Noname manuscript No.
(will be inserted by the editor)

A New Abstraction Framework for Affine Transformers

Tushar Sharma · **Thomas Reps**

Received: date / Accepted: date

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 re-purposed 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

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.

T. Sharma
University of Wisconsin; Madison, WI, USA
(Now at) Synopsys Inc., San Francisco, CA, USA
Tel.: +1 323-337-5546
E-mail: tsharma@cs.wisc.edu

T. Reps
University of Wisconsin; Madison, WI, USA
GrammaTech, Inc.; Ithaca, NY, USA
E-mail: reps@cs.wisc.edu

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 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 [22]. 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, 32]. 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 [23] 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, $\left[1 | \overrightarrow{v}'\right] := \left[1 | \overrightarrow{v}\right] T$. 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 $a_i, b \in \mathbb{Z}_{2^w}$ and $\mathbb{Z}_{2^w}$ is the set of w-bit bitvectors [12,6] (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[$\mathcal{B}$]. §6 provides discussion and related work. Proofs are given in App. A, App. B, and App. C.

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

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*
$\mathcal{A}$	$(a_1 \circ a_2)$	*composition*

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*
$\mathcal{A}$	$\alpha(Cond)$	*abstraction for guards*

2.1 Affine Programs

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

We borrow the notion of affine programs from [23], and add affine inequality and affine congruence guards. We restrict our affine programs to consist of a single procedure. The statements are restricted to either affine assignments or nondeterministic 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, an affine congruence, 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 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. Similarly, the $\alpha(Cond)$ operation performs abstraction on a guard (in a branch of a conditional-jump statement). 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 [23]. 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, $\left[1 | \overrightarrow{v}'\right] := \left[1 | \overrightarrow{v}\right] T$.

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, $\gamma_{\text{MOS}}(\mathcal{M}) \stackrel{\text{def}}{=} \left\{ (\overrightarrow{v}, \overrightarrow{v}') \,\middle|\, \overrightarrow{v}, \overrightarrow{v}' \in \mathbb{Z}_{2^w}^k \wedge \exists T \in \langle \mathcal{M} \rangle : [1|v]\, T = [1|v'] \right\}$.

Example 2.1 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 a 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*
$\mathcal{A}$	$\alpha(v_j := d_0 + \sum_{i=1}^{k} c_{ij} * v_i)$	$\left\{ \left[\begin{array}{c\|cccc} 1 & 0 & d_0 & 0 \\ \hline 0 & I_{j-1} & [c_{1j}, c_{2j}, \ldots c_{(j-1)j}]^t & 0 \\ 0 & 0 & c_{jj} & 0 \\ 0 & 0 & [c_{(j+1)j}, c_{(j+2)j}, \ldots c_{kj}]^t & I_{k-j} \end{array} \right] \right\}$
$\mathcal{A}$	$\alpha(v_j := ?)$	$\left\{ \left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & I_{j-1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{k-j} \end{array} \right], \left[\begin{array}{c\|ccc} 1 & 0 & 1 & 0 \\ \hline 0 & I_{j-1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{k-j} \end{array} \right] \right\}$
$\mathcal{A}$	Id	$\{I_{k+1}\}$
$\mathcal{A}$	$(M_1 \circ M_2)$	$\{A_2 A_1 \| A_i \in M_i\}$

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 $\text{AG}[\overrightarrow{v}; \overrightarrow{v}']$ 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 $\begin{array}{c} \begin{array}{ccc} 1 & \overrightarrow{v} & \overrightarrow{v}' \end{array} \\ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & I \end{array} \right] \end{array}$. The $\text{AG}[\{v_1, v_2\}; \{v_1', v_2'\}]$ element $\begin{array}{c} \begin{array}{ccccc} 1 & v_1 & v_2 & v_1' & v_2' \end{array} \\ \left[\begin{array}{c|cccc} 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{array} \right] \end{array}$ 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 view them as two different abstractions of two-vocabulary relations, and provide (approximate) inter-conversion methods [6, §4.2–4.4]. The other is to use a variant of the AG domain to represent the elements of the MOS domain exactly (see §2.5.2).

2.5.1 Comparison of MOS and AG elements as abstractions 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 2.2 For example, when $k = 1$, the AG element for "assume $x = 2$" is $\overset{\begin{smallmatrix}1 & x & x'\end{smallmatrix}}{\left[\,1 \middle| 2 \;\; 2\,\right]}$, i.e., "$x = 2 \wedge x' = 2$." In contrast, there is no MOS element that represents "$x = 2 \wedge x' = 2$" exactly. Moreover, in general there does not exist a *smallest* MOS transformer that overapproximates a given AG element. For instance, two minimal (and incomparable) MOS elements that overapproximate "assume $x = 2$" are (i) the identity transformer, i.e., $\left\{ \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right] \right\}$, and (ii) "$x' = 2$", i.e., $\left\{ \left[\begin{array}{c|c} 1 & 2 \\ \hline 0 & 0 \end{array}\right] \right\}$.

The latter example shows that an MOS element can capture a 1-vocabulary guard on the post-state (e.g., $x' = 2$); however, an MOS element cannot capture a 1-vocabulary guard on the pre-state (e.g., $x = 2$). □

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

Example 2.3 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

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

Affine spaces are closed under affine combinations of their elements. Thus, $\gamma_{\mathrm{MOS}}(M)$ is not an affine space because some affine combinations of its elements are not in $\gamma_{\mathrm{MOS}}(M)$. For instance, let $a = \left[1\ {-1}\ 1\ 1\right]$, $b = \left[2\ {-2}\ 6\ 6\right]$, and $c = \left[0\ 0\ {-4}\ {-4}\right]$. By Eqn. (1), we have $a \in \gamma_{\mathrm{MOS}}(M)$ when $p = 0$ in Eqn. (1), $b \in \gamma_{\mathrm{MOS}}(M)$ when $p = -1$, and $c \notin \gamma_{\mathrm{MOS}}(M)$ (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, $\gamma_{\mathrm{MOS}}(M)$ is not closed under affine combinations of its elements, and so $\gamma_{\mathrm{MOS}}(M)$ 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]).

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)
$\left\{ \begin{array}{c} \scriptstyle 1\ x\ y \\ \left[\begin{array}{c\|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right] \end{array}, \begin{array}{c} \scriptstyle 1\ x\ y \\ \left[\begin{array}{c\|cc} 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right] \end{array} \right\}$	$\begin{array}{c} \scriptstyle 1\ \ x\ \ y\ \ x'\ \ y' \\ \left[\begin{array}{c\|cccc} 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{array}\right] \end{array}$	$\begin{array}{c} \scriptstyle 1\ \ a_{01}\ \ a_{02}\ \ a_{10}\ \ a_{11}\ \ a_{12}\ \ a_{20}\ \ a_{21}\ \ a_{22} \\ \left[\begin{array}{c\|cccccccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right] \end{array}$

2.5.2 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 2.4 Tab. 3 shows the two ways MOS and AG elements can be related. Column 1 shows the MOS element M from Ex. 2.3, 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. Note that even though the AG domain is defined as a two-vocabulary matrix, the affine generators can also be defined over $(k+1) \times (k+1)$ symbolic constants, as shown in A_2. Because A_2 is just a reformulation of M, $\gamma_{\text{AG}[(k+1)\times(k+1)]}(A_2) = \langle M \rangle$. □

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 $\in \mathbb{Z}_{2^w}$: $(1, d_1, d_2, \ldots, d_k, 0, c_{11}, c_{12}, \ldots, 0, c_{21}, \ldots c_{kk})$.*[1] *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.* □

Example 3.1 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. □

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

Example 3.2 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: $\begin{bmatrix} \begin{array}{c|cc} 1 & x & y \\ [1,1] & [0,10] & [0,0] \\ \hline [0,0] & [1,1] & [2,3] \\ [0,0] & [0,0] & [1,1] \end{array} \end{bmatrix}$, 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 3.1 and 3.2 both exploit Observation 2, but use different abstract domains. Ex. 3.1 uses the AG domain with $(k+1)^2$ columns, whereas Ex. 3.2 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 3.3 In this example, the variable r of function f is initialized to 0 and non-deterministically incremented by $2x$ inside a loop with 10 iterations.

```
ENT: unsigned f(unsigned x) {
L0:    unsigned 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.[2] Each matrix of the form $\begin{bmatrix} \begin{array}{c|ccc} 1 & d_1 & d_2 & d_3 \\ \hline 0 & c_{11} & c_{12} & c_{13} \\ 0 & c_{21} & c_{22} & c_{23} \\ 0 & c_{31} & c_{32} & c_{33} \end{array} \end{bmatrix}$ 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.

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 $L2$. After iteration (i), the result represents $(x' = x) \wedge (i' = 0) \wedge (r' = 0)$. After iteration (ii), it adds the affine transformer

[2] 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. 3.1).

<table>
<tr><th>Edge</th><th>Transformer</th></tr>
<tr><td>$L0 \to L1$</td><td>$\left\{\left[\begin{array}{c|ccc}1&0&0&0\\\hline 0&1&0&0\\0&0&0&0\\0&0&0&0\end{array}\right]\right\}$</td></tr>
<tr><td>$L3 \to L4$</td><td>$\left\{\left[\begin{array}{c|ccc}1&0&0&0\\\hline 0&1&0&2\\0&0&1&0\\0&0&0&1\end{array}\right]\right\}$</td></tr>
<tr><td>$L4 \to L1$</td><td>$\left\{\left[\begin{array}{c|ccc}1&0&1&0\\\hline 0&1&0&0\\0&0&1&0\\0&0&0&1\end{array}\right]\right\}$</td></tr>
</table>

(a)

<table>
<tr><th>Iteration</th><th>Node L2</th></tr>
<tr><td>(i)</td><td>$\left\{\left[\begin{array}{c|ccc}1&0&0&0\\\hline 0&1&0&0\\0&0&0&0\\0&0&0&0\end{array}\right]\right\}$</td></tr>
<tr><td>(ii)</td><td>$\left\{\left[\begin{array}{c|ccc}1&0&0&0\\\hline 0&1&0&0\\0&0&0&0\\0&0&0&0\end{array}\right], \left[\begin{array}{c|ccc}1&0&1&0\\\hline 0&1&0&2\\0&0&0&0\\0&0&0&0\end{array}\right]\right\}$</td></tr>
<tr><td>(iii)</td><td>$\left\{\left[\begin{array}{c|ccc}1&0&0&0\\\hline 0&1&0&0\\0&0&0&0\\0&0&0&0\end{array}\right], \left[\begin{array}{c|ccc}1&0&1&0\\\hline 0&1&0&2\\0&0&0&0\\0&0&0&0\end{array}\right], \left[\begin{array}{c|ccc}1&0&2&0\\\hline 0&1&0&4\\0&0&0&0\\0&0&0&0\end{array}\right]\right\}$</td></tr>
</table>

(b)

Fig. 1 Abstract transformers and snapshots in the fixpoint analysis with the MOS domain for Ex. 3.3.

$(x' = x) \wedge (i' = 1) \wedge (r' = 2x)$ to the summary. The iteration stabilizes 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.

$\underline{\mathcal{B} = \mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}}$*:* 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 $\perp_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]}$ is $\left[\begin{array}{c|ccc}1&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}\\\hline 0&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}\\0&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}\\0&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}&\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}\end{array}\right]$.[3]

Fig. 2 shows the abstract transformers and the fixpoint analysis for the node $L2$ with the ATA$[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ domain. One advantage of using intervals as the base domain is that they can express inequalities. For instance, the abstract transformer for the edge $L1 \to L2$ can be specified by the transformation $(x' = x) \wedge (0 \leqslant i' \leqslant 10) \wedge (r' = r)$. As shown in §4.3, there can exist multiple

[3] 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 $\perp_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]}$. Thus, if any of the coefficients in an abstract-domain element $b \in \text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ is $\perp_{\mathcal{I}_{\mathbb{Z}_{2^w}}}$, then b is smashed to $\perp_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]}$.

<table>
<tr><th>Edge</th><th>Transformer</th></tr>
<tr><td>$L0 \rightarrow L1$</td><td>$\left[\begin{array}{c|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$</td></tr>
<tr><td>$L1 \rightarrow L2$</td><td>$a_g = \left[\begin{array}{c|ccc} 1 & [0,0] & [0,10] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$, or
$a_{id} = \left[\begin{array}{c|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$</td></tr>
<tr><td>$L3 \rightarrow L4$</td><td>$\left[\begin{array}{c|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [2,2] \\ 0 & [0,0] & [1,1] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$</td></tr>
<tr><td>$L4 \rightarrow L1$</td><td>$\left[\begin{array}{c|ccc} 1 & [0,0] & [1,1] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [1,1] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$</td></tr>
</table>

(a)

<table>
<tr><th>Iteration</th><th>Node L2</th></tr>
<tr><td>(i)</td><td>$\left[\begin{array}{c|ccc} 1 & [0,0] & [0,10] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$</td></tr>
<tr><td>(ii)</td><td>$\left[\begin{array}{c|ccc} 1 & [0,0] & [0,10] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,20] \\ 0 & [0,0] & [0,0] & [0,0] \\ 0 & [0,0] & [0,0] & [0,0] \end{array}\right]$</td></tr>
</table>

(b)

Fig. 2 Abstract transformers and fixpoint analysis with the $\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ domain for Ex. 3.3.

incomparable abstract transformers for guards in the ATA framework. Consequently, the abstract transformer for $L1 \rightarrow L2$, denoted by $a_{L1 \rightarrow L2}$, can either be the identity transformer a_{id} or the transformer a_g. (See Fig. 2.) If we use $a_{L1 \rightarrow L2} = a_g$, the analysis is able to obtain bounds constraints on the loop variable i. The guard constraints are propagated on the first iteration, and consequently, the iteration stabilizes on the second iteration. Thus, the function summary that $\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ learns, denoted by $S_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]}$, is $r' = [0, 20]x$. This summary captures the bounds property, but not the evenness property. Notice that $S_f = S_{\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]} \wedge S_{\text{MOS}}$.

Consider the instantiation of the ATA framework with strided-intervals over bitvectors [26], denoted by $\mathcal{SI}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$. 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, $\text{ATA}[\mathcal{SI}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ 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 $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$ and $\mathcal{SI}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$ 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 $\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ nor $\text{ATA}[\mathcal{SI}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ will be able to capture the bounds property for Ex. 3.3, because they are missing relational information between the loop counter i and the variable r. However, the reduced product of $\text{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ (or $\text{ATA}[\mathcal{SI}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$) and MOS can learn the exact function summary.

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,

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 $ct \in symbols(C : \overrightarrow{d}) \rightarrow \mathbb{Z}_{2^w}$*

join, and widen operations in the base domain. The equality operation is not the exact equality operation; that is, $(a_1 \widetilde{==} a_2)$ can return false, even if $\gamma(a_1) = \gamma(a_2)$. For instance, consider the following two abstract transformers in ATA$[\mathcal{I}_{\mathbb{Z}_{2^w}}{}^2]$, $a_1 = \begin{bmatrix} 1 & \top \\ 0 & [0,0] \end{bmatrix}$ and $a_2 = \begin{bmatrix} 1 & \top \\ 0 & [1,1] \end{bmatrix}$ (columns labelled 1, x). Clearly, $(a_1 \widetilde{==} a_2)$ is false, although $\gamma(a_1) = \gamma(a_2) = \{(x', x) : x', x \in \mathbb{Z}_{2^w}\}$; i.e., the concretization of both is the universe set.

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 App. A, with the exception of widening and abstraction operations.

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 $a, a' \in$ 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 $a'' = a' \circ^\sharp a$, $\gamma(a'') \supseteq$

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

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

$\{ (\overrightarrow{v}; \overrightarrow{v}'') \mid \exists \overrightarrow{v}'.R[\overrightarrow{v}; \overrightarrow{v}'] \wedge R'[\overrightarrow{v}'; \overrightarrow{v}''] \}$. This condition translates to:

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

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, abstract each result of matrix multiplication, 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.

4.2.1 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 $\widehat{\alpha}(\varphi)$, is the best value in $\mathcal{B}$ that over-approximates the set of all concrete affine transformers $(C : \overrightarrow{d})$ that satisfy φ [27,35]. For all $b \in \mathcal{B}$, the *symbolic concretization* of $\mathcal{B}$, denoted by $\widehat{\gamma}(b)$, maps b to a formula $\widehat{\gamma}(b) \in \mathbb{L}$ such that b and $\widehat{\gamma}(b)$ represent the same set of concrete affine transformers (i.e., $\gamma(b) = [\![\widehat{\gamma}(b)]\!]$). We expect the base domain $\mathcal{B}$ to provide the $\widehat{\gamma}$ operation. In our framework, there are slightly different variants of $\widehat{\alpha}$ and $\widehat{\gamma}$ according to which vocabulary of symbolic constants are involved. For instance, we use $\widehat{\gamma}'$ to denote symbolic concretization in terms of the primed symbolic constants $symbols(C' : \overrightarrow{d}')$. Similarly, $\widehat{\alpha}''$ 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:

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

(Note that $\widehat{\gamma}(base(a))$ and $\widehat{\gamma}'(base(a'))$ are formulas over $symbols(C : \overrightarrow{d})$ and $symbols(C' : \overrightarrow{d}')$ respectively.) Past literature [27,35,6] 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 [27,35] 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 [27], where the essential difference is in how variable *lower* is initialized (shown in gray in lines 1–3). Alg. 1 needs a method to enumerate a generator set gs for each $b \in \mathcal{B}$. The generator set gs is a set of concrete affine transformers, whose abstractions—when joined—results in b. Thus, $gs(b) = \{t_1, t_2, \ldots, t_l\}$, such that $b = \bigsqcup_{i=0}^{l} \beta(t_i)$, where the representation function $\beta(t)$ maps a concrete model t to the least value in $\mathcal{B}$ that overapproximates $\{t\}$ [27]; i.e., $\beta(t) = \alpha(\{t\})$. Such a set exists for any abstract domain with finite height, and 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 combinations 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 $\left[\begin{array}{c|c} 1 & \overrightarrow{d}' + \overrightarrow{d} \cdot C' \\ \hline 0 & C \cdot C' \end{array}\right]$. Because *lower* is initialized to $\bigsqcup \{\beta(t \times t') \mid t \in gs_1, t' \in gs_2\}$ rather than $\bot$, the number of SMT calls in the symbolic abstraction is significantly reduced, compared to the algorithm from [27]. The function $GetModel$, used at line 5, returns the model $M \in symbols(C'' : \overrightarrow{d}'') \rightarrow \mathbb{Z}_{2^w}$ satisfying the formula $(\varphi \wedge \neg\widehat{\gamma}(lower))$ given to the SMT solver at line 4, where φ is defined in Eqn. (3). Thus, the model M is a concrete affine transformer in a''. 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 $\varphi \Rightarrow \widehat{\gamma}(lower)$. The loop terminates when the height of the base domain $\mathcal{B}$ is finite. Thus, this algorithm is not applicable when the base domain has infinite height. Devising a symbolic abstraction algorithm for abstract domains with infinite height is beyond the scope of this work.

Algorithm 1 Abstract Composition via Symbolic Abstraction

1: $gs_1 \leftarrow \{t_1, t_2, ..., t_{l_1}\}$ ▷ where $base(a) = \bigsqcup_{i=0}^{l_1} \beta(t_i)$
2: $gs_2 \leftarrow \left\{t'_1, t'_2, ..., t'_{l_2}\right\}$ ▷ where $base(a') = \bigsqcup_{i=0}^{l_2} \beta(t'_i)$
3: $lower \leftarrow \bigsqcup \{\beta(t \times t') \mid t \in gs_1, t' \in gs_2\}$
4: **while** $r \leftarrow SMTCall(\varphi \wedge \neg\widehat{\gamma}(lower))$ *is Sat* **do**
5: $\quad M \leftarrow GetModel(r)$
6: $\quad lower \leftarrow lower \sqcup \beta(M)$
7: **return** *lower*

4.2.2 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 $\mathbb{Z}_{2^w}$. We introduce the concept of a *foundation domain*, denoted by $\mathcal{F}_{\mathcal{B}}$, to represent the abstractions of subsets of $\mathbb{Z}_{2^w}$ 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 $\mathbb{Z}_{2^w}$. 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* $bv \in \mathbb{Z}_{2^w}$
$\mathcal{F}$	$(f_1 +^\sharp f_2)$	*abstract addition*
$\mathcal{F}$	$(f_1 \times^\sharp f_2)$	*abstract multiplication*

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

$$a' \circ_{\mathrm{NR}} a = a'', \textit{ where } base(a'') \in (symbols(C : \overrightarrow{d}) \to \mathcal{F}) \tag{4}$$
$$\wedge \Big(\bigwedge_{1 \leqslant i,j \leqslant k} (\; base(a'')[c_{ij}] = \underset{1 \leqslant l \leqslant k}{\Sigma^\sharp} (base(a)[c_{il}] \times^\sharp base(a')[c_{lj}]) \Big)$$
$$\wedge \Big(\bigwedge_{1 \leqslant j \leqslant k} base(a'')[d_j] = \underset{1 \leqslant l \leqslant k}{\Sigma^\sharp} (base(a)[d_l] \times^\sharp base(a')[c_{lj}]) +^\sharp base(a')[d_j] \Big) \Big\}.$$

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).

Theorem 4.1 *ComposeNonRelationalSoundness of* $\circ_{NR}$

$$\gamma(a') \circ \gamma(a) \subseteq \gamma(a' \circ_{NR} a). \tag{5}$$

We provide a proof of soundness for $a' \circ_{\mathrm{NR}} a$ in App. 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 constructing the non-relational small-set, interval [2], and strided-interval [26] base domains.

Small sets. $\mathcal{F}_{SS_n} \stackrel{def}{=} \{\top\} \cup \{S | S \subseteq \mathbb{Z}_{2^w} \wedge |S| \leqslant n\}$. The join operation is defined by: $(f_1 \sqcup f_2) = \begin{cases} f_1 \cup f_2 & \textit{if } |f_1 \cup f_s| \leqslant n \\ \top & \textit{otherwise} \end{cases}$
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 [26,31] for the details of the abstract-domain operations.)

4.2.3 Affine-Closed Base Domain.

We discuss the special case when the base domain $\mathcal{B}$ is affine-closed, i.e., $\mathcal{B} =$ AG. In this case, abstract composition is defined as follows:

$$a' \circ_{\mathrm{AG}} a = a'',\ where\ \gamma_{\mathrm{AG}}(base(a'')) = \left\langle \{t_i \times t'_j \mid 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l'\} \right\rangle \wedge \\ \gamma_{\mathrm{AG}}(base(a)) = \langle \{t_1, t_2, ...t_l\} \rangle \wedge \gamma_{\mathrm{AG}}(base(a')) = \left\langle \{t'_1, t'_2, ...t'_{l'}\} \right\rangle \quad (6)$$

Lemma 5.1 in [23] 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. (6), but makes an extra SMT call to ensure that the result is sound. Because Eqn. (6) 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.

4.2.4 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: $\gamma(\alpha(t_i)) = \gamma(\beta(t_i))$.
- 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 $b = \bigsqcup_{i=0}^{l} \beta(t_i)$, 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 t_i in the rational space $\mathbb{Q}^{(k+k^2)}$.

Any convex abstract domain over rationals, such as polyhedra [5] or octagons [20], can be used to create a weakly-convex domain over bitvectors [34, 33]. Abstract composition for weakly-convex base domains is defined as fol-

lows:

$$a' \circ_{\mathrm{WC}} a = a'', \text{ where } base(a'') = \tag{7}$$

$$\begin{cases} \bigsqcup \left\{ \beta(t_i \times t'_j) \mid 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\} & \textit{if there are no overflows in any matrix multiplication } t_i \times t'_j \\ \top_{\mathcal{B}} & \textit{otherwise} \end{cases}$$

$$\text{where } base(a) = \sqcup \{\beta(t_1), \beta(t_2), \ldots, \beta(t_l)\}$$
$$\textit{and } base(a') = \sqcup \{\beta(t'_1), \beta(t'_2), \ldots, \beta(t'_{l'})\}.$$

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.)

Theorem 4.2 *ComposeWeaklyConvexSoundness of* $\circ_{WC}$

$$\gamma(a') \circ \gamma(a) \subseteq \gamma(a' \circ_{WC} a). \tag{8}$$

We provide a proof of soundness for $a' \circ_{\mathrm{WC}} a$ in App. B.2. Similar to the affine-closed case, abstract composition has time complexity $\mathcal{O}(H^2 k^3)$, where H is the height of the $\mathcal{B}$ lattice.

Practical concerns. With the exception of the non-relational base domain, the complexity of the abstract-composition algorithms is dependent on the height of the abstract-domain elements involved in the composition, i.e., h and h'. Practical implementations of abstract composition might decide to return $\top$ for abstract composition if the number of matrices to multiply is beyond some threshold, say t, so that the complexity of the abstract composition is $\mathcal{O}(tk^3)$.

4.3 Guards

In this subsection, we discuss abstract-transformer generation for conditional statements in the program. For many abstract domains, these can be implemented by performing the intersection of the identity transformation with the abstraction of the guard (expressed in the post-state vocabulary). Unfortunately, the abstract-domain elements in our framework are not necessarily closed under intersection. Consider the abstraction of the true branch of the conditional statement $'$if(v==0)$'$, which can be represented as the intersection of the two abstract values a_1 and a_2 for the vocabulary $V = \{v\}$. a_1 represents the affine transformation for the guard $v' = 0$, and a_2 represents the identity affine transformation $v' = v$. 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 conjunction of these constraints is satisfied only by the point $p = (v' = 0, v = 0)$. 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 $v' = v \cdot c$, and thus must contain all points of the form $(v' = k \cdot c, v = k)$, where $k \in \mathbb{Z}_{2^w}$. (See Fig. 3.) In general, the ATA framework cannot express guards because it is not closed under intersection.

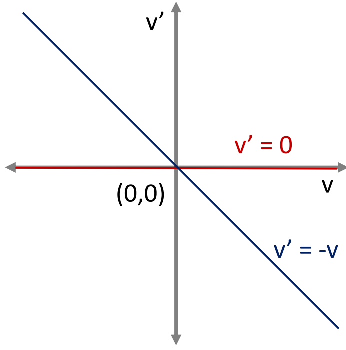

Fig. 3 Abstraction for the true branch of conditional statement "$if(v == 0)$."

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 $S_G = \{v \leqslant 10\}$, with the ATA$[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ domain. Consider $a_3 = \left[\begin{array}{c|c} 1 & [0,10] \\ \hline 0 & [0,0] \end{array}\right]$ and $a_4 = \left[\begin{array}{c|c} 1 & [0,0] \\ \hline 0 & [1,1] \end{array}\right]$. a_3 specifies the guard constraint $0 \leqslant v' \leqslant 10$, while a_4 is the identity transformation $v' = v$. 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 (for edge $L1 \rightarrow L2$) in Ex. 3.3. If an identity transfer had been used for the guard statement the analysis would not have established that the desired bounds constraints hold.

4.3.1 Handling Guards through explicit-form reformulation

As mentioned before, there can exist multiple incomparable abstract transformers for guards in the ATA framework. In this subsection, we show how to obtain those abstract transformers for different kind of guards.

The key idea is to reformulate the guard conditions explicitly as an affine transformation over program variables. Below is the discussion of several guard conditions and their corresponding reformulations:

Affine Equality Guards. An affine equality guard g_{ae} is of the form $\Sigma_{i=1}^{k} g_i \cdot v_i + g_0 = 0$, where $g_0, g_1, \ldots, g_k \in \mathbb{Z}_{2^w}$. A reformulation is possible, if there exists a g_i that has a multiplicative inverse. In particular, each odd element in $\mathbb{Z}_{2^w}$ has a multiplicative inverse (which may be found in time O(log w) [36, Fig. 10-5]), but no even element has a multiplicative inverse. Let us say that g_j is one such odd coefficient. Then, the reformulation is

$$v'_j = v_j \wedge v'_j = -g_j^{-1}(\Sigma_{i \neq j} g_i \cdot v_i + g_0),$$

where g_j^{-1} is the multiplicative inverse of g_j. The reformulation provides two (incomparable) abstract transformers for $\alpha(g_{ae})$:

1. $\alpha(v'_j = v_j) = \alpha(I_{k+1})$
2. $\alpha(v'_j = -g_j^{-1}(\Sigma_{i \neq j} g_i \cdot v_i + g_0)) =$

$$\alpha\left(\left[\begin{array}{c|cccc} 1 & 0 & -g_j^{-1} \cdot g_0 & 0 \\ \hline 0 & I_{j-1} & [-g_j^{-1} \cdot g_1, -g_j^{-1} \cdot g_2, \ldots - g_j^{-1} \cdot g_{j-1}]^t & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & [-g_j^{-1} \cdot g_{j+1}, -g_j^{-1} \cdot g_{j+2}, \ldots, -g_j^{-1} \cdot g_k]^t & I_{k-j} \end{array}\right]\right)$$

Note that because a multiplicative inverse exists for all values (with the exception of zero) in the case of rational numbers or real numbers, this reformulation can be applied for any affine equality guard over rationals or reals.

Example 4.1 Consider the guard condition $g_{ae_ex} := 2x + 3y = 7$, where x and y are 4-bit values. Because the coefficient of y is odd, its inverse exists; that is, $3^{-1} = 11$. Thus, g_{ae_ex} can be reformulated as $y' = y \wedge y = 10x + 13$. (Note that $10 = (-2) \cdot 3^{-1}$ and $13 = 7 \cdot 3^{-1}$ in modulo-16 bit-vector arithmetic.) Thus, for ATA$[\mathcal{I}^6_{\mathbb{Z}_{16}}]$, the abstract transformer for g_{ae_ex} can be either the identity transformer or the following transformer:

$$\begin{array}{c} \begin{array}{ccc} 1 & x & y \end{array} \\ \left[\begin{array}{c|cc} 1 & [0,0] & [13,13] \\ \hline 0 & [1,1] & [10,10] \\ 0 & [0,0] & [0,0] \end{array}\right] \end{array}.$$

□

Affine Congruences on a power of 2. An affine-congruence guard on a power of two, denoted by g_{ac}, is of the form $(\Sigma_{i=1}^{k} g_i \cdot v_i)\%2^m = g_0$, where $g_0, g_1, \ldots, g_k \in \mathbb{Z}_{2^w}, 1 \leqslant m \leqslant w$, and $0 \leqslant g_0 \leqslant 2^m$. A reformulation is possible, if there exists a g_i that has a multiplicative inverse. Let us say that g_j is one such odd coefficient. Then, the reformulation is

$$(v'_j = v_j) \wedge (v'_j = -g_j^{-1}(\Sigma_{i \neq j} g_i \cdot v_i) + g_j^{-1} \cdot d_j) \wedge (d_j\%2^m = g_0).$$

Similar to affine equality guards, the reformulation provides two (incomparable) abstract transformers for $\alpha(g_{ac})$:

1. $\alpha(v'_j = v_j) = \alpha(I_{k+1})$

2. $\alpha(v'_j = -g_j^{-1}(\Sigma_{i \neq j} g_i \cdot v_i) + g_j^{-1} \cdot d_j) \wedge (d_j \% 2^m = g_0)) =$

$$\alpha\left(\left\{\left[\begin{array}{c|cccc} 1 & 0 & g_0 + c \cdot 2^m & & 0 \\ \hline 0 & I_{j-1} & [-g_j^{-1} \cdot g_1, -g_j^{-1} \cdot g_2, \ldots, -g_j^{-1} \cdot g_{j-1}]^t & & 0 \\ 0 & 0 & 0 & & 0 \\ 0 & 0 & [-g_j^{-1} \cdot g_{j+1}, -g_j^{-1} \cdot g_{j+2}, \ldots, -g_j^{-1} \cdot g_k]^t & & I_{k-j} \end{array}\right] \middle| \; c \in \mathbb{Z}_{2^{w-m}} \right\}\right).$$

Example 4.2 Consider the guard condition $g_{ac_ex} := (3x + 2y \equiv 1(mod\ 4))$, where x and y are 4-bit values. Because the coefficient of x is odd, its inverse exists, that is, $3^{-1} = 11$. Thus, g_{ac_ex} can be reformulated as $x' = x \wedge x' = 10y + d_x \wedge d_x \equiv 3(mod\ 4)$. Thus, for ATA[AG], the abstract transformer for g_{ac_ex} can be either the identity transformer or the following abstract transformer:

$$\left\{ \begin{array}{c} \scriptstyle 1 \;\; x \;\; y \\ \left[\begin{array}{c|cc} 1 & 3 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 10 & 1 \end{array}\right] \end{array}, \begin{array}{c} \scriptstyle 1 \;\; x \;\; y \\ \left[\begin{array}{c|cc} 1 & 7 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 10 & 1 \end{array}\right] \end{array}, \begin{array}{c} \scriptstyle 1 \;\; x \;\; y \\ \left[\begin{array}{c|cc} 1 & 11 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 10 & 1 \end{array}\right] \end{array}, \begin{array}{c} \scriptstyle 1 \;\; x \;\; y \\ \left[\begin{array}{c|cc} 1 & 15 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 10 & 1 \end{array}\right] \end{array} \right\}.$$ □

Single-Variable Inequalities. A single-variable inequality guard, denoted by g_{svi}, is of the form $v_j \leqslant_u g_0$, where $g_0 \in \mathbb{Z}_{2^w}$ and subscript u stands for unsigned operation. g_{svi} can be reformulated by,

$$(v'_j = v_j) \wedge (v'_j = d_j) \wedge (d_j \in [0, g_0]).$$

The reformulation provides two (incomparable) abstract transformers for $\alpha(g_{svi})$:

1. $\alpha(v'_j = v_j) = \alpha(I_{k+1})$
2. $\alpha((v'_j = d_j) \wedge (d_j \in [0, g_0])) = \alpha\left(\left\{\left[\begin{array}{c|ccc} 1 & 0 & d_j & 0 \\ \hline 0 & I_{j-1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{k-j} \end{array}\right] \middle| \; d_j \in [0, g_0] \right\}\right).$

Example 4.3 Consider the guard condition $g_{svi_ex} := x \leqslant_u 7$, where x and y are 4-bit values. Thus, g_{svi_ex} can be reformulated as $x' = d_x \wedge d_x \in [0, 7]$. Thus, for ATA[$\mathcal{I}^6_{\mathbb{Z}_{16}}$], the abstract transformer for g_{svi_ex} can be either the identity transformer or the following transformer:

$$\begin{array}{c} \scriptstyle 1 \quad\; x \qquad\; y \\ \left[\begin{array}{c|cc} 1 & [0,7] & [0,0] \\ \hline 0 & [0,0] & [0,0] \\ 0 & [0,0] & [1,1] \end{array}\right] \end{array}.$$ □

Inequalities with multiple variables. Consider a simple two-variable inequality $v_i \leqslant_u v_j$. Due to overflow in bitvector arithmetic, this constraint is not equivalent to $v'_j = v_i + [0, max]$. Consequently, there does not exist an explicit reformulation of a multiple-variable inequality as an affine transformation. However, such a reformulation should be possible for affine transformations over rationals or reals.

4.3.2 Guards in the MOS domain.

Section 6 in [23] showed an alternative method to deal with guards, by adding an *indicator variable* for each (affine-equality) guard expression in the program. They suggest to postpone the decision taken at the guard. Instead of performing the intersection, an extra indicator variable is introduced for every guard expression in the program. Suppose that edges with guards are numbered $k+1,\ldots,m$. They instrument the original program by introducing fresh indicator variables $v_{k+1},\ldots,v_m$, one for each guard, which are initialized to zero. Each of the indicator variable, $\{v_j\}$, is assigned to the appropriate guard-equality affine expression at the corresponding edge. For instance, suppose that v_{k+1} corresponds to the guard $g := \Sigma_{i=0}^{k} g_i \cdot v_i + g_0 = 0$. Then, the guard condition will be treated as affine assignment for the indicator variable $v'_{k+1} : v'_{k+1} = \Sigma_{i=0}^{k} g_i \cdot v_i + g_0$. Finally, after the fix-point analysis, only those vectors are selected from the result whose indicator variables all equal 0.

Example 4.4 Consider the guard condition $g_{ae_ex} := 2x+3y = 7$ (from Ex. 4.1, where x and y are 4-bit values. An indicator variable g is added to track the guard condition g_{ae_ex}. Thus, for the MOS domain, the abstract transformer for g_{ae_ex} is: $\left\{ \begin{array}{c} \scriptstyle 1\ x\ y\ g \\ \left[\begin{array}{c|ccc} 1 & 0 & 0 & -7 \\ \hline 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{array}\right] \end{array} \right\}$. □

While this approach is more precise than our treatment of guards, it is costly because it requires tracking extra indicator variables in the analysis. However, their approach to indicator variables can be adopted into the ATA framework in a similar manner.

Example 4.5 Consider the guard condition $g_{svi_ex} := x \leqslant_u 7$ shown in Ex. 4.3. Recall that g_{svi_ex} can be reformulated as $x' = x \wedge x' = d_x \wedge d_x \in [0,7]$. Similar to Ex. 4.4, an indicator variable $g := x - [0,7]$ is added. Then, the guard condition g_{svi_ex} will be treated as an assignment for the indicator variable g. Thus, for ATA$[\mathcal{I}^{9}_{\mathbb{Z}_{16}}]$, the abstract transformer for g_{svi_ex} is: $\begin{array}{c} \scriptstyle 1\quad x\quad y\quad g \\ \left[\begin{array}{c|ccc} 1 & [0,0] & [0,0] & [-7,0] \\ \hline 0 & [1,1] & [0,0] & [1,1] \\ 0 & [0,0] & [1,1] & [0,0] \end{array}\right] \end{array}$. Observe that we are able to express the guard condition g_{svi_ex} precisely, but need three more symbolic constants for the variable g. Finally, after the fix-point analysis, only those vectors are selected from the result where g equals 0. □

4.4 Merge Functions

Müller-Olm and Seidl [23] discussed local variables for the interprocedural analysis over the MOS domain. In this subsection, we discuss local variable for the interprocedural analysis for the ATA framework. Knoop and Steffen [13] extended the Sharir and Pnueli [32] algorithm for interprocedural dataflow analysis to handle local variables. Suppose at a call site CS, procedure P calls

procedure Q. The global variables, denoted by $\overrightarrow{g}$, are accessible to Q, but the local variables, denoted by $\overrightarrow{l}$, in P are inaccessible to Q. Thus, the values of local variables after the call site CS come from the values before the call point, and the values of global variables after the call site CS come from the values at the return site in procedure Q. A *merge function* is used to combine the abstract-domain element before the call to Q with the abstract-domain element returned by Q to create the abstract-domain element to use in P after the call to Q has finished.

We assume that in each function, the local variables are initialized to 0. This assumption makes it easier to construct merge functions, and thus easy to prove their soundness. (See the proof of Thm. C.1.) To simplify the discussion, assume that all scopes have the same number of locals, and that each vocabulary $\overrightarrow{v}$ consists of subvocabularies $\overrightarrow{g}$ and $\overrightarrow{l}$—that is, $\overrightarrow{v} = (\overrightarrow{g}, \overrightarrow{l})$ Suppose that we have two relations, $R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$ and $R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$, each of which is a subset of $\mathbb{Z}^k_{2^w} \times \mathbb{Z}^k_{2^w}$, where R is the transition relation from the start state of the calling procedure P to the call site CS, and R' is the transition relation from the start state to the return site of the called procedure Q. Operationally, after completing the call at the call site CS, we want $\textsc{Merge}(R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'], R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'])$ to act as a modified relational composition in which R' acts like the identity function on locals, so that $\overrightarrow{l}'$ values from R are passed through R' unchanged to become the $\overrightarrow{l}'$ values of the result. This semantics can be specified as follows:

$$\begin{aligned}&\textsc{Merge}(R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'], R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']) \\ &\quad = \textsc{RevertLocals}(R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']) \circ R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']\end{aligned} \tag{9}$$

We define $\textsc{RevertLocals}(R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'])$ as follows:

$$\textsc{RevertLocals}(R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']) \stackrel{def}{=} \{(\overrightarrow{g}, \overrightarrow{q}, \overrightarrow{g}', \overrightarrow{q}) \mid R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']\} \tag{10}$$

We can partition $symbols(C\colon \overrightarrow{d})$ into globals and locals to write the matrix as $\left[\begin{array}{c|c|c} 1 & \overrightarrow{d}_g & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{lg} & C_{ll} \end{array}\right]$, which represents the affine transformation

$$(\overrightarrow{g}' = \overrightarrow{g} \cdot C_{gg} + \overrightarrow{l} \cdot C_{lg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l}' = \overrightarrow{g} \cdot C_{gl} + \overrightarrow{l} \cdot C_{ll} + \overrightarrow{d_l}). \tag{11}$$

Let $a, a' \in \text{ATA}[\mathcal{B}]$ be the abstract transformers that represent the relations $R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$ and $R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$, respectively. Then the merge function for a_1 and a_2 is defined as follows:

$$\begin{aligned} Merge(a, a') &= a'', \text{ where} \\ base(a'') &= (havoc(base(a'), lsyms) \sqcap havoc(base(Id), gsyms)) \circ a \\ &\quad lsyms = symbols(C_{gl}) \cup symbols(C_{ll}) \cup \\ &\qquad\qquad symbols(d_l) \cup symbols(C_{lg}) \\ &\quad gsyms = symbols(C_{gg}) \cup symbols(d_g) \end{aligned} \tag{12}$$

$lsyms$ are the symbols in the affine transformation that involve local variables. $gsyms$ are the symbols in the affine transformation that are not in $lsyms$. The expression $b_{gi} = ((havoc(base(a'), lsyms) \sqcap havoc(base(Id), gsyms))$ transforms each affine transformer $\left[\begin{array}{c|c|c} 1 & \vec{d}_g & \vec{d}_l \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{lg} & C_{ll} \end{array}\right] \in \gamma(base(a))$ to $\left[\begin{array}{c|c|c} 1 & \vec{d}_g & 0 \\ \hline 0 & C_{gg} & 0 \\ 0 & 0 & I \end{array}\right]$. In this way, b_{gi} ensures that the modifications of the globals at the return point of Q are accounted for, while the locals for P pass through Q unmodified.

Theorem 4.3 *MergeIsSoundSoundness of the merge function*

$$\textsc{Merge}(\gamma(a), \gamma(a')) \subseteq \gamma(Merge(a, a')). \tag{13}$$

We provide a proof of Thm. C.1 in App. C.

5 Implementation Design

In this section, we discuss key implementation design decisions that can affect the performance and the precision of the analysis.

Given source files to be analyzed, a call graph and control-flow graphs (CFG) for each procedure are created. LLVM is well-suited for this purpose [15, 18]. Then, the call graph and CFG can be abstracted into a constraint system, which can be solved through iterative fixed-point computation. The WALi [11, 14] system is well-suited to creating such a constraint system in the form of a Weighted Pushdown System (WPDS). The transitions in the WPDS correspond to CFG edges from a basic block to one of its successors. The weights on the edges are abstract transformers for the ATA framework. Finally, interprocedural analysis is done by performing post*, followed by the path-summary operation [28] to obtain overapproximating function summaries and an overapproximation of the reachable states at all program points.

The key design decisions that can affect an analysis based on the ATA framework are as follows:

- Abstract Transformer Generation: The implementation in [17] generates abstract transformers for the MOS domain using semantic reinterpretation [10, 24, 25, 19, 16]. Semantic reinterpretation is a principled method based on the idea of factoring the concrete semantics of a programming language into two parts: (i) a client specification, and (ii) a semantic core. The interface to the core consists of certain base-types, map-types, and operators (sometimes called a semantic algebra [30]), and the client is expressed in terms of this interface. This organization permits the core to be reinterpreted to produce an alternative semantics for the programming language. The reinterpretation of the core must consist of a domain of abstract integers, a domain of abstract states, abstract multiplication and abstract addition of abstract integers, and operations to lookup a variable's value in an abstract state and to create an updated version of a given abstract

state. [17] presents the semantic core for the MOS domain. The semantic core can be extended to apply to the ATA framework, which will enable a systematic generation of abstract transformers.

- Guards: In §4.3, we show that affine equality, affine congruence, and single-variable inequality guards can be reformulated in an explicit form, which allows them to be expressed as constraints over symbolic constants that can be abstracted in the base domain using the $\alpha(Cond)$ operations. Furthermore, an extra indicator variable can be added for a guard expression to enhance precision, at the cost of performance. Thus, there are two key design decisions for guards: (i) whether to use indicator variables, (ii) in case indicator variables are not used, whether to use the guard transformation a_g or the identity transformation a_{id}.

  Item (i) can be addressed by performing experiments on real-world benchmarks. Item (ii) can be addressed by a simple heuristic that delays the decision to the fix-point iteration. In other words, the abstract composition at the guard is calculated with both abstract-transformer choices for the guards, and the more precise result is chosen. In cases where the result is incomparable, the non-identity transformer is picked. For instance, in §3, the abstract transformer for $L1 \rightarrow L2$, denoted by $a_{L1 \rightarrow L2}$, is either the identity transformer a_{id} or the transformer a_g. In first ten iterations, abstract composition is more precise with a_{id}. However, after ten iterations it is more precise to perform abstract composition with a_g to precisely deduce the constraint $i \in [0, 10]$ and reach a fix-point.
- Sparse Representation: In real-world programs, most of the $k + k^2$ symbolic constants in an affine transformation tend to be zero. Thus, the ATA framework implementation will benefit from using sparse representations of affine-transformer matrices. For instance, the complexity of the sparse matrix-multiplication operation used in the abstract-composition operation can be $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$.

6 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 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. However, for base domains, such as polyhedra, the

abstract-composition methods shown in this paper are not directly applicable, because the domain has infinite height.

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 $\mathrm{ATA}[\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}]$ domain because the coefficients in the abstract values are intervals.

Miné [21] 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 $\mathrm{ATA}[\mathcal{B}]$, 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. [29] 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.

Goubault et al. [8] presented the zonotope domain, which, like ATA framework, is also a functional abstraction; that is, they provide abstractions of concrete transformers. The zonotope domain is relational, where the abstraction of the program state at each program point gives a relation between the initial value and the current value of each variable. While their abstract domain is different from ATA framework, they showed that performing abstract interpretation through functional abstraction leads to a modular and scalable analysis.

An abstract-domain element in $\mathrm{ATA}[\mathcal{B}]$ can be seen as an abstraction over sets of functions: $\mathbb{Z}_{2^w}^k \rightarrow \mathbb{Z}_{2^w}^k$. Jeannet et al. [9] provide a theoretical treatment of the relational abstraction of functions. They describe existing and new methods of abstracting functions of signature: $D_1 \rightarrow D_2$, resulting in a family of relational abstract domains. $\mathrm{ATA}[\mathcal{B}]$ is not captured by their framework of functional abstractions.

References

1. Chen, L., Miné, A., Wang, J., Cousot, P.: Interval polyhedra: An abstract domain to infer interval linear relationships. In: SAS (2009)

2. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. 2nd. Int. Symp on Programming. Paris (1976)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL, pp. 238–252 (1977)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL (1979)
5. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: POPL (1978)
6. Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract domains of affine relations. TOPLAS (2014)
7. Gange, G., Navas, J., Schachte, P., Søndergaard, H., Stuckey, P.: Abstract interpretation over non-lattice abstract domains. In: SAS (2013)
8. Goubault, E., Putot, S., Védrine, F.: Modular static analysis with zonotopes. In: CAV (2012)
9. Jeannet, B., Gopan, D., Reps, T.: A relational abstraction for functions. In: SAS (2005)
10. Jones, N., Mycroft, A.: Data flow analysis of applicative programs using minimal function graphs. In: POPL, pp. 296–306 (1986)
11. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (2007). Www.cs.wisc.edu/wpis/wpds/download.php
12. King, A., Søndergaard, H.: Automatic abstraction for congruences. In: VMCAI (2010)
13. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: CC (1992)
14. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: CAV (2005)
15. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Int. Symp. on Code Generation and Optimization (2004)
16. Lim, J., Reps, T.: A system for generating static analyzers for machine instructions. In: CC (2008)
17. Lim, J., Reps, T.: TSL: A system for generating abstract interpreters and its application to machine-code analysis. TOPLAS **35**(1) (2013)
18. LLVM: Low level virtual machine. Llvm.org
19. Malmkjær, K.: Abstract interpretation of partial-evaluation algorithms. Ph.D. thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas (1993)
20. Miné, A.: The octagon abstract domain. In: WCRE (2001)
21. Miné, A.: A few graph-based relational numerical abstract domains. In: SAS (2002)
22. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL (2004)
23. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. TOPLAS **29**(5) (2007)
24. Mycroft, A., Jones, N.: A relational framework for abstract interpretation. In: Programs as Data Objects (1985)
25. Nielson, F.: Two-level semantics and abstract interpretation. Theor. Comp. Sci. **69**, 117–242 (1989)
26. Reps, T., Balakrishnan, G., Lim, J.: Intermediate-representation recovery from low-level code. In: Part. Eval. and Semantics-Based Prog. Manip. (2006)
27. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: VMCAI (2004)
28. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP **58**(1–2) (2005)
29. Sankaranarayanan, S., Sipma, H., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: VMCAI (2005)
30. Schmidt, D.: Denotational Semantics. Allyn and Bacon, Inc., Boston, MA (1986)
31. Sen, R., Srikant, Y.: Executable analysis using abstract interpretation with circular linear progressions. In: MEMOCODE (2007)
32. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981)
33. Sharma, T., Reps, T.: Sound bit-precise numerical domains. In: VMCAI (2017)
34. Simon, A., King, A.: Taming the wrapping of integer arithmetic. In: SAS (2007)
35. Thakur, A., Elder, M., Reps, T.: Bilateral algorithms for symbolic abstraction. In: SAS (2012)
36. Warren Jr., H.: Hacker's Delight. Addison-Wesley (2003)

A Soundness of the Abstract-Domain Operations

In this section, we show that the abstract-domain operations for the ATA[$\mathcal{B}$] framework are sound with respect to the concrete semantics of the programming language.

Lemma A.1 *The bottom element represents the empty set.*

$$\gamma(\bot) = \emptyset \tag{14}$$

Proof

$$\begin{aligned} &\gamma(\bot) = \{(\vec{v}, \vec{v}') : \vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \gamma(\bot_{\mathcal{B}})\} \\ \Rightarrow &\gamma(\bot) = \{(\vec{v}, \vec{v}') : \vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \emptyset)\} \\ \Rightarrow &\gamma(\bot) = \emptyset \end{aligned}$$

□

Lemma A.2 *The equality operation is sound.*

$$(a_1 \widetilde{==} a_2) \Rightarrow (\gamma(a_1) == \gamma(a_2)) \tag{15}$$

Proof We will prove Lemma A.2 by contradiction. Assume that $a_1 \widetilde{==} a_2$ but $\gamma(a_1) \neq \gamma(a_2)$. Without loss of generality, we can assume that there exists $(\vec{v}, \vec{v}')$ such that:

$$\begin{aligned} &(\vec{v}, \vec{v}') \in \gamma(a_1) \wedge (\vec{v}, \vec{v}') \notin \gamma(a_2) \\ \Rightarrow\ &\vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \gamma(base(a_1)) \wedge (C : \vec{d}) \notin \gamma(base(a_2)) \\ \Rightarrow\ &\exists b.\ b \in \gamma(base(a_1)) \wedge b \notin \gamma(base(a_2)) \\ \Rightarrow\ &\gamma(base(a_1)) \neq \gamma(base(a_2) \\ \Rightarrow\ &base(a_1) \neq base(a_2) \text{ (by soundness of equality on } \mathcal{B}) \\ \Rightarrow\ &a_1 \neq a_2 \text{ (Contradiction!)} \end{aligned}$$

□

Lemma A.3 *The join operation is sound.*

$$\gamma(a_1 \tilde{\sqcup} a_2) \supseteq \gamma(a_1) \cup \gamma(a_2) \tag{16}$$

Proof Assume that Lemma A.3 is incorrect. Then there exists $(\vec{v}, \vec{v}') \notin \gamma(a_1 \tilde{\sqcup} a_2)$ such that:

$$\begin{aligned} &(\vec{v}, \vec{v}') \in \gamma(a_1) \cup \gamma(a_2) \\ \Rightarrow\ &\vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \gamma(base(a_1)) \\ &\text{(Without loss of generality.)} \\ \Rightarrow\ &\vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \gamma(base(a_1) \sqcup base(a_2)) \\ \Rightarrow\ &(\vec{v}, \vec{v}') \in \gamma(a_1 \tilde{\sqcup} a_2) \text{ (by soundness of join on } \mathcal{B}) \\ &\text{(Contradiction!)} \end{aligned}$$

□

The soundness of widening, statement abstractions, and identity function are easy to prove, and follow similar reasoning.

B Soundness of Abstract Composition

In this section, we show that the abstract-composition operations defined in §4.2 are sound. From Eqn. (2), an exact abstract composition $C_e = \gamma(a' \circ a)$ is defined as follows:

$$C_e = \{(\vec{v}, \vec{v}') \mid \exists (C : \vec{d}) \in \gamma(base(a)), (C' : \vec{d}') \in \gamma(base(a')), (C'' : \vec{d}'') : \\ (\vec{v}' = \vec{v} \cdot C'' + \vec{d}'') \wedge (C'' = C \cdot C') \wedge (\vec{d}'' = \vec{d} \cdot C' + \vec{d}')\}$$

B.1 Non-Relational Base Domain

In this section, we show that the fast abstract composition for ATA$[\mathcal{B}]$ (Eqn. (4)), when $\mathcal{B}$ is non-relational, is sound. Remember that any non-relational domain can be formulated as follows: $\mathcal{B} \overset{def}{=} symbols(C : \vec{d}) \to \mathcal{F}_{\mathcal{B}}$. The term $b[s]$, where $b \in \mathcal{B}$ and $s \in symbols(C : \vec{d})$, refers to the element in the foundation domain $f \in \mathcal{F}_{\mathcal{B}}$ corresponding to the symbol s.

Axiom 1 *Abstract addition is sound for $\mathcal{F}_{\mathcal{B}}$; i.e.,*

$$e_1 \in \gamma(f_1) \wedge e_2 \in \gamma(f_2) \Rightarrow e_1 + e_2 \in \gamma(f_1 +^{\sharp} f_2) \tag{17}$$

Axiom 2 *Abstract multiplication is sound for $\mathcal{F}_{\mathcal{B}}$; i.e.,*

$$e_1 \in \gamma(f_1) \wedge e_2 \in \gamma(f_2) \Rightarrow e_1 \times e_2 \in \gamma(f_1 \times^{\sharp} f_2) \tag{18}$$

Theorem B.1 *(Soundness of $\circ_{NR}$)*

$$C_e \subseteq \gamma(a' \circ_{NR} a). \tag{19}$$

Proof We will prove Thm. B.1 by contradiction. Consider a model $m = (\vec{v}, \vec{v}'')$, such that $m \in C_e$ and $m \notin \gamma(a' \circ_{\text{NR}} a)$. We will show that such a model cannot exists.

$$\begin{aligned}
& m \in \gamma(C_e) \\
\Leftrightarrow\ & \exists (C : \vec{d}) \in \gamma(base(a)), (C' : \vec{d}') \in \gamma(base(a')), (C'' : \vec{d}'') : \\
& \quad (\vec{v}' = \vec{v} \cdot C'' + \vec{d}'') \wedge (C'' = C \cdot C') \wedge (\vec{d}'' = \vec{d} \cdot C' + \vec{d}') \\
\Leftrightarrow\ & \exists (C : \vec{d}), (C' : \vec{d}'), (C'' : \vec{d}'') : (\vec{v}' = \vec{v} \cdot C'' + \vec{d}'') \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} (C''[i,j] = \sum_{1 \leqslant l \leqslant k} (C[i,l] \times C'[l,j])) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} (\vec{d}''[j] = (\sum_{1 \leqslant l \leqslant k} (\vec{d}[l] \times C'[l,j]')) + \vec{d}'[j]) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} C[i,j] \in \gamma(base(a)[c_{ij}]) \right) \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} \vec{d}[j] \in \gamma(base(a)[d_j]) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} C'[i,j] \in \gamma(base(a')[c_{ij}]) \right) \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} \vec{d}'[j] \in \gamma(base(a')[d_j]) \right) \\
\Rightarrow\ & \exists (C'' : \vec{d}'') : (\vec{v}' = \vec{v} \cdot C'' + \vec{d}'')
\end{aligned}$$

$$\wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} (C''[i,j] \in \underset{1 \leqslant l \leqslant k}{\Sigma^\sharp} (base(a)[c_{il}] \times^\sharp base(a')[c_{l,j}])) \right)$$

$$\wedge \left(\bigwedge_{1 \leqslant j \leqslant k} (\overrightarrow{d}''[j] \in \underset{1 \leqslant l \leqslant k}{\Sigma^\sharp} (base(a)[d_l] \times^\sharp base(a')[c_{l,j}]) +^\sharp base(a')[d_j]) \right)$$

(by application of axioms 1 and 2 to the expressions

$\underset{1 \leqslant l \leqslant k}{\Sigma} (C[i,l] \times C'[l,j])$ and $\underset{1 \leqslant l \leqslant k}{\Sigma} (\overrightarrow{d}[l] \times C'[l,j]') + \overrightarrow{d}'[j])$

$$\Leftrightarrow \exists (C'' : \overrightarrow{d}'') : (\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge b \in (symbols(C'' : \overrightarrow{d}'') \to \mathcal{F})$$

$$\wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} \left(b[c''_{ij}] = \underset{1 \leqslant l \leqslant k}{\Sigma^\sharp} (base(a)[c_{il}] \times^\sharp base(a')[c_{lj}]) \right) \right)$$

$$\wedge \left(\bigwedge_{1 \leqslant j \leqslant k} b[d''_j] = \underset{1 \leqslant l \leqslant k}{\Sigma^\sharp} (base(a)[d_l] \times^\sharp base(a')[c_{lj}]) +^\sharp base(a')[d_j] \right)$$

$\Leftrightarrow m \in \gamma(a' \circ_{\mathrm{NR}} a)$ (by Eqn. (4))

□

B.2 Weakly-Convex Base Domain

In this subsection, we present a proof of soundness of abstract composition for weakly-convex base domains, denoted by $a' \circ_{\mathrm{WC}} a$ (Eqn. (7)).

We present some useful axioms and lemmas before presenting the soundness theorem and its proof. Let $min_{\mathbb{Z}_{2^w}}$ and $max_{\mathbb{Z}_{2^w}}$ be the minimum and maximum bitvector values in $\mathbb{Z}_{2^w}$. Let $min_{\mathbb{Q}} = min_{\mathbb{Z}_{2^w}}$ and $max_{\mathbb{Q}} = max_{\mathbb{Z}_{2^w}}$.

Axiom 3 *$cast_{\mathbb{Q}}$ is distributive over bitvector addition in the absence of overflows: that is, if $min_{\mathbb{Q}} \leqslant cast_{\mathbb{Q}}(bv_1) + cast_{\mathbb{Q}}(bv_2) \leqslant max_{\mathbb{Q}}$, where $bv_1, bv_2 \in \mathbb{Z}_{2^w}$, then*

$$cast_{\mathbb{Q}}(bv_1) + cast_{\mathbb{Q}}(bv_2) = cast_{\mathbb{Q}}(bv_1 + bv_2) \tag{20}$$

Axiom 4 *$cast_{\mathbb{Q}}$ is distributive over bitvector multiplication in the absence of overflows: that is, if $min_{\mathbb{Q}} \leqslant cast_{\mathbb{Q}}(bv_1) \cdot cast_{\mathbb{Q}}(bv_2) \leqslant max_{\mathbb{Q}}$, where $bv_1, bv_2 \in \mathbb{Z}_{2^w}$, then*

$$cast_{\mathbb{Q}}(bv_1) \cdot cast_{\mathbb{Q}}(bv_2) = cast_{\mathbb{Q}}(bv_1 \cdot bv_2) \tag{21}$$

Lemma B.1 *$cast_{\mathbb{Q}}$ is distributive over matrix multiplication for bitvectors, if there are no overflows in the matrix multiplication. That is, for $n \times n$ matrices M and M' where $\forall_{1 \leqslant i,j \leqslant n} M[i,j], M'[i,j] \in \mathbb{Z}_{2^w}$,*

$$cast_{\mathbb{Q}}(M) \times cast_{\mathbb{Q}}(M') = cast_{\mathbb{Q}}(M \times M'). \tag{22}$$

Proof Let $M'' = cast_{\mathbb{Q}}(M) \times cast_{\mathbb{Q}}(M')$. Then,

$$\forall_{1 \leqslant i,j \leqslant n} : M''[i,j] = \Sigma_{1 \leqslant l \leqslant n} cast_{\mathbb{Q}}(M[i,l]) \cdot cast_{\mathbb{Q}}(M[l,j])$$

$$\Rightarrow \forall_{1 \leqslant i,j \leqslant n} : M''[i,j] = \Sigma_{1 \leqslant l \leqslant n} cast_{\mathbb{Q}}(M[i,l] \cdot M[l,j]) \text{ (by Axiom 4)}$$

$$\Rightarrow \forall_{1 \leqslant i,j \leqslant n} : M''[i,j] = cast_{\mathbb{Q}}(\Sigma_{1 \leqslant l \leqslant n} M[i,l] \cdot M[l,j]) \text{ (by Axiom 3)}$$

$$\Rightarrow cast_{\mathbb{Q}}(M) \times cast_{\mathbb{Q}}(M') = cast_{\mathbb{Q}}(M \times M')$$

□

Lemma B.2 *A convex combination of a set of rationals is inside bitvector boundaries if each of the rational values in the set is inside bitvector boundaries. Given any* $0 \leqslant \lambda_1, \lambda_2, \ldots, \lambda_l \leqslant 1$, *such that* $(\sum_{i=1}^{l} \lambda_i = 1)$.

$$min_{\mathbb{Q}} \leqslant q_1, q_2, \cdots, q_l \leqslant max_{\mathbb{Q}} \Rightarrow min_{\mathbb{Q}} \leqslant \sum_{i=1}^{l} \lambda_i q_i \leqslant max_{\mathbb{Q}} \tag{23}$$

Proof Suppose $min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i q_i$.

$$\begin{aligned} &\Rightarrow (min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i min_{\mathbb{Q}}) \text{ (because } min_{\mathbb{Q}} \leqslant q_1, q_2, \cdots, q_l) \\ &\Leftrightarrow \ min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i min_{\mathbb{Q}} \Rightarrow (min_{\mathbb{Q}} > min_{\mathbb{Q}}) \text{ (because } (\sum_{i=1}^{l} \lambda_i = 1).) \\ &\Leftrightarrow \ false \end{aligned}$$

Consequently, $min_{\mathbb{Q}} \leqslant \sum_{i=1}^{l} \lambda_i q_i$. The other inequality $\sum_{i=1}^{l} \lambda_i q_i \leqslant max_{\mathbb{Q}}$ can be proved in a similar fashion. □

Lemma B.3 *There are no overflows in a matrix multiplication of a convex combination of matrices, if there are no overflows in the matrix multiplications of the underlying matrices involved in the convex combination.*

$$\forall_{1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l'} : \ (t_i \times t'_j) \textit{ does not overflow} \Rightarrow (t \times t') \textit{ does not overflow, where}$$

$$\left(cast_{\mathbb{Q}}(t) = \sum_{i=1}^{l} \lambda_i \ cast_{\mathbb{Q}}(t_i)\right) \wedge \bigwedge_{i=1}^{l} (0 \leqslant \lambda_i \leqslant 1) \wedge \sum_{i=1}^{l} \lambda_i = 1, \tag{24}$$

$$\textit{and, } \left(cast_{\mathbb{Q}}(t') = \sum_{j=1}^{l'} \lambda'_j \ cast_{\mathbb{Q}}(t'_j)\right) \wedge \bigwedge_{j=1}^{l'} (0 \leqslant \lambda'_j \leqslant 1) \wedge \sum_{j=1}^{l'} \lambda'_j = 1. \tag{25}$$

Proof Because $(t_i \times t'_j)$ does not overflow, we know that each entry in the computation of the matrix multiplication does not overflow:

$$\forall_{1 \leqslant p, q \leqslant o} : min_{\mathbb{Q}} \leqslant \Sigma_{n=1}^{o} cast_{\mathbb{Q}}(t_i[p, n]) \cdot cast_{\mathbb{Q}}(t'_j[n, q]) \leqslant max_{\mathbb{Q}} \tag{26}$$

where t_i and t'_j are $o \times o$ matrices.

Suppose that $l'' = l \cdot l'$ and $\bigwedge_{m=1}^{l''} (0 \leqslant (\lambda''_m = \lambda_{\lceil m/l' \rceil} \cdot \lambda'_{(m-1)\%l'+1}) \leqslant 1)$. Then, $\sum_{m=1}^{l''} \lambda''_m = \sum_{i=1}^{l'} \lambda_i \cdot \sum_{j=1}^{l'} \lambda'_j = 1 \cdot 1 = 1$. Then, by applying Lem. B.2 to Eqn. (26), we get for all $1 \leqslant p, q \leqslant o$:

$$\begin{aligned} &min_{\mathbb{Q}} \leqslant \Sigma_{m=1}^{l''} \lambda''_m (\Sigma_{n=1}^{o} cast_{\mathbb{Q}}(t_{\lceil m/l \rceil}[p, n]) \cdot cast_{\mathbb{Q}}(t'_{m\%l+1}[n, q])) \leqslant max_{\mathbb{Q}} \\ \Leftrightarrow \ &min_{\mathbb{Q}} \leqslant \Sigma_{i=1}^{l} \Sigma_{j=1}^{l'} \lambda_i \lambda'_j (\Sigma_{n=1}^{o} cast_{\mathbb{Q}}(t_i[p, n]) \cdot cast_{\mathbb{Q}}(t'_j[n, q])) \leqslant max_{\mathbb{Q}} \\ \Leftrightarrow \ &min_{\mathbb{Q}} \leqslant (\Sigma_{n=1}^{o} (\Sigma_{i=1}^{l} \lambda_i cast_{\mathbb{Q}}(t_i[p, n])) \cdot (\Sigma_{j=1}^{l'} \lambda'_j cast_{\mathbb{Q}}(t'_j[n, q]))) \leqslant max_{\mathbb{Q}} \\ &\text{(by distributivity of multiplication over addition for rationals)} \\ \Leftrightarrow \ &min_{\mathbb{Q}} \leqslant (\Sigma_{n=1}^{o} cast_{\mathbb{Q}}(t[p, n]) \cdot cast_{\mathbb{Q}}(t'_j[n, q])) \leqslant max_{\mathbb{Q}} \\ &\text{(by the definition of } cast_{\mathbb{Q}}(t) \text{ and } cast_{\mathbb{Q}}(t') \text{ in Eqn. (24) and Eqn. (25))} \end{aligned}$$

Hence $(t \times t')$ does not overflow. □

Theorem B.2 *(Soundness of $\circ_{WC}$)*

$$C_e \subseteq \gamma(a' \circ_{WC} a). \tag{27}$$

Proof Consider any model $m = (\overrightarrow{v}, \overrightarrow{v}'')$, such that $m \in C_e$. To prove Thm. B.2, we need to show that $m \in \gamma(a' \circ_{\text{WC}} a)$. Eqn. (7) defines $a'' = a' \circ_{\text{WC}} a$ as follows

$$base(a'') = \begin{cases} \bigsqcup \left\{ \beta(t_i \times t'_j) |\ 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\} & \textit{if there are no overflows in any} \\ & \textit{matrix multiplication } t_i \times t'_j \\ \top_{\mathcal{B}} & \textit{otherwise} \end{cases}$$
$$\text{where } base(a) = \{t_1, t_2, ..., t_l\} \ \textit{and } base(a') = \left\{t'_1, t'_2, ..., t'_{l'}\right\}. \tag{28}$$

We know that for $m = (\overrightarrow{v}, \overrightarrow{v}')$,

$$\exists (C : \overrightarrow{d}) \in \gamma(base(a)), (C' : \overrightarrow{d}') \in \gamma(base(a')), (C'' : \overrightarrow{d}'') : \tag{29}$$
$$(\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge (C'' = C \cdot C') \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}').$$

By the properties of weakly-convex domains (see §4.2.4), we know that

$$cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right]\right) = \sum_{i=1}^{l} \lambda_i \ cast_{\mathbb{Q}}(t_i), \text{ for some } \lambda_1, \lambda_2, \ldots, \lambda_l \in \mathbb{Q} \text{ such that} \tag{30}$$
$$\bigwedge_{i=1}^{l} (0 \leqslant \lambda_i \leqslant 1) \wedge (\sum_{i=1}^{l} \lambda_i = 1) \wedge \bigwedge_{i=1}^{l} \left(t_i = \left[\begin{array}{c|c} 1 & \overrightarrow{d}_i \\ \hline 0 & C_i \end{array}\right]\right), \textit{ and}$$
$$cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d}' \\ \hline 0 & C' \end{array}\right]\right) = \sum_{i=1}^{l'} \lambda'_i \ cast_{\mathbb{Q}}(t'_i), \text{ for some } \lambda'_1, \lambda'_2, \ldots, \lambda'_l \in \mathbb{Q} \text{ such that} \tag{31}$$
$$\bigwedge_{i=1}^{l'} (0 \leqslant \lambda'_i \leqslant 1) \wedge (\sum_{i=1}^{l'} \lambda'_i = 1) \wedge \bigwedge_{i=1}^{l'} \left(t'_i = \left[\begin{array}{c|c} 1 & \overrightarrow{d}'_i \\ \hline 0 & C'_i \end{array}\right]\right).$$

To show that $m \in \gamma(a' \circ_{\text{WC}} a)$, we consider two cases.

Overflows in matrix multiplication. If there is an overflow encountered in any matrix multiplication $t_i \times t'_j$, then $base(a'') = \top_{\mathcal{B}}$ and consequently, $m \in \gamma(a' \circ_{\text{WC}} a)$ is true trivially.

No overflows in matrix multiplication. If there is no overflow encountered in any of the matrix multiplications $t_i \times t'_j$, then it suffices to prove that

$$(C'' : \overrightarrow{d}'') \in \bigsqcup_{i=1}^{l} \bigsqcup_{j=1}^{l'} \{\beta(t_i \times t'_j)\}. \tag{32}$$

Eqn. (32) translates to proving that for some $\{\lambda''_1, \lambda''_2, ..., \lambda''_{l''}\}$:

$$(cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d}'' \\ \hline 0 & C'' \end{array}\right]\right) = \sum_{i=1}^{l''} \lambda''_i \ cast_{\mathbb{Q}}(t''_i)), \text{ for some } \lambda''_1, \lambda''_2, \ldots, \lambda''_{l''} \in \mathbb{Q} \text{ such that} \tag{33}$$
$$\bigwedge_{i=1}^{l''} (0 \leqslant \lambda''_i \leqslant 1) \wedge (\sum_{i=1}^{l''} \lambda''_i = 1) \wedge \bigwedge_{i=1}^{l''} \left(t''_i = \left[\begin{array}{c|c} 1 & \overrightarrow{d}''_i \\ \hline 0 & C''_i \end{array}\right]\right) \wedge (t''_i = t_{\lceil i/l' \rceil} \times t'_{(i-1)\%l'+1}).$$

$$\begin{aligned}
& cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c}1 & \vec{d}''\\ \hline 0 & C''\end{array}\right]\right)\\
&= cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c}1 & \vec{d}\\ \hline 0 & C\end{array}\right] \times \left[\begin{array}{c|c}1 & \vec{d}'\\ \hline 0 & C'\end{array}\right]\right) \text{ (by Eqn. (29))}\\
&= cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c}1 & \vec{d}\\ \hline 0 & C\end{array}\right]\right) \times cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c}1 & \vec{d}'\\ \hline 0 & C'\end{array}\right]\right)\\
& \quad \text{(by Lem. B.3, because } \left[\begin{array}{c|c}1 & \vec{d}\\ \hline 0 & C\end{array}\right] \times \left[\begin{array}{c|c}1 & \vec{d}'\\ \hline 0 & C'\end{array}\right] \text{ does not overflow)}\\
&= \sum_{i=1}^{l} \lambda_i \; cast_{\mathbb{Q}}(t_i) \times \sum_{j=1}^{l'} \lambda'_j \; cast_{\mathbb{Q}}(t'_j)\\
& \quad \text{(by Eqn. (30) and Eqn. (31).)}\\
&= \sum_{i=1}^{l}\sum_{j=1}^{l'} \lambda_i \lambda'_j \; cast_{\mathbb{Q}}(t_i) \times \; cast_{\mathbb{Q}}(t'_j)\\
& \quad \text{(by distributivity of matrix multiplication over addition)}\\
&= \sum_{i=1}^{l}\sum_{j=1}^{l'} \lambda_i \lambda'_j \; cast_{\mathbb{Q}}(t_i \times t'_j)\\
& \quad \text{(by Lem. B.1)}\\
&= \sum_{m=1}^{l''} \lambda''_m \; cast_{\mathbb{Q}}(t''_m)
\end{aligned}$$

where $l'' = l \cdot l'$, $\bigwedge_{m=1}^{l''} (0 \leqslant (\lambda''_m = \lambda_{\lceil m/l' \rceil} \cdot \lambda'_{(m-1)\%l'+1}) \leqslant 1) \wedge (t''_m = t_{\lceil m/l' \rceil} \times t'_{(m-1)\%l'+1})$, and
$\sum_{m=1}^{l''} \lambda''_m = \sum_{i=1}^{l'} \lambda_i \cdot \sum_{j=1}^{l'} \lambda'_j = 1 \cdot 1 = 1.$ □

C Soundness of Merge Functions

In this section, we show that the merge operation defined in §4.4 is sound. Recall that the merge function is defined as:

$$\begin{aligned}
Merge(a, a') &= a'', \text{ where} \\
base(a'') &= (havoc(base(a'), lsyms) \sqcap havoc(base(Id), gsyms)) \circ a \\
& \quad lsyms = symbols(C_{gl}) \cup symbols(C_{ll}) \cup \\
& \quad\quad symbols(d_l) \cup symbols(C_{lg}) \\
& \quad gsyms = symbols(C_{gg}) \cup symbols(d_g)
\end{aligned} \tag{34}$$

As mentioned in Eqn. (9), the exact merge-function semantics are specified as follows:

$$\textsc{Merge}(\gamma(a), \gamma(a')) = \textsc{RevertLocals}(\gamma(a')) \circ \gamma(a) \tag{35}$$

Theorem C.1 *MergeIsSoundSoundness of the merge function*

$$\textsc{Merge}(\gamma(a), \gamma(a')) \subseteq \gamma(Merge(a, a')). \tag{36}$$

Proof We will prove Thm. C.1 by contradiction. Consider a model $m = (\overrightarrow{g_m}, \overrightarrow{l_m}; \overrightarrow{g_m}', \overrightarrow{l_m}')$, such that $m \in \textsc{Merge}(\gamma(a), \gamma(a'))$ and $m \notin \gamma(Merge(a, a'))$. Let $a_{RevLocs} \in \text{ATA}[\mathcal{B}]$ be an abstract domain value such that

$$base(a_{RevLocs}) \;=\; havoc(base(a'), lsyms) \sqcap havoc(base(Id), gsyms) \tag{37}$$

By the soundness of abstract composition, existence of m implies existence of $n = (\overrightarrow{g_n}, \overrightarrow{l_n}; \overrightarrow{g_n}', \overrightarrow{l_n}')$, such that $n \in \textsc{RevertLocals}(\gamma(a'))$ and $n \notin \gamma(a_{RevLocs})$. We will show that n cannot exist. Consequently, m cannot exist, and thus merge is sound.

$n \in \textsc{RevertLocals}(\gamma(a'))$

$\Leftrightarrow\ (\overrightarrow{g_n}, \overrightarrow{l_n}; \overrightarrow{g_n}', \overrightarrow{l_n}') \in \{(\overrightarrow{g}, \overrightarrow{q}, \overrightarrow{g}', \overrightarrow{q}) \mid (\overrightarrow{g}, \overrightarrow{l}, \overrightarrow{g}', \overrightarrow{l}') \in \gamma(a')\}$ (by Eqn. (10) and Eqn. (11))

$$\Leftrightarrow\ \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')), \overrightarrow{l}, \overrightarrow{l}', \overrightarrow{q} :$$

$$(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{l} \cdot C_{lg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l}' = \overrightarrow{g_n} \cdot C_{gl} + \overrightarrow{l} \cdot C_{ll} + \overrightarrow{d_l}),$$
$$\wedge\, (\overrightarrow{l_n} = \overrightarrow{q}) \wedge (\overrightarrow{l_n}' = \overrightarrow{q}) \wedge (\overrightarrow{l} = \overrightarrow{0})$$

($\overrightarrow{l}$ are initialized to 0 in each function)

$$\Leftrightarrow\ \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$$
$$(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l_n} = \overrightarrow{l_n}')$$

(by removing the existential variables $\overrightarrow{l}$, $\overrightarrow{l}$' and $\overrightarrow{q}$)

$$\Leftrightarrow\ \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$$
$$havoc(\left[\, 1 \middle| \overrightarrow{g_n}\ \overrightarrow{l_n} \,\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & 0 & C_{ll} \end{array} \right] = \left[\, 1 \middle| \overrightarrow{g_n}'\ \overrightarrow{l_n}' \,\right], lsyms) \wedge (\overrightarrow{l_n}' = \overrightarrow{l_n})$$

(because $(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{d_g})$ is the result of havoc on *lsyms* for $(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l_n}' = \overrightarrow{g_n} \cdot C_{gl} + \overrightarrow{l_n} \cdot C_{ll} + \overrightarrow{d_l})$)

$$\Leftrightarrow\ \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$$
$$havoc(\left[\, 1 \middle| \overrightarrow{g_n}\ \overrightarrow{l_n} \,\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array} \right] = \left[\, 1 \middle| \overrightarrow{g_n}'\ \overrightarrow{l_n}' \,\right], lsyms) \wedge (\overrightarrow{l_n}' = \overrightarrow{l_n})$$

(because $\overrightarrow{l_n}$ are initialized to zero)

$$\Leftrightarrow\ \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$$
$$havoc(\left[\, 1 \middle| \overrightarrow{g_n}\ \overrightarrow{l_n} \,\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array} \right] = \left[\, 1 \middle| \overrightarrow{g_n}'\ \overrightarrow{l_n}' \,\right], lsyms) \wedge$$
$$havoc(\left[\, 1 \middle| \overrightarrow{g_n}\ \overrightarrow{l_n} \,\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array} \right] = \left[\, 1 \middle| \overrightarrow{g_n}'\ \overrightarrow{l_n}' \,\right], gsyms)$$

(because havoc of *gsyms* on the identity transformation yields $\overrightarrow{l_n}' = \overrightarrow{l_n}$)

$$\Rightarrow\ \Bigg(\exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(havoc(base(a'), lsyms)) :$$
$$\left[\, 1 \middle| \overrightarrow{g_n}\ \overrightarrow{l_n} \,\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array} \right] = \left[\, 1 \middle| \overrightarrow{g_n}'\ \overrightarrow{l_n}' \,\right] \Bigg) \bigwedge$$

$$\left(\exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(havoc(base(Id), gsyms)) : \right.$$

$$\left. \begin{bmatrix} 1 \,|\, \overrightarrow{g_n} \; \overrightarrow{l_n} \end{bmatrix} \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array} \right] = \begin{bmatrix} 1 \,|\, \overrightarrow{g_n}' \; \overrightarrow{l_n}' \end{bmatrix} \right)$$

$\Leftrightarrow$ $(\overrightarrow{g_n}, \overrightarrow{l_n}, \overrightarrow{g_n}', \overrightarrow{l_n}') \in \gamma(a_{RevLocs})$

$\Leftrightarrow$ Model $(\overrightarrow{g_n}, \overrightarrow{l_n}, \overrightarrow{g_n}', \overrightarrow{l_n}')$ does not exist.

$\Rightarrow$ Model $(\overrightarrow{g_m}, \overrightarrow{l_m}, \overrightarrow{g_m}', \overrightarrow{l_m}')$ does not exist. (Contradiction.)
(by soundness of abstract composition)

If the abstract meet is exact, then the implication in the fourth-from-last step of the proof becomes an if and only if ($\Leftrightarrow$). Furthermore, if the abstract-composition operation is exact, then the implication in the last step of the proof becomes an if and only if ($\Leftrightarrow$). Thus, if abstract meet and abstract composition are exact, the merge operation is exact. $\square$