A Practical Construction for Decomposing Numerical Abstract Domains

GAGANDEEP SINGH, ETH Zurich, Switzerland
MARKUS PÜSCHEL, ETH Zurich, Switzerland
MARTIN VECHEV, ETH Zurich, Switzerland

Numerical abstract domains such as Polyhedra, Octahedron, Octagon, Interval, and others are an essential component of static program analysis. The choice of domain offers a performance/precision tradeoff ranging from cheap and imprecise (Interval) to expensive and precise (Polyhedra). Recently, significant speedups were achieved for Octagon and Polyhedra by manually decomposing their transformers to work with the Cartesian product of projections associated with partitions of the variable set. While practically useful, this manual process is time consuming, error-prone, and has to be applied from scratch for every domain.

In this paper, we present a generic approach for decomposing the transformers of sub-polyhedra domains along with conditions for checking whether the decomposed transformers lose precision with respect to the original transformers. These conditions are satisfied by most practical transformers, thus our approach is suitable for increasing the performance of these transformers without compromising their precision. Furthermore, our approach is "black box:" it does not require changes to the internals of the original non-decomposed transformers or additional manual effort per domain.

We implemented our approach and applied it to the domains of Zones, Octagon, and Polyhedra. We then compared the performance of the decomposed transformers obtained with our generic method versus the state of the art: the (non-decomposed) PPL for Polyhedra and the much faster ELINA (which uses manual decomposition) for Polyhedra and Octagon. Against ELINA we demonstrate finer partitions and an associated speedup of about 2x on average. Our results indicate that the general construction presented in this work is a viable method for improving the performance of sub-polyhedra domains. It enables designers of abstract domains to benefit from decomposition without re-writing all of their transformers from scratch as required by prior work.

CCS Concepts: • **Theory of computation → Program verification**; **Program analysis**; **Abstraction**;

Additional Key Words and Phrases: Abstract Interpretation, Numerical Domains, Domain Decomposition, Performance Optimization

ACM Reference Format:
Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. A Practical Construction for Decomposing Numerical Abstract Domains. *Proc. ACM Program. Lang.* 2, POPL, Article 55 (January 2018), 28 pages. https://doi.org/10.1145/3158143

1 INTRODUCTION

Numerical abstract domains are a key component of modern static program analyzers [Blanchet et al. 2003; Gurfinkel et al. 2015]. The design of these domains remains an art as one is faced

Authors' addresses: Gagandeep Singh, Department of Computer Science, ETH Zurich, Zürich, Switzerland, gsingh@inf.ethz.ch; Markus Püschel, Department of Computer Science, ETH Zurich, Zürich, Switzerland, pueschel@inf.ethz.ch; Martin Vechev, Department of Computer Science, ETH Zurich, Zürich, Switzerland, martin.vechev@inf.ethz.ch.

2475-1421/2018/1-ART55
https://doi.org/10.1145/3158143

with two critical choices while fine-tuning the cost and precision of their domain: (a) the shape of constraints which determines the domain's expressivity, and (b) the precision and scalability of its abstract transformers.

Improving scalability of abstract transformers is an inherently hard problem since limiting the shape of the constraints allowed in the domain does not necessarily guarantee reduction in the transformer's asymptotic complexity. For example, the best transformers for assignments in weakly relational domains such as Octagon [Miné 2006], Zones [Miné 2002], and TVPI [Simon and King 2010] have the same worst-case exponential complexity as those in the most expensive Polyhedra [Cousot and Halbwachs 1978] domain.

Because of the importance of scaling the analysis to realistic applications, there has been increased interest in improving the performance of existing domains. The approaches can be roughly divided into two classes: (a) implement less precise transformers tuned to the specific verification task [Blanchet et al. 2003; Heo et al. 2016; Venet and Brat 2004], or (b) maintain the same precision as the existing implementation and improve performance by designing specialized algorithms and data structures optimized for the particular domain [Gange et al. 2016; Jourdan 2017]. The former approach uses approximation (of the best or standard transformers) with the hope of improving performance in practical scenarios while maintaining sufficient precision needed to verify the property of interest. The latter approach, while challenging to devise, is appealing because it does not explicitly lose precision yet can still dramatically increase overall performance.

One technique for achieving this goal is the concept of *decomposition*. It is based on the observation that abstract elements may be decomposed into Cartesian products over disjoint subsets of variables; hence, a given domain transformer does not need to be applied on the complete abstract element but rather only on some part of it, thus reducing cost. The first attempt at decomposition was for the Polyhedra [Halbwachs et al. 2003, 2006] domain where the abstract elements were decomposed based on partitioning a variable set into subsets such that constraints exist only between the variables in the same subset. The partitioning was performed on the fly, however the partitions produced were too coarse and no implementation is publicly available.

Recently, the concept of online decomposition was revisited and applied to achieve orders of magnitude speed-ups over the state of the art [Singh et al. 2015, 2017]. The underlying idea is to maintain and continuously update the partitions based on the transformer semantics. Implementations exist for the Octagon [Singh et al. 2015] and Polyhedra [Singh et al. 2017] domains. In both cases, the decomposition was manually designed from scratch for the standard transformers of the particular domain. The downside of this approach is that the substantial effort invested in decomposing the transformers of the specific implementation of the domain cannot be reused and needs to be repeated for every new implementation. This task is difficult and error-prone as it requires devising new algorithms and data structures from scratch each time.

To illustrate the issue, consider an element $\mathcal{I} = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0\}$ in the Octagon domain (which captures constraints of the form $\pm x_i \pm x_j \leq c, c \in \mathbb{R}$, between the program variables) and the conditional expression $x_2 + x_3 + x_4 \leq 1$. There are multiple ways to define a sound conditional transformer in the Octagon domain for the given conditional expression. One may define a sound conditional transformer T_1 that adds the non-redundant constraint $-x_1 + x_4 \leq 1$ to $\mathcal{I}$ resulting in the output $\mathcal{I}' = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0, -x_1 + x_4 \leq 1\}$ whereas another transformer T_2 may add $x_2 + x_3 \leq 1$ to $\mathcal{I}$ resulting in $\mathcal{I}'' = \{-x_1 - x_2 \leq 0, -x_3 \leq 0, -x_4 \leq 0, x_2 + x_3 \leq 1\}$. The specialized decomposition for the Octagon domain [Singh et al. 2015] requires access to the exact definition of the transformer, i.e., it will produce different decomposition for T_1 and T_2 as the set of variables in the constraints added by the two transformers are disjoint.

This work. The key objective of this work is to bring the power of decomposition to all sub-polyhedra domains without requiring complex manual effort from the domain designer. This enables domain designers to achieve speed-ups without requiring them to rewrite all abstract transformers from scratch each time.

More formally, our goal is to provide a systematic correct-by-construction method that, given a sound abstract transformer T in a sub-polyhedra domain (e.g., Zones), generates a sound decomposed version of T that is faster than T and *does not* require any change to the internals of T. In this paper we present a construction that achieves this objective under certain conditions. We also show that the obtained decomposed transformers are faster than the prior, hand-tuned decomposed domains from [Singh et al. 2015, 2017].

Main contributions. Our paper makes the following contributions:

- We introduce a general construction for obtaining decomposed transformers from given non-decomposed transformers of existing numerical domains. Our construction is "black-box:" it does not require changes to the underlying algorithms implemented in the original non-decomposed transformers.
- We provide conditions on the non-decomposed transformers under which our decomposition maintains precision and equivalence at fixpoint.
- We apply our method to decompose standard transformers of three popular and expensive domains: Polyhedra, Octagon, and Zones. For these we provide complete end-to-end implementations as part of ELINA [eli].
- We evaluate the effectiveness of our decomposed analysis against state-of-the-art implementations on large real-world benchmarks including Linux device drivers. Our evaluation shows up to 6x and 2x speedups compared to state-of-the-art manually decomposed domains and orders of magnitude speedups compared to non-decomposed Polyhedra and Octagon. For Zones, we achieve speedups of up to 6x compared to our own, non-decomposed implementation.

2 GENERIC MODEL FOR NUMERICAL ABSTRACT DOMAINS

An abstract domain consists of a set of abstract elements and a set of transformers that model the effect of program statements and expressions (assignment, conditionals, etc.) on the abstract elements. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of variables. In this paper, we consider sub-polyhedra domains, i.e., numerical abstract domains $\mathcal{D}$ that encode linear relationships between the variables in $\mathcal{X}$ of the form:

$$\sum_{i=1}^{n} a_i x_i \otimes c, \qquad \text{where } x_i \in \mathcal{X}, a_i \in \mathbb{Z}, \otimes \in \{\leq, =\}, c \in \mathcal{C}. \tag{1}$$

Typical choices for $\mathcal{C}$ include $\mathbb{Q}$ (rationals) and $\mathbb{R}$ (reals). As with any abstraction, the design of a numerical domain is guided by the cost vs. precision tradeoff. For instance, the Polyhedra domain [Cousot and Halbwachs 1978] is the most precise numerical domain yet it is also the most expensive. On the other hand, the Interval (Box) domain is cheap but also very imprecise as it does not preserve relational information between variables. Between these two sit a number of domains with varying degrees of precision and cost: examples include Two Variables Per Inequality (TVPI) [Simon and King 2010], Octagon [Miné 2006], and Zones [Miné 2002].

Representing domain constraints. We introduce notation for describing the set of constraints a given domain $\mathcal{D}$ can express for the variables in $\mathcal{X}$. This set of constraints is referred to as $\mathcal{L}_{\mathcal{X},\mathcal{D}}$ and is determined by four components $(n, \mathcal{R}, \mathcal{T}, \mathcal{C})$:

- The size n of the variable set $\mathcal{X}$.

Table 1. Instantiation of constraints expressible in various numerical domains.

Domain	$\mathcal{R}$	$\mathcal{T}$	$\mathcal{C}$	Reference
Polyhedra	$\mathbb{Z}^n$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Cousot and Halbwachs 1978]
Linear equality	$\mathbb{Z}^n$	$\{=\}$	$\mathbb{Q}, \mathbb{R}$	[Karr 1976]
Octahedron	$\mathbb{U}^n$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[ClarisÃş and Cortadella 2007]
Stripes	$\{(a, a, -1, 0, \dots, 0) \mid a \in \mathbb{Z}\} \cup$ $\{(0, a, -1, 0, \dots, 0) \mid a \in \mathbb{Z}\}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Ferrara et al. 2008]
TVPI	$\mathbb{Z}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Simon and King 2010]
Octagon	$\mathbb{U}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Miné 2006]
Logahedra	$\mathbb{L}^2 \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Howe and King 2009]
Zones	$\{1, 0\} \times \{0, -1\} \times \{0\}^{n-2}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Miné 2002]
Upper bound	$\{1\} \times \{-1\} \times \{0\}^{n-2}$	$\{\leq\}$	$\{0\}$	[Logozzo and Fähndrich 2008]
Interval	$\{1, -1\} \times \{0\}^{n-1}$	$\{\leq, =\}$	$\mathbb{Q}, \mathbb{R}$	[Cousot and Cousot 1976]

- A relation $\mathcal{R} \subseteq \mathcal{R}_1 \times \mathcal{R}_2 \times \cdots \times \mathcal{R}_n$ to describe the universe of possible coefficients. Each $\mathcal{R}_i \subseteq \mathbb{Z}$ is a set of integers defining the allowed values for the coefficient a_i. Typical examples for $\mathcal{R}_i$ include $\mathbb{Z}, \mathbb{U} = \{-1, 0, 1\}$, and $\mathbb{L} = \{-2^k, 0, 2^k \mid k \in \mathbb{Z}\}$.
- The set $\mathcal{T} \subseteq \{\leq, =\}$ determining equality/inequality constraints.
- The set $\mathcal{C}$ containing the allowed values for the constant c in (1). Typical examples include $\mathbb{Q}$ and $\mathbb{R}$.

Table 1 shows common constraints in the above notation allowed by different numerical domains. The set of constraints $\mathcal{L}_{\mathcal{X},\mathcal{D}}$ representable by a domain $\mathcal{D}$ contains all constraints of the form $\sum_{i=1}^{n} a_i x_i \otimes c$ where: (i) the coefficient list of each expression $\sum_{i=1}^{n} a_i x_i$ is a permutation of a tuple in $\mathcal{R}$, (ii) $\otimes \in \mathcal{T}$, and (iii) the constant $c \in \mathcal{C}$. For instance, the possible constraints $\mathcal{L}_{\mathcal{X},\text{Octagon}}$ for the Octagon domain over real numbers are described via the tuple $(n, \mathbb{U}^2 \times \{0\}^{n-2}, \{\leq, =\}, \mathbb{R})$.

Example 2.1. Consider a program with four variables and a fictive domain that can relate at most two:

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\} \text{ and } \mathcal{L}_{\mathcal{X},\mathcal{D}} : (4, \mathbb{U}^2 \times \{0\}^2, \{\leq, =\}, \{1, 2\}).$$

Here, the constraint $2x_1 + 3x_4 \leq 2 \notin \mathcal{L}_{\mathcal{X},\mathcal{D}}$ as no permutation of tuples in $\mathbb{U}^2 \times \{0\}^2$ can produce $(2, 0, 0, 3)$. Similarly, $x_2 - x_3 \leq 3 \notin \mathcal{L}_{\mathcal{X},\mathcal{D}}$ even though there exists a permutation of tuples in $\mathbb{U}^2 \times \{0\}^2$ that can produce $(0, 1, -1, 0)$, but $3 \notin \mathcal{C}$. However, the constraints $x_2 - x_3 \leq 1$ and $x_2 - x_3 = 2$ are in $\mathcal{L}_{\mathcal{X},\mathcal{D}}$.

Defining an abstract domain. An abstract element $\mathcal{I}$ in a domain $\mathcal{D}$ is a conjunction of a finite number of constraints from $\mathcal{L}_{\mathcal{X},\mathcal{D}}$. By abuse of notation we will represent $\mathcal{I}$ as a set of constraints (interpreted as a conjunction of the constraints in the set). The set of all possible abstract elements in $\mathcal{D}$ is denoted with $\mathcal{P}_{\mathcal{D}}$ and typically forms a lattice $(\mathcal{P}_{\mathcal{D}}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ with respect to the defined domain order $\sqsubseteq$. Given abstract elements $\mathcal{I}$ and $\mathcal{I}'$, $\mathcal{I} \sqcup \mathcal{I}'$ is the smallest element in the domain covering both $\mathcal{I}$ and $\mathcal{I}'$ and is computed or approximated by the join transformer. Similarly $\mathcal{I} \sqcap \mathcal{I}'$ is the meet, computed, e.g., as $\mathcal{I} \cup \mathcal{I}'$. There are usually around 40 abstract transformers in a given domain $\mathcal{D}$. While our theory handles all transformers, our presentation focuses on the core transformers, namely: conditional containing a linear constraint, assignment with a linear expression, meet ($\sqcap$), join ($\sqcup$), and widening ($\triangledown$). We chose these because they are the most expensive transformers in the domain and thus their design shows the most variation, i.e., they can be implemented in multiple ways. We note that there is an equivalent representation of an abstract element based on the generator representation where the element is encoded as a collection of vertices, rays, and lines. In this paper, we use the constraint representation as it leads to a clearer exposition of the ideas. However, our technical results are also valid with the generator representation.

As is standard, we use the meet-preserving concretization function γ to denote with $\gamma(\mathcal{I})$ the concrete element (polyhedron) represented by the abstract element $\mathcal{I}$. We note that it is possible for $\mathcal{I}$ to include redundant constraints, that is, removing a constraint from $\mathcal{I}$ may not change the represented concrete element $\gamma(\mathcal{I})$. Further, the minimal (without any redundancy) representation of a concrete element $\gamma(\mathcal{I})$ need not be unique, i.e., there could be two distinct abstract elements $\mathcal{I}$ and $\mathcal{I}'$ with $\gamma(\mathcal{I}) = \gamma(\mathcal{I}')$:

Example 2.2. $\mathcal{I} = \{x_1 = 0, x_2 = 0\}$ and $\mathcal{I}' = \{x_1 = 0, x_2 = 0, x_1 = x_2\}$ represent the same concrete element $\gamma(\mathcal{I})$ in the Polyhedra domain. However, $\mathcal{I}'$ contains the redundant constraint $x_1 = x_2$. $\mathcal{I}$ is not the only minimal representation as $\mathcal{I}'' = \{x_1 = 0, x_1 = x_2\}$ is also minimal for $\gamma(\mathcal{I})$.

We next define what it means for an abstract transformer to be sound.[1]

Definition 2.1. A given abstract transformer T is sound w.r.t to its concrete transformer $T^{\#}$ iff for any element $\mathcal{I} \in \mathcal{D}$, $T^{\#}(\gamma(\mathcal{I})) \subseteq \gamma(T(\mathcal{I}))$.

The soundness criterion is naturally extended to transformers with multiple arguments.

Definition 2.2. We say an abstract domain $\mathcal{D}$ is *closed* (also called forward complete in [Giacobazzi et al. 2000; Ranzato and Tapparo 2004]) for a concrete transformer $T^{\#}$ (e.g., conditional, meet) iff it can be done precisely in the domain, i.e., if there exists an abstract transformer T corresponding to that concrete transformer such that for any abstract element $\mathcal{I}$ in $\mathcal{D}$, $\gamma(T(\mathcal{I})) = T^{\#}(\gamma(\mathcal{I}))$.

The Polyhedra domain is closed for conditional, assignment, and meet, but not for the join. All other domains in Table 1 are only closed for the meet. Indeed, a crucial aspect of abstract interpretation is to permit sound approximations for transformers for which the domain is not closed.

Example 2.3. The Octagon domain is not closed for the conditional transformer. For example, if the condition is $x_1 - 2x_2 \leq 0$ and the abstract element is $\mathcal{I} = \{x_1 \leq 1, x_2 \leq 0\}$, then the concrete element $T^{\#}(\gamma(\mathcal{I})) = \{x_1 \leq 1, x_2 \leq 0, x_1 - 2x_2 \leq 0\}$ is not representable exactly in the Octagon domain (because the constraint $x_1 - 2x_2 \leq 0$ is not exactly representable).

A useful concept in analysis (and one we refer to throughout the paper) is that of a best abstract transformer.

Definition 2.3. A (unary) abstract transformer T in $\mathcal{D}$ is best iff for any other sound unary abstract transformer T' (corresponding to the same concrete transformer $T^{\#}$) it holds that for any element $\mathcal{I}$ in $\mathcal{D}$, T always produces a more precise result (in the concrete), that is, $\gamma(T(\mathcal{I})) \subseteq \gamma(T'(\mathcal{I}))$. The definition is naturally lifted to multiple arguments.

In Example 2.3, a possible sound approximation for the output in the Octagon domain is $\mathcal{I}'' = \mathcal{I}$ while one best transformer would produce $\{x_1 \leq 0, x_2 \leq 0, x_1 - x_2 \leq 0\}$. Since there can be multiple abstract elements with the same concretization, there can be multiple best abstract transformers in $\mathcal{D}$. The sub-polyhedra abstract domains we consider in this paper always come equipped with a best transformer and are closed under meet.

[1] Throughout the paper we will simply use the term transformer to mean a sound abstract transformer.

3 DECOMPOSING ABSTRACT ELEMENTS

In this section we introduce the needed notation and concepts for decomposing abstract elements and transformers. As in [Halbwachs et al. 2003; Singh et al. 2015, 2017], our approach to decomposition is based on the observations that: (a) not all variables get related by a constraint in a given abstract element $\mathcal{I}$, and (b) the number of variables affected by a program statement is small compared to the size n of the set of program variables $\mathcal{X}$. These observations enable the decomposition of $\mathcal{I}$ into smaller pieces which, in turn, allow the derivation of abstract domain transformers with reduced asymptotic complexity. Note that the decomposition is not fixed: during the iterations of the analysis, new abstract elements are created and their decomposition is computed *dynamically*. Overall, this results in better performance with respect to the original non-decomposed transformer.

We address the decomposition of abstract elements and transformers for $\mathcal{D}$ based on partitioning the variable set $\mathcal{X}$. The set $\mathcal{P}_{\mathcal{X}}$ which consists of all partitions of $\mathcal{X}$ forms the *partition lattice* $(\mathcal{P}_{\mathcal{X}}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The elements of the lattice are ordered as follows: $\pi \sqsubseteq \pi'$, if every subset of π is included in some subset of π' (π "is finer" than π'). The lattice contains the usual *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) operators. In the partition lattice, $\top = \{\mathcal{X}\}$ and $\bot = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$.

Given an abstract element $\mathcal{I}$, we partition the set of program variables $\mathcal{X}$ into subsets $\mathcal{X}_k$ that we call *blocks* such that constraints only exist between variables in the same block. Each unconstrained variable x_i yields the singleton block $\{x_i\}$. We write $\pi_{\mathcal{I},\mathcal{D}} = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$ to denote the unique finest partition for an element $\mathcal{I}$. For simplicity, we usually omit $\mathcal{D}$ from the subscript and write $\pi_{\mathcal{I}}$.

The partition $\pi_{\mathcal{I}}$ decomposes $\mathcal{I}$ into a set of smaller abstract elements $\mathcal{I}_k$ on the variables in a block $\mathcal{X}_k$ which we call *factors*. Each factor $\mathcal{I}_k \subseteq \mathcal{I}$ is defined by the constraints that exist between the variables in the corresponding block $\mathcal{X}_k$. $\mathcal{I}$ can be recovered from the set of factors by taking the union of the constraint sets $\mathcal{I}_k$.

Example 3.1. Consider the element $\mathcal{I} = \{x_1 - x_2 \leq 1, x_3 \leq 0, x_4 \leq 0\}$ in the TVPI domain

$$\mathcal{X} = \{x_1, x_2, x_3, x_4\} \text{ and } \mathcal{L}_{\mathcal{X},\text{TVPI}} : (4, \mathbb{Z}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}).$$

Here $\mathcal{X}$ can be partitioned into three blocks with respect to $\mathcal{I}$ resulting in three factors:

$$\pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\}, \quad \mathcal{I}_1 = \{x_1 - x_2 \leq 1\}, \quad \mathcal{I}_2 = \{x_3 \leq 0\}, \quad \text{and } \mathcal{I}_3 = \{x_4 \leq 0\}.$$

For a given $\mathcal{D}$, $\pi_{\bot} = \pi_{\top} = \bot = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$. More generally, note that $\mathcal{I} \sqsubseteq \mathcal{I}'$ does not imply that $\pi_{\mathcal{I}'}$ is finer, coarser, or comparable to $\pi_{\mathcal{I}}$.

Different partitions for equivalent elements. To gain a deeper understanding of partitions for abstract elements, there are two interesting points worth noting. First, it is possible that two semantically equivalent abstract elements $\mathcal{I}, \mathcal{I}'$ in the domain have different partitions. That is, even if $\gamma(I) = \gamma(I')$, it may be the case that $\pi_{\mathcal{I}} \neq \pi_{\mathcal{I}'}$ or $\pi_{\mathcal{I}} \sqsubset \pi_{\mathcal{I}'}$:

Example 3.2. Consider $\mathcal{I} = \{x_1 \leq x_2, x_2 = 0, x_3 = 0\}$ with the finest partition $\pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3\}\}$, $\mathcal{I}' = \{x_1 \leq 0, x_2 = 0, x_3 = 0\}$ with $\pi_{\mathcal{I}'} = \{\{x_1\}, \{x_2\}, \{x_3\}\}$ and $\mathcal{I}'' = \{x_1 \leq x_3, x_2 = 0, x_3 = 0\}$ with $\pi_{\mathcal{I}''} = \{\{x_1, x_3\}, \{x_2\}\}$ in the Polyhedra domain. Here $\gamma(\mathcal{I}) = \gamma(\mathcal{I}') = \gamma(\mathcal{I}'')$, but the partitions are pairwise different.

Second, it is possible that for a given abstract element $\mathcal{I}$, there exists an equivalent element $\mathcal{I}'$ with finer partition but $\mathcal{I}'$ is not representable in the domain. This shows a potential limitation of syntactic partitions.

Example 3.3. Consider the Stripes domain

$$\mathcal{X} = \{x_1, x_2, x_3\}, \quad \mathcal{L}_{\mathcal{X},\text{Stripes}} : \{3, \{(a, a, -1) \mid a \in \mathbb{Z}\} \cup \{(0, a, -1) \mid a \in \mathbb{Z}\}, \{\leq, =\}, \mathbb{Q}\},$$

and the abstract element

$$\mathcal{I} = \{x_1 + x_2 - x_3 = 0, -x_2 + x_3 = 0\} \text{ with } \pi_{\mathcal{I}} = \{x_1, x_2, x_3\}.$$

This domain cannot represent the equivalent element $\mathcal{I}' = \{x_1 = 0, x_2 - x_3 = 0\}$ with partition $\pi_{\mathcal{I}'} = \{\{x_1\}, \{x_2, x_3\}\}$, which is finer than $\pi_{\mathcal{I}}$. This is because the constraint $x_1 = 0$ is not representable in the Stripes domain.

It is important we guarantee that regardless of how approximate a given transformer T is, the partition we end up computing for T is always sound (permissible) for the output abstract element $\mathcal{I}$ produced by T. Next, we formalize the notion of permissiveness [Singh et al. 2017]:

Definition 3.1. A partition $\overline{\pi}$ is permissible for an abstract element $\mathcal{I}$ if it is *coarser* than $\pi_{\mathcal{I}}$, that is, if $\overline{\pi} \sqsupseteq \pi_{\mathcal{I}}$.

The variables related in $\pi_{\mathcal{I}}$ are also related in any permissible partition of $\mathcal{I}$, but not vice-versa. In Example 3.1, $\{\{x_1, x_2\}, \{x_3, x_4\}\}$ is permissible for $\mathcal{I}$ while $\{\{x_1\}, \{x_2, x_3, x_4\}\}$ is not. We will generally use $\overline{\pi}_{\mathcal{I}}$ to denote a permissible partition for $\mathcal{I}$.

4 RECIPE FOR DECOMPOSING TRANSFORMERS

A primary objective of this work is to define a mechanical recipe which takes as input a sound abstract transformer and produces as output a sound and decomposed variant of that transformer, thus resulting in better analysis performance. In this section we describe the general recipe and illustrate its actual use.

At first glance the above challenge appears fundamentally difficult because there are many ways to define a sound transformer in a domain $\mathcal{D}$. Standard implementations of popular numerical domains like Octagon, Zones, TVPI, and others, do not necessarily implement the best transformers as they can be expensive; instead the domains often approximate them. Interestingly, as pointed out earlier, such an approximation can make the associated partition both coarser or finer. That is, the partitioning function is not monotone. Here is an example illustrating this point:

Example 4.1. Consider the elements $\mathcal{I} = \{x_1 \leq 0, x_2 \leq 0, x_1 - x_2 \leq 0\}$ with $\pi_{\mathcal{I}} = \{\{x_1, x_2\}\}$ and $\mathcal{I}' = \{x_1 \leq 0, x_2 \leq 0\}$ with $\pi_{\mathcal{I}'} = \{\{x_1\}, \{x_2\}\}$ in the Polyhedra domain. Here, $\gamma(\mathcal{I}) \subset \gamma(\mathcal{I}')$ and $\pi_{\mathcal{I}} \sqsupseteq \pi_{\mathcal{I}'}$. Now consider the element $\mathcal{I}'' = \{x_1 + x_2 \leq 0\}$ with $\pi_{\mathcal{I}''} = \{\{x_1, x_2\}\}$. Here, $\gamma(\mathcal{I}') \subset \gamma(\mathcal{I}'')$ and $\pi_{\mathcal{I}'} \sqsubseteq \pi_{\mathcal{I}''}$.

Definition 4.1. A transformer T in $\mathcal{D}$ is decomposable for input $\mathcal{I}$ iff the output $\mathcal{I}_O = T(\mathcal{I})$ results in a partition $\pi_{\mathcal{I}_O} \neq \top$. For binary transformers, the definition is analogous.

There are many ways to define sound approximations of the best transformers in $\mathcal{D}$. As a consequence, it is possible to have two transformers T_1, T_2 in $\mathcal{D}$ on the same input $\mathcal{I}$ such that one produces the $\top$ partition for the output while the other does not. There are two principal ways to obtain a decomposable transformer: (a) white box: here, one designs the transformer from scratch, maintaining the (changing) partitions during analysis, and (b) black box: here, one provides a construction for decomposing existing transformers without knowing their internals. In the next section, we pursue the second approach and show that it is possible and, under certain conditions, without losing precision. As a preview, we now describe the high-level steps that one needs to perform dynamically in our black-box decomposition.

A construction for online transformer decomposition. There are three main steps for decomposing a given transformer:

(1) compute (if needed) partitions for the input(s) of the transformer,

(2) compute a partition for the output based on the program statement/expression and input partition(s) from step 1,
(3) apply the transformer on one or more factors of the inputs from step 2.

We next describe these steps in greater detail.

In an ideal setting, one would always work with the finest partition for the inputs and the output to (optimally) reduce the cost of the transformer. The finest partition for the inputs of a given transformer can always be computed from scratch by taking the abstract element and connecting the variables that occur in the same constraint in that element. The downside is that this computation may incur significant overhead. For example, computing the finest partition for an element in the Octagon domain from scratch has the same quadratic complexity (in the number of variables) as the conditional, meet, and assignment transformers. Thus, it may nullify potential performance gains from decomposing these transformers. In our approach we thus iteratively maintain permissible partitions.

To compute the output partition, a naive way is to first run the transformer, obtain an abstract element as a result, and then compute the partition for that element. Of course, this approach is useless since running the standard transformer prevents performance gains. Thus, the challenge is to determine first a permissible output partition at little cost so that then the transformers can be applied only to relevant factors. Indeed, in our construction we compute a permissible partition for the output based on permissible partitions of the input, the program statement, and possibly additional information that is cheaply available.

In the last step, once the output partition is obtained, the associated abstract element is computed directly in decomposed form by applying the original transformer to one or more factor(s) of the input(s). Applying this transformer on smaller factors reduces its complexity and results in increased performance. In certain cases, the permissible partition for the output can be further refined after applying the transformer and without adding significant overhead. We identify such cases in Section 5.

Our approach is generic in nature and can decompose the standard transformers of the existing sub-polyhedra numerical abstract domains. We implemented our recipe and applied it to Polyhedra, Octagon, and Zones. Using a set of large Linux device drivers, we show later in Section 6 the performance of our generated decomposed transformers vs. transformers obtained via state-of-the-art hand-tuned decomposition [Singh et al. 2015, 2017]. Our approach leads up to 6x speed-ups for Polyhedra and up to 2x speed-ups for Octagon. This speed-up is due to our decomposition theorems (discussed next) that enable, in certain cases, finer decomposition of abstract elements than previously possible. Speedups compared to the original transformers without decomposition are orders of magnitude larger. Further, we also decompose the Zones domain using our approach (for which no previous decomposition exists) without changing the existing domain transformers. We obtain a speedup up to 6x over non-decomposed implementation of the Zones domain. In summary, our recipe is generic in nature yet leads to state-of-the-art performance for classic abstract transformers.

5 DECOMPOSING DOMAIN TRANSFORMERS

In this section we show a construction that takes as input a sound and monotone transformer in a given domain $\mathcal{D}$ and produces a decomposed variant of the same transformer that operates on part(s) of the input(s). The resulting decomposed transformer is always sound. We define classes of transformers for which the output produced by the decomposed transformer has the same concretization as the original non-decomposed transformer, i.e., there is no loss of precision. Although our results apply to all transformers, we focus on the conditional, assignment, meet, join, and widening transformers. We also show how to obtain finer partitions than the manually decomposed transformers for Polyhedra and Octagon considered in prior work [Singh et al. 2015, 2017].

Partitioned abstract elements and factors. Given an abstract element $\mathcal{I}$ with permissible partition $\overline{\pi}_{\mathcal{I}} = \{\mathcal{X}_1, \ldots, \mathcal{X}_r\}$, we denote the associated factors with $\mathcal{I}(\mathcal{X}_k)$, $1 \leq k \leq r$. We will write a decomposed abstract element $\mathcal{I}$ as a set of constraints (and not as set of factors) just as we did before, but always explicitly mention the associated partition.

Common partition. We assume that inputs $\mathcal{I}, \mathcal{I}'$ for binary transformers are partitioned according to a common permissible partition $\overline{\pi}_{\text{common}}$. This partition can always be computed as $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$, where $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ are permissible partitions for $\mathcal{I}, \mathcal{I}'$ respectively.

Abstract ordering. Ordering in the decomposed abstract domain is defined as $\mathcal{I} \sqsubseteq \mathcal{I}' \equiv \gamma(\mathcal{I}) \subseteq \gamma(\mathcal{I}')$.

Precision of decomposed transformers. We define classes of conditional, assignment, meet, join and widening transformers for which the outputs of both the given non-decomposed T and our associated decomposed T^d have the same concretization, i.e., $\gamma(T(\mathcal{I})) = \gamma(T^d(\mathcal{I}))$ for all $\mathcal{I}$ in $\mathcal{D}$, which implies that T^d inherits monotonicity from T. If the transformer is not in these classes, then both $\gamma(T(\mathcal{I})) \subset \gamma(T^d(\mathcal{I}))$ or $\gamma(T(\mathcal{I})) \supset \gamma(T^d(\mathcal{I}))$ are possible and monotonicity may not hold.

If in addition to $\gamma(T(\mathcal{I})) = \gamma(T^d(\mathcal{I}))$, the given transformers satisfy

$$\gamma(\mathcal{I}) = \gamma(\mathcal{I}') \Rightarrow \gamma(T(\mathcal{I})) = \gamma(T(\mathcal{I}')),$$

then the analysis with our associated decomposed transformers produces the same semantic invariants at fixpoint (fixpoint equivalence).

5.1 Conditional

We consider conditional statements of the form $e \otimes c$ where $e = \sum_{i=1}^{n} a_i x_i$ with $a_i \in \mathbb{Z}, \otimes \in \{\leq, =\}$, and $c \in \mathbb{Q}, \mathbb{R}$, on an abstract element $\mathcal{I}$ with an associated permissible partition $\overline{\pi}_{\mathcal{I}}$ in domain $\mathcal{D}$. The conditional transformer computes the effect of adding the constraint $e \otimes c$ to $\mathcal{I}$. As discussed in Section 2, most existing domains are not closed for the conditional transformer. Moreover, computing the best transformers is expensive in these domains and thus the transformer is usually approximated to strike a balance between precision and cost. The example below illustrates two sound conditional transformers on the same input: the first transformer produces a decomposable output whereas the output of the second results in the $\top$ partition.

Example 5.1. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{Polyhedra}} : (6, \mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{x_1 + x_2 \leq 0, x_3 + x_4 \leq 5\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\}.$$

For the conditional statement $x_5 + x_6 \leq 0$, a best transformer T_1 may return:

$$\mathcal{I}_O = \{x_1 + x_2 \leq 0, x_3 + x_4 \leq 5, x_5 + x_6 \leq 0\} \text{ with } \overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\},$$

which is decomposable. However, another sound transformer T_2 may return the non-decomposable output:

$$\mathcal{I}'_O = \{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 5\} \text{ with } \overline{\pi}_{\mathcal{I}'_O} = \pi_{\mathcal{I}'_O} = \top.$$

Let $\mathcal{B}_{\text{cond}} = \{x_i \mid a_i \neq 0\}$ be the set of variables with non-zero coefficients in the constraint $\sum_{i=1}^{n} a_i x_i \otimes c$. The block $\mathcal{B}^*_{\text{cond}} = \bigcup_{\mathcal{X}_k \cap \mathcal{B}_{\text{cond}} \neq \emptyset} \mathcal{X}_k$ fuses all blocks $\mathcal{X}_k \in \overline{\pi}_{\mathcal{I}}$ that have non-empty intersection with $\mathcal{B}_{\text{cond}}$.

Algorithm 1 Decomposed conditional transformer T^d_{cond}

```
function CONDITIONAL($(\mathcal{I}, \overline{\pi}_{\mathcal{I}})$, stmt, $T_{\text{cond}}$)
    $\mathcal{B}^*_{\text{cond}} := \text{compute\_block}(\text{stmt}, \overline{\pi}_{\mathcal{I}})$
    $\mathcal{I}_{\text{cond}} := \mathcal{I}(\mathcal{B}^*_{\text{cond}})$
    $\overline{\pi}_{\mathcal{I}_O} := \{\mathcal{A} \in \overline{\pi}_{\mathcal{I}} \mid \mathcal{A} \cap \mathcal{B}^*_{\text{cond}} = \emptyset\} \cup \{\mathcal{B}^*_{\text{cond}}\}$
    $\mathcal{I}_{\text{rest}} := \mathcal{I}(\mathcal{X} \setminus \mathcal{B}^*_{\text{cond}})$
    $\mathcal{I}_O := T_{\text{cond}}(\mathcal{I}_{\text{cond}}) \cup \mathcal{I}_{\text{rest}}$
    return $(\mathcal{I}_O, \overline{\pi}_{\mathcal{I}_O})$
```

Example 5.2. Consider $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and an element $\mathcal{I}$ in the Polyhedra domain with $\overline{\pi}_{\mathcal{I}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{x_6\}\}$. For the conditional $x_3 + x_6 \leq 0$, $\mathcal{B}_{\text{cond}} = \{x_3, x_6\}$ and $\mathcal{B}^*_{\text{cond}} = \{x_1, x_2, x_3, x_6\}$.

Construction for conditional. Algorithm 1 shows our construction for decomposing a given conditional transformer T_{cond}. Given an input element $\mathcal{I}$ with a permissible partition $\overline{\pi}_{\mathcal{I}}$ in domain $\mathcal{D}$, the algorithm first extracts the block $\mathcal{B}^*_{\text{cond}}$ based on the conditional statement and the permissible partition $\overline{\pi}_{\mathcal{I}}$ as described above. The block coarsens the input partition to yield the output partition. Finally, the original transformer is applied to the associated abstract element $\mathcal{I}_{\text{cond}}$; the remaining constraints are kept as is in the result.

In Algorithm 1 the output of the decomposed transformer T^d_{cond} on input $\mathcal{I}$ is computed as $T^d_{\text{cond}}(\mathcal{I}) = T_{\text{cond}}(\mathcal{I}_{\text{cond}}) \cup \mathcal{I}_{\text{rest}}$. One can show that T^d_{cond} is sound but we focus on also maintaining precision and thus monotonicity. Thus, we define a class $\text{Cond}(\mathcal{D})$ of conditional transformers T_{cond} where $\gamma(T_{\text{cond}}(\mathcal{I})) = \gamma(T^d_{\text{cond}}(\mathcal{I}))$ (this is one of the two conditions discussed earlier that ensure fixed point equivalence).

Definition 5.1. A (sound and monotone) transformer T_{cond} for the conditional expression $e \otimes c$ is in $\text{Cond}(\mathcal{D})$ iff for any element $\mathcal{I}$ and any associated permissible partition $\overline{\pi}_{\mathcal{I}}$, the output $T_{\text{cond}}(\mathcal{I})$ satisfies:

- $T_{\text{cond}}(\mathcal{I}) = \mathcal{I} \cup \mathcal{I}' \cup \mathcal{I}''$ where $\mathcal{I}'$ contains non-redundant constraints between the variables from $\mathcal{B}^*_{\text{cond}}$ only and $\mathcal{I}''$ is a set of redundant constraints between the variables in $\mathcal{X}$.
- $\gamma(T_{\text{cond}}(\mathcal{I}_{\text{cond}})) = \gamma(\mathcal{I}' \cup \mathcal{I}_{\text{cond}})$.

THEOREM 5.2. *If $T_{cond} \in \text{Cond}(\mathcal{D})$, then $\gamma(T_{cond}(\mathcal{I})) = \gamma(T^d_{cond}(\mathcal{I}))$ for all inputs $\mathcal{I}$ in $\mathcal{D}$. In particular, T^d_{cond} is sound and monotone.*

PROOF.

$$\begin{aligned}
\gamma(T_{\text{cond}}(\mathcal{I})) &= \gamma(\mathcal{I} \cup \mathcal{I}' \cup \mathcal{I}'') && \text{(by Definition 5.1)}\\
&= \gamma(\mathcal{I} \cup \mathcal{I}') && (\mathcal{I}'' \text{ is redundant})\\
&= \gamma(\mathcal{I}_{\text{rest}} \cup (\mathcal{I}_{\text{cond}} \cup \mathcal{I}')) && (\text{as } \mathcal{I} = \mathcal{I}_{\text{rest}} \cup \mathcal{I}_{\text{cond}})\\
&= \gamma(\mathcal{I}_{\text{rest}}) \cap \gamma(\mathcal{I}_{\text{cond}} \cup \mathcal{I}') && (\gamma \text{ is meet-preserving})\\
&= \gamma(\mathcal{I}_{\text{rest}}) \cap \gamma(T_{\text{cond}}(\mathcal{I}_{\text{cond}})) && \text{(by Definition 5.1)}\\
&= \gamma(\mathcal{I}_{\text{rest}} \cup T_{\text{cond}}(\mathcal{I}_{\text{cond}})) && (\gamma \text{ is meet-preserving})\\
&= \gamma(T^d_{\text{cond}}(\mathcal{I})).
\end{aligned}$$

□

Note that we can strengthen the condition in Definition 5.1 by replacing $\mathcal{B}^*_{\text{cond}}$ with $\mathcal{B}_{\text{cond}}$. This makes it independent of permissible partitions but would reduce the class $\text{Cond}(\mathcal{D})$.

In Example 5.1, $T_1 \in \mathrm{Cond}(\mathcal{D})$ whereas $T_2 \notin \mathrm{Cond}(\mathcal{D})$ since T_2 does not keep the original constraints. Most standard transformers used in practice satisfy the two conditions and can thus be decomposed with our construction without losing any precision. The following example illustrates our construction for decomposing the standard conditional transformer T_{cond} in the Octagon domain.

Example 5.3. Consider

$$\mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X},\mathrm{Octagon}} : (3, \mathbb{U}^2 \times \{0\}, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{x_1 \leq 0, x_2 + x_3 \leq 0\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1\}, \{x_2, x_3\}\}.$$

Consider the conditional statement $x_3 \leq 0$ with $\mathcal{B}_{\mathrm{cond}} = \{x_3\}$. T_{cond} adds the constraint $x_3 \leq 0$ to $\mathcal{I}$ and then applies Octagon closure on the resulting element to produce the output:

$$T_{\mathrm{cond}}(\mathcal{I}) = \{x_1 \leq 0, x_2 + x_3 \leq 0, x_3 \leq 0, x_1 + x_3 \leq 0\},$$

which matches Definition 5.1 with $\mathcal{I}' = \{x_3 \leq 0\}$ and $\mathcal{I}'' = \{x_1 + x_3 \leq 0\}$.

Algorithm 1 computes $\mathcal{B}^*_{\mathrm{cond}} = \{x_2, x_3\}$, $\overline{\pi}_O = \{\{x_1\}, \{x_2, x_3\}\}$, $\mathcal{I}_{\mathrm{cond}} = \{x_2 + x_3 \leq 0\}$ and $\mathcal{I}_{\mathrm{rest}} = \{x_1 \leq 0\}$. The algorithm applies T_{cond} on $\mathcal{I}_{\mathrm{cond}}$ and keeps $\mathcal{I}_{\mathrm{rest}}$ untouched to produce:

$$T^d_{\mathrm{cond}}(\mathcal{I}) = \{x_1 \leq 0, x_2 + x_3 \leq 0, x_3 \leq 0\} \quad \text{with } \overline{\pi}_O.$$

Since $\mathcal{I}' \cup \mathcal{I}_{\mathrm{cond}} = T_{\mathrm{cond}}(\mathcal{I}_{\mathrm{cond}})$, T_{cond} satisfies the conditions for $\mathrm{Cond}(\mathcal{D})$ in this case and there is no change in precision.

Note that best transformers are not necessarily in $\mathrm{Cond}(\mathcal{D})$. This is due to constraints on the coefficient set $\mathcal{R}$ or the constant set $\mathcal{C}$ in $\mathcal{D}$. We provide an example of a domain $\mathcal{D}$ which does not have any best transformer in $\mathrm{Cond}(\mathcal{D})$.

Example 5.4. We consider a fictive domain

$$\mathcal{X} = \{x_1, x_2\} \text{ and } \mathcal{L}_{\mathcal{X},\mathcal{D}} : (2, \mathbb{Z}^2, \{\leq, =\}, \{0, 1, 1.5\}).$$

We assume $\mathcal{I} = \{x_1 \leq 1, x_2 \leq 1\}$ with permissible partition $\{\{x_1\}, \{x_2\}\}$ and the conditional $x_2 \leq 0.5$. In this case $\mathcal{B}^*_{\mathrm{cond}} = \{x_2\}$. Using only the constraints with variables in $\mathcal{B}^*_{\mathrm{cond}}$ yields $T_{\mathrm{cond}}(\mathcal{I}_{\mathrm{cond}}) = \{x_2 \leq 1\}$ as $0.5 \notin \mathcal{C}$. This means that the most precise result we can express which fits our conditions in the definition will be semantically equivalent to $\mathcal{I}$. However, a best transformer would produce an abstract element semantically equivalent to $\{x_1 \leq 1, x_2 \leq 1, x_1 + x_2 \leq 1.5\}$, which is more precise than $\mathcal{I}$. Thus, no best transformer is in $\mathrm{Cond}(\mathcal{D})$.

The following obvious corollary provides a condition under which the output partition $\overline{\pi}_{\mathcal{I}_O}$ computed by Algorithm 1 is finest, i.e., $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$.

Corollary 5.3. *For the conditional* $e \otimes c$, $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$, *if* $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$ *and* $\mathcal{I}_O = \mathcal{I} \cup \{e \otimes c\}$.

5.2 Assignment

We consider linear assignments of the form $x_j := e$ on an abstract element $\mathcal{I}$ with an associated permissible partition $\overline{\pi}_{\mathcal{I}}$ in $\mathcal{D}$ where $e := \sum_{i=1}^{n} a_i x_i + c$ with $a_i \in \mathbb{Z}$ and $c \in \mathbb{Q}, \mathbb{R}$. An assignment is *invertible* if $a_j \neq 0$ (for example $x_1 := x_1 + x_2$). We write $\mathcal{I}_{x_j} \subseteq \mathcal{I}$ for the subset of constraints that contain the variable x_j.

As discussed in Section 2, a number of existing domains are not closed under assignment. As for the conditional, the best assignment transformers are usually expensive and may need to be approximated. We provide an example, very similar to Example 5.1, that shows how approximation can affect decomposition.

Algorithm 2 Decomposed assignment transformer T^d_{assign}

```
1: function ASSIGNMENT($(\mathcal{I},\overline{\pi}_{\mathcal{I}}), \text{stmt}, T_{\text{assign}}$)
2:     $\mathcal{B}^*_{\text{assign}} := \text{compute\_block}(\text{stmt}, \overline{\pi}_{\mathcal{I}})$
3:     $\mathcal{I}_{\text{assign}} := \mathcal{I}(\mathcal{B}^*_{\text{assign}})$
4:     $\overline{\pi}_{\mathcal{I}_O} := \{\mathcal{A} \in \overline{\pi}_{\mathcal{I}} \mid \mathcal{A} \cap \mathcal{B}^*_{\text{assign}} = \emptyset\} \cup \{\mathcal{B}^*_{\text{assign}}\}$
5:     $\mathcal{I}_{\text{rest}} := \mathcal{I}(\mathcal{X} \setminus \mathcal{B}^*_{\text{assign}})$
6:     $\mathcal{I}_O := T_{\text{assign}}(\mathcal{I}_{\text{assign}}) \cup \mathcal{I}_{\text{rest}}$
7:     $\overline{\pi}_{\mathcal{I}_O} := \text{refine}(\overline{\pi}_{\mathcal{I}_O})$
8:     return $(\mathcal{I}_O, \overline{\pi}_{\mathcal{I}_O})$
```

Example 5.5. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{Polyhedra}} : (6, \mathbb{Z}^6, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{x_1 + x_2 = 0, x_3 + x_4 = 5, x_5 - x_3 = 0\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4, x_5\}, \{x_6\}\}.$$

For the assignment $x_5 := -x_6$, a best sound assignment transformer T_1 may return the decomposable output:

$$\mathcal{I}_O = \{x_1 + x_2 = 0, x_3 + x_4 = 5, x_5 + x_6 = 0\} \text{ with } \overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}.$$

However, another sound transformer T_2 may return the non-decomposable output:

$$\mathcal{I}'_O = \{x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 5\} \text{ with } \overline{\pi}_{\mathcal{I}'_O} = \pi_{\mathcal{I}'_O} = \top.$$

Let $\mathcal{B}_{\text{assign}} = \{x_i \mid a_i \neq 0\} \cup \{x_j\}$ be the set of variables affected by $e := \sum_{i=1}^{n} a_i x_i + c$ (we also include x_j). The block $\mathcal{B}^*_{\text{assign}} = \bigcup_{\mathcal{X}_k \cap \mathcal{B}_{\text{assign}} \neq \emptyset} \mathcal{X}_k$ fuses all blocks $\mathcal{X}_k \in \overline{\pi}_{\mathcal{I}}$ having non-empty intersection with $\mathcal{B}_{\text{assign}}$.

Example 5.6. Consider $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and an element $\mathcal{I}$ in the Polyhedra domain with $\overline{\pi}_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}$. For the assignment $x_3 := x_1 + x_2$, $\mathcal{B}_{\text{assign}} = \{x_1, x_2, x_3\}$ and $\mathcal{B}^*_{\text{assign}} = \{x_1, x_2, x_3, x_4\}$.

We briefly explain the standard assignment transformer as background to motivate the later definition of Assign($\mathcal{D}$) (the class of assignment transformers that do not lose precision with our decomposition).

Standard transformer: invertible assignment. The standard assignment transformer first removes all constraints in $\mathcal{I}_{x_j}$ from $\mathcal{I}$. It then computes a set of constraints $\mathcal{I}_{\text{inv}}$ by substituting, in all constraints in $\mathcal{I}_{x_j}$, x_j with $(x_j - \sum_{i \neq j} a_i x_i - c)/a_j$. Finally, $\mathcal{I}_{\text{inv}}$ may be approximated by a set $\mathcal{I}'_{\text{inv}}$ of representable constraints (in $\mathcal{D}$) over the same variable set. The result is $\mathcal{I}_O = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}'_{\text{inv}}$.

Standard transformer: non-invertible assignment. For a non-invertible assignment, the transformer also first removes $\mathcal{I}_{x_j}$ from $\mathcal{I}$. Next, it computes a set of constraints $\mathcal{I}_{\text{non-inv}}$ by *projecting out* x_j from all constraints in $\mathcal{I}_{x_j}$ using variable elimination [louis Imbert 1993]. Then it adds $\{x_j - e = 0\}$ to $\mathcal{I}_{\text{non-inv}}$. Finally, $\mathcal{I}_{\text{non-inv}}$ may be approximated by $\mathcal{I}'_{\text{non-inv}}$ to make it representable over the same variable set. The result is $\mathcal{I}_O = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}'_{\text{non-inv}}$.

Construction for assignment. Algorithm 2 shows our construction for decomposing a given assignment transformer T_{assign}. It operates analogous to the decomposed conditional transformer, except for the partition refinement in line 7, which we explain at the end of this section. The output of the decomposed transformer T^d_{assign} on $\mathcal{I}$ is $T^d_{\text{assign}}(\mathcal{I}) = T_{\text{assign}}(\mathcal{I}_{\text{assign}}) \cup \mathcal{I}_{\text{rest}}$. Next we define a class Assign($\mathcal{D}$) of assignment transformers T_{assign} where $\gamma(T_{\text{assign}}(\mathcal{I})) = \gamma(T^d_{\text{assign}}(\mathcal{I}))$ for all $\mathcal{I}$. Again, this will ensure soundness and monotonicity.

Definition 5.4. A (sound and monotone) assignment transformer T in $\mathcal{D}$ for the statement $x_j := e$ is in Assign($\mathcal{D}$) iff for any element $\mathcal{I}$ and any associated permissible partition $\overline{\pi}_{\mathcal{I}}$ in $\mathcal{D}$, the output $T_{\text{assign}}(\mathcal{I})$ satisfies the following conditions:

- $T_{\text{assign}}(\mathcal{I}) = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}' \cup \mathcal{I}''$ where $\mathcal{I}'$ contains non-redundant constraints between the variables from $\mathcal{B}^*_{\text{assign}}$ only, and $\mathcal{I}''$ is a set of redundant constraints between the variables in $\mathcal{X}$.
- $\gamma(T_{\text{assign}}(\mathcal{I}_{\text{assign}})) = \gamma((\mathcal{I}_{\text{assign}} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}')$.

Theorem 5.5. *If $T_{assign} \in$ Assign($\mathcal{D}$), then $\gamma(T_{assign}(\mathcal{I})) = \gamma(T^d_{assign}(\mathcal{I}))$ for all inputs $\mathcal{I}$ in $\mathcal{D}$. In particular, T^d_{assign} is sound and monotone.*

Proof. $\mathcal{B}^*_{\text{assign}}$ contains x_j by definition, thus $\mathcal{I} \setminus \mathcal{I}_{x_j} = \mathcal{I}_{\text{rest}} \cup (\mathcal{I}_{\text{assign}} \setminus \mathcal{I}_{x_j})$. We have,

$$\begin{aligned}
\gamma(T_{\text{assign}}(\mathcal{I})) &= \gamma((\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}' \cup \mathcal{I}'') && \text{(by Definition 5.4)}\\
&= \gamma((\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}') && (\mathcal{I}'' \text{ is redundant})\\
&= \gamma(\mathcal{I}_{\text{rest}} \cup (\mathcal{I}_{\text{assign}} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}') && \text{(from above)}\\
&= \gamma(\mathcal{I}_{\text{rest}}) \cap \gamma((\mathcal{I}_{\text{assign}} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}') && (\gamma \text{ is meet-preserving})\\
&= \gamma(\mathcal{I}_{\text{rest}}) \cap \gamma(T_{\text{assign}}(\mathcal{I}_{\text{assign}})) && \text{(by Definition 5.4)}\\
&= \gamma(\mathcal{I}_{\text{rest}} \cup T_{\text{assign}}(\mathcal{I}_{\text{assign}})) && (\gamma \text{ is meet-preserving})\\
&= \gamma(T^d_{\text{assign}}(\mathcal{I}))
\end{aligned}$$

□

As for Cond($\mathcal{D}$), Definition 5.4 can be tightened to make it independent of permissible partitions at the price of a smaller Assign($\mathcal{D}$).

In Example 5.5, $T_1 \in$ Assign($\mathcal{D}$) whereas $T_2 \notin$ Assign($\mathcal{D}$) as T_2 does not keep the constraints in $\mathcal{I} \setminus \mathcal{I}_{x_5}$. Most standard assignment transformers used in practice satisfy the two conditions and can thus be decomposed with our construction without losing any precision. The following example illustrates our construction for decomposing the standard assignment transformer T_{assign} in the TVPI domain.

Example 5.7. Consider

$$\mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X},\text{TVPI}} : (3, \mathbb{Z}^2 \times \{0\}, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{x_1 \leq 0, x_2 + x_3 \leq 0, x_3 \leq 3\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1\}, \{x_2, x_3\}\}.$$

Consider the non-invertible assignment $x_2 := 2x_3$ with $\mathcal{B}_{\text{assign}} = \{x_2, x_3\}$. T_{assign} determines that $\mathcal{I}_{x_2} = \{x_2 + x_3 \leq 0\}$, projects out x_2, which yields the empty set, and then adds $x_2 - 2x_3 = 0$ to obtain $\mathcal{I}_{\text{non-inv}}$, which is representable. Overall this results in $\{x_1 \leq 0, x_2 - 2x_3 = 0, x_3 \leq 3\}$.

Next, the transformer applies TVPI completion to produce the final output:

$$T_{\text{assign}}(\mathcal{I}) = \{x_1 \leq 0, x_2 - 2x_3 = 0, x_3 \leq 3, x_2 \leq 6, x_1 + x_2 \leq 6, x_1 + x_3 \leq 3, x_2 + x_3 \leq 9\},$$

which has the form of Definition 5.4 with

$$\mathcal{I}' = \{x_2 - 2x_3 = 0\} \text{ and } \mathcal{I}'' = \{x_2 \leq 6, x_1 + x_2 \leq 6, x_1 + x_3 \leq 3, x_2 + x_3 \leq 9\}.$$

Algorithm 2 computes $\mathcal{B}^*_{\text{assign}} = \{x_2, x_3\}$, $\mathcal{I}_{\text{assign}} = \{x_2 + x_3 \leq 0, x_3 \leq 3\}$, $\overline{\pi}_{\mathcal{I}_O} = \{\{x_1\}, \{x_2, x_3\}\}$, and $\mathcal{I}_{\text{rest}} = \{x_1 \leq 0\}$. The algorithm applies T_{assign} on $\mathcal{I}_{\text{assign}}$ and keeps $\mathcal{I}_{\text{rest}}$ untouched to produce:

$$\mathcal{I}_O = T^d_{\text{assign}}(\mathcal{I}) = \{x_1 \leq 0, x_2 - 2x_3 = 0, x_3 \leq 3, x_2 \leq 6, x_2 + x_3 \leq 9\} \quad \text{with } \overline{\pi}_{\mathcal{I}_O}.$$

Here $\mathcal{I}'$ contains non-redundant constraints between variables from $\mathcal{B}^*_{\text{assign}}$ only and we have $\gamma((\mathcal{I}_{\text{assign}} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}') = \gamma(\{x_2 - 2x_3 = 0, x_2 \leq 3\}) = \gamma(T_{\text{assign}}(\mathcal{I}_{\text{assign}}))$. Thus T_{assign} satisfies the conditions for Assign$(\mathcal{D})$ in this case and there is no loss of precision.

Refinement. So far we have assumed that line 7 of Algorithm 2 is the identity (that is, refinement does not affect $\overline{\pi}_{\mathcal{I}_O}$). We now discuss refinement in more detail.

Definition 5.6 (Refinement condition). The output partition $\overline{\pi}_{\mathcal{I}_O}$ of a non-invertible assignment transformer T_{assign} satisfying Definition 5.4 is a candidate for refinement if $\mathcal{X}_t \cap \mathcal{B}_{\text{assign}} = \{x_j\}$ where $\mathcal{X}_t$ is the block of $\overline{\pi}_{\mathcal{I}}$ containing x_j. Here, $\mathcal{I}$ is the abstract element upon which the transformer is applied and $\overline{\pi}_{\mathcal{I}}$ is a permissible partition.

If the above condition holds during analysis (it can be checked efficiently), then refinement can split $\mathcal{B}^*_{\text{assign}}$ from $\overline{\pi}_{\mathcal{I}_O}$ into two blocks $\mathcal{X}_t \setminus \{x_j\}$ and $\mathcal{B}^*_{\text{assign}} \setminus (\mathcal{X}_t \setminus \{x_j\})$, provided no redundant constraint ($\mathcal{I}''$ in Definition 5.4) fuses these two blocks. The result is a finer partition $\overline{\pi}_{\mathcal{I}_O}$.

Example 5.8. Consider

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5\}, \mathcal{L}_{\mathcal{X},\text{Zones}} : (5, \{1,0\} \times \{0,-1\} \times \{0\}^3, \{\leq, =\}, \mathbb{Q}),$$
$$\mathcal{I} = \{x_1 \leq x_2, x_2 \leq x_3, x_4 \leq x_5\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2, x_3\}, \{x_4, x_5\}\}.$$

Consider the non-invertible assignment $x_2 := x_4$ with $\mathcal{B}_{\text{assign}} = \{x_2, x_4\}$ and the standard Zones assignment transformer T_{assign}. Without refinement, we obtain the partition $\overline{\pi}_{\mathcal{I}_O} = \{\mathcal{X}\} = \top$. However, our refinement condition enables us to obtain a finer output partition. We have that $\mathcal{X}_t = \{x_1, x_2, x_3\}$ and $\mathcal{X}_t \cap \mathcal{B}_{\text{assign}} = \{x_2\}$ and thus the dynamic refinement condition applies, splitting the block $\mathcal{B}^*_{\text{assign}} = \mathcal{X}$ into two blocks: $\mathcal{X}_t \setminus \{x_2\} = \{x_1, x_3\}$ and $\mathcal{B}^*_{\text{assign}} \setminus (\mathcal{X}_t \setminus \{x_2\}) = \{x_2, x_4, x_5\}$. This produces a finer partition for the output:

$$\mathcal{I}_O = \{x_1 \leq x_3, x_2 - x_4 = 0, x_4 \leq x_5, x_2 \leq x_5\} \text{ with } \overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O} = \{\{x_1, x_3\}, \{x_2, x_4, x_5\}\}.$$

As with the conditional, $\overline{\pi}_{\mathcal{I}_O} \neq \pi_{\mathcal{I}_O}$ in general even if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$. The following corollary provides conditions under which $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$ after applying Algorithm 2.

Corollary 5.7. *For the assignment $x_j := e$, $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$ holds if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$ and, in the invertible case $\mathcal{I}_O = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}_{inv}$ or, in the non-invertible case $\mathcal{I}_O = (\mathcal{I} \setminus \mathcal{I}_{x_j}) \cup \mathcal{I}_{\text{non-inv}}$.*

5.3 Meet ($\sqcap$)

As discussed in Section 2, all domains we consider are closed under the meet ($\sqcap$) and thus it is common to implement a precise transformer. In fact, any meet transformer $T_\sqcap$ that obeys $T_\sqcap(\mathcal{I}, \mathcal{I}') \sqsubseteq \mathcal{I}, \mathcal{I}'$ is precise. Thus, we assume a given precise meet transformer, i.e., $\gamma(T_\sqcap(\mathcal{I}, \mathcal{I}')) = \gamma(\mathcal{I}) \cap \gamma(\mathcal{I}')$ for all $\mathcal{I}, \mathcal{I}'$. As a consequence, our decomposed construction will always yield an equivalent transformer, without any conditions.

Construction for meet ($\sqcap$). Algorithm 3 shows our construction of a decomposed transformer for a given meet transformer $T_\sqcap$ on input elements $\mathcal{I}, \mathcal{I}'$ with the respective permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ in domain $\mathcal{D}$. The algorithm computes a common permissible partition $\overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ for the inputs and then applies $T_\sqcap$ separately on the individual factors of $\mathcal{I}, \mathcal{I}'$ corresponding to the blocks in $\overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$.

Theorem 5.8. *$\gamma(T_\sqcap(\mathcal{I}, \mathcal{I}')) = \gamma(T^d_\sqcap(\mathcal{I}, \mathcal{I}'))$ for all inputs $\mathcal{I}, \mathcal{I}'$ in $\mathcal{D}$. In particular, $T^d_\sqcap$ is sound and monotone.*

Algorithm 3 Decomposed meet transformer $T^d_\sqcap$

```
function MEET($(\mathcal{I}, \overline{\pi}_\mathcal{I}), (\mathcal{I}', \overline{\pi}_{\mathcal{I}'}), T_\sqcap$)
    $\overline{\pi}_{\mathcal{I}_O} := \overline{\pi}_\mathcal{I} \sqcup \overline{\pi}_{\mathcal{I}'}$
    $\mathcal{I}_O := \bigcup_{\mathcal{A} \in \overline{\pi}_{\mathcal{I}_O}} T_\sqcap(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))$
    return $(\mathcal{I}_O, \overline{\pi}_{\mathcal{I}_O})$
```

Proof.

$$\begin{aligned}
\gamma(T_\sqcap(\mathcal{I}, \mathcal{I}')) &= \gamma(\mathcal{I}) \cap \gamma(\mathcal{I}') \\
&= \bigcap_{\mathcal{A}} \gamma(\mathcal{I}(\mathcal{A})) \cap \gamma(\mathcal{I}'(\mathcal{A})) && (\mathcal{I} = \bigcup \mathcal{I}(\mathcal{A}),\ \mathcal{I}' = \bigcup \mathcal{I}'(\mathcal{A})) \\
&= \bigcap_{\mathcal{A}} \gamma(T_\sqcap(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))) && (T_\sqcap \text{ is precise}) \\
&= \gamma(\bigcup_{\mathcal{A}} T_\sqcap(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))) && (\gamma \text{ is meet-preserving}) \\
&= \gamma(T^d_\sqcap(\mathcal{I}, \mathcal{I}'))
\end{aligned}$$

□

The following example illustrates the decomposition of a best meet transformer $T_\sqcap$ in the Octahedron domain using Algorithm 3.

Example 5.9. Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\text{Octahedron}} = (4, \mathbb{U}^4, \{\leq, =\}, \mathbb{Q}), \\
\mathcal{I} &= \{x_1 \leq 1, x_2 \leq 0, x_3 + x_4 \leq 1\} \text{ with } \overline{\pi}_\mathcal{I} = \pi_\mathcal{I} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\} \text{ and} \\
\mathcal{I}' &= \{x_1 - x_3 - x_4 \leq 2, x_2 \leq 1\} \text{ with } \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'} = \{\{x_1, x_3, x_4\}, \{x_2\}\}.
\end{aligned}$$

$T_\sqcap$ computes the union $\mathcal{I} \cup \mathcal{I}'$ and then removes redundant constraints to produce the output:

$$T_\sqcap(\mathcal{I}, \mathcal{I}') = \{x_1 \leq 1, x_2 \leq 0, x_3 + x_4 \leq 1, x_1 - x_3 - x_4 \leq 2\}.$$

Algorithm 3 computes the common permissible partition $\overline{\pi}_\mathcal{I} \sqcup \overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_3, x_4\}, \{x_2\}\}$ and applies $T_\sqcap$ separately on the factors of $\mathcal{I}, \mathcal{I}'$ corresponding to the common partition and produces:

$$\begin{aligned}
&T^d_\sqcap(\mathcal{I}, \mathcal{I}') = \{x_1 \leq 1, x_2 \leq 0, x_3 + x_4 \leq 1, x_1 - x_3 - x_4 \leq 2\} \\
&\text{with } \overline{\pi}_{\mathcal{I}_O} = \{\{x_1, x_3, x_4\}, \{x_2\}\}.
\end{aligned}$$

The following corollary provides conditions under which the output partition is finest.

Corollary 5.9. *$\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$ if $\overline{\pi}_\mathcal{I} = \pi_\mathcal{I}$, $\overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$, and $\mathcal{I}_O = \mathcal{I} \cup \mathcal{I}'$.*

5.4 Join (⊔)

As discussed in Section 2, none of the sub-polyhedra domains we consider are closed for the join (⊔). Thus, the join transformer approximates the union of $\mathcal{I}$ and $\mathcal{I}'$ in $\mathcal{D}$ and is usually the most expensive transformer in $\mathcal{D}$. As with other transformers, an arbitrary approximation can result in the ⊤ partition. The example below shows this situation with two sound join transformers in the Zones domain.

Algorithm 4 Decomposed join transformer $T_{\sqcup}^{d}$

```
function JOIN($(\mathcal{I}, \overline{\pi}_{\mathcal{I}}), (\mathcal{I}', \overline{\pi}_{\mathcal{I}'}), T_{\sqcup}$)
    $\overline{\pi}_{\text{common}} := \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$
    $\mathcal{N} = \bigcup\{\mathcal{A} \in \overline{\pi}_{\text{common}} \mid \mathcal{I}(\mathcal{A}) \neq \mathcal{I}'(\mathcal{A})\}$
    $\mathcal{I}_{\sqcup} := \mathcal{I}(\mathcal{N})$
    $\mathcal{I}'_{\sqcup} := \mathcal{I}'(\mathcal{N})$
    $\mathcal{I}_{\text{rest}} := \mathcal{I}(\mathcal{X} \setminus \mathcal{N})$
    $\overline{\pi}_{\mathcal{I}_O} := \{\mathcal{A} \in \overline{\pi}_{\text{common}} \mid \mathcal{A} \cap \mathcal{N} = \varnothing\} \cup \{\mathcal{N}\}$
    $\mathcal{I}_O := T_{\sqcup}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup}) \cup \mathcal{I}_{\text{rest}}$
    $\overline{\pi}_{\mathcal{I}_O} := \text{refine}(\overline{\pi}_{\mathcal{I}_O})$
    return $(\mathcal{I}_O, \overline{\pi}_{\mathcal{I}_O})$
```

Example 5.10. Consider

$$\begin{aligned}
\mathcal{X} &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \mathcal{L}_{\mathcal{X},\text{Zones}} : (6, \{1,0\} \times \{0,-1\} \times \{0\}^4, \{\leq, =\}, \mathbb{R}),\\
\mathcal{I} &= \{x_1 = 1, x_2 = 2, x_3 \leq 3, x_4 = 4, x_5 = 0, x_6 = 0\} \text{ and}\\
\mathcal{I}' &= \{x_1 = 1, x_2 = 2, x_3 \leq 3, x_4 = 4, x_5 = 1, x_6 = 1\} \text{ with}\\
\overline{\pi}_{\mathcal{I}} &= \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}} = \bot.
\end{aligned}$$

A sound transformer T_1 may return the decomposed output:

$$\mathcal{I}_O = \{x_1 = 1, x_2 = 2, x_3 \leq 3, x_4 = 4, -x_5 \leq 0, x_5 \leq 1, -x_6 \leq 0, x_6 \leq 1\} \quad \text{with } \overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O} = \bot.$$

Another sound transformer T_2 for the join could produce the output $\mathcal{I}'_O$ with the $\top$ partition:

$$\mathcal{I}'_O = \{x_2 - x_1 \leq 1, x_1 - x_5 \leq 1, x_3 - x_2 \leq 1, x_3 - x_4 \leq -1, x_5 - x_6 = 0\} \quad \text{with } \overline{\pi}_{\mathcal{I}'_O} = \pi_{\mathcal{I}'_O} = \top.$$

Let $\overline{\pi}_{\text{common}} := \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ and $\mathcal{N} = \bigcup\{\mathcal{A} \in \overline{\pi}_{\text{common}} \mid \mathcal{I}(\mathcal{A}) \neq \mathcal{I}'(\mathcal{A})\}$ be the union of all blocks for which the corresponding factors $\mathcal{I}(\mathcal{A})$ and $\mathcal{I}'(\mathcal{A})$ are not semantically equal. In Example 5.10, we have $\mathcal{N} = \{x_5, x_6\}$.

Construction for $\sqcup$. Algorithm 4 shows our construction of a decomposed join transformer for a given $T_{\sqcup}$ on input elements $\mathcal{I}, \mathcal{I}'$ with the respective permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ in domain $\mathcal{D}$. The algorithm first computes a common permissible partition $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$. For each block $\mathcal{A} \in \overline{\pi}_{\text{common}}$, it checks if the corresponding factors $\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A})$ are (semantically) equal. If equality holds, the algorithm adds $\mathcal{I}(\mathcal{A})$ to the output $\mathcal{I}_O$ and adds the corresponding block $\mathcal{A}$ to the partition $\overline{\pi}_{\mathcal{I}_O}$. Those not equal are collected in the bigger factors $\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup}$ on which $T_{\sqcup}$ is applied, which reduces complexity. The associated block in $\overline{\pi}_{\mathcal{I}_O}$ is $\mathcal{N}$. The possible partition refinement is explained at the end of this section.

In Algorithm 4 the output of the decomposed transformer $T_{\sqcup}^{d}$ on inputs $\mathcal{I}, \mathcal{I}'$ is computed as $T_{\sqcup}^{d}(\mathcal{I}, \mathcal{I}') = T_{\sqcup}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup}) \cup \mathcal{I}_{\text{rest}}$. Next we define a class $\text{Join}(\mathcal{D})$ of join transformers $T_{\sqcup}$ for which $\gamma(T_{\sqcup}(\mathcal{I}, \mathcal{I}')) = \gamma(T_{\sqcup}^{d}(\mathcal{I}, \mathcal{I}'))$ for all inputs $\mathcal{I}, \mathcal{I}'$ in $\mathcal{D}$. This ensures soundness and monotonicity.

Definition 5.10. A join transformer $T_{\sqcup}$ is in $\text{Join}(\mathcal{D})$ iff for all pairs of input elements $\mathcal{I}, \mathcal{I}'$ and all associated common permissible partitions $\overline{\pi}_{\text{common}}$, the output $T_{\sqcup}(\mathcal{I}, \mathcal{I}')$ satisfies the following conditions:

- $T_{\sqcup}(\mathcal{I}, \mathcal{I}') = \mathcal{I}_{\text{rest}} \cup \mathcal{J}' \cup \mathcal{J}''$ where $\mathcal{I}_{\text{rest}} = \mathcal{I}(\mathcal{X} \setminus \mathcal{N})$, $\mathcal{J}'$ contains non-redundant constraints between only the variables from $\mathcal{N}$ and $\mathcal{J}''$ contains redundant constraints between the variables in $\mathcal{X}$.
- $\gamma(T_{\sqcup}(\mathcal{I}_{\sqcup}, \mathcal{I}'_{\sqcup})) = \gamma(\mathcal{J}')$.

THEOREM 5.11. *If $T_{\sqcup} \in \text{Join}(\mathcal{D})$, then $\gamma(T_{\sqcup}(\mathcal{I}, \mathcal{I}')) = \gamma(T_{\sqcup}^{d}(\mathcal{I}, \mathcal{I}'))$ for all inputs $\mathcal{I}, \mathcal{I}'$ in $\mathcal{D}$. In particular, $T_{\sqcup}^{d}$ is sound and monotone.*

Proof.

$$\begin{aligned}
\gamma(T_\sqcup(\mathcal{I},\mathcal{I}')) &= \gamma(\mathcal{I}_{\text{rest}} \cup \mathcal{J}' \cup \mathcal{J}'') && \text{(by Definition 5.10)}\\
&= \gamma(\mathcal{I}_{\text{rest}} \cup \mathcal{J}') && (\mathcal{J}'' \text{ is redundant})\\
&= \gamma(\mathcal{I}_{\text{rest}}) \cap \gamma(\mathcal{J}') && (\gamma \text{ is meet-preserving})\\
&= \gamma(\mathcal{I}_{\text{rest}}) \cap \gamma(T_\sqcup(\mathcal{I}_\sqcup,\mathcal{I}'_\sqcup)) && \text{(by Definition 5.10)}\\
&= \gamma(\mathcal{I}_{\text{rest}} \cup T_\sqcup(\mathcal{I}_\sqcup,\mathcal{I}'_\sqcup)) && (\gamma \text{ is meet-preserving})\\
&= \gamma(T^d_\sqcup(\mathcal{I},\mathcal{I}'))
\end{aligned}$$

□

The join transformer T_1 in Example 5.10 is in Join($\mathcal{D}$) whereas T_2 is not in Join($\mathcal{D}$) as it does not keep the constraints in $\mathcal{I}_{\text{rest}}$. Most standard and best join transformers in practice satisfy the conditions for Join($\mathcal{D}$) and thus they are decomposable with our construction without any change in precision.

The following example illustrates the decomposition of a best join transformer $T_\sqcup$ in the Octagon domain using Algorithm 4.

Example 5.11. Consider

$$\begin{aligned}
&\mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{L}_{\mathcal{X},\text{Octagon}} : (3, \mathbb{U}^2 \times \{0\}, \{\leq, =\}, \mathbb{R}),\\
&\mathcal{I} = \{x_1 \leq 2, x_2 \leq 1, x_3 \leq 3\}, \mathcal{I}' = \{x_1 \leq 1, x_2 \leq 3, x_3 \leq 3\} \text{ with}\\
&\overline{\pi}_{\mathcal{I}} = \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}} = \bot.
\end{aligned}$$

$T_\sqcup$ performs strong closure on both $\mathcal{I}, \mathcal{I}'$ to produce:

$$\begin{aligned}
\mathcal{I}^* &= \{x_1 \leq 2, x_2 \leq 1, x_3 \leq 3, x_1 + x_2 \leq 3, x_1 + x_3 \leq 5, x_2 + x_3 \leq 4\}\\
\mathcal{I}'^* &= \{x_1 \leq 1, x_2 \leq 3, x_3 \leq 3, x_1 + x_2 \leq 4, x_1 + x_3 \leq 4, x_2 + x_3 \leq 6\}.
\end{aligned}$$

It then takes the pairwise maximum of bounds for each constraint to produce the output:

$$T_\sqcup(\mathcal{I},\mathcal{I}') = \{x_1 \leq 2, x_2 \leq 3, x_3 \leq 3, x_1 + x_2 \leq 4, x_1 + x_3 \leq 5, x_2 + x_3 \leq 6\}.$$

which matches Definition 5.10 with

$$\mathcal{I}_{\text{rest}} = \{x_3 \leq 3\}, \mathcal{J}' = \{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4\}, \text{ and } \mathcal{J}'' = \{x_1 + x_3 \leq 5, x_2 + x_3 \leq 6\}.$$

Since $\overline{\pi}_{\mathcal{I}} = \overline{\pi}_{\mathcal{I}'}$, we have $\overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}}$. Here $\mathcal{I}_1 \neq \mathcal{I}'_1, \mathcal{I}_2 \neq \mathcal{I}'_2$ and $\mathcal{I}_3 = \mathcal{I}'_3$. Algorithm 4 computes $\mathcal{N} = \{x_1, x_2\}$ and combines $\mathcal{I}_1, \mathcal{I}_2$ into a single factor $\mathcal{I}_\sqcup$. Similarly, it combines $\mathcal{I}'_1, \mathcal{I}'_2$ into $\mathcal{I}'_\sqcup$.

$$\mathcal{I}_\sqcup = \{x_1 \leq 2, x_2 \leq 1\}, \mathcal{I}'_\sqcup = \{x_1 \leq 1, x_2 \leq 3\}.$$

The algorithm applies $T_\sqcup$ only on $\mathcal{I}_\sqcup, \mathcal{I}'_\sqcup$ whereas $\mathcal{I}_3$ is added to the output directly:

$$\mathcal{I}_O = T^d_\sqcup(\mathcal{I},\mathcal{I}') = \{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4, x_3 \leq 3\} \text{ with } \overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O} = \{\{x_1, x_2\}, \{x_3\}\}.$$

In the example, $\mathcal{I}_{\text{rest}}$ contains non-redundant constraints only between the variables from $\mathcal{X} \setminus \mathcal{N}$, $\mathcal{J}'$ contains non-redundant constraints between only the variables from $\mathcal{N}$ and $\gamma(T_\sqcup(\mathcal{I}_\sqcup, \mathcal{I}'_\sqcup)) = \{x_1 \leq 2, x_2 \leq 3, x_1 + x_2 \leq 4\} = \gamma(\mathcal{J}')$, and thus $T_\sqcup$ satisfies the conditions for Join($\mathcal{D}$) in this case.

Refinement. We can sometimes refine the output partition $\overline{\pi}_{\mathcal{I}_O}$ after computing the output $\mathcal{I}_O$ without inspecting or modifying $\mathcal{I}_O$. Namely, if a variable x_i is unconstrained in either $\mathcal{I}$ or $\mathcal{I}'$, then it is also unconstrained in $\mathcal{I}_O$. $\overline{\pi}_{\mathcal{I}_O}$ can thus be refined by removing x_i from the block containing it and adding the singleton set $\{x_i\}$ to $\overline{\pi}_{\mathcal{I}_O}$. This refinement can be performed after applying $T_{\sqcup}^{d}$. The following theorem formalizes this refinement.

Theorem 5.12. *Let $\mathcal{I}, \mathcal{I}'$ be abstract elements in $\mathcal{D}$ with the associated permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ respectively. Let $\mathcal{U} = \{x \in \mathcal{X} \mid x$ is unconstrained in $\mathcal{I}$ or $\mathcal{I}'\} = \{u_1, \ldots, u_r\}$ and let $\overline{\pi}_{\mathcal{I}_O}$ as computed in line 7 of Algorithm 4. Then the following partition is permissible for the output $\mathcal{I}_O$:*

$$\overline{\pi}_{\mathcal{I}_O} \sqcap \{\mathcal{X} \setminus \mathcal{U}, \{u_1\}, \cdots, \{u_r\}\}.$$

The proof of Theorem 5.12 is immediate from the discussion above. Unlike other transformers, we do not know of conditions for checking whether $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$.

5.5 Widening ($\triangledown$)

The widening transformer $T_{\triangledown}$ is applied during analysis to accelerate convergence towards a fixpoint. It is a binary transformer that guarantees: (i) the output satisfies $T_{\triangledown}(\mathcal{I}, \mathcal{I}') \sqsupseteq \mathcal{I}, \mathcal{I}'$, and (ii) the analysis terminates after a finite number of steps. In general, widening transformers are not monotonic or commutative. Further, best widening transformers do not exist for any numerical domain. In theory, it may be possible to design arbitrary widening transformers that always result in the $\top$ partition. In practice, the standard widening transformers are of two types:

Syntactic. For syntactic widening [Miné 2002, 2006], the output satisfies $\mathcal{I}_O \subseteq \mathcal{I}$. A constraint $\iota := \sum_{i=1}^{n} a_i x_i \leq c \in \mathcal{I}$ is in the output $\mathcal{I}_O$ iff there is a constraint $\iota' := \sum_{i=1}^{n} a_i x_i \leq c' \in \mathcal{I}'$ with the same linear expression and $c' \leq c$.

Semantic. The semantic widening [Cousot et al. 2005] requires the set of constraints in the input $\mathcal{I}$ to be non-redundant. The output satisfies $\mathcal{I}_O \subseteq \mathcal{I} \cup \mathcal{I}'$. Specifically, $\mathcal{I}_O$ contains the constraints from $\mathcal{I}$ that are satisfied by $\mathcal{I}'$ and the constraints ι' from $\mathcal{I}'$ that are mutually redundant with a constraint ι in $\mathcal{I}$.

Both these transformers are decomposable in practice. The following example illustrates the semantic and the syntactic widening on the Octagon domain.

Example 5.12. Consider

$$\begin{aligned}
&\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\text{Octagon}} : (4, \mathbb{U}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}),\\
&\mathcal{I} = \{x_1 - x_2 = 0, x_2 = 0, x_3 \leq 0, x_4 \leq 1\} \text{ with } \overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\} \text{ and}\\
&\mathcal{I}' = \{x_1 = 0, x_3 + x_4 \leq 2\} \text{ with } \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\}.
\end{aligned}$$

The semantic widening transformer T_1 yields:

$$\mathcal{I}_O = \{x_1 = 0\} \text{ with } \overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O} = \bot.$$

On the other hand, the syntactic widening transformer T_2 yields:

$$\mathcal{I}'_O = \varnothing \text{ with } \overline{\pi}_{\mathcal{I}'_O} = \pi_{\mathcal{I}'_O} = \bot.$$

Thus, both are decomposable in this case.

Construction for widening. Algorithm 5 shows our construction of a decomposed widening transformer on input elements $\mathcal{I}, \mathcal{I}'$ with respective permissible partitions $\overline{\pi}_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'}$ in $\mathcal{D}$. The algorithm computes a common permissible partition $\overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$ and then applies the widening transformer $T_{\triangledown}$ separately on the individual factors of $\mathcal{I}, \mathcal{I}'$ corresponding to the blocks of $\overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$. The refinement of the out partition is explained at the end of this section.

Algorithm 5 Decomposed widening transformer $T^d_\triangledown$

```
function WIDENING($(\mathcal{I}, \overline{\pi}_\mathcal{I}), (\mathcal{I}', \overline{\pi}_{\mathcal{I}'}), T_\triangledown$)
    $\overline{\pi}_{\mathcal{I}_O} := \overline{\pi}_\mathcal{I} \sqcup \overline{\pi}_{\mathcal{I}'}$
    $\mathcal{I}_O := \bigcup_{\mathcal{A} \in \overline{\pi}_{\mathcal{I}_O}} T_\triangledown(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))$
    $\overline{\pi}_{\mathcal{I}_O} :=$ refine($\overline{\pi}_{\mathcal{I}_O}$)
    return $(\mathcal{I}_O, \overline{\pi}_{\mathcal{I}_O})$
```

Next, we define a class Widen($\mathcal{D}$) of widening transformers $T_\triangledown$ for which $\gamma(T_\triangledown(\mathcal{I}, \mathcal{I}')) = \gamma(T^d_\triangledown(\mathcal{I}, \mathcal{I}'))$ for all inputs $\mathcal{I}, \mathcal{I}'$ in $\mathcal{D}$.

Definition 5.13. A widening transformer $T_\triangledown$ is in Widen($\mathcal{D}$) iff for all pairs of input elements $\mathcal{I}, \mathcal{I}'$ and all associated common permissible partitions $\overline{\pi}_{\text{common}}$, the output $T_\triangledown(\mathcal{I}, \mathcal{I}')$ satisfies:

$$\gamma(T_\triangledown(\mathcal{I}, \mathcal{I}')) = \bigcap_{\mathcal{A}} \gamma(T_\triangledown(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))).$$

THEOREM 5.14. *If* $T_\triangledown \in$ Widen($\mathcal{D}$)*, then* $\gamma(T_\triangledown(\mathcal{I}, \mathcal{I}')) = \gamma(T^d_\triangledown(\mathcal{I}, \mathcal{I}'))$ *for all inputs* $\mathcal{I}, \mathcal{I}'$ *in* $\mathcal{D}$*. Thus,* $T^d_\triangledown$ *is sound.*

PROOF.

$$\begin{aligned}\gamma(T_\triangledown(\mathcal{I}, \mathcal{I}')) &= \bigcap_{\mathcal{A}} \gamma(T_\triangledown(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))) \qquad \text{(by Definition 5.13)}\\ &= \gamma(\bigcup_{\mathcal{A}} T_\triangledown(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A}))) \qquad (\gamma \text{ is meet-preserving})\\ &= \gamma(T^d_\triangledown(\mathcal{I}, \mathcal{I}'))\end{aligned}$$

□

Both syntactic and semantic Octagon widening transformers from Example 5.12 are in Widen($\mathcal{D}$). It can be shown that the standard transformers in existing domains are in Widen($\mathcal{D}$). For syntactic widening, $\gamma(\mathcal{I}) = \gamma(\mathcal{I}')$ does not imply $\gamma(T_\triangledown(\mathcal{I}'', \mathcal{I})) = \gamma(T_\triangledown(\mathcal{I}'', \mathcal{I}'))$ in general, and thus fixpoint equivalence is not guaranteed with the decomposed transformer. The following example illustrates the decomposition of the standard semantic TVPI widening transformer $T_\triangledown$ using Algorithm 5.

Example 5.13. Consider

$$\begin{aligned}&\mathcal{X} = \{x_1, x_2, x_3, x_4\}, \mathcal{L}_{\mathcal{X},\text{TVPI}} = (\mathbb{Z}^2 \times \{0\}^2, \{\leq, =\}, \mathbb{Q}),\\ &\mathcal{I} = \{x_1 = 1, x_2 = 0, x_3 + x_4 \leq 1\}, \text{ with } \overline{\pi}_\mathcal{I} = \{\{x_1\}, \{x_2\}, \{x_3, x_4\}\} \text{ and}\\ &\mathcal{I}' = \{2x_1 - 3x_2 \leq 2, x_1 + x_2 = 1, x_3 \leq 0, x_4 \leq 0\} \text{ with } \overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}\}.\end{aligned}$$

$T_\triangledown$ keeps the constraint $x_3 + x_4 \leq 1$ from $\mathcal{I}$ as it is satisfied by $\mathcal{I}'$ (using $x_3 \leq 0, x_4 \leq 0$). It also adds the constraint $x_1 + x_2 = 1$ from $\mathcal{I}'$ to the output as it is mutually redundant with the constraint $x_1 = 1$ in $\mathcal{I}$. The output of $T_\triangledown$ is:

$$T_\triangledown(\mathcal{I}, \mathcal{I}') = \{x_1 + x_2 = 1, x_3 + x_4 \leq 1\}.$$

Algorithm 5 computes the common permissible partition $\mathcal{I}_O = \overline{\pi}_\mathcal{I} \sqcup \overline{\pi}_{\mathcal{I}'} = \{\{x_1, x_2\}, \{x_3, x_4\}\}$ and then computes the output $\mathcal{I}_O$ by applying $T_\triangledown$ separately on the individual factors of $\mathcal{I}, \mathcal{I}'$ corresponding to the blocks of $\overline{\pi}_{\text{common}}$:

$$\mathcal{I}_O = T^d_\triangledown(\mathcal{I}, \mathcal{I}') = \{x_1 + x_2 \leq 1, x_3 + x_4 \leq 1\} \text{ with } \overline{\pi}_{\mathcal{I}_O} = \{\{x_1, x_2\}, \{x_3, x_4\}\}.$$

Here $\gamma(T_\triangledown(\mathcal{I}, \mathcal{I}')) = \bigcap_{\mathcal{A}} \gamma(T_\triangledown(\mathcal{I}(\mathcal{A}), \mathcal{I}'(\mathcal{A})))$ and thus $T_\triangledown$ is in Widen($\mathcal{D}$).

Refinement. $T_{\triangledown}$ in Algorithm 5 does not create constraints between variables in different blocks of the common partition in the output $\mathcal{I}_O$. By construction $\overline{\pi}_{\mathcal{I}_O} = \overline{\pi}_{\text{common}} = \overline{\pi}_{\mathcal{I}} \sqcup \overline{\pi}_{\mathcal{I}'}$. For syntactic widening, $\mathcal{I}_O \subseteq \mathcal{I}$ and thus the output partition $\overline{\pi}_{\mathcal{I}_O}$ can be refined to $\overline{\pi}_{\mathcal{I}}$ after computing the output $\mathcal{I}_O$.

The following corollaries provide conditions when $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$ for the semantic and the syntactic widening respectively.

Corollary 5.15. *For semantic widening, $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$ if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}, \overline{\pi}_{\mathcal{I}'} = \pi_{\mathcal{I}'}$ and $\mathcal{I}_O = \mathcal{I} \cup \mathcal{I}'$.*

Corollary 5.16. *For syntactic widening, $\overline{\pi}_{\mathcal{I}_O} = \pi_{\mathcal{I}_O}$ if $\overline{\pi}_{\mathcal{I}} = \pi_{\mathcal{I}}$ and $\mathcal{I}_O = \mathcal{I}$.*

6 EXPERIMENTAL EVALUATION

In this section we evaluate the performance of our generic decomposition approach on three popular domains: Polyhedra, Octagon, and Zones. Using standard implementations of these domains, we show that our decomposition of their transformers leads to substantial performance improvements, often surpassing existing transformers designed for specific domains.

Our decomposed implementation for these domains is available as part (i.e., an update) of the ELINA library [eli]. Below, we compare to the prior ELINA as described in [Singh et al. 2015, 2017].

Experimental setup. All of our experiments were performed on a 3.5 GHz Intel Quad Core i7-4771 Haswell CPU. The machine has L1, L2, and L3 caches of sizes 256 KB, 1024 KB, and 8192 KB, respectively, and 16 GB of main memory. Turbo boost and hyper threading were disabled for consistency of measurements. All libraries were compiled with gcc 5.2.1 using the flags `-O3 -m64 -march=native`. We used a time limit of four hours for our experiments.

Benchmarks. The benchmarks were taken from the popular software verification competition [Beyer 2016]. The benchmark suite is divided into categories suited for different kinds of analyses, e.g., pointer, array, numerical, and others. We chose two categories suited for numerical analysis: (i) Linux Device Drivers (LD), and (ii) Control Flow (CF). Each of these categories contains hundreds of benchmarks and we evaluated the performance of our analysis on each of these. We use the crab-llvm analyzer which is part of the SeaHorn verification framework [Gurfinkel et al. 2015] for performing the analysis. The analyzer is written in C++ and performs intraprocedural analysis of LLVM bitcode. The analyzer explicitly checks for unconstrained variables during runtime and removes them. Thus, the total number of variables for Polyhedra, Octagon, and Zones can be different on the same benchmark.

6.1 Polyhedra

The standard implementation of the Polyhedra domain is based on the double representation method, i.e., it maintains both the constraints and the generator representation. This is because transformers such as meet are cheap with the constraint representation but expensive with the generator representation. On the other hand transformers such as join are cheap with the generator representation but expensive with the constraint representation. The Polyhedra analysis thus applies the domain transformer on one representation and then updates the other representation using a standard conversion algorithm [Chernikova 1968; Verge 1994]. The standard implementation contains the best conditional, assignment, meet, and join transformers together with a semantic widening transformer. All these transformers are in the classes of decomposable transformers defined in Section 5.

Table 2 shows the asymptotic complexity of Polyhedra transformers in the standard implementation with and without decomposition [Singh et al. 2017]. For the non-decomposed column in the table, n is the number of variables, m is the number of constraints, and g is the number of

Table 2. Asymptotic time complexity of the Polyhedra transformers with and without decomposition.

Transformer	Non-Decomposed	Decomposed
Conditional	$O(n)$	$O(n_{\max})$
Assignment	$O(ng)$	$O(n_{\max}g_{\max})$
Meet ($\sqcap$)	$O(nm)$	$O(\sum_{i=1}^{r} n_i m_i)$
Join ($\sqcup$)	$O(ng)$	$O(\sum_{i=1}^{r} n_i g_i m_i + n_{\max}g_{\max})$
Widening (∇)	$O(ngm)$	$O(\sum_{i=1}^{r} n_i g_i m_i)$
Conversion	$O(\exp(n, g))$	$O(\sum_{i=1}^{r} \exp(n_i, g_i))$

generators. In the decomposed column, r is the number of blocks in the partition, n_i is the number of variables in the ith block, $n_{\max}$ is the number of variables in the largest block, m_i and g_i are the number of constraints and generators, respectively, in the ith factor and $g_{\max}$ is the number of generators in the largest factor. It holds that $n = \sum_{i=1}^{r} n_i$, $m = \sum_{i=1}^{r} m_i$ and $g = \prod_{i=1}^{r} g_i$. We also show the complexity of the conversion algorithm for converting from the constraints to the generators. It has the same exponential complexity (in terms of n and g) for conversion in either direction. Thus, it is the most expensive operation in the standard implementation.

We compare the runtime and memory consumption for end-to-end Polyhedra analysis with our generic decomposed transformers versus the original non-decomposed transformers from the Parma Polyhedra Library (PPL) [Bagnara et al. 2008] and the decomposed transformers from ELINA [Singh et al. 2017]. PPL, ELINA, and our implementation store the constraints and the generators using matrices with 64-bit integers. PPL stores a single matrix for either representation whereas both ELINA and our implementation use a set of matrices corresponding to the factors, which requires exponential space in the worst case.

Table 3 shows the results on 13 large, representative benchmarks. These benchmarks were chosen based on the following criteria:

- The most time consuming function in the benchmark did not produce any integer overflow with ELINA or our approach.
- The benchmark ran for at least 2 minutes with PPL.

Our decomposition maintains semantic equivalence with both ELINA and PPL as long as there is no integer overflow and thus gets the same semantic invariants. All three implementations set the abstract element to $\top$ when an integer overflow occurs. The total number of integer overflows on the chosen benchmarks were 58, 23 and 21 for PPL, ELINA, and our decomposition, respectively. We also had fewer integer overflows than both ELINA and PPL on the remaining benchmarks. Thus, our decomposition improves in some cases also the precision of the analysis with respect to both ELINA and PPL.

Table 3 shows our experimental findings. The entry *MO* (memory-out) in the table means that the analysis ran out of memory whereas the entry *TO* (time-out) means the analysis did not finish within four hours. Whenever there is memory overflow and our analysis finishes, we show the corresponding speedup as ∞, because the analysis can never finish on the given machine even if given arbitrary time. The speedups in case of a time-out are lower bounds obtained by assuming optimistically that PPL had finished right after four hours.

In the table, PPL either ran out of memory or did not finish within four hours on 8 out of the 13 benchmarks. Both ELINA and our decomposition are able to analyze all benchmarks. We are faster than ELINA on all benchmarks with a maximum speedup of 5.9x on the `P19_159` benchmark. We also save significant memory over ELINA. The speedups on the remaining (not shown) benchmarks over the decomposed version of ELINA varies from 1.1x to 4x with an average of about 1.4x..

Table 3. Speedup for the Polyhedra domain analysis with our decomposition over PPL and ELINA.

Benchmark	PPL		ELINA		Our Decomposition		Speedup vs.	
	time(s)	memory(GB)	time(s)	memory(GB)	time(s)	memory(GB)	PPL	ELINA
`firewire_firedtv`	331	0.9	0.4	0.2	0.2	0.2	1527	2
`net_fddi_skfp`	6142	7.2	9.2	0.9	4.4	0.3	1386	2
`mtd_ubi`	MO	MO	4	0.9	1.9	0.3	∞	2.1
`usb_core_main0`	4003	1.4	65	2	29	0.7	136	2.2
`tty_synclinkmp`	MO	MO	3.4	0.1	2.5	0.1	∞	1.4
`scsi_advansys`	TO	TO	4	0.4	3.4	0.2	>4183	1.2
`staging_vt6656`	TO	TO	2	0.4	0.5	0.1	>28800	4
`net_ppp`	10530	0.1	924	0.3	891	0.1	11.8	1
`p10_l00`	121	0.9	11	0.8	5.4	0.2	22.4	2
`p16_l40`	MO	MO	11	3	2.9	0.4	∞	3.8
`p12_l57`	MO	MO	14	0.8	6.5	0.3	∞	2.1
`p13_l53`	MO	MO	54	2.7	25	0.9	∞	2.2
`p19_l59`	MO	MO	70	1.7	12	0.6	∞	5.9

Table 4. Partition statistics for the Polyhedra domain analysis.

Benchmark	Category	LOC	n		$n_{\max}^{\text{elina}}$		$n_{\max}^{\text{our}}$		$n_{\max}^{\text{finest}}$	
			max	avg	max	avg	max	avg	max	avg
`firewire_firedtv`	LD	14506	159	25	81	7	40	4	39	3
`net_fddi_skfp`	LD	30186	589	88	111	25	45	9	13	4
`mtd_ubi`	LD	39334	528	59	111	14	28	5	23	4
`usb_core_main0`	LD	52152	365	72	267	30	60	11	40	7
`tty_synclinkmp`	LD	19288	332	49	48	10	40	6	26	4
`scsi_advansys`	LD	21538	282	63	117	18	49	12	41	9
`staging_vt6656`	LD	25340	675	53	204	17	25	4	12	3
`net_ppp`	LD	15744	218	58	112	40	51	28	43	20
`p10_l00`	CF	592	303	174	234	54	79	16	14	6
`p16_l40`	CF	1783	874	266	86	31	39	14	5	3
`p12_l57`	CF	4828	921	261	461	78	21	7	4	3
`p13_l53`	CF	5816	1631	342	617	111	26	10	9	3
`p19_l59`	CF	9794	1272	358	867	187	31	8	12	3

Better partitioning leads to performance improvements. Table 4 shows further statistics about the category (LD or CF) and the number of lines of code in each benchmark. As can be seen, the benchmarks are quite large and contain up to 50K lines of code. Further, after each join, we measured the total number of variables n and report the maximum and the average. For the decomposed analyses (ELINA and ours) we measured the size of the largest block and report again maximum and average under $n_{\max}^{\text{elina}}$, $n_{\max}^{\text{our}}$. To assess the quality of the partitions, we also computed (with the needed overhead) the finest partition after each join and show the largest blocks under $n_{\max}^{\text{finest}}$ (maximum and average). As can be observed, our partitions are strictly finer than the ones produced by ELINA on all benchmarks due to the refinements for the assignment and join transformers. Moreover, it can be seen that our partitions are sometimes close to the finest partition but in many cases there is room for further improvement (an interesting item for future work).

Table 5. Asymptotic time complexity of the Octagon transformers with and without decomposition.

Transformer	Non-Decomposed	Decomposed
Conditional	$O(n^2)$	$O(n_{\max}^2)$
Assignment	$O(n^2)$	$O(n_{\max}^2)$
Meet ($\sqcap$)	$O(n^2)$	$O(\sum_{i=1}^{r} n_i^2)$
Join ($\sqcup$)	$O(n^2)$	$O(\sum_{i=1}^{r} n_i^2)$
Widening ($\triangledown$)	$O(n^2)$	$O(\sum_{i=1}^{r} n_i^2)$
Strong Closure	$O(n^3)$	$O(\sum_{i=1}^{r} n_i^3)$

6.2 Octagon

The standard implementation of the Octagon domain works only with the constraint representation and approximates the best conditional and best assignment transformers but implements best join and meet transformers. The widening is defined syntactically. All of these transformers are in the classes of (decomposable) transformers from Section 5. Since the syntactic widening does not produce semantically equivalent outputs for semantically equivalent but syntactically different inputs, our fixpoint can be different than the one computed by non-decomposed analysis. However, we still get the same semantic invariants at fixpoint on most of our benchmarks. The standard implementation requires a strong closure operation for the efficiency and precision of transformers such as join, conditional, assignment, and others.

Table 5 shows the asymptotic complexity of standard Octagon transformers as well as the strong closure operation with and without decomposition [Singh et al. 2015]. In the table, $n, n_i, n_{\max}$ have the same meaning as in Table 2. In can be seen that strong closure is the most expensive operation in this domain with cubic complexity. It is possible to apply it incrementally for the conditional and the assignment transformers.

We compare the performance of our approach for the standard Octagon analysis, using the non-decomposed ELINA (ELINA-ND) and the decomposed (ELINA-D) transformers from ELINA. All of these implementations store the constraint representation using a single matrix with 64-bit doubles. The matrix requires quadratic space in n. Thus, overall memory consumption is the same for all implementations.

We compare the runtime and report speedups for the end-to-end Octagon analysis in Table 6. We achieve up to 40x speedup for the end-to-end analysis over the non-decomposed implementation. More importantly, we are either faster or have the same runtime as the decomposed version of ELINA on all benchmarks but one. The maximum speedup over the decomposed version of ELINA is 2.2x. The speedups on the remaining (not shown) benchmarks vary between 1x and 1.6x with an average of about 1.2x. Notice that on the `mtd_ubi` benchmark, the Octagon analysis takes longer than the Polyhedra analysis. This is because the Octagon widening takes longer to converge compared to the Polyhedra widening.

Table 7 shows the partition statistics for the Octagon analysis (as we did for the Polyhedra analysis). It can be seen that while our refinements often produce finer partitions than the decomposed version of ELINA, they are coarser on 3 of the 13 benchmarks. This is because the decomposed transformers in ELINA are specialized for the standard approximations of the conditional and assignment transformers. We still achieve comparable performance on these benchmarks. Note that the partitions are close to the finest in most cases.

Table 6. Speedup for the Octagon domain analysis with our decomposition over the non-decomposed and the decomposed versions of ELINA.

Benchmark	ELINA-ND	ELINA-D	Our Decomposition	Speedup vs.	
	time(s)	time(s)	time(s)	ELINA-ND	ELINA-D
`firewire_firedtv`	0.4	0.07	0.07	5.7	1
`net_fddi_skfp`	28	2.6	1.9	15	1.4
`mtd_ubi`	3411	979	532	6.4	1.8
`usb_core_main0`	107	6.1	4.9	22	1.2
`tty_synclinkmp`	8.2	1	0.8	10	1.2
`scsi_advansys`	9.3	1.5	0.8	12	1.9
`staging_vt6656`	4.8	0.3	0.2	24	1.5
`net_ppp`	11	1.1	1.2	9.2	0.9
`p10_l00`	20	0.5	0.5	40	1
`p16_l40`	8.8	0.6	0.5	18	1.2
`p12_l57`	19	1.2	0.7	27	1.7
`p13_l53`	43	1.7	1.3	33	1.3
`p19_l59`	41	2.8	1.2	31	2.2

Table 7. Partition statistics for the Octagon domain analysis.

Benchmark	Category	LOC	n		$n_{\max}^{\text{elina}}$		$n_{\max}^{\text{our}}$		$n_{\max}^{\text{finest}}$	
			max	avg	max	avg	max	avg	max	avg
`firewire_firedtv`	LD	14506	159	25	31	6	40	4	27	3
`net_fddi_skfp`	LD	30186	573	86	49	18	30	10	14	7
`mtd_ubi`	LD	39334	553	46	111	65	22	9	16	9
`usb_core_main0`	LD	52152	364	72	59	22	39	9	35	7
`tty_synclinkmp`	LD	19288	324	49	84	15	26	6	25	4
`scsi_advansys`	LD	21538	293	64	94	19	41	6	20	5
`staging_vt6656`	LD	25340	651	52	63	7	25	4	14	3
`net_ppp`	LD	15744	218	54	40	23	55	29	39	19
`p10_l00`	CF	592	305	173	19	10	77	16	17	9
`p16_l40`	CF	1783	874	266	32	12	13	7	10	5
`p12_l57`	CF	4828	954	265	55	15	13	4	11	4
`p13_l53`	CF	5816	1635	337	41	12	22	7	10	5
`p19_l59`	CF	9794	1291	363	79	14	22	4	18	3

6.3 Zones

The standard Zones domain uses only the constraint representation. The conditional and assignment transformers are approximate whereas the meet and join are best transformers [Miné 2002]. The widening is defined syntactically. All of these transformers are in the class of (decomposable) transformers from Section 5. As for Octagon, fixpoint equivalence is not guaranteed due to widening being defined syntactically. However, we still get the same semantic invariants at fixpoint on most of our benchmarks. As for the Octagon domain, a cubic closure operation is required. The domain transformers have the same asymptotic complexity as in the Octagon domain.

We implemented both, a non-decomposed version of the transformers as well as a version with our decomposition method of the standard transformers. Both implementations store the

Table 8. Speedup for the Zones domain analysis with our decomposition over non-decomposed implementation.

Benchmark	Non-Decomposed	Our Decomposition	Speedup vs.
	time(s)	time(s)	Non-Decomposed
firewire_firedtv	0.05	0.05	1
net_fddi_skfp	3	1.5	2
mtd_ubi	1.4	0.7	2
usb_core_main0	10.3	4.6	2.2
tty_synclinkmp	1.1	0.7	1.6
scsi_advansys	0.9	0.7	1.3
staging_vt6656	0.5	0.2	2.5
net_ppp	1.1	0.7	1.5
p10_l00	1.9	0.4	4.6
p16_l40	1.7	0.7	2.5
p12_l57	3.5	0.9	3.9
p13_l53	8.7	2.1	4.2
p19_l59	9.8	1.6	6.1

Table 9. Partition statistics for the Zones domain analysis.

Benchmark	Category	LOC	n		$n_{\max}^{\text{our}}$		$n_{\max}^{\text{finest}}$	
			max	avg	max	avg	max	avg
firewire_firedtv	LD	14506	159	25	40	4	17	3
net_fddi_skfp	LD	30186	578	88	30	9	13	5
mtd_ubi	LD	39334	553	59	23	5	14	3
usb_core_main0	LD	52152	362	71	37	8	33	7
tty_synclinkmp	LD	19288	328	49	26	6	25	5
scsi_advansys	LD	21538	293	65	41	8	21	7
staging_vt6656	LD	25340	675	53	25	3	13	2
net_ppp	LD	15744	219	58	54	29	47	24
p10_l00	CF	592	303	174	77	16	17	8
p16_l40	CF	1783	856	261	13	7	10	6
p12_l57	CF	4828	882	249	12	4	10	3
p13_l53	CF	5816	1557	317	22	7	20	5
p19_l59	CF	9794	1243	331	14	4	13	3

constraints using a single matrix with 64-bit doubles that requires quadratic space in n. We compare the runtime and report speedups for the Zones analysis in Table 8. Our decomposition achieves speedups of up to 6x over the non-decomposed implementation. The speedups over the remaining benchmarks not shown in the table vary between 1.1x and 5x with an average of about 1.6x.

Table 9 shows the partition statistics for the Zones analysis. It can be seen that partitioning is the core reason for the speed-ups obtained and that the partitions are close to the finest in most cases.

6.4 Summary

Overall, our results show that the generic decomposition method proposed in this paper works well. It speeds up analysis compared to non-decomposed domains significantly, and, importantly, the more expensive the domain, the higher the speed-ups. Our generic method also compares favorably with the prior manually decomposed domains provided by ELINA due to refined partitioning. We also show that the partitions computed during analysis are close to optimal for Octagon and Zones but with further room for improvement for Polyhedra. The challenge is how to obtain those with reasonable cost. Further speed-ups can also be obtained by different implementations of the transformers that are, for example, selectively approximate to achieve finer partitions.

7 RELATED WORK

We discussed dynamic partitioning specialized for standard implementations of sub-polyhedra domains throughout the paper [Halbwachs et al. 2003; Singh et al. 2015, 2017]. We now discuss other related work related to improving the performance of numerical domain analysis.

Variable packing [Blanchet et al. 2003; Heo et al. 2016] has been used for decomposing the Octagon transformers. It partitions $\mathcal{X}$ statically before running the analysis based on certain criteria. For example, two variables are in the same block of the partition if they occur together in the same program statement. Although variable packing could also be generalized to decompose transformers of other domains, it is fundamentally different from our dynamic decomposition. Namely, in many cases the enforced static partition would not be permissible throughout analysis in our framework and thus it loses precision. Further, the dynamic decomposition often yields even finer partitions than can be detected statically. So dynamic decomposition (of transformers within the classes defined) provides both higher precision and faster execution. The work of [Venet and Brat 2004] dynamically maintains partitions based on a syntactic criteria for the Zones domain. The generated transformers are less precise than the ones produced by our approach. The works of [Gange et al. 2016] and [Jourdan 2017] are focussed on designing sparse algorithms for standard transformers of Zones and Octagon. While these algorithms cannot be extended to more expressive domains, they could be combined with our decomposition to potentially achieve better performance.

The work of [Maréchal et al. 2017] and [Maréchal and Périn 2017] focuses on improving the performance of standard Polyhedra transformers based on constraint representation using parametric linear programming. We believe their transformers could benefit from our decomposition approach. Both [Simon and King 2005] and [Miné et al. 2010] focus on improving the performance of the best join transformer in the Polyhedra domain based on the constraint representation. In [Simon and King 2005] the authors exploit sparsity by noticing that a given variable occurs only a few times in the constraint representations of the Polyhedra. If the output becomes too large, they approximate. Frequent calls to the linear solver limit the performance of their approach. In [Miné et al. 2010] the authors decompose the best join transformer by decomposing the inputs into two parts each. The join transformer is then applied on one of the pieces. The partitions obtained with this method are very coarse and thus the decomposed transformer has worse performance than achieved using our decomposition.

8 CONCLUSION

Partitioning abstract elements is a promising avenue to make abstract domain analysis faster, possibly by orders of magnitude, and thus practical for many real world program analysis and verification tasks. This is made possible thanks to the inherent "locality" in the way program statements, and sequences of such, access variables. This paper advances partitioning by showing that it is generally applicable to all sub-polyhedra domains and shows how to construct decomposed transformers from existing, non-decomposed transformers. This way, existing implementations can be re-factored to incorporate decomposition. We also showed that our decomposition does not lose precision on most practical transformers already in use. Finally, we provided techniques to refine the output partitions of important transformers in certain cases, an improvement over prior work. We evaluated our approach on three expensive abstract domains: Zones, Octagon, and Polyhedra and showed significant speed-ups compared to prior work, including domains that were previously manually decomposed. Most importantly, the more expensive a domain is, the higher the speed-ups offered by domain decomposition, reaching orders of magnitude for Polyhedra. This means that decomposition can level the playing field among domains, requiring rethinking of the fundamental question which domain to use for which application.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback. This research was supported by the Swiss National Science Foundation (SNF) grant number 163117.

REFERENCES

ELINA: ETH Library for Numerical Analysis. http://elina.ethz.ch.

R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 887–904, 2016.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

N. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282 – 293, 1968.

R. ClarisÃş and J. Cortadella. The octahedron abstract domain. *Science of Computer Programming*, 64:115 – 139, 2007.

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. International Symposium on Programming*, pages 106–130, 1976.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. Principles of Programming Languages (POPL)*, pages 84–96, 1978.

R. Cousot, R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1):28 – 56, 2005.

P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .net. *SIGPLAN Not.*, 43:329–346, 2008.

G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Exploiting sparsity in difference-bound matrices. In *Proc. Static Analysis Symposium (SAS)*, pages 189–211, 2016.

R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, Mar. 2000.

A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *Proc. Computer Aided Verification (CAV)*, pages 343–361, 2015.

N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 355–365, 2003.

N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design (FMSD)*, 29(1):79–95, 2006.

K. Heo, H. Oh, and H. Yang. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 237–256, 2016.

J. M. Howe and A. King. Logahedra: A new weakly relational domain. In *Proc. Automated Technology for Verification and Analysis (ATVA)*, pages 306–320, 2009.

J.-H. Jourdan. Sparsity preserving algorithms for octagons. *Electronic Notes in Theoretical Computer Science*, 331:57 – 70, 2017. Workshop on Numerical and Symbolic Abstract Domains (NSAD).

M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *Proc. Symposium on Applied Computing (SCP)*, pages 184–188, 2008.

J. louis Imbert. Fourier's elimination: Which to choose? *Principles and Practice of Constraint Programming*, pages 117–129, 1993.

A. Maréchal and M. Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *Proc. Verification, Model Checking, and Abstract Interpretation, (VMCAI)*, pages 367–385, 2017.

A. Maréchal, D. Monniaux, and M. Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *Proc. Static Analysis Symposium (SAS)*, pages 212–231, 2017.

A. Miné. A few graph-based relational numerical abstract domains. In *Proc. Static Analysis Symposium (SAS)*, pages 117–132, 2002.

A. Miné. The octagon abstract domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006.

A. Miné, E. RodrÃŋguez-Carbonell, and A. Simon. Speeding up polyhedral analysis by identifying common constraints. *Electronic Notes in Theoretical Computer Science*, 267(1):127 – 138, 2010.

F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In *Proc. European Symposium on Programming (ESOP)*, pages 18–32, 2004.

A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 336–351, 2005.

A. Simon and A. King. The two variable per inequality abstract domain. *Higher Order Symbolic Computation (HOSC)*, 23: 87–143, 2010.

G. Singh, M. Püschel, and M. Vechev. Making numerical program analysis fast. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 303–313, 2015.

G. Singh, M. Püschel, and M. Vechev. Fast polyhedra abstract domain. In *Proc. Principles of Programming Languages (POPL)*, pages 46–59, 2017.

A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 231–242, 2004.

H. L. Verge. A note on Chernikova's algorithm. Technical report, 1994.