A New Abstraction Framework for Affine Transformers*

Tushar Sharma[1] and Thomas Reps[1,2] **

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. This paper addresses the problem of abstracting a set of affine transformers $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, where $\overrightarrow{v}$ and $\overrightarrow{v}'$ represent the pre-state and post-state, respectively. We introduce a framework to harness any base abstract domain $\mathcal{B}$ in an abstract domain of affine transformations. Abstract domains are usually used to define constraints on the variables of a program. In this paper, however, abstract domain $\mathcal{B}$ is repurposed to constrain the elements of C and $\overrightarrow{d}$—thereby defining a set of affine transformers on program states. This framework facilitates intra- and interprocedural analyses to obtain function and loop summaries, as well as to prove program assertions.

1 Introduction

Most critical applications, such as airplane and rocket controllers, need correctness guarantees. Usually these correctness guarantees can be described as safety properties in the form of assertions. Verifying an assertion amounts to showing that the assertion holds true for all possible runs of an application. Proving an assertion is, in general, an undecidable problem. Nevertheless, there exist static-analysis techniques that are able to verify automatically some kinds of program assertions. One such technique is abstract interpretation [3], which soundly abstracts the concrete executions of the program to elements in an abstract domain, and checks the correctness guarantees using the abstraction.

In this paper, we provide analysis techniques to abstract the behavior of the program as a set of affine transformations over bit-vectors. An affine transformer is a relation on states, defined by $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, where $\overrightarrow{v}'$ and $\overrightarrow{v}$ are row vectors that represent the post-transformation state and the pre-transformation state, respectively. C is the linear component of the transformation and $\overrightarrow{d}$ is a constant vector. For example, $\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix} + \begin{bmatrix} 10 & 0 \end{bmatrix}$ denotes the affine

* Supported, in part, by a gift from Rajiv and Ritu Batra; DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.

transformation ($x' = x + 2y + 10 \wedge y' = 0$) over variables $\{x, y\}$. We denote an affine transformation by $C : \overrightarrow{d}$. The paper is based on the following observation:

Observation 1 *Abstract domains are usually used to define constraints on the variables of a program. However, they can be re-purposed to constrain the elements of* $C : \overrightarrow{d}$ *—thereby defining a set of affine transformers on program states.*

The need for abstraction over affine transformers. Abstractions of affine transformers can be used to obtain affine-relation invariants at each program point in the program [12]. An affine relation is a linear-equality constraint between numeric-valued variables of the form $\sum_{i=1}^{n} a_i v_i + b = 0$. For a given set of variables $\{v_i\}$, affine-relation analysis (ARA) identifies affine relations that are invariants of a program. The results of ARA can be used to determine a more precise abstract value for a variable via *semantic reduction* [4], or detect the relationship between program variables and loop-counter variables.

Furthermore, when the abstract-domain elements are abstractions of affine transformers, abstract interpretation can be used to provide useful function summaries or loop summaries [2, 18]. In principle, summaries can be computed offline for large libraries of code so that client static analyses can use them to provide verification results more efficiently.

Previous work [6] compared two abstract domains for affine-relation analysis over bitvectors: (i) an affine-closed abstraction of relations over program variables (AG), and (ii) an affine-closed abstraction of affine transformers over program variables (MOS). Müller-Olm and Seidl [13] introduced the MOS domain, whose elements are the affine-closed sets of affine transformers. An MOS element can be represented by a set of square matrices. Each matrix T is an affine transformer of the form $T = \left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right]$, which represents the state transformation $\overrightarrow{v}' := \overrightarrow{v} \cdot C + \overrightarrow{d}$, or, equivalently, $\left[1 \middle| \overrightarrow{v}'\right] := \left[1 \middle| \overrightarrow{v}\right] T$. In [6], the authors observe that the MOS domain can encode two-vocabulary relations that are not affine-closed even though the affine transformers themselves are affine closed. (See §2.5 for an example.) Thus, moving the abstraction from affine relations over program variables to affine relations over affine transformations possibly offers some advantages because it allows some non-affine-closed sets to be representable.

While the MOS domain is useful for finding affine-relation invariants in a program, the join operation used at confluence points can lose precision in many cases, leading to imprecise function summaries. Furthermore, the analysis does not scale well as the number of variables in the vocabulary increases. In other words, it has one baked-in performance-versus-precision aspect.

Problem Statement. Our goal is to generalize the ideas used in the MOS domain—in particular, to have an abstraction of *sets of affine transformers*—but to provide a way for a client of the abstract domain to have some control over the performance/precision trade-off. Toward this end, we define a new family of numerical abstract domains, denoted by ATA$[\mathcal{B}]$. (ATA stands for Affine-Transformers Abstraction.) Following Obs. 1, ATA$[\mathcal{B}]$ is parameterized by a

base numerical abstract domain $\mathcal{B}$, and allows one to represent a set of affine transformers (or, alternatively, certain disjunctions of transition formulas).

Summary of the Approach. Let the $(k+k^2)$-tuple $(d_1, d_2, \ldots, d_k, c_{11}, c_{12}, \ldots, c_{1k}, c_{21}, c_{22}, ..., c_{kk})$ denote the affine transformation $\bigwedge_{j=1}^{k} \left(v'_j = \sum_{i=1}^{k} (c_{ij} v_i) + d_j \right)$, also written as "$C : \overrightarrow{d}$." The key idea is that we will use $(k + k^2)$ symbolic constants to represent the $(k+k^2)$ coefficients in a transformation of the form $C : \overrightarrow{d}$, and use a base abstract domain $\mathcal{B}$—provided by a client of the framework—to represent *sets* of possible values for these symbolic constants. In particular, $\mathcal{B}$ is an abstract domain for which, for all $b \in \mathcal{B}$, $\gamma(b)$ is a set of $(k + k^2)$-tuples—each tuple of which provides values for $\{d_i\} \cup \{c_{ij}\}$, and can thus be interpreted as an affine transformation $C : \overrightarrow{d}$.

With this approach, a given $b \in \mathcal{B}$ represents the disjunction $\bigvee\{(C : \overrightarrow{d}) \in \gamma(b)\}$. When $\mathcal{B}$ is a non-relational domain, each $b \in \mathcal{B}$ constrains the values of $\{d_i\} \cup \{c_{ij}\}$ *independently*. When $\mathcal{B}$ is a relational domain, each $b \in \mathcal{B}$ can impose *intra-component* constraints on the allowed tuples $(d_1, d_2, \ldots, d_k, c_{11}, c_{12}, \ldots, c_{1k}, c_{21}, c_{22}, \ldots, c_{kk})$.

ATA[$\mathcal{B}$] generalizes the MOS domain, in the sense that the MOS domain is exactly ATA[AG], where AG is a relational abstract domain that captures affine equalities of the form $\sum_i a_i k_i = b$, where $a_i, b \in Z_{2^w}$ and $\mathbb{Z}_{2^w}$ is the set of w-bit bitvectors [9, 6] (see §2.4). For instance, an element in ATA[AG] can capture the set of affine transformers "$x' = k_1 * x + k_1 * y + k_2$, where k_1 is odd, k_2 is even, and k_1 is the coefficient of both x and y." On the other hand, an element in the abstract domain ATA[$\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+k^2)}$], where $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+k^2)}$ is the abstract domain of $(k + k^2)$-tuples of intervals over bitvectors, can capture a set of affine transformers such as $x' = k_3 * x + k_4 * y + k_5$, where $k_3 \in [0, 1]$, $k_4 \in [2, 2]$, and $k_5 \in [0, 10]$.

This paper addresses a wide variety of issues that arise in defining the ATA[$\mathcal{B}$] framework, including describing the abstract-domain operations of ATA[$\mathcal{B}$] in terms of the abstract-domain operations available in the base domain $\mathcal{B}$.

Contributions. The overall contribution of our work is the framework ATA[$\mathcal{B}$], for which we present

- methods to perform basic abstract-domain operations, such as equality and join.
- a method to perform abstract composition, which is needed to perform abstract interpretation.
- a faster method to perform abstract composition when the base domain is non-relational.

§2 introduces the terminology used in the paper; and presents some needed background material. §3 demonstrates the framework with the help of an example. §4 formally introduces the parameterized abstract domain ATA[$\mathcal{B}$]. §5 provides discussion and related work. Proofs are given in Appendices A and B of [19].

2 Preliminaries

All numeric values in this paper are integers in $\mathbb{Z}_{2^w}$ for some bit width w. That is, values are w-bit machine integers with the standard operations for machine addition and multiplication. Addition and multiplication in $\mathbb{Z}_{2^w}$ form a ring, not a field, so some facets of standard linear algebra do not apply.

Throughout the paper, k is the size of the *vocabulary* $V = \{v_1, v_2, .., v_k\}$—i.e., the variable-set under analysis. We use $\overrightarrow{v}$ to denote the vector $[v_1 v_2 .. v_k]$ of variables in vocabulary V. A *two-vocabulary* relation $R[V; V']$ is a transition relation between values of variables in the *pre-state* vocabulary V and values of variables in the *post-state* vocabulary V'. For instance, a transition relation $R[V; V']$ in the concrete collecting semantics is a subset of $\mathbb{Z}_{2^w}^k \times \mathbb{Z}_{2^w}^k$ (which is isomorphic to $\mathbb{Z}_{2^w}^{2k}$).

Matrix addition and multiplication are defined as usual, forming a matrix ring. We denote the transpose of a matrix M by M^t. A *one-vocabulary matrix* is a matrix with $k+1$ columns. A *two-vocabulary matrix* is a matrix with $2k+1$ columns. In each case, the "+1" is related to the fact that we capture affine rather than linear relations. I_n denotes the $n \times n$ identity matrix. Given a matrix C, we use $C[i, j]$ to refer to the entry at the i-th column and j-th row of C. Given a vector $\overrightarrow{d}$, we use $\overrightarrow{d}[j]$ to refer to the j-th entry in $\overrightarrow{d}$.

2.1 Affine Programs

$$
\begin{array}{rcl}
\langle Block \rangle & :: & l : (\langle Stmt \rangle \ \mathbf{;})^* \ \langle Next \rangle \\
\langle Next \rangle & :: & \mathbf{jump}\ l\mathbf{;} \\
 & | & \mathbf{jump}\ \langle Cond \rangle\ \mathbf{?}\ l_1\ \mathbf{:}\ l_2 \\
\langle Cond \rangle & :: & ? \mid \langle Expr \rangle\ Op\ \langle Expr \rangle \\
\langle Op \rangle & :: & = \mid \neq \mid \geqslant \mid \leqslant \\
\langle Expr \rangle & :: & c_0 + \sum_{i=1}^{k} c_i * v_i \\
\langle Stmt \rangle & :: & v_j := \langle Expr \rangle \\
 & | & v_j := ?
\end{array}
$$

We borrow the notion of affine programs from [13]. We restrict our affine programs to consist of a single procedure. The statements are restricted to either affine assignments or non-deterministic assignments. The control-flow instruction consists of either an unconditional jump statement, or a conditional jump with an affine equality, an affine disequality, an affine inequality, or unknown guard condition.

2.2 Abstract-Domain Operations

The two important steps in abstract interpretation (AI) are:

1. Abstraction: The abstraction of the program is constructed using the abstract domain and abstract semantics.
2. Fixpoint analysis: Fixpoint iteration is performed on the abstraction of the program to identify invariants.

For the purpose of our analysis, the program is abstracted to a control-flow graph, where each edge in the graph is labeled with an abstract transformer. An abstract transformer is a *two-vocabulary* transition relation $R[V; V']$. Concrete states described by an abstract transformer are represented by row vectors of

Table 1. Abstract-domain operations.

Type	Operation	Description
$\mathcal{A}$	$\bot$	*bottom element*
bool	$(a_1 == a_2)$	*equality*
$\mathcal{A}$	$(a_1 \sqcup a_2)$	*join*
$\mathcal{A}$	$(a_1 \nabla a_2)$	*widen*
$\mathcal{A}$	Id	*identity element*

Type	Operation	Description
$\mathcal{A}$	$\alpha(v_j :=?)$	*abstraction for nondeterministic assignments*
$\mathcal{A}$	$\alpha(v_j := c_0 + \sum_{i=1}^{k} c_{ij} * v_i)$	*abstraction for affine assignments*
$\mathcal{A}$	$(a_1 \circ a_2)$	*composition*

length $2k$. A (two-vocabulary) concrete state is sometimes called an *assignment* to the variables of the pre-state and the post-state vocabulary.

Tab. 1 lists the abstract-domain operations needed to generate the program abstraction and perform fixpoint analysis on it. Bottom, equality, and join are standard abstract-domain operations. The *widen* operation is needed for domains with infinite ascending chains to ensure termination. The two operations of the form $\alpha(Stmt)$ perform abstraction on an assignment statement $Stmt$ to generate an abstract transformer. Id is the identity element; which represents the identity transformation $(\bigwedge_{i=1}^{k} v_i' = v_i)$. Finally, the abstract-composition operation $a_1 \circ a_2$ returns a sound overapproximation of the composition of the abstract transformation a_1 with the abstract transformation a_2.

2.3 The Müller-Olm/Seidl Domain

An element in the Müller-Olm/Seidl domain (MOS) is an affine-closed set of affine transformers, as detailed in [13]. An MOS element is represented by a set of $(k+1)$-by-$(k+1)$ matrices. Each matrix T is a one-vocabulary transformer of the form $T = \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$, which represents the state transformation $\overrightarrow{v}' := \overrightarrow{v} \cdot M + b$, or, equivalently, $\left[1 \middle| \overrightarrow{v}'\right] := \left[1 \middle| \overrightarrow{v}\right] T$.

An MOS element $\mathcal{M}$, consisting of a set of matrices, represents the affine span of the set, denoted by $\langle \mathcal{M} \rangle$. $\langle \mathcal{M} \rangle$ is defined as follows: $\langle \mathcal{M} \rangle \stackrel{\text{def}}{=} \left\{ T \;\middle|\; \exists \overrightarrow{u} \in \mathbb{Z}_{2^w}^{|\mathcal{M}|} : T = \sum_{M \in \mathcal{M}} u_M M \wedge T_{1,1} = 1 \right\}$. The meaning of $\mathcal{M}$ is the union of the graphs of the affine transformers in $\langle \mathcal{M} \rangle$. Thus, $\gamma_{\text{MOS}}(\mathcal{M}) \stackrel{\text{def}}{=} \left\{ (\overrightarrow{v}, \overrightarrow{v}') \;\middle|\; \overrightarrow{v}, \overrightarrow{v}' \in \mathbb{Z}_{2^w}^{k} \wedge \exists T \in \langle \mathcal{M} \rangle : \left[1 \middle| v\right] T = \left[1 \middle| v'\right] \right\}$.

Example 1. If $w = 4$, the MOS element $\mathcal{M} = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 2 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\}$ represents the affine span $\langle \mathcal{M} \rangle = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 2 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 4 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \ldots, \left[\begin{array}{c|cc} 1 & 0 & 12 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 14 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\}$, which corresponds to the transition relation in which $v_1' = v_1$, v_2 can have any value, and v_2' can have any even value. □

Tab. 2 gives the abstract-domain operations for the MOS domain. The bottom element of the MOS domain is the empty set $\varnothing$, and the MOS element that represents the identity relation is the singleton set $\{I\}$. The equality check can be done by checking if the span of the matrices in the two values is equal. [6] provides an normal form for the MOS domain, which can be used to reduce the equality

check to *syntactic* equality checks on the matrices in M_1 and M_2. The widening operation is not needed for MOS because it is a finite-height lattice. The abstraction operation for the affine-assignment statement $\alpha(v_j := d_0 + \sum_{i=1}^{k} c_{ij} * v_i)$ gives back an MOS-element with a single matrix where every variable $v \in V - \{v_j\}$ is left unchanged, and the variable v_j is transformed to reflect the assignment by updating the corresponding column in the matrix with the assignment coefficients. The abstraction operation for the non-deterministic assignment statement $\alpha(v_j :=?)$ gives back an MOS-element containing two matrices. Similar to the abstraction for affine assignment operation, every variable $v \in V - v_j$ is left unchanged in both the matrices. v_j is set to 0 in the first and and 1 in the second matrix. The affine-closed set of these two matrices ensures that v_j is assigned to non-deterministically. The abstract-composition operation performs multiplication for each pair of the matrices in M_1 and M_2.

Table 2. Abstract-domain operations for the MOS-domain.

Type	Operation	Description
$\mathcal{A}$	$\bot_{\text{MOS}}$	$\varnothing$
bool	$(M_1 == M_2)$	$\langle M_1 \rangle == \langle M_2 \rangle$
$\mathcal{A}$	$(M_1 \sqcup M_2)$	$M_1 \cup M_2$
$\mathcal{A}$	$(a_1 \nabla a_2)$	*not applicable*
$\mathcal{A}$	$\alpha(v_j := d_0 + \sum_{i=1}^{k} c_{ij} * v_i)$	$\left\{ \left[\begin{array}{c\|ccc} 1 & 0 & d_0 & 0 \\ \hline 0 & I_{j-1} & [c_{1j}, c_{2j}, \ldots c_{(j-1)j}]^t & 0 \\ 0 & 0 & c_{jj} & 0 \\ 0 & 0 & [c_{(j+1)j}, c_{(j+2)j}, \ldots c_{kj}]^t & I_{k-j} \end{array}\right] \right\}$
$\mathcal{A}$	$\alpha(v_j :=?)$	$\left\{ \left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & I_{j-1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{k-j} \end{array}\right], \left[\begin{array}{c\|ccc} 1 & 0 & 1 & 0 \\ \hline 0 & I_{j-1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{k-j} \end{array}\right] \right\}$
$\mathcal{A}$	Id	$\{I_{k+1}\}$
$\mathcal{A}$	$(M_1 \circ M_2)$	$\{A_2 A_1 \mid A_i \in M_i\}$

2.4 The Affine-Generator Domain

An element in the Affine Generator domain ($\text{AG}[\overrightarrow{v}; \overrightarrow{v}']$) is a two-vocabulary matrix whose rows are the affine generators of a two-vocabulary relation over variables $\overrightarrow{v}$. An $\text{AG}[\overrightarrow{v}; \overrightarrow{v}']$ element is an r-by-$(2k+1)$ matrix G, with $0 < r \leqslant 2k+1$. The concretization of an $\text{AG}[\overrightarrow{v}; \overrightarrow{v}']$ element is

$$\gamma_{\text{AG}}(G) \stackrel{\text{def}}{=} \left\{ (\overrightarrow{v}, \overrightarrow{v}') \mid \overrightarrow{v}, \overrightarrow{v}' \in \mathbb{Z}_{2^w}^k \wedge \left[1 | v \; v'\right] \in \text{row}\, G \right\}.$$

The *row space* of a matrix G is defined by $\text{row}\, G \stackrel{\text{def}}{=} \{r \mid \exists \overrightarrow{u} : \overrightarrow{u} G = r\}$.

The $\text{AG}[\overrightarrow{v}; \overrightarrow{v}']$ domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states.

The bottom element of the AG domain is the empty matrix, and the $\text{AG}[\overrightarrow{v}; \overrightarrow{v}']$ element that represents the identity relation is the matrix $\left[\begin{array}{c\|cc} {\scriptstyle 1} & {\scriptstyle \overrightarrow{v}} & {\scriptstyle \overrightarrow{v}'} \\ \hline 1 & I & I \end{array}\right]$.

The AG$[\{v_1, v_2\}; \{v_1', v_2'\}]$ element $\left[\begin{array}{c|cccc} & 1 & v_1 & v_2 & v_1' & v_2' \\ \hline 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 \end{array}\right]$ represents the transition relation in which $v_1' = v_1$, v_2 can have any value, and v_2' can have any even value.

To compute the join of two AG elements, stack the two matrices vertically and get the canonical form of the result [6, §2.1].

2.5 Relating MOS and AG

There are two ways to relate the MOS and AG domains. One way is to use them as abstractions of two-vocabulary relations and provide (approximate) interconversion methods. The other is to use a variant of the AG domain to represent the elements of the MOS domain exactly.

Comparison of MOS and AG elements as abstraction of two-vocabulary relations. As shown in [6, §4.1], the MOS and AG domains are incomparable: some relations are expressible in each domain that are not expressible in the other. Intuitively, the central difference is that MOS is a domain of sets of *functions*, while AG is a domain of *relations*.

AG can capture 1-vocabulary guards on both the pre-state and post-state vocabularies, while MOS can capture 1-vocabulary guards only on its post-state vocabulary.

Example 2. For example, when $k = 1$, the AG element for "assume $x = 2$" is $\left[\begin{array}{c|cc} 1 & x & x' \\ \hline 1 & 2 & 2 \end{array}\right]$, i.e., "$x = 2 \wedge x' = 2$". In contrast, there is no MOS element that represents $x = 2 \wedge x' = 2$. The smallest MOS element that over-approximates "assume $x = 2$" is the identity transformer $\left\{\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right]\right\}$. □

On the other hand, the MOS-domain can encode two-vocabulary relations that are not affine-closed.

Example 3. One example is the matrix basis $M = \left\{\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right]\right\}$. The set that M encodes is

$$\begin{aligned} \gamma_{\text{MOS}}(M) &= \left\{ [x\ y\ x'\ y'] \;\middle|\; \begin{array}{c} \exists u_0, u_1 : \left[\begin{array}{c|cc} 1 & x & y \end{array}\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & u_0 & u_0 \\ 0 & u_1 & u_1 \end{array}\right] = \left[\begin{array}{c|cc} 1 & x' & y' \end{array}\right] \\ \wedge\ u_0 + u_1 = 1 \end{array} \right\} \\ &= \left\{ [x\ y\ x'\ y'] \,\middle|\, \exists u_0 : x' = y' = u_0 x + (1 - u_0) y \right\} \\ &= \left\{ [x\ y\ x'\ y'] \,\middle|\, \exists u_0 : x' = y' = x + (1 - u_0)(y - x) \right\} \\ &= \left\{ [x\ y\ x'\ y'] \,\middle|\, \exists p : x' = y' = x + p(y - x) \right\} \end{aligned} \tag{1}$$

Affine spaces are closed under affine combinations of their elements. Thus, $\gamma_{\text{MOS}}(M)$ is not an affine space because some affine combinations of its elements are not in $\gamma_{\text{MOS}}(M)$. For instance, let $a = \begin{bmatrix} 1 & -1 & 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 2 & -2 & 6 & 6 \end{bmatrix}$, and $c = \begin{bmatrix} 0 & 0 & -4 & -4 \end{bmatrix}$. By Eqn. (1), we have $a \in \gamma_{\text{MOS}}(M)$ when $p = 0$ in Eqn. (1), $b \in \gamma_{\text{MOS}}(M)$ when $p = -1$, and $c \notin \gamma_{\text{MOS}}(M)$ (the equation "$-4 = 0 + p(0 - 0)$"

has no solution for p). Moreover, $2a - b = c$, so c is an affine combination of a and b. Thus, $\gamma_{\text{MOS}}(M)$ is not closed under affine combinations of its elements, and so $\gamma_{\text{MOS}}(M)$ is not an affine space. □

Soundly converting an MOS element M to an overapproximating AG element is equivalent to stating two-vocabulary affine constraints satisfied by M [6, §4.2]).

Reformulation of MOS elements as AG elements. An MOS element $M = \{M_1, M_2, ..., M_n\}$ represents the set of $(k+1)\times(k+1)$ matrices in the affine closure of the matrices in M. Each matrix can be thought of as a $(k+1)\times(k+1)$ vector, and hence M can be represented by an AG element of size $n \times ((k+1)\times(k+1))$.

Example 4. Tab. 3 shows the two ways MOS and AG elements can be related. Column 1 shows the MOS element M from Ex. 3, which represents the set of matrices in the affine closure of the two $(k+1)\times(k+1)$ matrices, with $k = 2$. The second column gives the AG element A_1 (a matrix with $2k+1$ columns) representing the affine-closed space over $\{x, y, x', y'\}$ satisfied by M. Consequently, $\gamma_{\text{AG}}(A_1) \supseteq \gamma_{\text{MOS}}(M)$. Column 3 shows the two matrices of M as the $2 \times ((k+1)\times(k+1))$ AG element A_2. Because A_2 is just a reformulation of M, $\gamma_{\text{AG}}(A_2) = \gamma_{\text{MOS}}(M)$. □

Table 3. Example demonstrating two ways of relating MOS and AG.

MOS element (M)	Overapproximating AG element (A_1)	Reformulation as abstraction over affine transformers (A_2)
$\left\{ \begin{bmatrix} 1 & x & y \\ \hline 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & x & y \\ \hline 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \right\}$	$\left[\begin{array}{c\|cccccc} 1 & x & y & x' & y' \\ \hline 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{array}\right]$	$\left[\begin{array}{c\|cccccccc} 1 & a_{01} & a_{02} & a_{10} & a_{11} & a_{12} & a_{20} & a_{21} & a_{22} \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right]$

3 Overview

In this section, we motivate and illustrate the ATA[$\mathcal{B}$] framework, with the help of several examples. The first two examples illustrate the following principle, which restates Obs. 1 more formally:

Observation 2 *Each affine transformation $C : \overrightarrow{d}$ in a set of affine transformations involves $(k+1)^2$ coefficients $\in \mathbb{Z}_{2^w}$: $(1, d_1, d_2, \ldots, d_k, 0, c_{11}, c_{12}, \ldots, 0, c_{21}, ...c_{kk})$.[3] Thus, we may use any abstract domain whose elements concretize to subsets of $\mathbb{Z}_{2^w}^{(k+1)^2}$ as a method for representing a set of affine transformers.* □

[3] k of the coefficients are always 0, and one coefficient is always 1 (i.e., the first column is always $(1|\ 0\ 0\ ...\ 0)^t$). For this reason, we really need only $k + k^2$ elements, but we will sometimes refer to $(k+1)^2$ elements for brevity.

Example 5. The AG element A_2 in column 3 of Tab. 3 illustrates how an AG element with $(k+1)^2$ columns represents the same set of affine transformers as the MOS element M shown in column 1. For instance, the first row of A_2 represents the first matrix in M. □

Example 6. Consider the element $E = ([1,1],[0,10],[0,0],[0,0],[1,1],[2,3],[0,0],[0,0],[1,1])$ of $\mathcal{I}^9_{\mathbb{Z}_{2^w}}$. E can be depicted more mnemonically as the following matrix: $\left[\begin{array}{c|cc} {\scriptstyle 1} & {\scriptstyle x} & {\scriptstyle y} \\ \hline [1,1] & [0,10] & [0,0] \\ [0,0] & [1,1] & [2,3] \\ [0,0] & [0,0] & [1,1] \end{array}\right]$, where every element in E is an interval ($\mathcal{I}_{\mathbb{Z}_{2^w}}$). E represents the point set $\{(x',y',x,y) : \exists i_1, i_2 \in \mathbb{Z}_{2^w} : x' = x + i_1 \wedge y' = i_2 x + y \wedge 0 \leqslant i_1 \leqslant 10 \wedge 2 \leqslant i_2 \leqslant 3\}$. □

Examples 5 and 6 both exploit Observation 2, but use different abstract domains. Ex. 5 uses the AG domain with $(k+1)^2$ columns, whereas Ex. 6 uses the domain $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$. In particular, an abstract-domain element in our framework ATA[$\mathcal{B}$] is a set of affine transformations $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, such that the allowed coefficients in the matrix C and the vector $\overrightarrow{d}$ are abstracted by a base abstract domain $\mathcal{B}$.

The remainder of this section shows how different instantiations of Observation 2 allow different properties of a program to be recovered.

Example 7. In this example, the variable r of function f is initialized to 0 and conditionally incremented by $2x$ inside a loop with 10 iterations.

```
ENT: int f(int x) {
L0:    int i = 0, r = 0;
L1:    while(i <= 10) {
L2:       if(*)
L3:          r = r + 2*x;
L4:       i = i + 1;
       }
L5:    return r;
    }
```

The exact function summary for function f, denoted by S_f, is $(\exists k.r' = 2kx \wedge\ 0 \leqslant k \leqslant 10)$. Note that S_f expresses two important properties of the function: (i) the return value r' is an even multiple of x, and (ii) the multiplicative factor is contained in an interval. □

$\mathcal{B}$ = AG with $(k+1)^2$ columns: Fig. 1(a) shows the abstract transformers generated with the MOS domain.[4] Each matrix of the form $\left[\begin{array}{c|ccc} 1 & d_1 & d_2 & d_3 \\ \hline 0 & c_{11} & c_{12} & c_{13} \\ 0 & c_{21} & c_{22} & c_{23} \\ 0 & c_{31} & c_{32} & c_{33} \end{array}\right]$ represents the state transformation $(x' = d_1 + c_{11}x + c_{21}i + c_{31}r) \wedge (i' = d_2 + c_{12}x + c_{22}i + c_{32}r) \wedge (r' = d_3 + c_{13}x + c_{23}i + c_{33}r)$.

For instance, the abstract transformer for $L3 \rightarrow L4$ is an MOS-domain element with a single matrix that represents the affine transformation: $(x' = x) \wedge (i' = i) \wedge (r' = 2x + r)$. The edges absent from Fig. 1(a), e.g., $L1 \rightarrow L2$, have the identity MOS-domain element.

[4] We will continue to refer to the MOS domain directly, rather than "the instantiation of Observation 2 with an AG element containing $(k+1)^2$ columns" (à la Ex. 5).

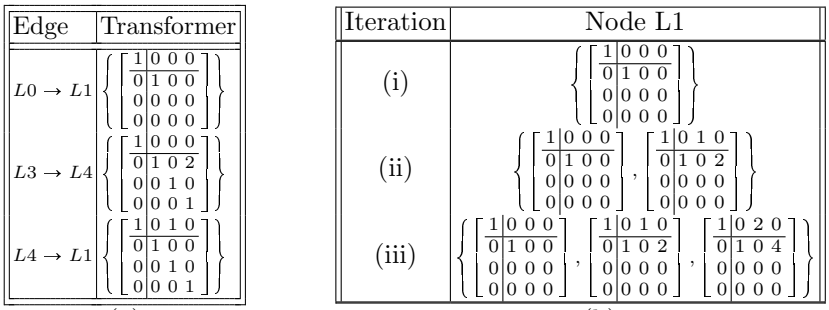

Fig. 1. Abstract transformers and snapshots in the fixpoint analysis with the MOS domain for Ex. 7.

To obtain function summaries, an iterative fixed-point computation needs to be performed. An abstract-domain element a is a *summary* at some program point L, if it describes a two-vocabulary transition relation that over-approximates the (compound) transition relation from the beginning of the function to program point L. Fig. 1(b) provides the iteration results for the summary at the program point $L1$. After iteration (i), the result represents $(x' = x) \wedge (i' = 0) \wedge (r' = 0)$. After iteration (ii), it adds the affine transformer $(x' = x) \wedge (i' = 1) \wedge (r' = 2x)$ to the summary. Quiescence is discovered on the third iteration because the affine-closure of the three matrices is the same as the affine-closure of the two matrices after the second iteration. As a result, the function summary that MOS learns, denoted by S_{MOS}, is $\exists k.r' = 2kx$, which is an overapproximation of the exact function summary S_f. Imprecision occurs because the MOS-domain is not able to represent inequality guards. Hence, the summary captures the evenness property, but not the bounds property.

$\mathcal{B} = \mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$*:* By using different $\mathcal{B}$s, an analyzer will be able to recover different properties of a program. Now consider what happens when the program above is analyzed with ATA$[\mathcal{B}]$ instantiated with the non-relational base domain of environments of intervals ($\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$). The identity transformation for the abstract domain ATA$[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ is $\left[\begin{array}{c|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [1,1] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$. The bottom element for the abstract domain ATA$[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$, denoted by $\bot_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]}$ is $\left[\begin{array}{c|ccc} 1 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \\ \hline 0 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \\ 0 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \\ 0 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \end{array}\right]$.[5]

[5] The abstract domain $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$ is the product domain of $(k+1)^2$ interval domains, that is, $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2} = \mathcal{I}_{\mathbb{Z}_{2^w}} \times \mathcal{I}_{\mathbb{Z}_{2^w}} \times \ldots \times \mathcal{I}_{\mathbb{Z}_{2^w}}$. $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$ uses smash product to maintain a canonical representation for $\bot_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]}$. Thus, if any of the coefficients

Edge	Transformer
$L0 \to L1$	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$
$L1 \to L2$	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [0,10] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$
$L3 \to L4$	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [2,2] \\ 0 & [0,0] & [1,1] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$
$L4 \to L1$	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [1,1] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [1,1] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$

(a)

Iteration	Node L1
(i)	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$
(ii)	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [0,1] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,2] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$
...	...
(xi)	$\left[\begin{array}{c\|ccc} 1 & [0,0] & [0,10] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,20] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$

(b)

Fig. 2. Abstract transformers and fixpoint analysis with the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain for Ex. 7.

Fig. 2 shows the abstract transformers and the fixpoint analysis for the node $L1$ with the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain. One advantage of using intervals as the base domain is that they can express inequalities. For instance, the abstract transformer for the edge $L1 \to L2$ specifies the transformation $(x' = x) \wedge (0 \leqslant i' \leqslant 10) \wedge (r' = r)$. Consequently, the function summary that ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ learns, denoted by $S_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]}$, is $r' = [0, 20]x$. This summary captures the bounds property, but not the evenness property. Notice that, $S_f = S_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]} \wedge S_{\text{MOS}}$.

Consider the instantiation of the ATA framework with strided-intervals over bitvectors [14], denoted by $\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$. A strided interval represents a set of the form $\{l, l+s, l+2s, ..., l+(n-1)s\}$. Here, l is the beginning of the interval, s is the stride, and n is the interval size. Consequently, ATA$[\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ learns the function summary $\exists k.r' = kx \wedge k = 2[0, 10]$, which captures both the evenness property and the bounds property. Note that a traditional (non-ATA-framework) analysis based on the strided-interval domain alone would not be able to capture the desired summary because the strided-interval domain is non-relational.

Widening concerns. In principle, abstract domains $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$ and $\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$ do not need widening operations because the lattice height is finite. However, the height is exponential in the bitwidth w of the program variables, and thus in practice we need widening operations to speed-up the fixpoint iteration. In the presence of widening, neither ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ nor ATA$[\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ will be able to capture the bounds property for Ex. 7, because they are missing relational information between the loop counter i and the variable r. However, the reduced product of ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ (or ATA$[\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$) and MOS can learn the exact function summary.

in an abstract-domain element $b \in \text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ is $\bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}}$, then b is smashed to $\bot_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]}$.

4 Affine-Transformer-Abstraction Framework

In this section, we formally introduce the Affine-Transformer-Abstraction framework (ATA) and describe abstract-domain operations for the framework. We also discuss some specific instantiations.

ATA$[\mathcal{B}]$ *Definition.* Let C be a k-by-k matrix: $[c_{ij}]$, where each c_{ij} is a symbolic constant for the entry at i-th row and j-th column. Let $\overrightarrow{d}$ be a k-vector, $[d_i]$, where each d_i is a symbolic constant for the i-th entry in the vector. As mentioned in §1, an affine transformer, denoted by $C : \overrightarrow{d}$, describes the relation $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, where $\overrightarrow{v}'$ and $\overrightarrow{v}$ are row vectors of size k that represent the post-transformation state and the pre-transformation state, respectively, on program variables.

Given a base abstract domain $\mathcal{B}$, the ATA framework generates a corresponding abstract domain ATA$[\mathcal{B}]$ whose elements represent a transition relation between the pre-state and the post-state program vocabulary. Each element $a \in$ ATA$[\mathcal{B}]$ is represented using an element $base(a) \in \mathcal{B}$, such that:

$$\gamma(a) = \{(\overrightarrow{v}, \overrightarrow{v}') | \ \exists (C : \overrightarrow{d}) \in \gamma(base(a)) : \overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}\}.$$

4.1 Abstract-Domain Operations for ATA$[\mathcal{B}]$

In this subsection, we provide all the abstract-domain operations for ATA$[\mathcal{B}]$, with the exception of abstract composition, which is discussed in §4.2.

In the ATA$[\mathcal{B}]$ framework, the symbolic constants in the base domain $\mathcal{B}$ are denoted by $symbols(C{:}\overrightarrow{d})$, where $symbols(C{:}\overrightarrow{d}) = (d_1, d_2, \ldots, d_n, c_{11}, c_{12}, \ldots, c_{1k}, c_{21}, c_{22}, \ldots, c_{2k}, \ldots, c_{kk})$ is the tuple of $k+k^2$ symbolic constants in the affine transformation. Tab. 4 lists the abstract-domain interface for the base abstract domain $\mathcal{B}$ needed to implement these operations for ATA$[\mathcal{B}]$. The first five operations in the interface are standard abstract-domain operations. $havoc(b_1, S)$ takes an element b_1 and a subset $S \subseteq symbols(C{:}\overrightarrow{d})$ of symbolic constants, and returns an element without any constraints on the symbolic constants in S. The last operation in Tab. 4 defines an abstraction for a concrete affine transformer ct. A concrete affine transformer is a mapping from the symbolic constants in the affine transformer to bitvectors of size w. We represent concrete state ct with the $(k+1) \times (k+1)$ matrix: $\left[\begin{array}{c|cccc} 1 & ct(d_1) & ct(d_2) & \ldots & ct(d_k) \\ \hline 0 & ct(c_{11}) & ct(c_{12}) & \ldots & ct(c_{1k}) \\ 0 & ct(c_{21}) & ct(c_{22}) & \ldots & ct(c_{2k}) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & ct(c_{k1}) & ct(c_{k2}) & \ldots & ct(c_{kk}) \end{array}\right]$, where $ct(s)$ denotes the concrete value in $\mathbb{Z}_{2^w}$ of symbol s in the concrete state ct.

Tab. 5 gives the abstract-domain operations for ATA$[\mathcal{B}]$ in terms of the base abstract-domain operations in $\mathcal{B}$. The first operation is the $\bot$ element, which is simply defined as $\bot_{\mathcal{B}}$, the bottom element in the base domain. Similarly, equality, join, and widen operations are defined as the equality, join, and widen operations in the base domain. The equality operation is not the exact equality operation;

Table 4. Base abstract-domain operations.

Type	Operation	Description
$\mathcal{B}$	$\bot$	*bottom element*
$\mathcal{B}$	$\top$	*top element*
bool	$(b_1 == b_2)$	*equality*
$\mathcal{B}$	$(b_1 \sqcup b_2)$	*join*
$\mathcal{B}$	$(b_1 \nabla b_2)$	*widen*
$\mathcal{B}$	$havoc(b_1, S)$	*remove all constraints on symbolic constants in* S
$\mathcal{B}$	$\alpha(ct)$	*abstraction for the concrete affine transformer* ct*, where* $ct \in symbols(C : \overrightarrow{d}) \rightarrow \mathbb{Z}_{2^w}$

that is, $(a_1 \widetilde{==} a_2)$ can return false, even if $\gamma(a_1) = \gamma(a_2)$. However, the equality operation is sound; that is, when $(a_1 \widetilde{==} a_2)$ returns true, then $\gamma(a_1) = \gamma(a_2)$. The $\tilde{\sqcup}$ operation for the ATA$[\mathcal{B}]$ is a quasi-join operation [7]. In other words, the least upper bound does not necessarily exist for ATA$[\mathcal{B}]$, but a sound upper-bound operation $\tilde{\sqcup}$ is available.

The abstraction operation for the affine-assignment statement $\alpha(v_j := d_0 + \sum_{i=1}^{k} c_{ij} * v_i)$ gives back an ATA$[\mathcal{B}]$-element with a single transformer where every variable $v \in V - \{v_j\}$ is left unchanged and the variable v_j is transformed to reflect the assignment by updating the coefficients of the corresponding column. The abstraction operation for the non-deterministic assignment statement $\alpha(v_j :=?)$ gives back an ATA$[\mathcal{B}]$-element, such that every variable $v \in V - \{v_j\}$ is left unchanged but the symbolic constant corresponding to the coefficients in the column j of the affine transformation can be any value. This operation is carried out by performing *havoc* on the identity transformation with respect to the set $\{d_j, c_{1j}, c_{2j}, ..., c_{kj}\}$ of symbolic constants. The identity transformation Id is obtained by abstracting the concrete affine transformer ct that represents the identity transformer. We provide proofs of soundness for these abstract-domain operations in [**?**, Appendix A].

Table 5. Abstract-domain operations for the ATA$[\mathcal{B}]$-domain.

Type	Operation	Description
$\mathcal{A}$	$\bot$	$\bot_{\mathcal{B}}$
bool	$(a_1 \widetilde{==} a_2)$	$(base(a_1) == base(a_2))$
$\mathcal{A}$	$(a_1 \tilde{\sqcup} a_2)$	$(base(a_1) \sqcup base(a_2))$
$\mathcal{A}$	$(a_1 \nabla a_2)$	$(base(a_1) \nabla base(a_2))$
$\mathcal{A}$	$\alpha(v_j := d_j + \sum_{i=1}^{k} c_{ij} * v_i)$	$\alpha\left(\left[\begin{array}{c\|ccc} 1 & 0 & d_j & 0 \\ \hline 0 & I_{j-1} & [c_{1j}, c_{2j}, ...c_{(j-1)j}]^t & 0 \\ 0 & 0 & c_{jj} & 0 \\ 0 & 0 & [c_{(j+1)j}, c_{(j+2)j}, ...c_{kj}]^t & I_{k-j} \end{array}\right]\right)$
$\mathcal{A}$	$\alpha(v_j :=?)$	$havoc(\alpha(I_{k+1}), \{d_j, c_{1j}, c_{2j}, ..., c_{kj}\})$
$\mathcal{A}$	Id	$\alpha(I_{k+1})$

4.2 Abstract Composition

We have shown that all the abstract-domain operations for ATA$[\mathcal{B}]$ can be implemented in terms of abstract-domain operations in $\mathcal{B}$, with the exception of

abstract composition. Let us consider the composition of two abstract values $a, a' \in \text{ATA}[\mathcal{B}]$, representing the two-vocabulary relations $R[\overrightarrow{v}; \overrightarrow{v}'] = \gamma(a)$ and $R'[\overrightarrow{v}'; \overrightarrow{v}''] = \gamma(a')$. An abstract operation $\circ^\sharp$ is a sound abstract-composition operation if, for all $a'' = a' \circ^\sharp a$, $\gamma(a'') \supseteq \{ (\overrightarrow{v}; \overrightarrow{v}'') | \exists \overrightarrow{v}'. R[\overrightarrow{v}; \overrightarrow{v}'] \wedge R'[\overrightarrow{v}'; \overrightarrow{v}''] \}$. This condition translates to:

$$\begin{aligned}\gamma(base(a'')) \supseteq \{(\overrightarrow{v}, \overrightarrow{v}'') \mid \exists (C : \overrightarrow{d}) \in \gamma(base(a)), (C' : \overrightarrow{d}') \in \gamma(base(a')), \\ (C'' : \overrightarrow{d}'') : (\overrightarrow{v}'' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge (C'' = C \cdot C') \\ \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}')\}\end{aligned} \tag{2}$$

The presence of the quadratic components $C \cdot C'$ and $\overrightarrow{d} \cdot C'$ makes the implementation of abstract composition non-trivial. One extremely expensive method to implement abstract composition is to enumerate the set of all concrete transformers $(C : \overrightarrow{d}) \in \gamma(base(a))$ and $(C' : \overrightarrow{d}') \in \gamma(base(a'))$, perform matrix multiplication for each pair of concrete transformers, and perform join over all pairs of them. This approach is impractical because the set of all concrete transformers in an abstract value can be very large.

First, we provide a general method to implement abstract composition. Then, we provide methods for abstract composition when the base domain $\mathcal{B}$ has certain properties, like non-relationality and weak convexity. The latter methods are faster, but are only applicable to certain classes of base abstract domains.

General Case. We present a general method to perform abstract composition by reducing it to the symbolic-abstraction problem. The *symbolic abstraction* of a formula φ in logic $\mathbb{L}$, denoted by $\hat{\alpha}(\varphi)$, is the best value in $\mathcal{B}$ that over-approximates the set of all concrete affine transformers $(C : \overrightarrow{d})$ that satisfy φ [15, 22]. For all $b \in \mathcal{B}$, the *symbolic concretization* of $\mathcal{B}$, denoted by $\hat{\gamma}(b)$, maps b to a formula $\hat{\gamma}(b) \in \mathbb{L}$ such that b and $\hat{\gamma}(b)$ represent the same set of concrete affine transformers (i.e., $\gamma(b) = [\![\hat{\gamma}(b)]\!]$). We expect the base domain $\mathcal{B}$ to provide the $\hat{\gamma}$ operation. In our framework, there are slightly different variants of $\hat{\alpha}$ and $\hat{\gamma}$ according to which vocabulary of symbolic constants are involved. For instance, we use $\hat{\gamma}'$ to denote symbolic concretization in terms of the primed symbolic constants $symbols(C' : \overrightarrow{d}')$. Similarly, $\hat{\alpha}''$ denotes symbolic abstraction in terms of the double-primed symbolic constants $symbols(C'' : \overrightarrow{d}'')$. The function *dropPrimes* shifts the vocabulary of symbolic constants by removing the primes from the symbolic constants that an abstract value represents.

We use $\mathbb{L} = QF\_BV$, i.e., quantifier-free bit-vector logic, to express abstract composition symbolically as follows:

$$\begin{aligned}base(a'') = dropPrimes\big(\hat{\alpha}''(\varphi)\big), \text{ where} \\ \varphi = (C'' = C \cdot C') \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}') \\ \wedge \hat{\gamma}(base(a)) \wedge \hat{\gamma}'(base(a')).\end{aligned} \tag{3}$$

(Note that $\hat{\gamma}(base(a))$ and $\hat{\gamma}'(base(a'))$ are formulas over $symbols(C : \overrightarrow{d})$ and $symbols(C' : \overrightarrow{d}')$ respectively.) Past literature [15, 22, 6] provides various algorithms to implement symbolic abstraction. Symbolic-abstraction methods are

usually slow because they make repeated calls to an SMT solver. Specifically, the symbolic-abstraction algorithms in [15, 22] require $\mathcal{O}(h'')$ calls to SMT, where h'' is the height of the abstract-domain element — i.e., $base(a'')$ in the lattice $\mathcal{B}$.

Alg. 1 is a variant of the symbolic-abstraction algorithm from [15]. Alg. 1 needs a method to enumerate a generator set gs for each $b \in \mathcal{B}$. Such a set can easily be obtained from the generator representation of $\mathcal{B}$. For instance, each row in an AG element is an affine transformer, and a generator set for the AG element is the set of all rows in the AG matrix: the affine combination of the rows generate the concrete affine transformers that the AG element (see §2.4) represents. Note that the generator set for an abstract value b is usually much smaller than the set of all affine transformers in b. For the AG domain, the generating set is worst-case polynomial size, whereas the set of all affine transformers is worst-case exponential in the number of variables k.

In Alg. 1, line 3 initializes the value *lower* to the product of each pair of abstract transformers. The product $t \times t'$, where $t = \left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right]$ and $t' = \left[\begin{array}{c|c} 1 & \overrightarrow{d}' \\ \hline 0 & C' \end{array}\right]$ is $\left[\begin{array}{c|c} 1 & \overrightarrow{d} \cdot C' + \overrightarrow{d}' \\ \hline 0 & C \cdot C' \end{array}\right]$. Because *lower* is initialized to $\{t \times t' \mid t \in gs_1, t' \in gs_2\}$ rather than $\bot$, the number of SMT calls in the symbolic abstraction is significantly reduced, compared to the algorithm from [15]. The function $GetModel$, used at line 5, returns the model $M \in symbols(C'' : \overrightarrow{d}'') \to \mathbb{Z}_{2^w}$ satisfying the formula $(\varphi \wedge \neg\hat{\gamma}(lower))$ given to the SMT solver at line 4. Thus, the model M is a concrete affine transformer in a''. The representation function β, used at line 6, maps a singleton model M to the least value in $\mathcal{B}$ that overapproximates $\{M\}$ [15]. While the SMT call at line 4 is satisfiable, the loop keeps improving the value of *lower* by adding the satisfying model M to *lower* via the representation function β and the join operation. When line 4 is unsatisfiable, the loop terminates and returns *lower*. This method is sound because the unsatisfiable call proves that $\varphi \Rightarrow \hat{\gamma}(lower)$. The loop terminates when the height of the base domain $\mathcal{B}$ is finite.

Algorithm 1 Abstract Composition via Symbolic Abstraction

```
1: gs1 ← {t1, t2, ..., t_{l1}}                    ▷ where base(a) = ⨆_{i=0}^{l1} t_i
2: gs2 ← {t'1, t'2, ..., t'_{l2}}                 ▷ where base(a') = ⨆_{i=0}^{l2} t'_i
3: lower ← {t × t' | t ∈ gs1, t' ∈ gs2}
4: while r ← SMTCall(φ ∧ ¬γ̂(lower)) is Sat do
5:     M ← GetModel(r)
6:     lower ← lower ⊔ β(M)
7: return lower
```

Non-relational base domains. In this section, we present a method to implement abstract composition for ATA[$\mathcal{B}$], when $\mathcal{B}$ is non-relational. We focus on the non-relational case separately because it allows us to implement a sound abstract-composition operation efficiently.

Foundation domain. Each element in the non-relational domain $\mathcal{B}$ is a mapping from symbols S to a subset of Z_{2^w}. We introduce the concept of a *foundation domain*, denoted by $\mathcal{F}_{\mathcal{B}}$, to represent the abstractions of subsets of Z_{2^w} in the base abstract-domain elements. We can define a non-relational base domain in terms of the foundation domain as follows: $\mathcal{B} \stackrel{def}{=} S \to \mathcal{F}_{\mathcal{B}}$. For instance, the non-relational domain of intervals $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$ can be represented by $S \to \mathcal{I}_{\mathbb{Z}_{2^w}}$, where $\mathcal{I}_{\mathbb{Z}_{2^w}}$ represents the interval lattice over $\mathbb{Z}_{2^w}$, and S is a set of $(k+1)^2$ symbolic constants that represent the coefficients of an affine transformer.

A foundation domain $\mathcal{F}$ is a lattice whose elements concretize to subsets of Z_{2^w}. Tab. 6 present the foundation-domain operations for $\mathcal{F}$. Bottom, equality, join, widen, and $\alpha(bv)$ are standard abstract-domain operations. The abstract addition and multiplication operations provide a sound reinterpretation of the collecting semantics of concrete addition and multiplication. For instance, with the interval foundation domain, $[0,7] +^\sharp [-3,17] = [-3,24]$ and $[0,6] \times^\sharp [-3,3] = [-18,18]$.

Table 6. Foundation-domain operations.

Type	Operation	Description
$\mathcal{F}$	$\bot$	*empty set*
bool	$(f_1 == f_2)$	*equality*
$\mathcal{F}$	$(f_1 \sqcup f_2)$	*join*
$\mathcal{F}$	$(f_1 \nabla f_2)$	*widen*

Type	Operation	Description
$\mathcal{F}$	$\alpha(bv)$	*abstraction for the bitvector value* $bv \in Z_{2^w}$
$\mathcal{F}$	$(f_1 +^\sharp f_2)$	*abstract addition*
$\mathcal{F}$	$(f_1 \times^\sharp f_2)$	*abstract multiplication*

Abstract composition for a non-relational domain is defined as follows:

$$
\begin{aligned}
a' \circ_{\text{NR}} a = \Big\{ (\overrightarrow{v}, \overrightarrow{v}') \mid \exists (C : \overrightarrow{d}) : (\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}) \wedge b \in (symbols(C : \overrightarrow{d}) \to \mathcal{F}) \quad (4)\\
\wedge \Big(\bigwedge_{1 \leqslant i,j \leqslant k} (b[c_{ij}] = \Sigma^\sharp_{1 \leqslant l \leqslant k} (base(a)[c_{il}] \times^\sharp base(a')[c_{lj}]) \Big)\\
\wedge \Big(\bigwedge_{1 \leqslant j \leqslant k} b[d_j] = \Sigma^\sharp_{1 \leqslant l \leqslant k} (base(a)[d_l] \times^\sharp base(a')[c_{lj}]) +^\sharp base(a')[d_j] \Big) \Big\}.
\end{aligned}
$$

The term $b[s]$, where $b \in \mathcal{B}$ and $s \in symbols(C : \overrightarrow{d})$, refers to the element in the foundation domain $f \in \mathcal{F}_{\mathcal{B}}$, that corresponds to the symbol s. $\Sigma^\sharp_{1 \leqslant l \leqslant k}$ is calculated by abstractly adding the k terms indexed by l. Abstract composition for a non-relational domain uses abstract addition and abstract multiplication to soundly overapproximate the quadratic terms occurring in Eqn. (2). We provide a proof of the soundness for $a' \circ_{\text{NR}} a$ in [**?**, Appendix B.1]. The abstract-composition operation requires $\mathcal{O}(k^3)$ abstract-addition operations and $\mathcal{O}(k^3)$ abstract-multiplication operations.

Examples of foundation domains. We now present a few foundation domains that allow to construct the non-relational small-set, interval [2], and strided-interval [14] base domains.

Small sets. $\mathcal{F}_{SS_n} \stackrel{def}{=} \{\top\} \cup \{S \mid S \subseteq \mathbb{Z}_{2^w} \wedge |S| \leqslant n\}$. The join operation is defined by: $(f_1 \sqcup f_2) = \begin{cases} f_1 \cup f_2 & \text{if } |f_1 \cup f_s| \leqslant n \\ \top & \text{otherwise} \end{cases}$

n denotes the maximum cardinality allowed in the non-top elements of $\mathcal{F}_{SS_n}$. Other abstract operators, including abstract addition and multiplication, are implemented in a similar manner.

Intervals. $\mathcal{F}_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \stackrel{def}{=} \{\bot\} \cup \{[a,b] |\ a,b \in \mathbb{Z}_{2^w}, a \leqslant b\}$. Most abstract operations are straightforward (See [2] for details). The abstract-addition and abstract-multiplication operations need to be careful about overflows to preserve soundness. For instance,

$$[a_1,b_1] +^\sharp [a_2,b_2] = \begin{cases} [a_1+a_2, b_1+b_2] & \textit{if neither } a_1+a_2 \textit{ nor } b_1+b_2 \\ & \textit{overflows} \\ [min, max] & \textit{otherwise} \end{cases}$$

Strided Interval. $\mathcal{F}_{\mathcal{SI}_{\mathbb{Z}_{2^w}}} \stackrel{def}{=} \{\bot\} \cup \{s[a,b] \mid a,b,s \in \mathbb{Z}_{2^w}, a \leqslant b\}$, where $\gamma(s[a,b]) = \{i \mid a \leqslant i \leqslant b, i \equiv a (mod\ s)\}$. (See [14, 17] for the details of the abstract-domain operations.)

Affine-Closed Base Domain. We discuss the special case when the base domain $\mathcal{B}$ is affine-closed, i.e., $\mathcal{B} = \text{AG}$. The abstract composition is defined as:

$$a' \circ_{\text{AG}} a = a'', \textit{ where } base(a'') = \left\langle \left\{ t_i \times t'_j \middle|\ 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\} \right\rangle \wedge \quad (5)$$
$$base(a) = \langle \{t_1, t_2, ...t_l\} \rangle \wedge base(a') = \left\langle \left\{ t'_1, t'_2, ...t'_{l'} \right\} \right\rangle$$

Lemma 5.1 in [13] asserts that the above abstract composition method is sound by linearity of affine-closed abstractions. The abstract composition has time complexity $\mathcal{O}(hh'k^3)$, h (respectively h') is the height of the abstract-domain element $base(a)$ (or $base(a')$) in the AG lattice. Because the height of the AG lattice with $(k+1)^2$ columns is $\mathcal{O}(k^2)$, the time complexity for the abstract composition operation translates to $\mathcal{O}(k^7)$. Alg. 1 essentially implements Eqn. (5), but makes an extra SMT call to ensure that the result is sound. Because Eqn. (5) is sound by linearity for the AG domain, the very first SMT call in the while-loop condition at line 4 in Alg. 1 will be unsatisfiable.

Weakly-Convex Base Domain We present methods to perform abstract composition when the base domain $\mathcal{B}$ satisfies a property we call *weak convexity*. Base domain $\mathcal{B}$ is *weakly convex* iff

- The abstraction of a single concrete affine transformer is exact: $\gamma(\alpha(t_i)) = \{t_i\}$.
- All abstract-domain elements $b \in \mathcal{B}$ are contained in a convex space over rationals: For any set of concrete affine transformers $\{t_0, t_1, ..., t_l\}$, such that $b = \bigsqcup_{i=0}^{l} t_i$, and any $t \in \gamma(b)$:
  $\exists \lambda_1, \lambda_2, \ldots, \lambda_l \in \mathbb{Q}. (0 \leqslant \lambda_1, \lambda_2, ..., \lambda_l \leqslant 1) \wedge \sum_{i=0}^{l} \lambda_i = 1 \wedge cast_{\mathbb{Q}}(t) = \sum_{i=0}^{l} \lambda_i\ cast_{\mathbb{Q}}(t_i)$.
  The $cast_{\mathbb{Q}}$ function is used to specify the convexity property by moving the point space from bitvectors to rationals. For instance, the expression $\Sigma_{i=0}^{l} \lambda_i . cast_{\mathbb{Q}}(t_i)$ specifies the convex combination of the concrete affine transformers m_i in the rational space $cast_{\mathbb{Q}}^{(k+k^2)}$.

Any convex abstract domain over rationals, such as polyhedra [5] or octagons [10], can be used to create a weakly-convex domain over bitvectors [21, 20]. Abstract composition for weakly-convex base domains is defined as follows:

$$a' \circ_{\text{WC}} a = a'', \textit{ where } base(a'') = \tag{6}$$
$$\begin{cases} \left\{ t_i \times t'_j \middle|\ 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\} & \textit{if there are no overflows in any matrix multiplication } t_i \times t'_j \\ \top_{\mathcal{B}} & \textit{otherwise} \end{cases}$$
$$\text{where } base(a) = \{t_1, t_2, ..., t_l\} \textit{ and } base(a') = \left\{ t'_1, t'_2, ..., t'_{l'} \right\}.$$

The intuition is that the weak-convexity properties are preserved under matrix multiplication in the absence of overflows. This principle is similar to the linearity argument used to show that abstract composition is sound when the base domain is affine-closed. (See above for more details.) We provide a proof of the soundness for $a' \circ_{\text{WC}} a$ in [**?**, Appendix B.2]. Similar to the affine-closed case, abstract composition has time complexity $\mathcal{O}(H^2k^3)$, where H is the height of the $\mathcal{B}$ lattice.

5 Discussion and Related Work

The abstract-domain elements in our framework abstract two-vocabulary relationships arising between the pre-transformation state and post-transformation state. For the sake of simplicity, we assumed that the variable sets in the pre-transformation and post-transformation state are the same, and an affine transformer is represented by a $(k+1) \times (k+1)$ matrix, where k is the number of variables in the pre-transformation state. However, this requirement is not mandatory. We can easily adapt our abstract-domain operations to work on $(k+1) \times (k'+1)$ matrices where k' is the number of variables in the post-transformation state.

The abstract-domain elements in our framework are not necessarily closed under intersection. Consider the two abstract values a_1 and a_2 for the vocabulary $V = \{v_1\}$. Let a_1 represent the affine transformation $v'_1 = 0$ and a_2 represent the identity affine transformation $v'_1 = v_1$. Thus, $a_1 = \alpha(\left[\begin{smallmatrix}1|0\\0|0\end{smallmatrix}\right])$, and $a_2 = \alpha(\left[\begin{smallmatrix}1|0\\0|1\end{smallmatrix}\right])$. The intersection of $\gamma(a_1)$ and $\gamma(a_2)$ is the point $p = (v'_1 = 0, v_1 = 0)$. There does not exist an abstract value in ATA$[\mathcal{B}]$, that can exactly represent the point p, because any abstract value containing p must contain at least one affine transformer of the form $v'_1 = v_1 \cdot c$, and thus must contain all points of the form $(v'_1 = t \cdot c, v_1 = t)$, where $t \in \mathcal{Z}_{2^w}$. As a consequence, there does not exist a Galois connection between ATA$[\mathcal{B}]$ and the concrete domain $\mathcal{C}$ of all two-vocabulary relations $R[V; V']$, which implies that there does not exist a best abstraction for a set of concrete points. For instance, consider the abstraction of the guard statement $S_G = \{v_1 \leqslant 10\}$, with the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain. Consider $a_3 = \left[\begin{smallmatrix}1|[0,10]\\0|[0,0]\end{smallmatrix}\right]$ and $a_4 = \left[\begin{smallmatrix}1|[0,0]\\0|[1,1]\end{smallmatrix}\right]$. a_3 specifies the guard constraint $0 \leqslant v'_1 \leqslant 10$, while a_4 is the identity transformation $v'_1 = v_1$. Note that these abstract values

are incomparable and can be used to represent the abstract transformer for S_G. Furthermore, $a_3 \sqcap a_4$ does not exist. Thus, an analysis has to settle for either a_3 or a_4. (In §3, we used an abstract transformer similar to a_3 for the guard in the while statement in Ex. 7. Using an identity transfer for the guard statement would not have been useful to capture the desired bounds constraint.)

The ATA constructor preserves finiteness; that is, if the base domain $\mathcal{B}$ is finite, then the domain ATA$[\mathcal{B}]$ is finite as well.

It is also possible to use the ATA constructor to infer affine transformations over rationals or reals. In these cases, the symbolic-composition methods for weakly-convex base domains (see §4.2) will carry over to affine transformations over rationals or reals for convex base domains (e.g., polyhedra) with only slight modifications. For instance. abstract composition for convex base domains over rationals or reals is defined as follows:

$$a' \circ a = a'', \textit{ where } base(a'') = \left\{ t_i \times t'_j \middle|\ 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\}$$
$$\text{where } base(a) = \{t_1, t_2, ..., t_l\} \textit{ and } base(a') = \left\{ t'_1, t'_2, ..., t'_{l'} \right\}.$$

Chen et al. [1] devised the *interval-polyhedra* domain which can express constraints of the form $\Sigma_k[a_k, b_k]x_k \leqslant c$ over rationals. Interval polyhedra are more expressive than classic convex polyhedra, and thus can express certain non-convex properties. Abstract-domain operations for interval polyhedra are constructed by linear programming and interval Fourier-Motzkin elimination. The domain has similarities to the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain because the coefficients in the abstract values are intervals.

Miné [11] introduced *weakly relational domains*, which are a parameterized family of relational domains, parameterized by a non-relational base abstract domain. They can express constraints of the form $(v_j - v_i) \in \mathcal{F}$, where $\mathcal{F}$ is an abstraction over $\mathcal{P}(\mathbb{Z})$. Similar to ATA$[\mathcal{B}]$, Miné's framework requires the base non-relational domain to provide abstract-addition and abstract-unary-minus operations. These operations are used to propagate information between constraints via a closure operation that is similar to finding shortest paths.

Sankaranarayanan et al. [16] introduced a domain based on template constraint matrices (TCMs) that is less powerful than polyhedra, but more general than intervals and octagons. Their analysis discovers linear-inequality invariants using polyhedra with a predefined fixed shape. The predefined shape is given by the client in the form of a template matrix. Our approach is similar because an affine transformer with symbolic constants can be seen as a template. However, the approaches differ because Sankaranarayanan et al. use an LP solver to find values for template parameters, whereas we use operations and values from an abstract domain to find and represent a set of allowed values for template parameters.

An abstract-domain element in ATA$[\mathcal{B}]$ can be seen as an abstraction over sets of functions: $\mathbb{Z}^k_{2^w} \rightarrow \mathbb{Z}^k_{2^w}$. Jeannet et al. [8] provide a theoretical treatment of the relational abstraction of functions. They describing existing and new methods of abstracting functions of signature: $D_1 \rightarrow D_2$, resulting in a family

of relational abstract domains. ATA[$\mathcal{B}$] is not captured by their framework of functional abstractions.

References

1. L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS*, 2009.
2. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd. Int. Symp on Programming*, Paris, Apr. 1976.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL*, pages 238–252, 1977.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.
6. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. *TOPLAS*, 2014.
7. G. Gange, J. Navas, P. Schachte, H. Søndergaard, and P. Stuckey. Abstract interpretation over non-lattice abstract domains. In *SAS*, 2013.
8. B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *SAS*, 2005.
9. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.
10. A. Miné. The octagon abstract domain. In *WCRE*, 2001.
11. A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, 2002.
12. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
13. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *TOPLAS*, 29(5), 2007.
14. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Part. Eval. and Semantics-Based Prog. Manip.*, 2006.
15. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.
16. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.
17. R. Sen and Y. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *MEMOCODE*, 2007.
18. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
19. T. Sharma and T. Reps. A new abstraction framework for affine transformers. Tech. Rep. TR-1846, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, May 2017. Revised July 2017.
20. T. Sharma and T. Reps. Sound bit-precise numerical domains. In *VMCAI*, 2017.
21. A. Simon and A. King. Taming the wrapping of integer arithmetic. In *SAS*, 2007.
22. A. Thakur, M. Elder, and T. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, 2012.