texttt{inv}(\ell)}{\twoheadrightarrow} n_1 \ (\textsc{PathSym})$$

Remark. Assumption 2 does not necessarily imply HUniqueLabel (i.e., that Assumption 1 is met on every $\mathcal{W}$). If (and only if) x and y are constant integers, there exists several suitable a, b such that $y = a \times x + b$ or $y = (x \texttt{ xor } a) \texttt{ rot } b$. This implies that we have to detect *conflicts* when we find two such relations to maintain HUniqueLabel, that we solve by propagating the information

that y and x are constant in another domain and dropping at least one of the relations. Relatedly, we may create new relations when we join pairs of constant values (Sections 5.1 and 7.2).

3.2 Union-Find over Groups of Edge Labels

3.2.1 A Minimal Data-Structure For Storing Relation Between Nodes. We are now interested in a data structure that would allow retrieving, for any two nodes n_1 and n_2, the label ℓ such that $n_1 \xrightarrow{\ell} n_2$ when it exists, exploiting the properties of Section 3.1 to make this efficient. If this data structure is a directed graph, one can retrieve this label as long as one undirected path exists from n_1 to n_2, by composing the labels on the edges on this path (and inverting them for reversed edges).

The most space-efficient topology for this data structure is that of a spanning forest. Indeed, we only need to remember enough edges such that any connected nodes in the original graph are still connected. Spanning forests correspond to such minimally-connected graphs. In a given tree, the most efficient topology to compute the relations between nodes is a star topology, where one distinguished element is the root and all other elements directly point to it. In this case, the relations between any two nodes can be computed using at most one composition.

3.2.2 Lazy Path Compression. When interleaving the addition of new edges with the query of the label between node pairs, it may not be optimal to immediately transform the graph to this star topology. This problem is very related to the dynamic connectivity problem, that maintains a spanning tree of a graph [Tarjan 1975]. If the labels were the unit type, then finding the label between two nodes amounts to finding if two nodes are connected; which is done efficiently by union-find. Our data structure is very similar. We just label the edges on the parent link of the union-find, in order to remember the label as well as the connectivity information. We therefore call it a *labeled union-find* data structure. It is a strict extension of union-find: classical union-find is just the case where the labels are the unit type.

3.2.3 Labeled Union-Find. Figure 4 presents our implementation of labeled union-find, which is a simple extension over classical union-find [Galler and Fischer 1964; Tarjan 1975]. The type of the union-find $\mathbb{U}$ uses a mutable map structure to represent a partial function, which can be e.g. an array, a hash table, or a pointer field inside a record corresponding to nodes.

The classic `find` function returns the representative element and performs path compression. Our version also returns the label pointing to the parent, and composes labels during compression. Our `union` takes as an extra argument the new relation between the two nodes being joined. It is therefore no longer a pure union between equivalence classes, so we renamed it to `add_relation`.[3] When joining two nodes which already have the same representative via a different relation, it calls a user-supplied conflict function. Finally, we added the `get_relation` function. Given two nodes, it returns the label of an edge connecting them (if they are in the same relational class), or $\top$ if not.

There are many variations of the union-find data structure [Patwary et al. 2010; Tarjan and van Leeuwen 1984] that can be immediately translated to our labeled union-find variant. The one we describe in Figure 4 uses randomized linking [Goel et al. 2014], which provides equivalent performance than smarter linking strategies in practice.

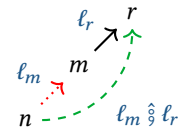
Remark. Labeled union-find operations can be made polymorphic to handle relations between nodes with type parameters (like `'a list` in OCaml or `std::vector<T>` in C++). This does not require changing the code, only updating the type signatures to those of Figure 4c, which require existential types. For simplicity, we stick to monomorphic types in the rest of the paper.

[3] Similarly, we use "relational class" instead of "equivalence class" for to the set of elements that have the same representative.

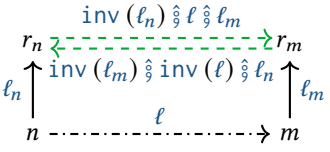
$$\begin{array}{l} \mathsf{nodes} : \mathbb{N} \\ \mathsf{map} : \mathbb{U} \triangleq \mathbb{N} \rightharpoonup \mathbb{N} \times \mathbb{L} \\ \\ \mathtt{init} \in \mathtt{unit} \to \mathbb{U} \\ \mathtt{init}\ () \triangleq \emptyset \end{array} \qquad \begin{array}{r} \mathsf{labels} : \mathbb{L} \\ \mathtt{id} \in \mathbb{L} \\ \fatsemi \in \mathbb{L} \times \mathbb{L} \to \mathbb{L} \\ \mathtt{inv} \in \mathbb{L} \to \mathbb{L} \end{array}$$

(a) Labeled union-find parameters.

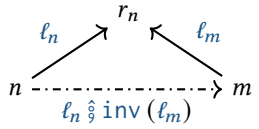
$\mathtt{find} \in \mathbb{U} \times \mathbb{N} \to \mathbb{N} \times \mathbb{L}$
$\mathtt{find}\,(U,\ n) \triangleq \textbf{if}\ n \notin \mathrm{dom}\ U\ \textbf{then}\ (n, \mathtt{id})\ \textbf{else}$
  $\textbf{let}\ (m, \ell_m) = U[n]\ \textbf{in let}\ (r, \ell_r) = \mathtt{find}\,(U,\ m)\ \textbf{in}$
  $U[n] \coloneqq (r,\ \ell_m \fatsemi \ell_r);$ // optional path compression
  $(r, \ell_m \fatsemi \ell_r)$



$\mathtt{add\_relation} \in \mathbb{U} \times \mathbb{N} \times \mathbb{N} \times \mathbb{L} \to \mathtt{unit}$
$\mathtt{add\_relation}\,(U,\ n,\ m,\ \ell) \triangleq$
  $\textbf{let}\ (r_n, \ell_n) = \mathtt{find}\,(U,\ n)\ \textbf{in let}\ (r_m, \ell_m) = \mathtt{find}\,(U,\ m)\ \textbf{in}$
  $\textbf{if}\ r_n = r_m\ \textbf{then}$
    $\textbf{if}\ \ell \neq \ell_n \fatsemi \mathtt{inv}\,(\ell_m)\ \textbf{then}\ \mathrm{conflict}(U, m, n, \ell, \ell_n \fatsemi \mathtt{inv}\,(\ell_m))$
  $\textbf{else if}\ \mathrm{rand}()\ \textbf{then}\ U[r_n] \coloneqq (r_m,\ \mathtt{inv}\,(\ell_n) \fatsemi \ell \fatsemi \ell_m)$
    $\textbf{else}\ U[r_m] \coloneqq (r_n,\ \mathtt{inv}\,(\ell_m) \fatsemi \mathtt{inv}\,(\ell) \fatsemi \ell_n)$



$\mathtt{get\_relation} \in \mathbb{U} \times \mathbb{N} \times \mathbb{N} \to \mathbb{L}_\top$
$\mathtt{get\_relation}\,(U,\ n,\ m) \triangleq$
  $\textbf{let}\ (r_n, \ell_n) = \mathtt{find}\,(U,\ n)\ \textbf{in let}\ (r_m, \ell_m) = \mathtt{find}\,(U,\ m)\ \textbf{in}$
  $\textbf{if}\ r_n = r_m\ \textbf{then}\ \ell_n \fatsemi \mathtt{inv}\,(\ell_m)\ \textbf{else}\ \top$



(b) Labeled union-find definition. Some functions have been illustrated by small graphs. Full arrows are those present in the map at the start; green dashed arrows are added during the function call; red dotted arrows are removed by the function call; and dash-dotted arrows are never present in the map.

$$\begin{array}{r} \mathbb{N}\langle\alpha\rangle,\ \mathbb{L}\langle\alpha,\beta\rangle \\ \mathtt{id} \in \mathbb{L}\langle\alpha,\alpha\rangle \\ \fatsemi \in \mathbb{L}\langle\alpha,\beta\rangle \times \mathbb{L}\langle\beta,\gamma\rangle \to \mathbb{L}\langle\alpha,\gamma\rangle \\ \mathtt{inv} \in \mathbb{L}\langle\alpha,\beta\rangle \to \mathbb{L}\langle\beta,\alpha\rangle \end{array} \qquad \begin{array}{r} \mathbb{U} \triangleq \mathbb{N}\langle\alpha\rangle \rightharpoonup \exists\,\beta.\mathbb{N}\langle\beta\rangle \times \mathbb{L}\langle\alpha,\beta\rangle \\ \mathtt{find} \in \mathbb{U} \times \mathbb{N}\langle\alpha\rangle \to \exists\,\beta.\mathbb{N}\langle\beta\rangle \times \mathbb{L}\langle\alpha,\beta\rangle \\ \mathtt{add\_relation} \in \mathbb{U} \times \mathbb{N}\langle\alpha\rangle \times \mathbb{N}\langle\beta\rangle \times \mathbb{L}\langle\alpha,\beta\rangle \to \mathtt{unit} \\ \mathtt{get\_relation} \in \mathbb{U} \times \mathbb{N}\langle\alpha\rangle \times \mathbb{N}\langle\beta\rangle \to \mathbb{L}\langle\alpha,\beta\rangle_\top \end{array}$$

(c) Signature of labeled union-find with type parameters.

Fig. 4. Labeled union-find parameters, definitions, and alternative polymorphic signatures.

Theorem 3.1. *Let $U \in \mathbb{U}$ be the union-find obtained by successive calls to* `init`, `add_relation`, `find`, *and* `get_relation`. *If* conflict *does not modify U, then, two nodes n and m are related (i.e., there exists ℓ such that $n \overset{\ell}{\longrightarrow\!\!\!\!\!\rightarrow} m$ in U) if and only if they belong to the same tree (whose root is the representative), and if and only if they would have been related in the graph whose edges are the arguments of the successive calls to* `add_relation` *which did not trigger* conflict. *Furthermore:*

(1) `find` *terminates, returns the representative r, and $(r, \ell) \triangleq \mathtt{find}\,(U,\ n) \Rightarrow n \overset{\ell}{\longrightarrow\!\!\!\!\!\rightarrow} r$;*
(2) `get_relation` $(U,\ n,\ m) = \ell \neq \top$ *if and only if $n \overset{\ell}{\longrightarrow\!\!\!\!\!\rightarrow} m$;*
(3) `get_relation` $(U,\ n,\ m) = \top$ *if and only if n and m are not related;*
(4) conflict *is not called if and only if* HUniqueLabel *holds.*

Proof. See proof in Appendix B.1 (page 29). □

Managing Conflicts. Conflicts occur when `add_relation` tries to add a new, different relation, between two already related nodes. As we will see in the next section, the suitable abstract relations for labeled union-find often form a flat lattice (Theorem 4.5), meaning no relation is more or less precise than the other. Thus, we cannot replace the old conflicting relation by a more precise one.

$\texttt{init}_{\mathbb{I}} \in \texttt{unit} \to \mathbb{U}\text{-}\mathbb{I}$ $\quad$ $\texttt{get\_info} \in \mathbb{U}\text{-}\mathbb{I} \times \mathbb{N} \to \mathbb{I}$

$\texttt{init}_{\mathbb{I}}\ () \triangleq (\emptyset, [n \in \mathbb{N} \mapsto \top_{\mathbb{I}}])$ $\quad$ $\texttt{get\_info}\ ((U, I),\ n) \triangleq \textbf{let}\ (r, \ell) = \texttt{find}\ (U,\ n)\ \textbf{in}\ \mathcal{A}\ (\ell,\ I[r])$

$\texttt{add\_info} \in \mathbb{U}\text{-}\mathbb{I} \times \mathbb{N} \times \mathbb{I} \to \texttt{unit}$

$\texttt{add\_info}\ ((U, I),\ n,\ i) \triangleq \textbf{let}\ (r, \ell) = \texttt{find}\ (U,\ n)\ \ \textbf{in}\ I[r] \coloneqq I[r] \sqcap_{\mathbb{I}} \mathcal{A}\ (\texttt{inv}\ (\ell)\,,\ i)$

$\texttt{add\_relation}_{\mathbb{I}} \in \mathbb{U}\text{-}\mathbb{I} \times \mathbb{N} \times \mathbb{N} \times \mathbb{L} \to \texttt{unit}$

$\texttt{add\_relation}_{\mathbb{I}}\ ((U, I),\ n,\ m,\ \ell) \triangleq$ same as `add_relation`, adding after each $U[r_b] \coloneqq (r_a, \ell')$:

$\quad I[r_a] \coloneqq I[r_a] \sqcap_{\mathbb{I}} \mathcal{A}\ (\texttt{inv}\ (\ell')\,,\ I[r_b])\,;$

$\quad \textbf{del}\ I[r_b];$

Fig. 5. Operations added to labeled union-find to store information for all nodes at representatives.

However, in many instances conflicts imply extra information on the related values, but that information cannot be represented by the labeled union-find. For example, when using the TVPE relation (Example 4.6) which relates variables via linear equalities $y = ax + b$, a conflict means we have two separate lines in 2D space that contain the point (x, y) whose coordinates are the values of both variables. Either these are parallel, and no suitable value exists (the state in unsatisfiable), or these intersect, and computing their intersection will yield exact values for both variables. In both cases, this information should be kept elsewhere, for instance in a non-relational domain.

3.3 Adding Information to Relational Classes

We now extend the classic use of the union-find data-structure which gathers information for the whole equivalence class at the root node. Since the paths are not simple equalities, we will often need to transform the information along the path. This is done via a *group action* from the labels to some set of information $\mathbb{I}$, meaning that each label can act as a function on those values. In this case, the labeled union-find data structure can efficiently compute the information of any node given the information of the representative elements of the data structure.

We can compress the representation of a map $\mathbb{N} \to \mathbb{I}$ from nodes to information, to a map from representative elements of the relational class to information. We will explore several applications of this technique in Section 5 and Section 6, such as constraint factorization (Section 2.2) where $\mathbb{I}$ is a non-relational abstraction attached to a program variable (e.g., its interval or known bits).

3.3.1 Group Actions. A *group action* of the labels $\mathbb{L}$ is a function $\mathcal{A} \in \mathbb{L} \times \mathbb{I} \to \mathbb{I}$ that satisfies:

$$\forall\ \ell\ \ell' \in \mathbb{L},\ \forall\ i \in \mathbb{I}, \qquad \mathcal{A}\ (\ell \mathbin{\hat{\S}} \ell',\ i) = \mathcal{A}\ (\ell,\ \mathcal{A}\ (\ell',\ i)) \qquad \text{(HActionCompose)}$$

$$\forall\ i \in \mathbb{I}, \qquad \mathcal{A}\ (\texttt{id},\ i) = i \qquad \text{(HActionIdentity)}$$

This implies that $\ell \mapsto \mathcal{A}\ (\ell,\ \cdot)$ is a homomorphism from the labels to the transformations over $\mathbb{I}$.

3.3.2 Action of a Labeled Union-Find. Once provided with an action, we can lift a map that stores information for representatives to a map storing information for all nodes (see function `get_info` in Figure 5): just use the `find` operation on the labeled union find to retrieve the representative and label, then apply the action using the label and the information on the representative. For instance in Figure 3, to recover the interval on x, the action for $+2$ would be the interval addition $\lambda i.\, i + [2; 2]$, that we apply on the interval $[7; 9]$ of the representative element y to find that $x \in [9; 11]$.

Formally, we change the type of our union-find $\mathbb{U}$ to add a second map, from representatives to information: $\mathbb{U}\text{-}\mathbb{I} \triangleq \mathbb{U} \times (\mathbb{N} \rightharpoonup \mathbb{I})$. Figure 5 presents the new operations of the data structure to manipulate information. In order to combine information from various source, we assume that the set $\mathbb{I}$ is a meet-semilattice with a top element $\top_{\mathbb{I}}$ representing the absence of information, and a commutative associative meet operator $\sqcap_{\mathbb{I}} \in \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ which combines information from multiple sources. First, the $\texttt{init}_{\mathbb{I}}$ function creates a new empty union-find, initializing all information to

top.[4] Then, `get_info` can be used to obtain the information attached to a given node. Information can be added via `add_info`, which stores the information on the root by combining it (with $\sqcap_{\mathbb{I}}$) with existing information. Finally, `add_relation` also needs to be modified to combine the information of both representatives being merged. One condition for this to work is that the action distributes over the meet, so that combining information can be done in any order (this is in particular the case when $\sqcap_{\mathbb{I}}$ and $\mathcal{A}$ are exact operations in an abstract domain). The following theorem states that the data structure computes the correct information:

Theorem 3.2. *Let $(U, I) \in \mathbb{U}\text{-}\mathbb{I}$ be the result of a sequence of calls to* `add_relation`$_{\mathbb{I}}$, `get_info`, `add_info` *after a first* `init`$_{\mathbb{I}}$ $()$, *with $(m_0, i_0) \dots (m_k, i_k)$ the arguments passed to* `add_info`. *Then:*

(1) the domain of I is exactly the set of representatives of U (all accesses to I are correct)
(2) if $\mathcal{A}$ distributes over $\sqcap_{\mathbb{I}}$ (i.e., $\mathcal{A}(\ell, i \sqcap_{\mathbb{I}} j) = \mathcal{A}(\ell, i) \sqcap_{\mathbb{I}} \mathcal{A}(\ell, j)$), we have for all nodes n:

$$\texttt{get\_info}((U, I), n) = \bigsqcap_{\mathbb{I}, 0 \le p \le k,\ n \text{ and } m_p \text{ in the same relational class}} \mathcal{A}\left(\texttt{get\_relation}(U, n, m_p), i_p\right)$$

Proof. See full proof in Appendix B.1 (page 31). □

4 Suitable Abstract Relations

Our goal is to implement the weakly relational domains of Section 2 ($\mathbb{W}^\sharp \triangleq \mathbb{X} \times \mathbb{X} \rightharpoonup \mathbb{R}^\sharp$) using the labeled union-find structure seen in Section 3 ($\mathbb{U} \triangleq \mathbb{N} \rightharpoonup \mathbb{N} \times \mathbb{L}$). From now on, we will have $\mathbb{N} = \mathbb{X}$ (the union-find represents relations between program variables) and $\mathbb{L} = \mathbb{R}^\sharp$, $\texttt{id} = \texttt{id}^\sharp$, $\texttt{inv} = \texttt{inv}^\sharp$, $\hat{\circ} = \circ^\sharp$ (we use abstract relations as labels). Because we can turn elements of $\mathbb{X} \rightharpoonup \mathbb{X} \times \mathbb{R}^\sharp$ into elements in $\mathbb{X} \times \mathbb{X} \rightharpoonup \mathbb{R}^\sharp$, we can consider labeled union-find domains as weakly relational domains (and reuse the concretization, definitions, and properties).

4.1 Properties of Abstract Relations as a Group

One question remains: what abstract relations can we use? Specifically, the operations on abstract relations ($\texttt{id}^\sharp$, $\texttt{inv}^\sharp$, and $\circ^\sharp$) must simultaneously be sound (i.e., obey the rules of Section 2.1.2) and obey the group axioms (Assumption 2). This implies surprising properties about abstract relations, which allow both ruling out unsuitable candidates (e.g., interval difference cannot be used) but also easily finding some, such as any composition of injective functions.

Lemma 4.1. $\texttt{inv}^\sharp$ *is exact:* $\forall\, \mathcal{R}^\sharp \in \mathbb{R}^\sharp,\ \gamma_{\mathbb{R}^\sharp}(\texttt{inv}^\sharp(\mathcal{R}^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)^{-1}$.

Proof. Uses involution of $\texttt{inv}^\sharp$ and HInverseSound. For details, see Appendix B.2 (page 33). □

Lemma 4.2. $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \in \mathcal{P}(\mathbb{V} \times \mathbb{V})$ *is an equivalence relation between values.*

Proof. See Appendix B.2 (page 33). □

Theorem 4.3. *An abstract relation $\mathcal{R}^\sharp \in \mathbb{R}^\sharp$ represents a concrete injective partial function $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ between the equivalence classes of the $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ relation.*

Proof. See Appendix B.2 (page 33). □

Example 4.4 (Parity comparison). In many of our abstract relations, $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ is equality on values. A counter-example is parity comparison, where $\mathbb{R}^\sharp = \{\texttt{sameparity}, \texttt{differentparity}\}$, with the obvious meaning. Then, $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp = \texttt{sameparity})$ is an equivalence relation whose classes are the set of odd and even values, and `differentparity` corresponds to $[\texttt{even} \mapsto \texttt{odd}; \texttt{odd} \mapsto \texttt{even}]$.

[4] To save space, we can avoid storing the tops in the map by having the map get operation return top on absent values.

A common situation is when abstract relations represent not only partial injective functions, but also total bijective functions (as is the case for the constant difference or the constant bitvector comparison abstract relations). The following theorem shows that this is the case whenever the abstract composition $\fatsemi^\sharp$ is exact; furthermore, the concrete functions are then also a group.

Theorem 4.5. *The following propositions are equivalent:*

(1) $\gamma_{\mathbb{R}^\sharp}$ *is a group morphism between* $\left\langle \mathbb{R}^\sharp, \fatsemi^\sharp, \texttt{inv}^\sharp, \texttt{id}^\sharp \right\rangle$ *and the group of relations between equivalence classes* $\left\langle \mathcal{P}\left(\mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \times \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)\right), \fatsemi, \bullet^{-1}, \{(\overline{v},\overline{v}) \mid \overline{v} \in (\mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp))\}\right\rangle$*;*

(2) $\fatsemi^\sharp$ *is exact:* $\forall\, \mathcal{R}_1^\sharp\ \mathcal{R}_2^\sharp \in \mathbb{R}^\sharp,\ \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$*;*

(3) for all $\mathcal{R}^\sharp$, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ *is a total function;*

(4) for all $\mathcal{R}^\sharp$, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ *is surjective;*

(5) for all $\mathcal{R}^\sharp$, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ *is bijective.*

Furthermore, if these hold then the lattice of abstract relations is flat: for all $\mathcal{R}_1^\sharp$ *and* $\mathcal{R}_2^\sharp$ *such that* $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$*, we have* $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$.

Proof. See Appendix B.2 (page 34). □

An important case where $\fatsemi^\sharp$ is non-exact is the TVPE abstract domain on integers:

Example 4.6 (Two-values per equality). Take the abstract relations $\mathbb{R}^\sharp \triangleq \mathbb{Q}_{\neq 0} \times \mathbb{Q}$, where $\mathbb{Q}$ is the set of rationals, concretized as $\gamma_{\mathbb{R}^\sharp}(a,b) \triangleq \{(x,y) \in \mathbb{V}^2 \mid y = ax + b\}$. We name it TVPE by similarity with the two-variables per inequality (TVPI) domain [Simon and King 2010]. There are (at least) two variants of TVPE, one where $\mathbb{V} = \mathbb{Z}$ and one where $\mathbb{V} = \mathbb{Q}$. When $\mathbb{V} = \mathbb{Z}$ the abstract composition is not exact. Consider the chain $z \xrightarrow{(2,0)} y \xrightarrow{(0.5,0)} x$ (i.e., $z = 2y \wedge y = 0.5x$). In the abstract, we have $(2,0) \fatsemi^\sharp (0.5,0) = (1,0) = \texttt{id}^\sharp$, which implies $z = x$. However, in the concrete, $\gamma_{\mathbb{R}^\sharp}(2,0) \fatsemi \gamma_{\mathbb{R}^\sharp}(0.5,0) = \{(z,z) \mid z \in 2\mathbb{Z}\}$. In other words, the abstraction forgot that z and x are even (this information could be stored in a non-relational domain, see Section 5).

4.2 Examples of Abstract Relations

The characterizations of Theorems 4.3 and 4.5 allow easily defining new suitable abstract relations: it suffices to consider invertible functions or their compositions. Thus, in addition to the constant difference (Example 2.1), constant bitvector comparison (Example 2.3), parity comparison (Example 4.4), and TVPE (Example 4.6) abstract relations, we can present some additional relations:

*Example 4.7 (*xor *and rotation).* Bitvectors ($\mathbb{V} \triangleq \mathbb{BV}_n$) with the relations $\mathbb{R}^\sharp \triangleq [0, n-1] \times \mathbb{BV}_n$ can be concretized as $\gamma_{\mathbb{R}^\sharp}(s,c) \triangleq \{(x,y) \in \mathbb{BV}_n \mid y = (x \texttt{ xor } c) \texttt{ rot } s\}$. While such direct rotations are uncommon, shifts are very common; and shifting a bitvector whose erased bits are known can be transformed into a (xor with constant, rotation) sequence. This domain is the composition of rotations with the constant bitvector comparison (Example 2.3) domain; it can be extended by considering arbitrary bit permutations instead of just rotations.

Example 4.8 (Modular arithmetic). In $\mathbb{Z}/2^n\mathbb{Z}$, addition with a constant, or multiplication with an odd value, are invertible operations. Thus, they can be seen as a TVPE abstract domain for modular arithmetic. Multiplication with a power of two is not an invertible operation, but it can be encoded as a xor + rotation if the erased bits are known (e.g. if there are no overflows).

Example 4.9 (Invertible matrix multiplications). If $\mathbb{V}$ is a vector of values in a field, then we can relate different vector values using matrix multiplication, provided the matrices are invertible.

Example 4.10 (Casts). Casts from integer to reals, or from a bitvector to its signed/unsigned integer value are invertible, and can thus be represented in an abstract domain. Combined with constraint factorization (Section 5), it allows sharing constraints across variables of different types.

5 Reduced Product with Non-Relational Domains

This section explains how the labeled union-find domain can be combined with non-relational abstract domains, which store numeric information for each variable independently (concretizes to $\mathbb{X} \to \mathcal{P}(\mathbb{V})$). The relations in labeled union-find can be used to propagate this numeric information across related variables.

5.1 Constraint Propagation

The classical way to have domains collaborate in abstract interpretation is through a reduced product [Cousot and Cousot 1979]. Here we'll combine a union-find $\mathbb{U}$ labeled with abstract relations $\mathbb{R}^\sharp$ together with a non-relational abstraction $\mathbb{X} \to \mathbb{V}^\sharp$, mapping program variables to *abstract values* $\mathbb{V}^\sharp$. These abstract values concretize to sets of concrete values: $\gamma_{\mathbb{V}^\sharp} \in \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$. We also suppose $\mathbb{V}^\sharp$ equipped with standard lattice operators $\sqcup_{\mathbb{V}^\sharp}$, $\sqcap_{\mathbb{V}^\sharp}$, and $\top_{\mathbb{V}^\sharp}$. Examples of such single-value abstractions include intervals [Cousot and Cousot 1977], bitwise/known bits abstraction [Michel and Hentenryck 2012; Miné 2012; Regehr et al. 2005], or congruence [Granger 1989].

We can view the labeled union-find as a set of constraints to propagate to the single value abstractions. For example, if we know that $x \in [0;3]$, $y \in [2;8]$ and that $y = x + 1$, we can refine these values to $x \in [1;3]$ and $y \in [2;4]$. Formally, this is done via the `refine` operator (which can be defined for arbitrary n-ary constraints, not just those that meet the conditions of Section 4):

$$\begin{array}{l}\texttt{refine} \in \mathbb{R}^\sharp \times \mathbb{V}^\sharp \times \mathbb{V}^\sharp \to \mathbb{V}^\sharp \times \mathbb{V}^\sharp \\ \gamma_{\mathbb{V}^\sharp}(\mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, v_1^\sharp, v_2^\sharp))) \supseteq \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \;\middle|\; \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(v_2^\sharp), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \right\} \\ \gamma_{\mathbb{V}^\sharp}(\mathrm{snd}(\texttt{refine}(\mathcal{R}^\sharp, v_1^\sharp, v_2^\sharp))) \supseteq \left\{ v_2 \in \gamma_{\mathbb{V}^\sharp}(v_2^\sharp) \;\middle|\; \exists v_1 \in \gamma_{\mathbb{V}^\sharp}(v_1^\sharp), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \right\}\end{array} \quad \text{(HRefineSound)}$$

Where $\mathrm{fst} \in X \times Y \to X$ and $\mathrm{snd} \in X \times Y \to Y$ are the canonical pair projections. Given two value abstractions $v_1^\sharp$ and $v_2^\sharp$ and a relation $\mathcal{R}^\sharp$, `refine` returns two new improved abstractions $v_1^{\sharp\prime}$ and $v_2^{\sharp\prime}$ which restrict $v_1^\sharp$ and $v_2^\sharp$ to values that satisfy $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$. We can use this to refine the values of all our variables using the relations in the labeled union-find. Constraint propagation is done by picking a constraint; using `refine` to improve the values of its variables; if any value has changed, adding all constraints linking to its variable back into the worklist; and repeating until a fixed point is reached. This is the basis of algorithms like AC-3 [Mackworth 1977] or HC-4 [Benhamou et al. 1999; Miné 2017]. Computing such a fixpoint may take a long time, which is why narrowing is useful to speed it up in practice; a simple narrowing is to stop propagation after a fixed number of steps.

Example 5.1. In constraint programming, `refine` is often replaced by directly computing the set of possible values for the variable that we want to update, and intersecting it with the previous values. However, with abstract values, this leads to a loss of precision. For instance, consider $x_1 + x_2 = 4$ with the initial bitwise abstraction `0b00?0` for both x_1 and x_2. Then, the most precise implementation for `refine` deduces that $x_1 = x_2 = 2 =$ `0b0010`, but computing possible values for x_2 from the values of x_1 will yield `0b0??0`, which returns `0b00?0` after intersection with `0b00?0`. As explained by Miné [2017], applying several maximally precise abstract operations is less precise than applying the best abstraction of the combination of these operations.

Constraint propagation should be performed any time a new constraint is added to the product: either a new binary relation in $\mathbb{U}$, or a more precise value abstraction in $\mathbb{X} \to \mathbb{V}^\sharp$. In practice, this usually happens after conditionals in the program, when the condition is assumed to be true or false depending on the branch taken.

In theory, for maximal precision, we would start by saturating the graph of relations before performing constraint propagation. However, this undermines the main strength of labeled union-find: minimizing the number of stored relations. Luckily, the following theorem shows that when

`refine` is exact (replace $\subseteq$ by $=$ in HRefineSound), constraint propagation on the minimal spanning tree is equivalent to constraint propagation on the saturated graph.

Theorem 5.2. *If* `refine` *is exact then, for all states* $(U, M) \in \mathbb{U} \times (\mathbb{X} \to \mathbb{V}^\sharp)$ *on which we performed constraint propagation, and for all edges* $x \xrightarrow{\mathcal{R}^\sharp} y$ *in the saturated graph of* U*, if we let* $(v_1^\sharp, v_2^\sharp) \triangleq \texttt{refine}(\mathcal{R}^\sharp, M[x], M[y])$*, we have* $\gamma_{\mathbb{V}^\sharp}(M[x]) \subseteq \gamma_{\mathbb{V}^\sharp}(v_1^\sharp)$ *and* $\gamma_{\mathbb{V}^\sharp}(M[y]) \subseteq \gamma_{\mathbb{V}^\sharp}(v_2^\sharp)$.

Proof. By structural induction on $x \xrightarrow{\mathcal{R}^\sharp} y$, full proof in Appendix B.3 (page 35). □

The reverse reduction, improving labeled union-find from the non-relational domain, is less common. One exception is when two variables are constant, in which case we can generally deduce the abstract relation between these variables (e.g. using the constant offset or bitvector comparison relation). Note that sometimes there is a large (xor-rotate) or infinite (TVPE) number of such relations. These reductions are typically not useful, except when joining two pairs (u_1, m_1) and (u_2, m_2) of abstract elements, where some variables no longer are constant. For instance, using the TVPE domain we can compute $\{x = 2, y = 5\} \sqcup \{x = 3, y = 7\} = \{x = [2;3]; y = [5;7]; y = 2x + 1\}$.

5.2 Map Factorization

As hinted by Figure 3, the previous reduction can be obtained more efficiently by factorizing the non-relational map into the labeled union-find, i.e., by attaching abstract values only to the representative element of relational classes (and not to every variable). This uses the results of Section 3.3, where the info that we associate with relational classes will be the abstract value, so here $\mathbb{I} = \mathbb{V}^\sharp$ and $\sqcap_{\mathbb{I}} = \sqcap_{\mathbb{V}^\sharp}$.

Factorization requires a group action $\mathcal{A} \in \mathbb{R}^\sharp \times \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ to transform the abstract value along an edge. In practice, $\mathcal{A}(\mathcal{R}^\sharp, \cdot)$ is an abstraction of the concrete injective function represented by $\mathcal{R}^\sharp$ that operates on abstract values. For instance, when using an interval value abstraction with the constant difference abstract relation, then the action is given by $\mathcal{A}\ (k,\ [a;b]) \triangleq [k;k] + [a;b]$. It uses interval addition, as a lifting for integer addition on intervals. Formally, this is expressed via a soundness hypothesis on $\mathcal{A}$:

$$\forall\, v^\sharp\ \mathcal{R}^\sharp, \quad \gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathcal{R}^\sharp,\ v^\sharp)) \quad \supseteq \quad \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} \qquad \text{(HActionSound)}$$

When the above $\supseteq$ is replaced by equality, we say the action is exact. Actions can be understood as a simplified version of `refine`:

Lemma 5.3. *If* `refine` *satisfies HRefineSound, then* $\mathcal{A}(\mathcal{R}^\sharp, v^\sharp) \triangleq \mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, \top_{\mathbb{V}^\sharp}, v^\sharp))$ *satisfies HActionSound. If* `refine` *is exact (HRefineSound with* $\supseteq$ *replaced by* $=$*), then so is* $\mathcal{A}$.

Proof. See proof in Appendix B.3 (page 36). □

In what follows, we make two assumptions to simplify the presentation of our results, with the complete statements deferred to Appendix B.3. The first is that $\gamma_{\mathbb{V}^\sharp}$ is injective: there are no two abstract values with the same concretization. This is not a strong constraint, as we can satisfy it by taking the quotient of the abstract domain by the equivalence relation "same concretization" (this is the usual construct to transform a Galois connection to a Galois insertion). Moreover, most value abstractions, like the bitwise and interval abstractions, naturally have an injective $\gamma_{\mathbb{V}^\sharp}$ (provided there is a single bottom element). The second assumption is that $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ is equality, which has been true in almost all the presented examples.

Just like in Section 4, combining the action hypotheses with soundness narrows down the possible candidates. Namely, we can show that being an action is equivalent to being exact:

Lemma 5.4. *Assume any of the points of Theorem 4.5 hold. If $\mathcal{A}$ satisfies HActionSound then $\mathcal{A}$ is exact (we can replace $\supseteq$ by $=$ in HActionSound) if and only if it is a group action.*

Proof. See proof in Appendix B.3 (page 37). □

Example 5.5. The existence of an exact action implies that the precision loss seen in Example 5.1 cannot occur. Indeed, defining it requires a precise addition on a bitwise abstraction. But, since addition propagates information across bits, a single unknown bit ? in the arguments can lead to multiple ? in the result. Thus, composing a constant addition with its inverse loses precision, while an action should return the initial value unchanged.

In practice, this means that we must use compatible abstract relations and values. For instance, the interval abstraction works well with constant difference or TVPE (since addition and multiplication by constants on intervals is exact), and bitwise abstractions work well with the xor-rotate abstract relations (because xor and rotation on bitwise abstractions is also exact).

The following theorem can be combined with Theorem 3.2 to prove that map factorization is as precise as performing full constraint propagation:

Theorem 5.6. *If $\mathcal{A}$ and $\sqcap_{\mathbb{V}^\sharp}$ are both sound and exact then:*

$$\forall\, v_1^\sharp\ v_2^\sharp\ \mathcal{R}^\sharp,\ \mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp \sqcap_{\mathbb{V}^\sharp} v_2^\sharp\right) = \mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp\right) \sqcap_{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}^\sharp,\ v_2^\sharp\right)$$

Proof. See proof in Appendix B.3 (page 39). □

Factorization both takes less memory space and allows for faster operations. The memory benefit comes from the values removed from the value map. Propagation is also faster since updating a single value updates all variables in the same relational class in constant time. Using AC-3 and no factorization we would have tried propagating constraints between every pair of variables in the class; thus we save $O(n^2)$ operations where n is the size of the relational class.

Remark. Factorization can lead to detecting a bottom state late. Consider the graph $x \xrightarrow{\times 0.5} y$. If we change the value of y to only include odd integers, then x has no valid value, so the whole state is bottom. In Section 5.1, this would be detected immediately since changing a value triggers constraint propagation. With the factorized map however, this will only be detected when accessing x. Note that this only happens when we are not in the case of Theorem 4.5: the problem comes from the fact that the function induced by $\times 0.5$ is not total, but only defined on even integers.

6 Reduced Product with Relational Domains

In this section, we focus on the interaction between labeled union-find domains and other relational domain. Specifically, we only examine the inter-reduction procedures since the join operation seems currently impracticable. We focus on two important communication channels [Cousot et al. 2006] between abstract domains, equalities and abstract relations. Note that we already discussed combination with weakly-relational domains in Section 2, and do not discuss it more here.

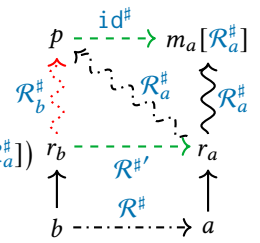
6.1 Labeled Union-Find and Equivalence Relations

We can update the labeled union-find structure to detect new equalities between variables (actually, we can detect the $\texttt{id}^\sharp$ relation, which is generally the equality). Equality is an important communication channel not only when combining decision procedures (e.g. Nelson-Oppen), but also in program analysis [Cousot et al. 2006; Lemerre 2023; Rosen et al. 1988].

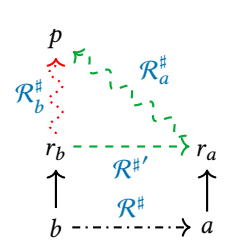
One issue is that labeled union-find can prove identity between pairs of variables when queried using `get_relation`, but it does not automatically detect the list of all identities. For instance, if we know that $y = x + 2$ and $z = x + 2$, we will not discover $y = z$ unless we ask for

```
// Called when setting U[r_b] := (r_a, R#')
// i.e. in add_relation_𝕀 ((U, I), b, a, R#)
m_a ⊓_𝕀 m_b ≜
  for (R#_a ↦ p) ∈ m_b do
    if R#_a ∈ dom m_a then new_id_rel(p, m_a[R#_a])
    else m_a[R#_a] := p
  return m_a
```

(a) Instantiation of $\sqcap_{\mathbb{I}}$.



(b) Case $\mathcal{R}^\sharp_a \in \operatorname{dom} m_a$.



(c) Case $\mathcal{R}^\sharp_a \notin \operatorname{dom} m_a$.

Fig. 6. Instantiation of $\sqcap_{\mathbb{I}}$ to update identity classes, illustrated using squiggly arrows for bindings in I. The illustrations represent the application of $\sqcap_{\mathbb{I}}$ and $\mathcal{A}$ as done in $\texttt{add\_relation}_{\mathbb{I}}\ ((U, I),\ b,\ a,\ \mathcal{R}^\sharp)$ (Figure 5).

$\texttt{get\_relation}\,(U,\ y,\ z)$. Instead, we want the labeled union-find domain to "push" new inferred equalities by calling a user-supplied function, $\texttt{new\_id\_rel}\,(y,\ z)$, whenever $y \xrightarrow{\texttt{id}^\sharp} z$.

To solve this, we will first partition the variables $\mathbb{X}$ using the $\texttt{id}^\sharp$ relation. This is an instance of constraint factorization as we now only need to relate equivalence classes of $\texttt{id}^\sharp$. Then, we add some information on relational classes, namely a mapping from labels to variables (i.e., following Section 3.3, with $\mathbb{I} = \mathbb{R}^\sharp \rightharpoonup \mathbb{X}$). The information can be seen as a trie representing the labeled union-find tree in the opposite direction, mapping the root to a representative element of the $\texttt{id}^\sharp$ equivalence class. Formally, the following invariants are preserved in a (U, I) pair:

- $\texttt{find}\,(U,\ x) = (r, \mathcal{R}^\sharp) \Rightarrow I[r][\mathcal{R}^\sharp] \xrightarrow{\texttt{id}^\sharp} x$
- $(\mathcal{R}^\sharp \mapsto x) \in I[r] \Rightarrow \texttt{find}\,(U,\ x) = (r, \mathcal{R}^\sharp)$

Initially, we initialize the (U, I) pair as follows: $\texttt{init}_{\mathbb{I}}\,() \triangleq (\emptyset, \left[n \in \mathbb{N} \mapsto [\texttt{id}^\sharp \mapsto n]\right])$. The action shifts all keys by the provided relation: $\mathcal{A}(\mathcal{R}^\sharp, I) \triangleq I[\texttt{inv}^\sharp(\mathcal{R}^\sharp) \fatsemi^\sharp \cdot]$. The $\sqcap_{\mathbb{I}}$ operation is defined in Figure 6. It merges the mappings, and calls $\texttt{new\_id\_rel}$ on discovered $\texttt{id}^\sharp$ relations.

The above algorithm guarantees that the transitive-reflexive closure of the calls to $\texttt{new\_id\_rel}$ (which can be constructed using a classic union-find), is the set of all the $\texttt{id}^\sharp$ that are implied by U. Moreover, there are no redundant calls to $\texttt{new\_id\_rel}$.

Using the same information, one can also detect more relations during the merge; for instance we can detect disequalities when merging incompatible constant difference relations (e.g. $y = x + 2$ and $y = x + 3$), or constant differences when merging TVPE relations (e.g. $y = 2x + 3$ and $z = 2x + 7$). As presented in Figure 3, constant difference is useful to perform constraint factorization on DBMs.

6.2 Reduced Product with Shostak Theory

While the previous section made use of abstract relations, this section describes how new relations can be discovered using an extension of Shostak theory [Shostak 1984]. Moreover, we will see how the labeled union-find can optimize the implementation of Shostak theory.

To summarize, a Shostak Theory $\mathcal{T}$ as defined in Barrett et al. [2002] with signature Σ, canonizer $\texttt{canon}$, and solver $\texttt{solve}$, is able to decide if a set of equations E in Σ implies an equality $t_1 = t_2$ in E. The function $\texttt{solve}$ returns a substitution from an equation, and $\texttt{canon}$ normalizes a term ($\mathcal{T} \models t_1 = t_2$ iff $\texttt{canon}\,(t_1) \equiv \texttt{canon}\,(t_2)$, where $\equiv$ is syntactic equality). The algorithm computes incrementally equisatisfiable sets S_i of equations where their left-hand side is a variable that appears only once in the set. In the following, such sets of equations are considered to be substitutions.

Example 6.1. The theory of linear real arithmetic is a good example of a Shostak theory. Here, $\texttt{canon}$ is obtained from an ordering on the variables, and $\texttt{solve}$ consists in Gaussian elimination

(expressing one variable as a linear equation using the others). The algorithm processes the set $E = \{\overbrace{-z+y-u=0}^{e_1}, \overbrace{x+2z=2z-u}^{e_2}, \overbrace{-t-2y=z+2v}^{e_3}, \overbrace{z-2=-y-v}^{e_4}\}$ as follows:

- $\sigma_i \triangleq \texttt{solve}\,(S_i(e_i))$, with $S_0 \triangleq \emptyset$ and $S_i \triangleq \sigma_i(S_{i-1}) \cup \sigma_i$ for $i \geq 1$
- $\sigma_1 = \{u = y - z\}$, $\sigma_2 = \{x = z - y\}$, $\sigma_3 = \{t = -2y - 2v - z\}$, $\sigma_4 = \{z = -y - v + 2\}$
- $S_3 = \{u = y - z, x = z - y, t = -2y - 2v - z\}$
- $S_4 = \{u = y - 3, x = 3 - y, t = -y - v - 2, z = -y - v + 2\}$

The entailed equalities between variables (such as $u = x$ in S_3 in Example 6.1) can be found by keeping a reverse mapping M from the canonized right-hand side to a representative of their left-hand side. This is useful for propagating equalities among theories. Moreover, a union-find-like data structure Δ usually allows remembering $u = x$, and stores the right-hand side, $y - 3$ for S_3, only once at the representative of x and u (as described in Conchon et al. [2007]). Additionally, it avoids substituting in both definitions of u and x when computing S_4.

We propose to use labeled union-find in order to further factorize the equations and find disequalities between variables (or other relations). Given labels ℓ which follow the hypothesis of Section 3, an extended Shostak Theory requires, in addition to `canon`, a function `canon_rel` that returns a term and a label: $\texttt{canon\_rel}\,(t) = (t', \ell)$. Similarly to `canon`, we say that $\mathcal{T} \models t_1 = t_2$ iff $t_1' \equiv t_2'$ and $\ell_1 = \ell_2$ (where $(t_1', \ell_1) = \texttt{canon\_rel}\,(t_1)$ and $(t_2', \ell_2) = \texttt{canon\_rel}\,(t_2)$). We also need a group action $\mathcal{A}$, that rebuilds a term from a term and a label. Formally, if $\texttt{canon\_rel}\,(t) = (t', \ell)$ then $\mathcal{A}(\ell, t') = \texttt{canon}\,(t)$. Given these building blocks, we can optimize M and Δ. The reverse mapping for $\texttt{canon\_rel}\,(t) = (t', \ell)$ only needs to map t'. Δ becomes a labelled union-find with label ℓ and with $\mathbb{I}$ the term at the right-hand side.

Example 6.2. Using the abstract relation constant difference (Example 2.1), `canon_rel` separates the constant part from the rest after normalization (e.g. $\texttt{canon\_rel}\,(-y - v - 2) = (-y - v, -2)$). In S_4, it allows storing and substituting only in $t = -y - v - 2$, while keeping the relation $z = t + 4$ in the labeled union-find Δ. Combined with the equivalence class described in Section 6.1, this still entails equality, and additionally entails disequalities.

This Example 6.2 is used in our evaluation in Section 7. The collaboration between the Shostak Theory of linear arithmetic and the abstract relation constant difference provides a qualitative gain on other domains. The newly found relational information allows new propagations.

Example 6.3. With the equations from Example 6.1, if we know $t \in [0; 10]$, forward and backward propagation of intervals on arithmetic operators and equalities cannot propagate any information through e_3 since, for example propagation from $z + 2v$ to z is imprecise. However, with the labeled union-find Δ for constant difference of Example 6.2 as in Section 5, $z \in [4; 14]$ is directly deduced.

Using the TVPE relational domain instead of constant difference gives even more benefits, since $x = 3u + 3v$ and $y = 8u + 8v$ (related by $(8/3, 0)$) would be factorized and their domains would be propagated directly. However, since it does not satisfy HUniqueLabel, some care is needed during conflict to propagate the learned constants.

7 Preliminary Implementation and Evaluation Results

7.1 Extending a Constraint Solver with Constant Difference

Colibri2 is a new constraint solver used for program verification. It handles quantified SMT formulas across many theories. It emphasizes propagation inside and between theories.

Motivation. While working on the theory of sequences (Dynamically sized arrays [Ait-El-Hara et al. 2024a; Sheng et al. 2023]), we found a lack of propagation of the interval domain between some arithmetic terms. For example, in Figure 7, `t` is represented as a unique sequence of size `100`, with access modeled as $\text{seq.nth}(t, 10i + (j+1))$. This requires proving that $10i + (j+1) \in [0; 99]$, which cannot be deduced from $10i + j \in [0; 89]$ using basic interval propagation as shown in Example 6.3 and as is done in the original solver, which will be referred to as BASE.

```
int t[10][10];
if(0 <= 10*i + j < 90){
  a = t[i][j + 1]; ...
}
```

Fig. 7. Fragment of C program.

Users of verification tools are often frustrated when seemingly simple problems are not solved. For example, the problem Example 7.1 that only use one multiplication is not solved by BASE. A decision procedure for non linear-arithmetic is difficult to implement and costly, but in that case, a propagation between $f(4)$ and $f(9)$ which are at a constant distance of 5 would suffice. The simplex algorithm can be theoretically used for this propagation, but adding the negation of every unknown comparisons or calling a maximization and minimization for every term is costly.

Example 7.1. Given two real variables a and b, a function $f(x) = 2a + x + 3b$ and the assertion that $10 < f(4)$ holds. $f(9)^2 \leq 225$ is unsatisfiable for any values of a and b.

Implementation. We want to propagate across terms that are related by the constant difference abstract relation (Example 2.1), so we can use the labeled union-find to group them (Section 6.1). The relations between these terms are found following Section 6.2. This lead to the variant of Colibri2 where we propagate the interval domain between elements of those labeled union-find relational classes, which we call LABELED-UF.

We also went further by factorizing (Section 5.2) the interval domain. This third version is referred to as GROUP-ACTION. Initially the interval domain used in Colibri2 was not a group action for the relation: since reals and integers are handled together (to reason about conversions), the domain had a flag to indicate if it only contained integers. This "is integer" flag was not a group action: e.g., adding and removing 0.5 loses precision. So we replaced this flag with the congruence domain on rationals [Granger 1989, 1997], which is a group action for the constant difference relations (and for TVPE, which will be useful when we integrate them). This congruence domain is also used by BASE and LABELED-UF for a fair comparison.

During development, we found that many regressions of LABELED-UF and GROUP-ACTION compared to BASE were due to slow convergences [Bordeaux et al. 2007], i.e., very long or infinite sequences of propagations. With the new propagations naturally came new slow convergences. In Colibri2, these are normally limited by stopping the propagations from a term when it has been updated too many times. But in this case, the presence of non-linear constraints, rationals and unbounded variables, lead here to an unforeseen kind of slow convergences: the rational numbers used in the bounds of the intervals grew too fast to fit into memory, leading to more "out of memory" errors. To remedy this, we limited the propagation of the interval domain when its bounds take more than 20 memory words. It helped LABELED-UF to be on par with BASE. For GROUP-ACTION, we added a second improvement, when the propagation of the multiplication creates a domain that uses numbers that are too big in memory, it is over-approximated with bounds that use smaller denominators. It is a sort of on-demand floating point approximation. Even though GROUP-ACTION benefited the most from these fixes, LABELED-UF and BASE also improved.

Both variants LABELED-UF and GROUP-ACTION are able to make the propagation between $10i + j$ and $10i + (j+1)$, for Figure 7, and solve Example 7.1.

Test Bench. To evaluate our new implementations on a large scale, we tested them on the SMT-LIB [Barrett et al. 2016] 2024 benchmarks [Preiner et al. 2024] for integer and real arithmetic theories, corresponding to the logics QF_LIA, QF_LRA, QF_LIRA, QF_IDL, QF_RDL, QF_NIA, QF_NRA, and QF_NIRA, totaling 55,449 problems which are not sourced only from program verification.

Table 1. Comparison of Colibri2 variants on SMT-LIB benchmarks. Number of newly solved (+), unsolved (-) problems and difference for each row compared to columns. In total 17465 problems are solved by BASE.

	BASE	LABELED-UF
LABELED-UF	-49 +61 (+12)	
GROUP-ACTION	-65 +70 (+5)	-39 +22 (-17)

We compared the LABELED-UF and GROUP-ACTION implementations against BASE, the original implementation in Colibri2. Experiments were conducted using six cluster nodes, each equipped with 72 cores at 3GHz and 187GB of RAM. 30 cores per node were used, with a time limit of one minute and a memory limit of 4GB per problem.

Results. Table 1 presents the results of our experimentation. Comparing only the number of solved problems before the 60s timeout biases the result towards the problems that are solved near the time limit. So we consider that a solver version improves on a problem compared to another solver version if it solves it in less than 55s (cutoff) while the other is not able to solve it in 60s. Both LABELED-UF and GROUP-ACTION improve upon BASE, with respectively 12 and 5 more improvements. Interestingly if we use a cutoff of 5s, 14 problems are solved by both variants in less than 5s, while BASE cannot solve them in 60s (conversely it is 1 and 3 respectively). However, for some problems that remains to be investigated, GROUP-ACTION solves them a lot slower than LABELED-UF or BASE.

In conclusion, the labeled union-find allowed us to easily implement new propagations compared to BASE. LABELED-UF and GROUP-ACTION have comparative results from BASE. For now the regressions found are due to the additional propagations. It is in a sense the price of success, we are able to solve a problem our user got, while losing a small percentage of time overall. GROUP-ACTION is behind LABELED-UF, its implementation is more complex and requires more future refinements. Finally, the next step will be to implement TVPE, which will provide even more new propagations.

7.2 Extending an Abstract Interpreter with TVPE

Our experiments focus on the following questions: what is the performance impact of using labeled union-find (with TVPE) in a non-relational abstract Interpreter in terms of runtime and memory usage? To what extent can labeled union-find improve precision of such an analyzer?

Implementation. We used labeled union-find in the Codex static analyzer for C code. Specifically, our implementation performs map factorization as seen in Section 5.2. It works with the TVPE abstract relations (Example 4.6) and a reduced product between interval and congruence [Granger 1989] as a non-relational abstraction.

```
int i = 0, j = 4;
while(i < 10) {
    i += 1;
    j += 3;
}
```

Fig. 8. Small C program.

One specificity of Codex is that it performs the numerical analysis after SSA translation [Lemerre 2023; Lesbre and Lemerre 2024]. This allows us to use mutable union-find to represent flow-insensitive relations between SSA variables (which are bound and not assigned), and thus avoids the performance costs caused by the use of a confluently persistent implementation (Appendix A). However, it limits the relations we can use: we cannot learn from conditionals (integration with flow-sensitive implementations of union-find is planned in future work, another option would be using an e-SSA [Bodík et al. 2000] or SSI [Ananian 2001; Boissinot et al. 2012] form). We thus only add relations in the following cases:

- **variable definitions:** when an SSA variable y is bound to an addition between a variable and a constant (e.g. $x + 1$), we can add the relation $(y \xrightarrow{+1} x)$. The same goes for subtraction and multiplication (if the analysis can prove absence of possible wrap-around);
- **joining related variables:** if a branch contains $y_0 \xrightarrow{\mathcal{R}^\sharp} x_0$ and the other has the same relation $y_1 \xrightarrow{\mathcal{R}^\sharp} x_1$, then the join $x_2 = \phi(x_0, x_1); y_2 = \phi(y_0, y_1)$ also satisfies $y_2 \xrightarrow{\mathcal{R}^\sharp} x_2$;
- **joining constants:** the labeled union-find can represent some information lost in the join of the non-relational domain. With TVPE, we can relate ϕ terms with constant arguments: it amounts to finding a line through two points. For instance, if a branch contains $x_0 = 1; y_0 = 3$ and the other $x_1 = 2; y_1 = 5$, then the join $x_2 = \phi(x_0, x_1); y_2 = \phi(y_0, y_1)$ satisfies $y_2 \xrightarrow{\bullet\times 2+3} x_2$. This may or may not be possible depending on the relation being used. With TVPE, we can relate all constant ϕ-terms: it amounts to finding a line through two points.

With this construction, conflicts will never occur since there is at most one relation per variable.

Example 7.2. The C code of Figure 8 illustrates how joining constants and related variables can improve precision of the analysis. Running a simple non-relational analysis on it yields $i = 10, j \in [4 : +\infty], j \equiv 1 \mod 3$ after the loop. However, with labeled union-find, the relation $j = 3 * i + 4$ is inferred and maintained through the loop and widening, leading to a final value of $i = 10, j = 34$.

Test Bench. We selected 584 C functions from the `ReachSafety` category of the SV-Comp benchmarks [Beyer 2024]. We selected the test folders that focus mainly on numerical tasks.

Method. We have run Codex, both with and without the labeled union-find domain, Codex has sophisticated constraint propagation: when learning new information about a variable x, Codex propagates it both upwards to the variables used to define x, and downwards to the variables that use x in their definitions. The default depth limit for this propagation is 1000. To mimic the behavior of a simpler analyzer, we ran the experiment again, setting the depth limit to 2.

For each run, we measured the runtime, memory usage, number of performed unions, maximum relational class size, and the count of unsolved alarms and assertions. The 584 tests totaled 16507 lines of code, with an average of 28 lines per tests and a maximum of 1408.

Results. 451 tests called `add_relation`, with an average of 40 calls per analysis. The relational classes are small, with an average of 2.4 SSA variables in the largest class, and a maximum of 12. Comparing the number of unions performed to the number of SSA variables shows that 12% of the bitvector variable are in unions on average, with a maximum of 43%.

In terms of performance: the memory consumption was comparable, and runtime was on average 10% slower when using union-find. This confirms the relatively cheap nature of this domain.

As expected, there were no cases where using labeled union-find lead to precision losses. There were only 23/584 cases where the labeled union-find allowed tightening the non-relational information. In 11 of these cases, the precision improvements allowed proving new alarms or assertions. When using the lower constraint propagation limit, we get 122/584 cases of precision improvements of non-relational information, with 22 cases leading to proving more alarms or assertions.

8 Related Work

We have previously published a work-in-progress short paper at a non-archival venue [Lemerre and Lesbre 2024], which summarized the concept of labeled union-find and some results of Section 4. It did not include the algorithms, detailed formalization, theorem statements, nor any experimental evaluation. The current paper provides a significantly expanded treatment, including all those missing elements.

Weakly Relational Domains. Our formalization of weakly-relational domains differs from the usual presentation. Schwarz et al. [2023] proposes a general presentation of weakly relational domains based on a 2-decomposability notion, requiring the domain to be expressible as a conjunction of constraints over single variables and pairs of variables, and the join to be applied to each component of the conjunction. In our formalization, we can represent all the 2-decomposable domains as a reduced product between our weakly relational abstraction $\mathbb{W}^\sharp = (\mathbb{X}\times\mathbb{X}) \to \mathbb{R}^\sharp$ and a non-relational abstraction $\mathbb{X} \to \mathbb{V}^\sharp$. This includes Octagons [Miné 2006] (as pairs of intervals constraining $x - y$ and $x + y$), DBMs [Miné 2001], Pentagons [Logozzo and Fähndrich 2008], or TVPI [Simon and King 2010], but not general polyhedra [Cousot and Halbwachs 1978] or linear equalities [Karr 1976]. There are conditions for the Floyd-Warshall algorithm to compute the transitive closure [Schwarz and Seidl 2023]. Some weakly relational domains that can be represented in our formalization, like DBMs with disequalities [Péron and Halbwachs 2007], do not fit these criteria.

The formalization that most closely resembles ours is that of [Miné 2002], but the relation is limited to the form $y - x \in \mathcal{R}^\sharp$, which cannot represent bitvector or octagon constraints. Bagnara [1998] describes other relations between pairs of reals, such as bounded quotient.

Making Relational Domains More Efficient. The supra-linear complexity of relational domains was quickly noted as a problem, and solutions were devised to work around this problem. One pragmatic solution is packing [Blanchet et al. 2003; Heo et al. 2016; Venet and Brat 2004], i.e., considering only the relations between different clusters of variables based on some heuristics. While effective, this strategy leads to precision losses, as we forget about some of the relations between variables.

One of the first attempts to gain precision in relational abstract domains (especially polyhedra) without losing precision is Halbwachs et al. [2006], which proposes three independent techniques: factoring/decomposition, i.e., detecting clusters of variables with no relations between them; substituting variables using linear equations to remove one variable, and base changes. They reported improvements of the factoring method, which has since been successfully explored in [Singh et al. 2015, 2017, 2018]. However, their results with the variable substitution method were disappointing, as the method allowed negligible gains (possibly because the implementation of Chernikova's algorithm already uses variable substitution) but could cause significant overhead (possibly because Karr's domain [Karr 1976], which they also used, has $O(|\mathbb{X}|^3)$ operations [Müller-Olm and Seidl 2004]). Our constraint factorization method has similarities with this technique, but the domains we propose in this paper are cheap. Variable substitution is not used in weakly-relational domains, and we can already observe speedups when using them (in the constraint programming setting), which makes this research direction worthy of re-exploration.

Outside of numerical analysis, Cox et al. [2015] describes the use of an abstract domain computing equalities, and its use for constraint factorization of domains computing relations between sets, which provides significant performance gains in equality-heavy loads.

Making Weakly Relational Domains More Efficient. Weakly-relational domains were developed as a remedy against the high computational complexity of relational domains, but their supra-linear complexity ($O(|\mathbb{X}|^2)$ storage, $O(|\mathbb{X}|^3)$ computation time), coming from the need to perform transitive closure, remains a bottleneck in program analysis for large instances. Outside of decomposition [Singh et al. 2015], we can also exploit the natural sparsity of the graph, i.e. not store edges that carry no information (e.g., if their interval difference relation is $[-\infty, +\infty]$). However, as soon as the values of two variables are constrained by an interval, some information on their difference is known, making the graph spuriously dense. Following this observation, Gange et al. [2021] and Jourdan [2016] propose a method that can be seen as constraint elimination when the eliminated

constraints can be recovered using the non-relational domain, making the graph sparse. This method is orthogonal to our constraint factorization method, so a combination of both is possible.

One way to make domains more efficient is sparse abstract interpretation [Mirliaz and Pichardie 2022; Paisante et al. 2016; Tavares et al. 2014], i.e. having a global flow-insensitive invariant, and rely on variable renaming/live-range splitting to avoid losing precision. This would be a natural candidate for labeled union-find, as these representations simultaneously often have many variables (which would benefit from factorization (Section 5)), and flow-insensitive information allows using mutable union-find and avoiding expensive join operations.

Semi-relational Domains. Semi-relational domains [Bodík et al. 2000; Logozzo and Fähndrich 2008; Nazaré et al. 2014] are another family of abstract domains that are cheaper than weakly relational methods. They are implemented using a $\mathbb{X} \to$ "abstraction" map ($O(|\mathbb{X}|)$ complexity), where "abstraction" can refer to other variables. Unlike our domains, they do not perform any transitive closure; instead, they just take advantage of the relations that they encounter. They are effective in solving bound-checking problems. As they are also inexpensive, a combination of these domains with labeled union-find domains could improve precision while remaining inexpensive.

Instances of Labeled Union-Find. The labeled union-find structure was introduced by Frühwirth [2007, 2009] (under the name generalized union-find), and applied on non-integer TVPE and single-bit `xor` relations. Zucker [2022] mentions this data structure, and gives examples of interesting and suitable groups.

Several existing works can also be viewed as specific instances of the labeled union-find data structure. Aspvall and Shiloach [1980] uses TVPE on a spanning forest to efficiently solve systems of linear equations with at most two variables per equation. Ghidini et al. [2024] describes an instance of labeled union-find using constant difference relations, and reports that the analysis scales well and infers useful properties. Ait-El-Hara et al. [2024b] describes a theory of n-indexed sequences featuring a family of equivalence modulo a relocation relation denoted $s_1 =_{\texttt{reloc}(d)} s_2$, between two n-indexed sequences s_1 and s_2, where the relation indicates that s_1 and s_2 have the same content but have indices shifted by d. They also hint at a variation of the union-find data structure based on this relation. Nieuwenhuis and Oliveras [2005] use identifiers of union operations as labels to compute the smallest set of union operations that connects two elements. This can be achieved using labeled union-find over the free group (ignoring that some elements are inversed when returning the set).

In all these works, the edges are labeled by bijective concrete mathematical relations. By using abstract relations, we can accomodate cases where the relations are not bijective (Example 4.6), and where the identity relation can be an equivalence relation distinct from equality.

Other Extensions of Union-Find. Tarjan [1979] describes an extension of the union-find data structure, the link-eval structure, notably used to compute dominators in Lengauer and Tarjan [1979]. The structure also composes labels on a path, where labels only have to obey the semigroup axioms. Despite these superficial similarities, the structure is actually quite different; notably in labeled union-find the label is carried on edges, whereas it is carried on nodes in link-eval.

9 Conclusion

We have presented a new family of cheap relational abstract domains based on the new labeled union-find data structure. We started from a formalization of weakly relational abstract domains as graphs labeled by abstract relations. To make dynamic computation of transitive closure efficient, we assumed a unique label assumption that led us to discover a data structure that we call *labeled union-find*. Labeled union-find further assumes that the composition of labels follows the group axioms. For abstract relations, this latter assumption implies that abstract relations represent

injective transformations between equivalence classes. We proposed many interesting instances of abstract relations, including TVPE (affine relations of the form $y = a * x + b$).

Domains based on labeled union-find can be combined with other numerical domains by reduced product or by constraint factorization (linking only to one element in the *relational class* of elements related in a labeled union-find), and we studied different applications, including combination with non-relational domains, with weakly relational domains, with equalities, or with domains exchanging information through an extension of Shostak theory. We have found use cases in both program analysis and constraint solving for SMT, with promising initial benchmarks that show that adding labeled union-find is cheap yet can store and propagate useful information.

We have described a few distinct instantiations of the framework, but its genericity could lead to many other applications, and this paper could be a first step in a promising line of research. In general, the union-find structure is often used in program verification, as an efficient way to compute closure of equivalence relations. It is used to perform efficient unification [Paterson and Wegman 1978], to speed up datalog computations [Nappa et al. 2019; Sahebolamri et al. 2023], in alias analysis [Steensgaard 1996], in abstract interpretation [Chang and Leino 2005; Cox et al. 2015], etc. *Labeled union-find extends the possibilities of using this efficient data structure to new usages.*

The labeled union-find data structure described in Section 3 could also find uses outside of program analysis and verification. Consider an unknown variable x_0. We repeatedly derive new variables x_i by applying invertible transformations on x_0 and its derived variables. Labeled union-find easily solves how one can transform one variable to another, or compute the value of a variable when the value of another is known. Applications could be solving geometry or Rubik's cube problems, but it would be interesting to find even more use cases for efficient transitive closure of group elements.

Acknowledgments

This publication was made possible by the use of the FactoryIA supercomputer, financially supported by the Ile-De-France Regional Council. This research was supported in part by the Agence Nationale de la Recherche (ANR) grant agreement ANR-22-CE39-0014-03 (EMASS project).

Data-Availability Statement

The software that supports Section 7 is open source and available under an GPU LGPL v2.1 License. Colibri2 can be found at https://colibri.frama-c.com/index.html and Codex at https://codex.top. For the specific version of these tools used in this paper, a software artifact including docker images with prebuilt binaries and instructions to run the experiments is available on Zenodo doi:10.5281/zenodo.15165896 [Lesbre et al. 2025a].

References

Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. 2024a. On SMT Theory Design: The Case of Sequences. In *LPAR 2024 Complementary Volume (Kalpa Publications in Computing, Vol. 18)*, Nikolaj Bjørner, Marijn Heule, and Andrei Voronkov (Eds.). EasyChair, 14–29. https://doi.org/10.29007/75tl

Hichem Rami Ait-El-Hara, François Bobot, and Guillaume Bury. 2024b. An SMT Theory for n-Indexed Sequences. In *Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories (CEUR Workshop Proceedings, Vol. 3725)*, Giles Reger and Yoni Zohar (Eds.). CEUR, Montreal, Canada, 64–74. https://ceur-ws.org/Vol-3725/#short13

C Scott Ananian. 2001. *The static single information form.* Master's thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/bitstream/handle/1721.1/86578/48072795-MIT.pdf

Bengt Aspvall and Yossi Shiloach. 1980. A fast algorithm for solving systems of linear equations with two variables per equation, Vol. 34. 117–124. https://doi.org/10.1016/0024-3795(80)90162-7

Roberto Bagnara. 1998. *Data-Flow Analysis for Constraint Logic-Based Languages.* Ph. D. Dissertation. Universita di Pisa.

Clark W. Barrett, David L. Dill, and Aaron Stump. 2002. A Generalization of Shostak's Method for Combining Decision Procedures. In *FroCoS '02*. Springer, 132–146. https://doi.org/10.5555/646821.706603

Clack W. Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org

Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. 1999. Revising Hull and Box Consistency. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, Danny De Schreye (Ed.). MIT Press, 230–244. https://doi.org/10.5555/341176.341208

Dirk Beyer. 2024. *SV-Benchmarks: Benchmark Set for Software Verification (SV-COMP 2024).* https://doi.org/10.5281/zenodo.10669723

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *PLDI'03*. ACM Press, San Diego, California, USA, 196–207. https://doi.org/10.1145/781131.781153

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In *PLDI 2000*, Monica S. Lam (Ed.). ACM, 321–333. https://doi.org/10.1145/349299.349342

Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. 2012. SSI Properties Revisited. *ACM Trans. Embed. Comput. Syst.* 11S, 1 (jun 2012), 21:1–21:23. https://doi.org/10.1145/2180887.2180898

Lucas Bordeaux, Youssef Hamadi, and Moshe Y. Vardi. 2007. An Analysis of Slow Convergence in Interval Propagation. In *Principles and Practice of Constraint Programming – CP 2007*, Christian Bessière (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 790–797. https://doi.org/10.1007/978-3-540-74970-7_56

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. *Abstract Interpretation with Alien Expressions and Heap Structures.* Springer, 147–163. https://doi.org/10.1007/978-3-540-30579-8_11

Zakaria Chihani, Bruno Marre, François Bobot, and Sébastien Bardin. 2017. Sharpening Constraint Programming Approaches for Bit-Vector Theory. In *CPAIOR '17 (Lecture Notes in Computer Science, Vol. 10335)*, Domenico Salvagnin and Michele Lombardi (Eds.). Springer, 3–20. https://doi.org/10.1007/978-3-319-59776-8_1

Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. 2007. CC (X): Efficiently combining equality and solvable theories without canonizers. *SMT* (2007). https://doi.org/10.1016/j.entcs.2008.04.080

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. *ML Workshop* (10 2007), 37–46. https://doi.org/10.1145/1292535.1292541

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL '79* (San Antonio, Texas). Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Revised Selected Papers from the 11th Asian Computing Science Conference on Advances in Computer Science - Secure Software and Related Issues – ASIAN 2006 (Lecture Notes in Computer Science, Vol. 4435)*. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program, In POPL '78. *ACM-SIGACT Symposium on Principles of Programming Languages*, 84–96. https://doi.org/10.1145/512760.512770

Arlen Cox, Bor-Yuh Evan Chang, Huisong Li, and Xavier Rival. 2015. Abstract Domains and Solvers for Sets Reasoning. In *LPAR-20 2015 (Lecture Notes in Computer Science, Vol. 9450)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer, 356–371. https://doi.org/10.1007/978-3-662-48899-7_25

James R. Driscoll, Daniel Dominic Sleator, and Robert Endre Tarjan. 1994. Fully Persistent Lists with Catenation. *J. ACM* 41, 5 (1994), 943–959. https://doi.org/10.1145/185675.185791

Amos Fiat and Haim Kaplan. 2003. Making data structures confluently persistent. *J. Algorithms* 48, 1 (2003), 16–58. https://doi.org/10.1016/S0196-6774(03)00044-0

Thom W. Frühwirth. 2007. Quasi-Linear-Time Algorithms by Generalisation of Union-Find in CHR. In *CSCLP 2007 (Lecture Notes in Computer Science, Vol. 5129)*, François Fages, Francesca Rossi, and Sylvain Soliman (Eds.). Springer, 91–108. https://doi.org/10.1007/978-3-540-89812-2_7

Thom W. Frühwirth. 2009. *Constraint Handling Rules.* Cambridge University Press, Chapter 10.4, 256–280. https://doi.org/10.1017/CBO9780511609886.014

Bernard A. Galler and Michael J. Fischer. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (1964), 301–303. https://doi.org/10.1145/364099.364331

Graeme Gange, Zequn Ma, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2021. A Fresh Look at Zones and Octagons. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021), 11:1–11:51. https://doi.org/10.1145/3457885

Rebecca Ghidini, Julian Erhard, Michael Schwarz, and Helmut Seidl. 2024. C-2PO: A Weakly Relational Pointer Domain: "These Are Not the Memory Cells You Are Looking For". In *NSAD@SAS 2024* (Pasadena, CA, USA). Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/3689609.3689994

Roberto Giacobazzi and Elisa Quintarelli. 2001. Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings (Lecture*

Notes in Computer Science, Vol. 2126), Patrick Cousot (Ed.). Springer, 356–373. https://doi.org/10.1007/3-540-47764-0_20

Ashish Goel, Sanjeev Khanna, Daniel H. Larkin, and Robert Endre Tarjan. 2014. Disjoint Set Union with Randomized Linking. In *SODA '14*, Chandra Chekuri (Ed.). SIAM, 1005–1017. https://doi.org/10.1137/1.9781611973402.75

Philippe Granger. 1989. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics* 30, 3-4 (1989), 165–190. https://doi.org/10.1080/00207168908803778

Philippe Granger. 1997. Static analyses of congruence properties on rational numbers (extended abstract). In *Static Analysis*, Pascal Van Hentenryck (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–292.

Nicolas Halbwachs, David Merchat, and Laure Gonnord. 2006. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods Syst. Des.* 29, 1 (2006), 79–95. https://doi.org/10.1007/S10703-006-0013-2

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *SAS '16 (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 237–256. https://doi.org/10.1007/978-3-662-53413-7_12

Jacques-Henri Jourdan. 2016. Sparsity Preserving Algorithms for Octagons. In *NSAD@SAS '16 (Electronic Notes in Theoretical Computer Science, Vol. 331)*, Isabella Mastroeni (Ed.). Elsevier, 57–70. https://doi.org/10.1016/J.ENTCS.2017.02.004

Michael Karr. 1976. Affine Relationships Among Variables of a Program. *Acta Informatica* 6 (1976), 133–151. https://doi.org/10.1007/BF00268497

Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (Jan. 2023), 30 pages. https://doi.org/10.1145/3571258

Matthieu Lemerre and Dorian Lesbre. 2024. Labeled union-find for constraint factorization. In *NSAD 24*. Pasadena, United States. https://cea.hal.science/cea-04996700

Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141. https://doi.org/10.1145/357062.357071

Dorian Lesbre and Matthieu Lemerre. 2024. Compiling with Abstract Interpretation. *Proc. ACM Program. Lang.* 8, PLDI (2024), 368–393. https://doi.org/10.1145/3656392

Dorian Lesbre, Matthieu Lemerre, Hichem Rami Ait-El-Hara, and François Bobot. 2025a. *Artifact for paper "Relational Abstractions Based on Labeled Union-Find"*. https://doi.org/10.5281/zenodo.15165896

Dorian Lesbre, Matthieu Lemerre, Hichem Rami Ait-El-Hara, and François Bobot. 2025b. Relational Abstractions Based on Labeled Union-Find. *Proc. ACM Program. Lang.* 9, PLDI, Article 195 (6 2025). https://doi.org/10.1145/3729298

Francesco Logozzo and Manuel Fähndrich. 2008. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 184–188. https://doi.org/10.1145/1363686.1363736

Alan K. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence* 8, 1 (1977). https://doi.org/10.1016/0004-3702(77)90007-8

Laurent D. Michel and Pascal Van Hentenryck. 2012. Constraint Satisfaction over Bit-Vectors. In *CP 2012 (Lecture Notes in Computer Science, Vol. 7514)*, Michela Milano (Ed.). Springer, 527–543. https://doi.org/10.1007/978-3-642-33558-7_39

Antoine Miné. 2001. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *PADO 2001 (Lecture Notes in Computer Science, Vol. 2053)*, Olivier Danvy and Andrzej Filinski (Eds.). Springer, 155–172. https://doi.org/10.1007/3-540-44978-7_10

Antoine Miné. 2002. A few graph-based relational numerical abstract domains. In *SAS '02 (Lecture Notes in Computer Science (LNCS), Vol. 2477)*. Springer, 117–132. https://doi.org/10.1007/3-540-45789-5_11 http://www.di.ens.fr/~mine/publi/article-mine-sas02.pdf.

Antoine Miné. 2004. *Weakly relational numerical abstract domains.* Ph. D. Dissertation. École Polytechnique. http://www.di.ens.fr/~mine/these/these-color.pdf.

Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19, 1 (2006), 31–100. https://doi.org/10.1109/WCRE.2001.957836

Antoine Miné. 2012. Abstract domains for bit-level machine integer and floating-point operations. In *WING'12 - 4th International Workshop on invariant Generation*. Manchester, United Kingdom, 16. https://doi.org/10.29007/b63g

Antoine Miné. 2017. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages* 4, 3-4 (2 2017), 120–372. https://doi.org/10.1561/2500000034

Solène Mirliaz and David Pichardie. 2022. A Flow-Insensitive-Complete Program Representation. In *VMCAI 2022* (Philadelphia, PA, USA). Springer-Verlag, Berlin, Heidelberg, 197–218. https://doi.org/10.1007/978-3-030-94583-1_10

Markus Müller-Olm and Helmut Seidl. 2004. A Note on Karr's Algorithm. In *Automata, Languages and Programming*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-27836-8_85

Patrick Nappa, David Zhao, Pavle Subotic, and Bernhard Scholz. 2019. Fast Parallel Equivalence Relations in a Datalog Compiler. In *PACT 2019*. IEEE, 82–96. https://doi.org/10.1109/PACT.2019.00015

Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa e Oliveira, Laure Gonnord, and Fernando Magno Quintão Pereira. 2014. Validation of memory accesses through symbolic analyses. In *OOPSLA '14*, Andrew P. Black and Todd D.

Millstein (Eds.). ACM, 791–809. https://doi.org/10.1145/2660193.2660205

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-producing congruence closure. In *RTA'05*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *Workshop on ML*. 77–86.

Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2016. Symbolic range analysis of pointers *(CGO '16)*. ACM, 171–181. https://doi.org/10.1145/2854038.2854050

M.S. Paterson and M.N. Wegman. 1978. Linear unification. *J. Comput. System Sci.* 16, 2 (1978), 158–167. https://doi.org/10.1016/0022-0000(78)90043-0

Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. 2010. Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure. In *SEA 2010 (Lecture Notes in Computer Science, Vol. 6049)*, Paola Festa (Ed.). Springer, 411–423. https://doi.org/10.1007/978-3-642-13193-6_35

Mathias Péron and Nicolas Halbwachs. 2007. An Abstract Domain Extending Difference-Bound Matrices with Disequality Constraints. In *VMCAI '07 (Lecture Notes in Computer Science, Vol. 4349)*, Byron Cook and Andreas Podelski (Eds.). Springer, 268–282. https://doi.org/10.1007/978-3-540-69738-1_20

Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. 2024. *SMT-LIB release 2024 (non-incremental benchmarks)*. https://doi.org/10.5281/zenodo.11061097

Francesco Ranzato. 2013. Complete Abstractions Everywhere. In *VMCAI 2013 (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 15–26. https://doi.org/10.1007/978-3-642-35873-9_3

John Regehr, Alastair Reid, and Kirk Webb. 2005. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.* 4, 4 (2005), 751–778. https://doi.org/10.1145/1113830.1113833

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1988)*. 12–27. https://doi.org/10.1145/73560.73562

Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1198–1223. https://doi.org/10.1145/3622840

Michael Schwarz, Simmo Saan, Helmut Seidl, Julian Erhard, and Vesal Vojdani. 2023. Clustered Relational Thread-Modular Abstract Interpretation with Local Traces. In *ETAPS '23 (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 28–58. https://doi.org/10.1007/978-3-031-30044-8_2

Michael Schwarz and Helmut Seidl. 2023. Octagons Revisited - Elegant Proofs and Simplified Algorithms. In *SAS 2023 (Lecture Notes in Computer Science, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 485–507. https://doi.org/10.1007/978-3-031-44245-2_21

Ying Sheng, Andres Nötzli, Andrew Reynolds, Yoni Zohar, David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Clark Barrett, and Cesare Tinelli. 2023. Reasoning About Vectors: Satisfiability Modulo a Theory of Sequences. *Journal of Automated Reasoning* 67, 3 (Sept. 2023), 32. https://doi.org/10.1007/s10817-023-09682-2

Robert E Shostak. 1984. Deciding combinations of theories. *Journal of the ACM (JACM)* 31, 1 (1984), 1–12. https://doi.org/10.1145/2422.322411

Axel Simon and Andy King. 2010. The two variable per inequality abstract domain. *High. Order Symb. Comput.* 23, 1 (2010), 87–143. https://doi.org/10.1007/s10990-010-9062-8

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2015. Making numerical program analysis fast. In *PLDI '15*, David Grove and Stephen M. Blackburn (Eds.). ACM, 303–313. https://doi.org/10.1145/2737924.2738000

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *POPL '17*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. https://doi.org/10.1145/3009837.3009885

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.* 2, POPL (2018), 55:1–55:28. https://doi.org/10.1145/3158143

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *POPL '96*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225. https://doi.org/10.1145/321879.321884

Robert Endre Tarjan. 1979. Applications of Path Compression on Balanced Trees. *J. ACM* 26, 4 (1979), 690–715. https://doi.org/10.1145/322154.322161

Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (1984), 245–281. https://doi.org/10.1145/62.2160

André Tavares, Benoit Boissinot, Fernando Pereira, and Fabrice Rastello. 2014. Parameterized Construction of Program Representations for Sparse Dataflow Analyses. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–39. https://doi.org/10.1007/978-3-642-54807-9_2

Arnaud Venet and Guillaume P. Brat. 2004. Precise and efficient static array bound checking for large embedded C programs. In *PLDI '04*. 231–242. https://doi.org/10.1145/996841.996869

Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 254–265. https://doi.org/10.1109/CGO53902.2022.9741267

Philip Zucker. 2022. https://www.philipzucker.com/union-find-groupoid/. Accessed 2025-03-10.

```
inter ∈ 𝕌𝕃 × 𝕌𝕃 → 𝕌𝕃
inter ((U₁, C₁), (U₂, C₂)) ≜
    let M ∈ ℕ × ℕ ⇀ List(ℕ × 𝕃 × 𝕃) = ∅ in   // Absent keys implicitly map to an empty list
    let C = C₁ ∩ C₂ combining non-equal values with n c₁ c₂ ↦
        M[n, n] ≔ [(n, id, id)]; return c₁ ∩ c₂                                  in
    let U = U₁ ∩ U₂ combining non-equal values with n (r₁, ℓ₁) (r₂, ℓ₂) ↦
        for (r, ℓ'₁, ℓ'₂) in M[r₁, r₂] do
            if ℓ₁ ⨟̂ ℓ'₁ = ℓ₂ ⨟̂ ℓ'₂ then                  // Same relation between n and r in 1 and 2
                if not on first iteration then C[r] ≔ C[r] ∪ {n};
                return (r, ℓ₁ ⨟̂ ℓ'₁)
            if on first iteration then C[r] ≔ C[r] \ {n};
        // No return occured, we iterate in ascending order so n is the lowest element of its class
        C[n] ≔ if M[r₁, r₂] = [] then C₁[r₁] ∩ C₂[r₂] else {n};
        M[r₁, r₂].append((n, inv (ℓ₁), inv (ℓ₂)));  return (n, id)                in
    return (U, C)
```

Fig. 9. Intersection on immutable labeled union-find (abstract join).

A Abstract Join: Immutable Labeled Union-Find Intersection

The union-find structure presented in Section 3 makes heavy use of mutability. This allows for excellent performance in union and find operations, but makes copying very costly. This is fine when representing flow insensitive relations (that hold at every point), but insufficient to represent flow-sensitive ones (local to a branch): in the latter case, we need several versions of the data structure that are updated from a common version, and we need to merge them. Merging implies that we need confluent persistence [Driscoll et al. 1994; Fiat and Kaplan 2003], so union-find implementations with a weaker notion of persistence [Conchon and Filliâtre 2007] do not suffice for our needs.

We thus need immutable maps to represent $\mathbb{U}$, which adds a moderate performance cost (`find` is now $O(\log n)$ instead of amortized $O(1)$). Specifically, we use fast mergeable maps [Okasaki and Gill 1998], as they allow for intersection in $O(\Delta \log n)$, where n is the size of the maps and Δ is their difference. As we join maps deriving from a common ancestor, we expect Δ to be small relative to n.

Theoretically, the most precise abstract join between two union-finds consists in saturating both; taking the intersection of the saturated graphs; and then returning to the union-find form by constraint elimination. Figure 9 presents an algorithm computing that join without having to saturate the graphs. In order to make use of fast intersection, which skips sub-maps consisting of elements with common representatives, it needs a union-find representation that only depends on its contents. Thus, we use collapsing union-find [Tarjan and van Leeuwen 1984], which performs eager path compression, and is reasonably efficient [Patwary et al. 2010]. Maintaining this in `add_relation` requires storing reverse maps from representatives to all their elements: $\mathbb{UC} \triangleq \mathbb{U} \times (\mathbb{N} \rightharpoonup \mathcal{P}(\mathbb{N}))$. Additionally, we require that representatives point to themselves; and that there is a total order on $\mathbb{N}$, with each representative being the smallest element of their connected component.

The algorithm identifies the classes of the intersection by a pair of classes (which we identify by their representatives) of the arguments. The labeled version, however, may split such a connected-component further since it only keeps edges with the same labels. Thus, we use a memoization map M that keeps, for each pair of connected components from the arguments, the list of new components in their intersection, along with the relations between the old representative to the new one. Note that it starts by assuming that the intersection of two connected components will

contain all their shared nodes (the first time a pair of components is encountered, the new C value is initialized to their intersection), but abandons this for further component splits (C only starts as a singleton).

Theorem A.1 (Inter correction and complexity). *For (U_1, C_1) and $(U_2, C_2) \in \mathbb{UC}$ satisfying the invariants, then $(U_i, C_i) \triangleq \texttt{inter}\,((U_1, C_1), (U_2, C_2))$ also satisfies these invariants and for all n, m, ℓ:*

$$\texttt{get\_relation}\,(U_1,\ n,\ m) = \ell = \texttt{get\_relation}\,(U_2,\ n,\ m) \quad\Leftrightarrow\quad \texttt{get\_relation}\,(U_i,\ n,\ m) = \ell$$

It runs in $O(\Delta^2 \log^2 n)$ where $n = |U_1| + |U_2|$ and Δ is the number of different bindings between them.

Proof. See proof in Appendix B.4 (page 39). □

The presented join only operates on the relational part, but it could easily be extended to support also having a factorized map of values as in Section 5.2.

B Proofs

B.1 Proofs: Union-Find with a Group of Edges Labels

Proof of Theorem 3.1: we start by proving a slightly different formulation of this theorem:

Lemma B.1. *Let $U \in \mathbb{U}$ be the union-find obtained by calling* `add_relation` *a number of times on* `init ()` *(with* conflict *being no-op). Then:*

1. `find` *terminates, does not change the saturated graph, and $(r, \ell) \triangleq \texttt{find}\,(U,\ n) \Rightarrow n \xrightarrow{\ell}\!\!\!\!\rightarrow_U r$;*
2. *the saturated graph of U satisfies HUniqueLabel and is included in the saturated graph of the edges passed to* `add_relation`*;*
3. *n and m are in the same connected component iff. they have the same representative;*
4. `get_relation`$\,(U,\ n,\ m) = \ell \neq \top$ *if and only if $n \xrightarrow{\ell}\!\!\!\!\rightarrow_U m$,*
5. `get_relation`$\,(U,\ n,\ m) = \top$ *if and only if n and m are in distinct components;*
6. conflict *is not called if and only if HUniqueLabel holds in the saturated graph of the edges passed to* `add_relation`*. In that case, both saturated graphs are equal.*

Proof.

(1) Start by proving that there are no looping chains in U (by induction on k):
- it is trivially true for the empty map
- for the recursive case: both `find` and `add_relation` can modify U. However, when they do, they set a node to point to another node which is returned by `find`. By definition of `find`, such a node cannot be in $\mathrm{dom}\,U$, it must point to nothing, so no chain is created.

This justifies that `find` terminates.
Now take $(r, \ell) \triangleq \texttt{find}\,(U,\ n)$, by induction on the chains in U, we can show that the induced graph hasn't changed and $n \xrightarrow{\ell}\!\!\!\!\rightarrow r$:
- If $n \notin \mathrm{dom}\,U$, U isn't changed at all and `find` returns $(n, \texttt{id})$ so PathRefl shows $n \xrightarrow{\ell}\!\!\!\!\rightarrow r$;
- If $U[n] = (m, \ell_m)$ and the property holds for m, we have $n \xrightarrow{\ell_m}\!\!\!\!\rightarrow m$ by EdgePath and $m \xrightarrow{\ell_r}\!\!\!\!\rightarrow r$ by induction hypothesis, so PathTrans yields $n \xrightarrow{\ell_m \hat{;} \ell_r}\!\!\!\!\rightarrow r$.
  $U[n]$ is then modified, however we can show that this doesn't change the induced graph. For one, the new saturated graph is included in the old one since we replaced $n \xrightarrow{\ell_m} m$ by $n \xrightarrow{\ell_m \hat{;} \ell_r} r$, which was already present in the saturated graph. For the other

inclusion, it suffices to show that, in the new graph, $n \xrightarrow{\ell_m} m$ holds. To do so simply apply PathTrans and PathSym to $m \xrightarrow{\ell_r} r$ and $n \xrightarrow{\ell_m \hat{\S} \ell_r} r$

Note that we know that $m \xrightarrow{\ell_r} r$ still holds because $m \neq n$ by the no-looping chains condition, and after the call to `find` $(U,\ m)$, either $m \notin \mathrm{dom}\, U$ (in which case $r = m$ and $\ell_r = \mathtt{id}$, so PathRefl gives the result) or $U[m] = (r, \ell_r)$ so EdgePath gives the result.

(2)-1 We already know that calling `find` does not change the graph induced by U, all we need to show is that the edges added by `add_relation` $(U,\ n,\ m,\ \ell)$ are implied by $n \xrightarrow{\ell} m$.

The added edges are either $r_n \xrightarrow{\mathtt{inv}(\ell_n) \hat{\S} \ell \hat{\S} \ell_m} r_m$ or $r_m \xrightarrow{\mathtt{inv}(\ell_m) \hat{\S} \mathtt{inv}(\ell) \hat{\S} \ell_n} r_n$. Since we have $n \xrightarrow{\ell_n} r_n$ and $m \xrightarrow{\ell_m} r_m$ by (1), both of these added edges can be built using PathTrans and PathSym (as seen in the graph of Figure 4).

Here we can also show that, as long as there are no conflict calls, then the graphs are equal since we can always reconstruct $n \xrightarrow{\ell} m$ from $n \xrightarrow{\ell_n} r_n$ and $m \xrightarrow{\ell_m} r_m$ and the added edge. This will be useful in (6).

We will prove that the graph of U satisfies HUniqueLabel later.

(3) Being in the same connected component is equivalent to being linked by an edge $n \xrightarrow{\ell} m$ in the saturated graph. Proceed by induction on that edge:

- EdgePath: this implies $U[n] = (m, \ell)$, which in turn would imply that their representatives match;
- PathRefl: this implies $n = m$, so their representatives match;
- PathSym: immediate by induction hypothesis;
- PathTrans: our edge decomposes as $n \xrightarrow{\ell_0} m' \xrightarrow{\ell_1} m$, by induction hypothesis, n and m' have same representative, and so do m' and m, therefore, so do n and m.

For the reciprocal, if two elements n, m have the same representative then by (1) they are both linked to that representative in U's saturated graph, which is a subset of the edge's graph by (2)

(4) $\Rightarrow$ Let $\ell =$ `get_relation` $(U,\ n,\ m)$, if $\ell \neq \top$ then by definition of `get_relation`, $\ell = \ell_n \mathbin{\hat{\S}} \mathtt{inv}\ (\ell_m)$ and n and m have same representative r. Furthermore, by (1) we have $n \xrightarrow{\ell_n} r$ and $m \xrightarrow{\ell_m} r$. We can conclude by applying PathSym and PathTrans.

(5) If $\ell = \top$ then, by definition of `get_relation`, n and m have different representatives. So by (3) they must also be in different connected components.

For the reciprocal, if n and m are in different connected components, then they have different representative by (3). Therefore, by definition of `get_relation`, `get_relation` $(U,\ n,\ m) = \top$

(6) If conflict is called, then HUniqueLabel does not hold. Indeed, when the call occurs, we can easily show that (similarly to (4)) we have $n \xrightarrow{\ell_n \hat{\S} \mathtt{inv}(\ell_m)} m$ in U, by (2), this is also true in the graph of the edges. Since the graph of the edges also contains the edge we are adding $n \xrightarrow{\ell} m$, and since $\ell \neq \ell_n \mathbin{\hat{\S}} \mathtt{inv}\ (\ell_m)$, then it does not satisfy HUniqueLabel.

Conversely, if HUniqueLabel does not hold, then conflict will be called. Suppose the first conflict comes at the k-th edge: take a list of edges $(n_1, \ell_1, m_1), \ldots (n_k, \ell_k, m_k)$ such that HUniqueLabel holds on the graph induced by $(n_1, \ell_1, m_1), \ldots (n_{k-1}, \ell_{k-1}, m_{k-1})$ but not the one induced by the full list.

We thus have two paths with different labels. One of these paths must include (n_k, ℓ_k, m_k). Using group operations on the edges, we can show that this implies there is another edge

from n_k to m_k with a different label ℓ that is already present in the graph induced by the first $k-1$ edges.
By the proof of (2), since conflict was not called when adding these first $k-1$ edges, the graph of U also contains that edge. Thus, n_k and m_k have the same representative by (3). Thus, when adding k-th edge, `add_relation` tests if the new edge ℓ_k matches the previous one. This test will succeed, we saw that $\ell_k \neq \ell$, and the edge built from the path to the representatives in `get_relation` will be equal to ℓ (since both is are edges of the graph of the $k-1$ first edges, which satisfy HUniqueLabel, and therefore unique).
To show the final point: we saw in the proof of (2) that when conflict is not called, both graphs are equal.

(2)-2 Since when a conflict occurs, U is not modified, we can remove the calls that lead to the conflict call without changing U. Doing so yields a smaller set of edges that triggers no calls and generates the same graph. By (4), that graph must satisfy HUniqueLabel is equal to the graph of U.

(4) $\Leftarrow$ Take $n \overset{\ell}{\twoheadrightarrow}_U m$ then by (3) they have the same representative. `get_relation` will return a non-$\top$ value, which must be equal to ℓ by (2). □

Theorem 3.1. *Let $U \in \mathbb{U}$ be the union-find obtained by successive calls to* `init`, `add_relation`, `find`, *and* `get_relation`. *If* conflict *does not modify U, then, two nodes n and m are related (i.e., there exists ℓ such that $n \overset{\ell}{\twoheadrightarrow} m$ in U) if and only if they belong to the same tree (whose root is the representative), and if and only if they would have been related in the graph whose edges are the arguments of the successive calls to* `add_relation` *which did not trigger* conflict. *Furthermore:*

(1) `find` *terminates, returns the representative r, and $(r, \ell) \triangleq \texttt{find}\,(U,\ n) \Rightarrow n \overset{\ell}{\twoheadrightarrow} r$;*
(2) `get_relation` $(U,\ n,\ m) = \ell \neq \top$ *if and only if $n \overset{\ell}{\twoheadrightarrow} m$;*
(3) `get_relation` $(U,\ n,\ m) = \top$ *if and only if n and m are not related;*
(4) conflict *is not called if and only if HUniqueLabel holds.*

Proof. All of these are easy consequences of Lemma B.1. □

Proof of Theorem 3.2:

Theorem 3.2. *Let $(U, I) \in \mathbb{U}\text{-}\mathbb{I}$ be the result of a sequence of calls to* `add_relation`$_{\mathbb{I}}$, `get_info`, `add_info` *after a first* `init`$_{\mathbb{I}}$ $()$, *with $(m_0, i_0) \ldots (m_k, i_k)$ the arguments passed to* `add_info`. *Then:*

(1) the domain of I is exactly the set of representatives of U (all accesses to I are correct)
(2) if $\mathcal{A}$ distributes over $\sqcap_{\mathbb{I}}$ (i.e., $\mathcal{A}\,(\ell,\ i \sqcap_{\mathbb{I}} j) = \mathcal{A}\,(\ell,\ i) \sqcap_{\mathbb{I}} \mathcal{A}\,(\ell,\ j)$), we have for all nodes n:

$$\texttt{get\_info}\,((U,I),\ n) = \prod_{0 \leq p \leq k,\ n \text{ and } m_p \text{ in the same relational class}}^{\mathbb{I}} \mathcal{A}\left(\texttt{get\_relation}\left(U,\ n,\ m_p\right),\ i_p\right)$$

Proof. We first show that:

- dom I is exactly the set of representatives of U
- for all $n \in \text{dom}\, I, I[n] = \prod^{\mathbb{I}}_{0 \leq p \leq k,\ n \text{ and } m_p \text{ in same connected comp}} \mathcal{A}\left(\texttt{get\_relation}\left(U,\ n,\ m_p\right),\ i_p\right)$

We proceed by induction on the list of functions (`add_relation`$_{\mathbb{I}}$ or `add_info`) applied to the empty union-find:

- The result is initially true: when no union is performed, all elements are their own representatives, so dom $[i \in \mathbb{I} \rightharpoonup \top_{\mathbb{I}}] = \mathbb{I}$ is indeed the set of all representatives.
Furthermore, as no information has been added, all meets are empty, thus equal to $\top_{\mathbb{I}}$.

- Suppose the result is true for (U, I), let us show it holds after $\texttt{add\_info}\,((U, I),\ n,\ i)$.
  By construction, $\texttt{add\_info}$ modifies I at r, where $(r, \ell) = \texttt{find}\,(U,\ n)$, which is indeed a representative, so $\operatorname{dom} I$ isn't changed.
  The result still holds for all representatives which are not r, since by Theorem 3.1, they are not in the same connected component as n, so both their value and the index-set of the $\sqcap_{\mathbb{I}}$ are unchanged.
  For r, the new $\sqcap_{\mathbb{I}}$ is equal to the meet of old one (which is equal to $I[r]$ by induction hypothesis) and $\mathcal{A}\,(\texttt{get\_relation}\,(U,\ r,\ n)\,,\ i)$. On the other hand, $I[r] := I[r] \sqcap_{\mathbb{I}} \mathcal{A}\,(\texttt{inv}\,(\ell)\,,\ i)$. Since $\texttt{find}\,(U,\ n) = (r, \ell)$, it is clear that $\texttt{get\_relation}\,(U,\ r,\ n) = \texttt{inv}\,(\ell)$, so the equality is indeed preserved.
- Suppose the result is true for (U, I), let us show it holds after $\texttt{add\_relation}_{\mathbb{I}}\,((U, I),\ n,\ m,\ \ell)$.
  If n and m are in the same class, then by Theorem 3.1, they have the same representative, so $\texttt{add\_relation}_{\mathbb{I}}$ does not change (U, I).
  If they are not, then their classes are merged, choosing a new representative randomly between r_n and r_m. Both cases are symmetric, so we only show the case where r_n is updated to point to r_m. In that case the new $\operatorname{dom} I$ is equal to the old minus r_n (since by induction hypothesis, both r_n and r_m where in $\operatorname{dom} I$). Thus, it is still exactly the set of all representatives.
  $I[r]$ has not been changed for all representatives not in the connected component of n and m, so the equality still holds there. For $I[r_n]$, the new value is $I[r_m] := I[r_m] \sqcap_{\mathbb{I}} \mathcal{A}\,(\texttt{inv}\,(\texttt{inv}\,(\ell_n) \fatsemi \ell \fatsemi \ell_m)\,,\ I[r_n])$. The big-meet also has new elements, all of those that come from the connected component of m:

$$
\begin{aligned}
I[r_m] &= I[r_m] \sqcap_{\mathbb{I}} \mathcal{A}\,(\texttt{inv}\,(\texttt{inv}\,(\ell_n) \fatsemi \ell \fatsemi \ell_m)\,,\ I[r_n]) \\
&= \bigsqcap_{\substack{0 \le p \le k,\ r_m \text{ and } m_p \\ \text{were in same connected comp}}} \mathcal{A}\,(\texttt{get\_relation}\,(U,\ r_m,\ m_p)\,,\ i_p) \\
&\quad \sqcap_{\mathbb{I}}\ \mathcal{A}\left(\texttt{inv}\,(\texttt{inv}\,(\ell_n) \fatsemi \ell \fatsemi \ell_m)\,,\ \bigsqcap_{\substack{0 \le p \le k,\ r_n \text{ and } m_p \\ \text{were in same connected comp}}} \mathcal{A}\,(\texttt{get\_relation}\,(U,\ r_n,\ m_p)\,,\ i_p)\right)
\end{aligned}
$$

By distributivity of meet/action and HActionCompose :

$$
\begin{aligned}
&= \bigsqcap_{\substack{0 \le p \le k,\ r_m \text{ and } m_p \\ \text{were in same connected comp}}} \mathcal{A}\,(\texttt{get\_relation}\,(U,\ r_m,\ m_p)\,,\ i_p) \\
&\quad \sqcap_{\mathbb{I}} \bigsqcap_{\substack{0 \le p \le k,\ r_n \text{ and } m_p \\ \text{were in same connected comp}}} \mathcal{A}\,(\texttt{inv}\,(\texttt{inv}\,(\ell_n) \fatsemi \ell \fatsemi \ell_m) \fatsemi \texttt{get\_relation}\,(U,\ r_n,\ m_p)\,,\ i_p)
\end{aligned}
$$

Since $U[r_n]$ is now $(r_m, \texttt{inv}\,(\ell_n) \fatsemi \ell \fatsemi \ell_m)$:

$$
\begin{aligned}
&= \bigsqcap_{\substack{0 \le p \le k,\ r_m \text{ and } m_p \\ \text{were in same connected comp}}} \mathcal{A}\,(\texttt{get\_relation}\,(U,\ r_m,\ m_p)\,,\ i_p) \\
&\quad \sqcap_{\mathbb{I}} \bigsqcap_{\substack{0 \le p \le k,\ r_n \text{ and } m_p \\ \text{were in same connected comp}}} \mathcal{A}\,(\texttt{get\_relation}\,(U,\ r_m,\ m_p)\,,\ i_p)
\end{aligned}
$$

The new connected comp of r_m is the disjoint union of the old comps of r_m and r_n :

$$
\begin{aligned}
&= \bigsqcap_{\substack{0 \le p \le k,\ r_m \text{ and } m_p \\ \text{in same connected comp}}} \mathcal{A}\,(\texttt{get\_relation}\,(U,\ r_m,\ m_p)\,,\ i_p)
\end{aligned}
$$

Finally, the full result on `get_info` is a direct consequence of the result on the representatives $I[r]$ combined with distributivity of the action and meet:

$$
\begin{aligned}
&\texttt{get\_info}\,(U,\ n)\\
&\quad= \mathcal{A}\,(\ell,\ I[r]) \quad \text{where } (r,\ell) = \texttt{find}\,(U,\ n)\\
&\quad= \mathcal{A}\left(\ell,\ \bigsqcap_{\substack{0\le p\le k,\ r \text{ and } m_p \text{ in same connected comp}}} \mathcal{A}\left(\texttt{get\_relation}\left(U,\ r,\ m_p\right),\ i_p\right)\right) \quad \text{by the above}\\
&\quad= \bigsqcap_{\substack{0\le p\le k,\ r \text{ and } m_p \text{ in same connected comp}}} \mathcal{A}\left(\ell,\ \mathcal{A}\left(\texttt{get\_relation}\left(U,\ r,\ m_p\right),\ i_p\right)\right) \quad \text{meet/action distribute}\\
&\quad= \bigsqcap_{\substack{0\le p\le k,\ r \text{ and } m_p \text{ in same connected comp}}} \mathcal{A}\left(\ell \fatsemi \texttt{get\_relation}\left(U,\ r,\ m_p\right),\ i_p\right) \quad \text{by HActionCompose}\\
&\qquad \text{since } n \text{ and } r \text{ have same connected component, whose repr is } r:\\
&\quad= \bigsqcap_{\substack{0\le p\le k,\ n \text{ and } m_p \text{ in same connected comp}}} \mathcal{A}\left(\texttt{get\_relation}\left(U,\ n,\ m_p\right),\ i_p\right) \qquad \square
\end{aligned}
$$

B.2 Proofs: Union-Find Labeled with Abstract Relations

Proof of Lemma 4.1:

Lemma 4.1. $\mathrm{inv}^\sharp$ *is exact:* $\forall\, \mathcal{R}^\sharp \in \mathbb{R}^\sharp,\ \gamma_{\mathbb{R}^\sharp}(\mathrm{inv}^\sharp(\mathcal{R}^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)^{-1}$.

Proof. Group theory says $\mathrm{inv}^\sharp$ is involutive: $\mathrm{inv}^\sharp \circ \mathrm{inv}^\sharp$ is identity. Proceed by double inclusion:

- $\supseteq$ directly by HInverseSound;
- $\subseteq$ $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathrm{inv}^\sharp(\mathrm{inv}^\sharp(\mathcal{R}^\sharp))) \supseteq \gamma_{\mathbb{R}^\sharp}(\mathrm{inv}^\sharp(\mathcal{R}^\sharp))^{-1}$ by HInverseSound. Thus, we have $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)^{-1} \supseteq \gamma_{\mathbb{R}^\sharp}(\mathrm{inv}^\sharp(\mathcal{R}^\sharp))$ by monotony and involution of $\bullet^{-1}$. $\square$

Proof of Lemma 4.2:

Lemma 4.2. $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) \in \mathcal{P}(\mathbb{V} \times \mathbb{V})$ *is an equivalence relation between values.*

Proof.

- Reflexivity: directly by HIdentitySound;
- Symmetry: since $\mathrm{inv}^\sharp(\mathtt{id}^\sharp) = \mathtt{id}^\sharp$, Lemma 4.1 implies $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)^{-1}$;
- Transitivity: $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp \fatsemi^\sharp \mathtt{id}^\sharp) \supseteq \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ by HComposeSound. $\square$

Proof of Theorem 4.3:

Theorem 4.3. *An abstract relation* $\mathcal{R}^\sharp \in \mathbb{R}^\sharp$ *represents a concrete injective partial function* $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ *between the equivalence classes of the* $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ *relation.*

Proof. Take $\mathcal{R}^\sharp$ an abstract relation, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ is compatible with the classes of $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$:

- for all $x,\ y,\ z$ such that $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ and $(z,x) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$, we can show that $(z,y) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp \fatsemi^\sharp \mathcal{R}^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ by symmetry of $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ (Lemma 4.2) and HComposeSound;
- similarly, $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ and $(y,z) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ implies $(x,z) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$.

Furthermore, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ is injective and functional on those classes:

- for all $(x,z) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ and $(y,z) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$, we have $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)^{-1}$, so by HInverseSound, monotony of $\fatsemi$ and HComposeSound, $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp \fatsemi^\sharp \mathrm{inv}^\sharp(\mathcal{R}^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$;
- similarly, for all $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$ and $(x,z) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$, we have $(y,z) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$. □

Proof of Theorem 4.5:

Theorem 4.5. *The following propositions are equivalent:*

(1) $\gamma_{\mathbb{R}^\sharp}$ *is a group morphism between* $\langle \mathbb{R}^\sharp, \fatsemi^\sharp, \mathrm{inv}^\sharp, \mathrm{id}^\sharp \rangle$ *and the group of relations between equivalence classes* $\langle \mathcal{P}\left(\mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp) \times \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)\right), \fatsemi, \bullet^{-1}, \{(\overline{v},\overline{v}) \mid \overline{v} \in (\mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp))\}\rangle$;
(2) $\fatsemi^\sharp$ *is exact:* $\forall\, \mathcal{R}_1^\sharp\ \mathcal{R}_2^\sharp \in \mathbb{R}^\sharp,\ \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$;
(3) for all $\mathcal{R}^\sharp$, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$ *is a total function;*
(4) for all $\mathcal{R}^\sharp$, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$ *is surjective;*
(5) for all $\mathcal{R}^\sharp$, $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \in \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp) \rightharpoonup \mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$ *is bijective.*

Furthermore, if these hold then the lattice of abstract relations is flat: for all $\mathcal{R}_1^\sharp$ *and* $\mathcal{R}_2^\sharp$ *such that* $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$, *we have* $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$.

Proof.

1 ⇒ 2: Since $\gamma_{\mathbb{R}^\sharp}$ is a group morphism (1), we have $\forall\, \mathcal{R}_1^\sharp\ \mathcal{R}_2^\sharp \in \mathbb{R}^\sharp,\ \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp) = \pi\big(\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp)\big) \fatsemi \pi\big(\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)\big)$ where π is the canonical projection from $\mathbb{V}$ to $\mathbb{V}/\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$ lifted to relations. Conclude using Theorem 4.3 which proved that $\pi\big(\gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\big) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$:

2 ⇒ 3: let $\mathcal{R}^\sharp \in \mathbb{R}^\sharp$, for $x \in \mathbb{V}$, by Lemma 4.2 we know that $(x,x) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp) = \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp \fatsemi^\sharp \mathrm{inv}^\sharp(\mathcal{R}^\sharp))$ since $\fatsemi^\sharp$ is exact (2), we have $(x,x) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathrm{inv}^\sharp(\mathcal{R}^\sharp))$. By definition of $\fatsemi$, there exists y such that $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)$;

3 ⇔ 4: $\gamma(\mathcal{R}^\sharp)^{-1} = \gamma(\mathrm{inv}^\sharp(\mathcal{R}^\sharp))$ by Lemma 4.1. If (3) holds then $\gamma(\mathrm{inv}^\sharp(\mathcal{R}^\sharp))$ is a total function, so its inverse must be subjective. Symmetrically, if (4) holds then the function is subjective, so its inverse is total;

3,4 ⇒ 5: $\gamma(\mathcal{R}^\sharp)$ is an injective partial function by Theorem 4.3, which is also total (3) and surjective (4). Thus, it is bijective;

5 ⇒ 1 We already know $\gamma_{\mathbb{R}^\sharp}$ preserves $\mathrm{inv}^\sharp$ (Lemma 4.1). It also preserves $\mathrm{id}^\sharp$ since we reason on equivalence classes. All that remains is to show it preserves composition $\fatsemi^\sharp$ (i.e. that (2) holds).
For $\mathcal{R}_1^\sharp\ \mathcal{R}_2^\sharp \in \mathbb{R}^\sharp$, HComposeSound gives $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp) \supseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$.
For the reverse, take $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp)$. Since $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp)$ is bijective (5) it is total, so there exists z such that $(x,z) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp)$. Similarly, there exists t such that $(z,t) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$. $(x,t) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp)$, so by bijectivity of $\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp$, $(t,y) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$. Thus $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$ modulo $\gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$.

For the final property, it suffices to prove the reverse inclusion. Take $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$, Since $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp)$ is total (3), there exists z such that $(x,z) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$. Since $\gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)$ is functional between equivalence classes (Theorem 4.3), $(y,z) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{id}^\sharp)$ and so, using Theorem 4.3 again gives $(x,y) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp)$ □

B.3 Proofs: Reduced Product with Non-Relational Domains

Proof of Theorem 5.2:

Theorem 5.2. *If* `refine` *is exact then, for all states* $(U,M) \in \mathbb{U} \times (\mathbb{X} \to \mathbb{V}^\sharp)$ *on which we performed constraint propagation, and for all edges* $x \xrightarrow{\mathcal{R}^\sharp} y$ *in the saturated graph of* U, *if we let* $(v_1^\sharp, v_2^\sharp) \triangleq \mathtt{refine}(\mathcal{R}^\sharp, M[x], M[y])$, *we have* $\gamma_{\mathbb{V}^\sharp}(M[x]) \subseteq \gamma_{\mathbb{V}^\sharp}(v_1^\sharp)$ *and* $\gamma_{\mathbb{V}^\sharp}(M[y]) \subseteq \gamma_{\mathbb{V}^\sharp}(v_2^\sharp)$.

Proof. Proceed by structural induction on $x \xrightarrow{\mathcal{R}^\sharp} y$

- EdgePath: this edge is used for constraint propagation, so it has stabilized when the fixpoint is reached;
- PathRefl: In that case $x = y$ and $\mathcal{R}^\sharp = \mathtt{id}^\sharp$, let $(v_1^\sharp, v_2^\sharp) \triangleq \mathtt{refine}(\mathtt{id}^\sharp, M[x], M[x])$, by HRefineSound we have:

$$\begin{array}{lll} \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(M[x]), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) \right\} \\ & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; (v_1, v_1) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) \right\} \quad \text{choosing } v_2 = v_1 \\ & \supseteq & \gamma_{\mathbb{V}^\sharp}(M[x]) \quad \text{since } (v_1, v_1) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp) \text{ by HIdentitySound} \end{array}$$

  and similarly for $v_2^\sharp$;
- PathSym: let $(v_1^\sharp, v_2^\sharp) = \mathtt{refine}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp), M[x], M[y])$, by HRefineSound, we have:

$$\begin{array}{ll} & \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \supseteq \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp)) \right\} \\ \wedge & \gamma_{\mathbb{V}^\sharp}(v_2^\sharp) \supseteq \left\{ v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]) \;\middle|\; \exists\, v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp)) \right\} \\ & \\ & \text{so, by } \textit{Lemma 4.1} \\ \Rightarrow & \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \supseteq \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), (v_2, v_1) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \right\} \\ \wedge & \gamma_{\mathbb{V}^\sharp}(v_2^\sharp) \supseteq \left\{ v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]) \;\middle|\; \exists\, v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]), (v_2, v_1) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp) \right\} \\ & \\ \Rightarrow & \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \supseteq \gamma_{\mathbb{V}^\sharp}(v_1^{\sharp\prime}) \wedge \gamma_{\mathbb{V}^\sharp}(v_2^\sharp) \supseteq \gamma_{\mathbb{V}^\sharp}(v_2^{\sharp\prime}) \text{ by HRefineSound and exact} \\ & \text{where } (v_2^{\sharp\prime}, v_1^{\sharp\prime}) = \mathtt{refine}(\mathcal{R}^\sharp, M[y], M[x]) \end{array}$$

  Finally, the induction hypothesis states $\gamma_{\mathbb{V}^\sharp}(M[y]) \subseteq \gamma_{\mathbb{V}^\sharp}(v_2^{\sharp\prime})$ and $\gamma_{\mathbb{V}^\sharp}(M[x]) \subseteq \gamma_{\mathbb{V}^\sharp}(v_1^{\sharp\prime})$;
- PathTrans: $x \xrightarrow{\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp} y$ is built from $x \xrightarrow{\mathcal{R}_1^\sharp} z$ and $z \xrightarrow{\mathcal{R}_2^\sharp} y$.
  Let $(v_1^\sharp, v_2^\sharp) = \mathtt{refine}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp, M[x], M[y])$. We will show the result for $v_1^\sharp$, as the result for $v_2^\sharp$ can be obtained in symmetrically:

$$\begin{array}{lll} \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) & = & \gamma_{\mathbb{V}^\sharp}(\mathrm{fst}(\mathtt{refine}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp, M[x], M[y]))) \\ & & \\ & & \text{by HRefineSound} \\ & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp) \right\} \\ & & \\ & & \text{applying HComposeSound gives:} \\ & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \right\} \\ & & \\ & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), \exists v_3, (v_1, v_3) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \wedge (v_3, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \right\} \\ & & \\ & & \text{since } \exists v_3, \phi \Rightarrow \exists v_3 \in \gamma_{\mathbb{V}^\sharp}(M[z]), \phi \\ & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \begin{array}{l} \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), \exists v_3 \in \gamma_{\mathbb{V}^\sharp}(M[z]), \\ (v_1, v_3) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \wedge (v_3, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \end{array} \right\} \\ & & \\ & & \text{The induction hypothesis on } z \xrightarrow{\mathcal{R}_2^\sharp} y \text{ and exactness of } \mathtt{refine} \text{ gives :} \\ & & \qquad \gamma_{\mathbb{V}^\sharp}(M[z]) \subseteq \left\{ v \in \gamma_{\mathbb{V}^\sharp}(M[z]) \;\middle|\; \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(M[y]), (v, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp) \right\} \\ & & \text{so we can remove the } \exists v_2 : \\ & \supseteq & \left\{ v_1 \in \gamma_{\mathbb{V}^\sharp}(M[x]) \;\middle|\; \exists v_3 \in \gamma_{\mathbb{V}^\sharp}(M[z]), (v_1, v_3) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \right\} \\ & \supseteq & \gamma_{\mathbb{V}^\sharp}(\mathrm{fst}(\mathtt{refine}(\mathcal{R}_1^\sharp,\ M[x],\ M[z]))) \quad \text{by exactness of } \mathtt{refine} \\ & \supseteq & \gamma_{\mathbb{V}^\sharp}(M[x]) \quad \text{by induction hypothesis} \end{array}$$

□

Proof of Lemma 5.3

Lemma 5.3. *If* `refine` *satisfies HRefineSound, then* $\mathcal{A}(\mathcal{R}^\sharp, v^\sharp) \triangleq \mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, \top_{\mathbb{V}^\sharp}, v^\sharp))$ *satisfies HActionSound. If* `refine` *is exact (HRefineSound with* $\supseteq$ *replaced by* $=$*), then so is* $\mathcal{A}$.

Proof. If `refine` is sound, then we have:

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathcal{R}^\sharp, v^\sharp)) &= \gamma_{\mathbb{V}^\sharp}(\mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, \top_{\mathbb{V}^\sharp}, v^\sharp))) && \text{by definition} \\
&\supseteq \left\{v_1 \in \gamma_{\mathbb{V}^\sharp}(\top_{\mathbb{V}^\sharp}) \;\middle|\; \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\} && \text{by HRefineSound} \\
&= \left\{v_1 \;\middle|\; \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\} && \text{since } \gamma_{\mathbb{V}^\sharp}(\top_{\mathbb{V}^\sharp}) = \mathbb{V}
\end{aligned}$$

Furthermore, in the exact case, $\supseteq$ is an equality. □

Proof of Lemma 5.4: Since soundness and exactness hypothesis only express inequalities in the concrete, we will have to reason under both $\gamma_{\mathbb{V}^\sharp}$ and $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$. To simplify notation we write:

- $\equiv \in \mathcal{P}(\mathbb{V} \times \mathbb{V})$ for equality modulo $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$: $v_0 \equiv v_1 \triangleq (v_0, v_1) \in \gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$.
- $\subseteqq \in \mathcal{P}(\mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}))$ for set inclusion modulo $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$: $V \subseteqq V' \triangleq \forall v \in V, \exists v' \in V', v \equiv v'$.
- $\eqqcolon \in \mathcal{P}(\mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}))$ for set equality modulo $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$, defined by double-inclusion.

Note that when $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ is equality, these definitions coincide with normal equality and subset relations. In the general case however, they are weaker than equality/subset. We will now restate our hypotheses using these weaker relations:

$$\gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}_1^\sharp \fatsemi \mathcal{R}_2^\sharp, v^\sharp\right)\right) \eqqcolon \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}_1^\sharp, \mathcal{A}\left(\mathcal{R}_2^\sharp, v^\sharp\right)\right)\right) \qquad \text{(HActionCompose2)}$$

$$\gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\texttt{id}^\sharp, v^\sharp\right)\right) \eqqcolon \gamma_{\mathbb{V}^\sharp}(v^\sharp) \qquad \text{(HActionIdentity2)}$$

$$\gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathcal{R}^\sharp, v^\sharp)) \subseteqq \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} \qquad \text{(HActionSound2)}$$

$$\gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathcal{R}^\sharp, v^\sharp)) \eqqcolon \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} \qquad \text{(HActionCompl2)}$$

$$\gamma_{\mathbb{V}^\sharp}(\mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, v_1^\sharp, v_2^\sharp))) \subseteqq \left\{v_1 \in \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \;\middle|\; \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(v_2^\sharp), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\} \qquad \text{(HRefineSound2)}$$

$$\gamma_{\mathbb{V}^\sharp}(\mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, v_1^\sharp, v_2^\sharp))) \eqqcolon \left\{v_1 \in \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \;\middle|\; \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(v_2^\sharp), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\} \qquad \text{(HRefineCompl2)}$$

These hypotheses are weaker than the ones stated using equality/subset. Meaning that, for example, HActionSound implies HActionSound2. When $\gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)$ is equality however, they are equivalent.

For instance, we can re-formulate Lemma 5.3 with the weaker premises. Note that it also has weaker conclusions, meaning this result is not implied by Lemma 5.3.

Lemma B.2. *If* `refine` *satisfies HRefineSound2 then* $\mathcal{A}(\mathcal{R}^\sharp, v^\sharp) \triangleq \mathrm{fst}(\texttt{refine}(\mathcal{R}^\sharp, \top_{\mathbb{V}^\sharp}, v^\sharp))$ *satisfies HActionSound2. Furthermore, if* `refine` *is satisfies HRefineCompl2, then* $\mathcal{A}$ *satisfies HActionCompl2.*

Proof. Same proof as in Lemma 5.3, swapping out the hypotheses for their weaker variants. □

Lemma B.3. *If* $\mathcal{A}$ *satisfies HActionCompl2 and any of the points of Theorem 4.5 hold, then it satisfies HActionCompose2 and HActionIdentity2.*

Proof. Let us start by proving HActionIdentity2:

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}(\mathcal{A}\left(\texttt{id}^\sharp, v^\sharp\right)) &\eqqcolon \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\texttt{id}^\sharp)\} && \text{by HActionCompl2} \\
&\subseteqq \gamma_{\mathbb{V}^\sharp}(v^\sharp) && \text{by definition of } \subseteqq
\end{aligned}$$

The other inclusion comes from HIdentitySound:

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}(v^\sharp) &= \{v_1 \mid v_1 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)\} \\
&\subseteq \{v_1 \mid v_1 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp) \wedge (v_1, v_1) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)\} && \text{by HIdentitySound} \\
&\subseteq \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)\} \\
&\subseteq\!\!\subseteq \gamma_{\mathbb{V}^\sharp}(\mathcal{A}\,(\mathtt{id}^\sharp,\ v^\sharp)) && \text{by HActionCompl2}
\end{aligned}$$

We can now work on HActionCompose2:

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}(\mathcal{A}\,(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp,\ v^\sharp)) &= \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp \fatsemi^\sharp \mathcal{R}_2^\sharp)\} && \text{by HActionCompl2} \\
&= \{v_1 \mid \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \fatsemi \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)\} && \text{by } \textit{Theorem } 4.5\ (2) \\
&= \{v_1 \mid \exists\, v_3, \exists\, v_2 \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v_1, v_3) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp) \wedge (v_3, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_2^\sharp)\} \\
&= \{v_1 \mid \exists\, v_3 \in \gamma_{\mathbb{V}^\sharp}(\mathcal{A}\,(\mathcal{R}_2^\sharp,\ v^\sharp)), (v_1, v_3) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}_1^\sharp)\} && \text{by HActionCompl2} \\
&= \gamma_{\mathbb{V}^\sharp}(\mathcal{A}\,(\mathcal{R}_1^\sharp,\ \gamma_{\mathbb{V}^\sharp}(\mathcal{A}\,(\mathcal{R}_2^\sharp,\ v^\sharp)))) && \text{by HActionCompl2}
\end{aligned}$$

□

Lemma B.4. *If $\mathcal{A}$ satisfies HActionCompose2 and HActionIdentity2, and any of the points of Theorem 4.5 hold, then it satisfies HActionCompl2.*

Proof.

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}(v^\sharp) &= \gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathtt{id}^\sharp,\ v^\sharp)) && \text{by HActionIdentity2} \\
&= \gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp) \fatsemi^\sharp \mathcal{R}^\sharp,\ v^\sharp)) \\
&= \gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp),\ \mathcal{A}(\mathcal{R}^\sharp,\ v^\sharp))) && \text{by HActionCompose2} \\
&\supseteq \{v \mid \exists\, v' \in \gamma_{\mathbb{V}^\sharp}(\mathcal{A}(\mathcal{R}^\sharp,\ v^\sharp)),\ (v, v') \in \gamma_{\mathbb{R}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp))\} && \text{by HActionSound2} \\
&\supseteq \{v \mid \exists v',\ \exists\, v'' \in \gamma_{\mathbb{V}^\sharp}(v^\sharp),\ (v, v') \in \gamma_{\mathbb{R}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp)) \wedge (v', v'') \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} && \text{by HActionSound2} \\
&\supseteq \{v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp) \mid \exists v',\ (v, v') \in \gamma_{\mathbb{R}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp)) \wedge (v', v) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} && \text{by choosing } v'' = v \\
&= \{v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp) \mid (v, v) \in \gamma_{\mathbb{V}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp)) \fatsemi \gamma_{\mathbb{V}^\sharp}(\mathcal{R}^\sharp)\} \\
&= \{v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp) \mid (v, v) \in \gamma_{\mathbb{V}^\sharp}(\mathtt{inv}^\sharp(\mathcal{R}^\sharp) \fatsemi^\sharp \mathcal{R}^\sharp)\} && \text{by } \textit{Theorem } 4.5\ (2) \\
&\supseteq \{v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp) \mid (v, v) \in \{(v, v) \mid v \in \mathbb{V}\}\} && \text{by HIdentitySound} \\
&= \gamma_{\mathbb{V}^\sharp}(v^\sharp)
\end{aligned}$$

Therefore, every $\supseteq$ in the chain is really an equivalence. In particular, the second usage of HActionSound is an equivalence, which implies the desired exactness equality. □

Lemma 5.4. *Assume any of the points of Theorem 4.5 hold. If $\mathcal{A}$ satisfies HActionSound then $\mathcal{A}$ is exact (we can replace $\supseteq$ by $=$ in HActionSound) if and only if it is a group action.*

Proof. It is a simplification of the statements of Lemma B.3 and Lemma B.4 when $\gamma_{\mathbb{V}^\sharp}$ is injective and $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ is equality. Note that Lemma B.4 also holds with the stronger hypotheses, and (proved in exactly the same way), so the assumption are only really needed for the direct implication (any sound $\mathcal{A}$ is an action). □

Proof of Theorem 5.6: We start by stating the general form of this theorem, when the assumption on $\gamma_{\mathbb{V}^\sharp}$ and $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ do not hold:

Lemma B.5. *If $\mathcal{A}$ and $\sqcap_{\mathbb{V}^\sharp}$ are both sound and exact then:*

$$\forall\, v^\sharp\ \mathcal{R}_1^\sharp\ \mathcal{R}_2^\sharp,\ \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp \sqcap_{\mathbb{V}^\sharp} v_2^\sharp\right)\right) = \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp\right) \sqcap_{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}^\sharp,\ v_2^\sharp\right)\right)$$

Thus, similarly to Theorem 3.2, for a union-find $(U, I) \in \mathbb{U}\text{-}\mathbb{I}$ whose known abstract values passed to `add_info` *where $(y_0, v_0^\sharp), \ldots, (y_k, v_k^\sharp)$, we have:*

$$\gamma_{\mathbb{V}^\sharp}(\texttt{get\_info}\,((U,I),\ x)) = \gamma_{\mathbb{V}^\sharp}\left(\prod_{x \text{ and } y_p \text{ related by } \mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right)$$

Therefore, in that case, using map factorization is just as precise as performing constraint propagation.

Proof. Let $\mathcal{R}^\sharp \in \mathbb{R}^\sharp$, $v_1^\sharp, v_2^\sharp \in \mathbb{V}^\sharp$ and $v \in \mathbb{V}$, then:

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp \sqcap_{\mathbb{V}^\sharp} v_2^\sharp\right)\right) &= \{v \mid \exists\, v' \in \gamma_{\mathbb{V}^\sharp}(v_1^\sharp \sqcap_{\mathbb{V}^\sharp} v_2^\sharp), (v, v') \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} && \mathcal{A} \text{ sound and exact} \\
&= \{v \mid \exists\, v' \in \gamma_{\mathbb{V}^\sharp}(v_1^\sharp) \cap \gamma_{\mathbb{V}^\sharp}(v_2^\sharp), (v, v') \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\} && \sqcap_{\mathbb{V}^\sharp} \text{ sound and exact} \\
&= \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp\right)\right) \cap \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ v_2^\sharp\right)\right) && \mathcal{A} \text{ sound and exact} \\
&= \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp\right) \sqcap_{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}^\sharp,\ v_2^\sharp\right)\right) && \sqcap_{\mathbb{V}^\sharp} \text{ sound and exact}
\end{aligned}$$

Next, we can prove the fact that the value of a variable has the same concretization as the meet of all its related values in the same way as in Theorem 3.2. While we do not have full action-meet commutation here, we have it under $\gamma_{\mathbb{V}^\sharp}$, which is sufficient to repeat the proof.

To show the fact that we are at least as precise, it suffices to show that for a given application of `refine` with $(v_x^\sharp, v_y^\sharp) \triangleq \texttt{refine}(\mathcal{R}^\sharp, \sigma(x), \sigma(y))$, if $\gamma_{\mathbb{V}^\sharp}(\sigma(x)) \subseteq \gamma_{\mathbb{V}^\sharp}(\texttt{get\_info}\,((U,I),\ x))$ and $\gamma_{\mathbb{V}^\sharp}(\sigma(y)) \subseteq \gamma_{\mathbb{V}^\sharp}(\texttt{get\_info}\,((U,I),\ y))$, then we also have $\gamma_{\mathbb{V}^\sharp}(v_x^\sharp) \subseteq \gamma_{\mathbb{V}^\sharp}(\texttt{get\_info}\,((U,I),\ x))$ and $\gamma_{\mathbb{V}^\sharp}(v_y^\sharp) \subseteq \gamma_{\mathbb{V}^\sharp}(\texttt{get\_info}\,((U,I),\ y))$. Since those inequalities hold at the start (where all abstract values are top), then by induction, they will never cease to hold.

By symmetry, we only have to show the result on x:

$$\begin{aligned}
\gamma_{\mathbb{V}^\sharp}(v_x^\sharp) &\supseteq \left\{v_1 \in \gamma_{\mathbb{V}^\sharp}(\sigma(x)) \,\middle|\, \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(\sigma(y)), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\} \quad \text{by HRefineSound2} \\
&\supseteq \left\{v_1 \in \gamma_{\mathbb{V}^\sharp}(\texttt{get_info}\,((U,I),\ x)) \,\middle|\, \exists v_2 \in \gamma_{\mathbb{V}^\sharp}(\texttt{get_info}\,((U,I),\ y)), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\}
\end{aligned}$$

therefore, by the equality on $\gamma_{\mathbb{V}^\sharp}(\texttt{get\_info}\,((U,I),\ x))$:

$$= \left\{v_1 \in \gamma_{\mathbb{V}^\sharp}\left(\prod_{x,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right) \,\middle|\, \exists v_2 \in \gamma_{\mathbb{V}^\sharp}\left(\prod_{y,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right), (v_1, v_2) \in \gamma_{\mathbb{R}^\sharp}(\mathcal{R}^\sharp)\right\}$$

$$= \gamma_{\mathbb{V}^\sharp}\left(\prod_{x,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right) \cap \gamma_{\mathbb{V}^\sharp}\left(\mathcal{A}\left(\mathcal{R}^\sharp,\ \prod_{y,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right)\right) \text{ by HActionCompl2}$$

$$= \gamma_{\mathbb{V}^\sharp}\left(\prod_{x,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right) \cap \gamma_{\mathbb{V}^\sharp}\left(\prod_{y,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}^\sharp \fatsemi \mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right) \text{ by previous result}$$

since $x \xrightarrow{\mathcal{R}^\sharp} y$ we can relabel the second meet :

$$= \gamma_{\mathbb{V}^\sharp}\left(\prod_{x,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right) \cap \gamma_{\mathbb{V}^\sharp}\left(\prod_{x,y_p,\mathcal{R}_p^\sharp}^{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}_p^\sharp,\ v_p^\sharp\right)\right) \text{ by previous result} \qquad \square$$

Theorem 5.6. *If $\mathcal{A}$ and $\sqcap_{\mathbb{V}^\sharp}$ are both sound and exact then:*

$$\forall\, v_1^\sharp\ v_2^\sharp\ \mathcal{R}^\sharp,\ \mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp \sqcap_{\mathbb{V}^\sharp} v_2^\sharp\right) = \mathcal{A}\left(\mathcal{R}^\sharp,\ v_1^\sharp\right) \sqcap_{\mathbb{V}^\sharp} \mathcal{A}\left(\mathcal{R}^\sharp,\ v_2^\sharp\right)$$

Proof. It is a simplification of Lemma B.5 when $\gamma_{\mathbb{V}^\sharp}$ is injective and $\gamma_{\mathbb{R}^\sharp}(\mathtt{id}^\sharp)$ is equality. □

B.4 Proofs: Abstract Join: Immutable Labeled Union-Find Intersection

Proof of Theorem A.1:

Theorem A.1 (Inter correction and complexity). *For (U_1, C_1) and $(U_2, C_2) \in \mathbb{UC}$ satisfying the invariants, then $(U_i, C_i) \triangleq \mathtt{inter}\,((U_1, C_1), (U_2, C_2))$ also satisfies these invariants and for all n, m, ℓ:*

$$\mathtt{get\_relation}\,(U_1,\ n,\ m) = \ell = \mathtt{get\_relation}\,(U_2,\ n,\ m) \iff \mathtt{get\_relation}\,(U_i,\ n,\ m) = \ell$$

It runs in $O(\Delta^2 \log^2 n)$ where $n = |U_1| + |U_2|$ and Δ is the number of different bindings between them.

Proof. First, looking at all times elements are added to M, it is easy to show that for all r_1, r_2:

$$\forall\, (n, \ell_1, \ell_2) \in M[r_1, r_2], \ell_1 = \mathtt{get\_relation}\,(U_1,\ r_1,\ n) \wedge \ell_2 = \mathtt{get\_relation}\,(U_2,\ r_2,\ n) \qquad \text{(HM)}$$

Proving the Invariants

Eager compression: lets examine the bindings added to U_i:

- Some come from the intersection, so they are equal to the bindings of U_1 and U_2, which point directly to their representative. It follows that this representative points to itself in both U_1 and U_2, so it is also pointing to itself in the intersection.
- Some come from the combining function, those are either self-pointing (therefore eager) or point to an element from M. Since all elements added to M are self pointing, this is still a directly pointing to a representative.

Representatives point to themselves: our intersection does not remove any element from U_1 and U_2 (which must be total functions if they satisfy the invariant). Thus, U_i is also a total function (the representatives must point somewhere). By the eager compression invariant, they must point to themselves.

Representatives are minimal: suppose n points to r, as seen above, if n's binding comes from the intersection, so does r's. They thus satisfy the invariant $n > r$.

If n's binding comes from the combining function, it is either a self-binding $n = r$, or a binding to an element added to M before n. If that element was added to M inside the combine function, then it must be smaller than n since the combine function sees elements in increasing order.

In the other case, it was added by the $C_1 \cap C_2$ merge. This means r was a representative in U_1 and U_2. Since we then have $n \xrightarrow{\ell_1}\!\!\!\!\rightarrow_{U_1} r_1$ and $r_1 \xrightarrow{\ell_1'}\!\!\!\!\rightarrow_{U_1} r$ (by HM), it follows that n is in the connected component of r in U_1, and must thus be bigger than r.

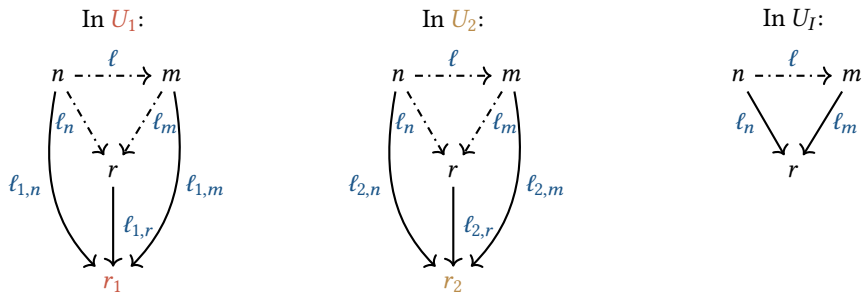
C_i is indeed the set of connected components of U_i: After the $C_1 \cap C_2$ intersection, C_i is equal to $[r \mapsto C_1[r] \cap C_2[r] \mid r \text{ representative in both } U_1, U_2]$. Now consider how C_i evolves during the $U_1 \cap U_2$ intersection:

- Elements that are not seen by the combining function have a representative r that was a representative in both U_1 and U_2, so they are already in the correct connected component and are not moved.

- The first item of $M[r_1, r_2]$ will always start with an associated class $C_1[r_1] \cap C_2[r_2]$, so elements that match this item also already in the correct connected component.
  Elements that don't match the first item are removed from this connected component, and added to the component they do match.
  Elements that match nothing are in no connected component (since they were removed if the list is not empty, and never contained in the intersection if it is). They are added to their own component once out of the loop (if n has representatives r_1 and r_2, then it must be in $C_1[r_1] \cap C_2[r_2]$).

Proving Correctness

Let us take n, m, ℓ such that $\texttt{get\_relation}\,(U_1,\, n,\, m) = \ell = \texttt{get\_relation}\,(U_2,\, n,\, m)$. Let r be the smallest element in the connected component of n and m such that $\texttt{get\_relation}\,(U_1,\, n,\, r) = \texttt{get\_relation}\,(U_1,\, n,\, r)$ and $\texttt{get\_relation}\,(U_1,\, m,\, r) = \texttt{get\_relation}\,(U_1,\, m,\, r)$ (it exists as a min of a non-empty set (it must contain n)). We also let $(r_1, \ell_{1,n}) = \texttt{find}\,(U_1,\, n)$, $(r_1, \ell_{1,m}) = \texttt{find}\,(U_1,\, m)$ (they have same representative by Theorem 3.1) and similarly for U_2. Graphically:



- Case $r = r_1 = r_2$: in that case, the nodes n and m are never seen in the custom combining function since $U_1[n] = (r, \ell_n) = U_2[n]$ and same for m. Thus, they are preserved by the intersection: $U_i[n] = (r, \ell_n)$ and same for m, so $\ell = \texttt{get\_relation}\,(U_i,\, n,\, m)$;
- Otherwise, $r_1 \neq r_2$ or $r \neq r_1$: In that case, r will be examined by the combining function. Indeed, if $r_1 \neq r_2$ then $U_1[r] = (r_1, \ell_{1,r}) \neq (r_2, \ell_{2,r}) = r_2$. But if $r_1 = r_2 \neq r$, we must have $\ell_{1,r} \neq \ell_{2,r}$. Otherwise, $\ell_{1,n} = \ell_n \mathbin{\hat{\S}} \ell_{1,r} = \ell_n \mathbin{\hat{\S}} \ell_{2,r} = \ell_{2,m}$ and same for m, which contradicts the fact that r is minimal.
  In the combining function, it will not match any element in $M[r_1, r_2]$, (if it does, then that element would be smaller than r and also have the same relation to n and m...). Therefore, $(r, \texttt{inv}\,(\ell_{1,r}), \texttt{inv}\,(\ell_{2,r}))$ will be added to $M[r_1, r_2]$, and $U_i[r] = (r, \texttt{id})$.
  We will thus have $U_i[n] = (r, \ell_n)$. If $n = r$ then the result is immediate, otherwise n and will (later) be examined by the combining function, It will match $(r, \texttt{inv}\,(\ell_{1,r}), \texttt{inv}\,(\ell_{2,r}))$ since $\ell_{1,n} \mathbin{\hat{\S}} \ell_{1,r} = \ell_n$ (and same for 2). Note that it cannot match any of the previous list items since r is minimal and the list is clearly increasing. The same reasoning gives $U_i[m] = (r, \ell_m)$.
  Since $\ell = \ell_n \mathbin{\hat{\S}} \ell_m$, the condition still holds.

For the reciprocal, we can show that the saturated graph generated by U_i is included in that of U_1. All points added to U_i either come from U_1 directly; are self representative; or are built by $U_i[n] = (r, \ell_1 \mathbin{\hat{\S}} \ell_1')$ where $\texttt{find}\,(U_1,\, n) = (r_1, \ell_1)$ and $\ell_1' = \texttt{get\_relation}\,(U_1,\, r_1,\, r)$ by HM. Thus, we clearly have $\texttt{get\_relation}\,(U_1,\, n,\, r) = \ell_1 \mathbin{\hat{\S}} \ell_1'$.

The saturated graph is also included in U_2, since the algorithm is quite symmetrical. Thus, and edge that appears in the graph generated by U_i must also appear in the graphs generated by U_1 and U_2.

Proving Complexity

Let us count operations following the algorithm. The first operation is the connected component intersection. Since it's a Patricia-tree fast intersection, it is in $O(\text{diff} \times \text{fun} \times \log(\text{size}))$. Here size is the number of connected components, bounded by n. Diff is the number of components with non-equal values, it is bounded by d (Each pair of non-equal connected components can be mapped to a point where they differ, such a point would be counted in d). Finally, fun is the complexity of the combining function. Here it performs a connected component intersection, so it also $O(d \log n)$ when using Patricia trees to represent sets. Thus, this first operation is $O(d^2 \log^2 n)$.

The intersection of U_1 and U_2 is also $O(\text{diff} \times \text{fun} \times \log(\text{size}))$, but here size is n, diff is d. For the combining function:

- The loop is in $O(d + \log n)$: indeed, the set insertion/deletion that occurs in the loop occur only on a single iteration, so they can be counted apart, as a single $O(\log n)$. The loop itself executes at most d times, because for a given r_1, r_2, the length of $M[r_1, r_2]$ is bound by 1 plus the number of differing elements (which are those seen in the combining function).
- Out of the loop, we have two connected component lookups ($O(\log n)$), one M lookup and write ($O(\log(n * n)) = O(2 \log(n)) = O(\log n)$), and one connected set intersection ($O(d \log n)$)

So the total cost of the combining function if $O(d + \log n + \log n + d \log n) = O(d \log n)$. Therefore, the cost of this second intersection is also $O(d^2 \log^2 n)$. □