Abstract Domains of Affine Relations*

M. Elder[1], J. Lim[1], T. Sharma[1], T. Andersen[1], and T. Reps[1,2]**

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

Abstract. This paper considers some known abstract domains for affine-relation analysis (ARA), along with several variants, and studies how they relate to each other. We show that the abstract domains of Müller-Olm/Seidl (MOS) and King/Søndergaard (KS) are, in general, incomparable, but give sound interconversion methods. We also show that the methods of King and Søndergaard can be applied without bit-blasting—while still using a bit-precise concrete semantics.

1 Introduction

The work reported in this paper was motivated by our work on TSL [16], which is a system for generating abstract interpreters for machine code. With TSL, one specifies an instruction set's concrete operational semantics by defining an interpreter

$$\texttt{interpInstr} : \texttt{instruction} \times \texttt{state} \rightarrow \texttt{state}.$$

For a given abstract domain $\mathcal{A}$, a sound abstract transformer for each instruction of the instruction set is obtained by defining a sound reinterpretation of each operation of the TSL meta-language as an operation over $\mathcal{A}$. By extending the reinterpretation to TSL expressions and functions—including `interpInstr`—the set of operator-level reinterpretations defines the desired set of abstract transformers for the instructions of the instruction set.

However, this method abstracts each TSL operation in isolation, and is therefore rather myopic. Moreover, the operators that TSL provides to specify an instruction set's concrete semantics include arithmetic, logical, and "bit-twiddling" operations. The latter include left-shift; arithmetic and logical right-shift; bitwise-and, bitwise-or, and bitwise-xor; etc. Few abstract domains retain precision over the full gamut of such operations.

A more global approach that considers the semantics of an entire instruction (or, even better, an entire basic block or other path fragment) can yield a more precise transformer. One way to specify the goals of such a global approach is through the notion of *symbolic abstraction* [22]:

* Supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. 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.

– An abstract domain $\mathcal{A}$ is said to support a *symbolic implementation of the α function* of a Galois connection if, for every logical formula ψ that specifies (symbolically) a set of concrete stores $[\![\psi]\!]$, there is a method $\widetilde{\alpha}$ that finds a sound abstract value $\widetilde{\alpha}(\psi) \in \mathcal{A}$ that over-approximates $[\![\psi]\!]$. That is, $[\![\psi]\!] \subseteq \gamma(\widetilde{\alpha}(\psi))$, where $[\![\psi]\!]$ denotes the meaning function for the logic.
– For some abstract domains, it is even known how to perform a *best* symbolic implementation of α, denoted by $\widehat{\alpha}$ [22]. For every ψ, $\widehat{\alpha}$ finds the best value in $\mathcal{A}$ that over-approximates $[\![\psi]\!]$.

In particular, the issue of "myopia" is addressed by first creating a logical formula φ_I that captures the concrete semantics of each instruction I (or basic block, or path fragment) in quantifier-free bit-vector logic (QFBV), and then performing $\widetilde{\alpha}(\varphi_I)$ or $\widehat{\alpha}(\varphi_I)$. (The generation of a QFBV formula that, with no loss of precision, captures the concrete semantics of an instruction or basic block is a problem that itself fits the TSL operator-reinterpretation paradigm [16, §3.4].)

We explored how to address these issues using two existing abstract domains for affine-relation analysis (ARA)—one defined by Müller-Olm and Seidl (MOS) [19, 21] and one defined by King and Søndergaard (KS) [11, 12]—as well as a third domain of *affine generators* that we introduce. (Henceforth, the three domains are referred to as MOS, KS, and AG, respectively.) All three domains represent sets of points that satisfy affine relations over variables that hold machine integers, and are based on an extension of linear algebra to modules over a ring [8, 7, 1, 25, 19, 21]. The contributions of our work can be summarized as follows:

– For MOS, it was not previously known how to perform $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ in a non-trivial fashion (e.g., other than defining $\widetilde{\alpha}_{\mathrm{MOS}}$ to be $\lambda f.\top$). In contrast, King and Søndergaard gave an algorithm for $\widehat{\alpha}_{\mathrm{KS}}$ [12, Fig. 2], which led us to examine more closely how MOS and KS are related. A KS value consists of a set of *constraints* on the values of variables. We introduce a third abstract domain, AG, which can be considered to be the *generator* counterpart of KS. A KS constraint-based value can be converted to an AG value with no loss of precision, and vice versa.

  In contrast, we show that MOS and KS/AG are, in general, *incomparable*. However, we give sound interconversion methods: we show that an AG value v_{AG} can be converted to an over-approximating MOS value v_{MOS}—i.e., $\gamma(v_{\mathrm{AG}}) \subseteq \gamma(v_{\mathrm{MOS}})$—and that an MOS value w_{MOS} can be converted to an over-approximating AG value w_{AG}—i.e., $\gamma(w_{\mathrm{MOS}}) \subseteq \gamma(w_{\mathrm{AG}})$.

  Consequently, by means of the conversion path $\varphi \to \mathrm{KS} \to \mathrm{AG} \to \mathrm{MOS}$, we show how to perform $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ (§4.5).
– To apply the techniques described in the two King and Søndergaard papers [11, 12], it is necessary to perform bit-blasting. Their goal is to create implementations of operations that are precise, modulo the inherent limitations of precision that stem from using KS. They use bit-blasting to express a bit-precise concrete semantics for a statement or basic block. Working at the bit level lets them track the effect of non-linear bit-twiddling operations, such as shift and xor.

One drawback of bit-blasting is the huge number of variables that it introduces (e.g., 32 or 64 bit-valued variables for each `int`-valued program variable). Given that one is performing numerous cubic-time operations on the matrices that arise, there is a question as to whether the bit-blasted version of KS could ever be applied to problems of substantial size. The times reported by King and Søndergaard are quite high [12, §7], although they state that there is room for improvement by, e.g., using sparse matrices.

In our work, we use an SMT solver rather than a SAT solver, and show that implementations of operations that are best operations for the KS domain can be obtained without resorting to bit-blasting. Instead, we work with QFBV formulas that capture symbolically the precise bit-level semantics of each instruction or basic block, and take advantage of the ability of $\widehat{\alpha}_{\mathrm{KS}}$ to create best word-level transformers.[3]

The greatly reduced number of variables that comes from working at word level opens up the possibility of applying our methods to much larger problems, and in particular to performing interprocedural analysis. We show how to use the KS domain as the basis for interprocedural ARA. In particular, we use a two-vocabulary version of KS to create a weight domain for a weighted pushdown system (WPDS) [23, 2, 10] (§5).

In addition to the specific contributions listed above, this paper provides insight on the range of options one has for performing affine-relation analysis, and how the different approaches relate to each other.

Organization. The remainder of the paper is organized as follows: §2 summarizes relevant features of the various ARA domains considered in the paper. §3 presents the AG domain, and shows how an AG value can be converted to a KS value, and vice versa. §4 presents our results on the incomparability of the MOS and KS domains, but gives sound methods to convert a KS value into an over-approximating MOS value, and vice versa. §5 explains how to use the KS domain for interprocedural analysis without bit-blasting. §6 presents experimental results. §7 discusses related work. Proofs can be found in a companion technical report [4].

2 Terminology and Notation

All numeric values in this paper are integers in $\mathbb{Z}_{2^w}$ for some bit width w. That is, values are machine integers with the standard 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 (and thus we must regard our intuitions about linear algebra with caution). In particular, all odd elements in $\mathbb{Z}_{2^w}$ have a multiplicative inverse (which may be found in time $O(\log w)$ [26, Fig. 10-5]), but no even elements have a multiplicative inverse. The *rank* of a value

[3] The two methods are not entirely comparable because the bit-blasting approach works with a great deal more variables (to represent the values of individual bits). However, for word-level properties the two are comparable. For instance, both can discover that the action of an xor-based swap is to exchange the values of two program variables.

$x \in \mathbb{Z}_{2^w}$ is the maximum integer $p \leq w$ such that $2^p | x$. For example, $\text{rank}(1) = 0$, $\text{rank}(12) = 2$, and $\text{rank}(0) = w$.

Throughout the paper, k is the size of the *vocabulary*, the variable set under analysis. A *two-vocabulary* relation is a relation between values of variables in its *pre-state* vocabulary to values of variables in its *post-state* vocabulary.

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 for technical reasons (which vary according to what kind of matrix we are dealing with). I denotes the (square) identity matrix (whose size can be inferred from context).

Actual states in the various abstract domains are represented by k-length row vectors. The *row space* of a matrix M is $\text{row}\, M \stackrel{\text{def}}{=} \{x \mid \exists w \colon wM = x\}$. When we speak of the "null space" of a matrix, we actually mean the set of row vectors whose transposes are in the traditional null space of the matrix. Thus, we define $\text{null}^t\, M \stackrel{\text{def}}{=} \{x \mid Mx^t = 0\}$.

Matrices in Howell Form. An appreciation of how linear algebra in rings differs from linear algebra in fields can be obtained by seeing how certain issues are finessed when converting a matrix to *Howell form* [8]. The Howell form of a matrix is an extension of reduced row-echelon form [17] suitable for matrices over $\mathbb{Z}_n$. Because Howell form is a canonical form for matrices over principal ideal rings [8, 25], it provides a way to test pairs of abstract-domain elements for equality of their concretizations—an operation needed by analysis algorithms to determine when a fixed point is reached.

Definition 1. *The leftmost nonzero value in a row vector is its* **leading value**, *and the leading value's index is the* **leading index**. *A matrix M is in* **row-echelon form** *iff*

- *All rows consisting entirely of zeroes are at the bottom.*
- *The sequence of leading indices of rows is strictly increasing.*

If M is in row-echelon form, let $[M]_i$ denote the matrix that consists of all rows of M whose leading index is i or greater.

A matrix M is in **Howell form** *iff*

1. *M is in row-echelon form,*
2. *the leading value of every row is a power of two,*
3. *each leading value is the largest value in its column, and*
4. *for every row r of M, for any $p \in \mathbb{Z}$, if i is the leading index of $2^p r$, then $2^p r \in \text{row}[M]_i$.*

Suppose that $w = 4$. Item 4 of Defn. 1 is illustrated by $M = \left[\begin{smallmatrix} 4 & 2 & 4 \\ 0 & 4 & 0 \end{smallmatrix}\right]$. The first row of M has leading index 1. Multiplying the first row by 4 produces [0 8 0], which has leading index 2. This meets condition 4 because $[0\ 8\ 0] = 2 \cdot [0\ 4\ 0]$, so $[0\ 8\ 0] \in \text{row}[M]_2$.

The Howell form of a matrix is unique among all matrices with the same row space (or null space) [8]. As mentioned above, Howell form provides a way to test pairs of KS or AG elements for equality of their concretizations.

Algorithm 1 HOWELLIZE: Put the matrix G in Howell form.

```
 1: procedure HOWELLIZE(G)
 2:     Let j = 0                          ▷ j is the number of already-Howellized rows
 3:     for all i from 1 to 2k + 1 do
 4:         Let R = {all rows of G with leading index i}
 5:         if R ≠ ∅ then
 6:             Pick an r ∈ R that minimizes rank r_i
 7:             Pick the odd u and rank p so that u2^p = r_i
 8:             r ← u^{-1} r                          ▷ Adjust r, leaving r_i = 2^p
 9:             for all s in R \ {r} do
10:                 Pick the odd v and rank t so that v2^t = s_i
11:                 s ← s − (v2^{t−p}) r                          ▷ Zero out s_i
12:                 if row s contains only zeros then
13:                     Remove s from G
14:             In G, swap r with G_{j+1}                ▷ Place r for row-echelon form
15:             for all h from 1 to j do          ▷ Set values above r_i to be 0 ≤ · < r_i
16:                 d ← G_{h,i} ≫ p                ▷ Pick d so that 0 ≤ G_{h,i} − dr_i < r_i
17:                 G_h ← G_h − dr                 ▷ Adjust row G_h, leaving 0 ≤ G_{h,i} < r_i
18:             if r_i ≠ 1 then                    ▷ Add logical consequences of r to G
19:                 Add 2^{w−p} r as last row of G          ▷ New row has leading index > i
20:             j ← j + 1
```

The notion of a *saturated set of generators* used by Müller-Olm and Seidl [21] is closely related to Howell form, but is defined for an unordered set of generators rather than row-vectors arranged in a matrix, and has no analogue of item 3. The algorithms of Müller-Olm and Seidl do not compute multiplicative inverses (see §7), so a saturated set has no analogue of item 2. Consequently, a saturated set is not canonical among generators of the same space.

Our technique for putting a matrix in Howell form is given as procedure HOWELLIZE (Alg. 1). Much of HOWELLIZE is similar to a standard Gaussian-elimination algorithm, and it has the same overall cubic-time complexity as Gaussian elimination. In particular, HOWELLIZE minus lines 15–19 puts G in row-echelon form (item 1 of Defn. 1) with the leading value of every row a power of two. (Line 8 enforces item 2 of Defn. 1.) HOWELLIZE differs from standard Gaussian elimination in how the pivot is picked (line 6) and in how the pivot is used to zero out other elements in its column (lines 7–13). Lines 15–17 of HOWELLIZE enforce item 3 of Defn. 1, and lines 18–19 enforce item 4. Lines 12–13 remove all-zero rows, which is needed for Howell form to be canonical.

The Affine Generator Domain. An element in the Affine Generator domain (AG) is a two-vocabulary matrix whose rows are the affine generators of a two-vocabulary relation.

An AG element is an r-by-$(2k+1)$ matrix G, with $0 < r \leq 2k+1$. The concretization of an AG element is

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

The AG 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 AG element that represents the identity relation is the matrix $\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & I \end{array}\right]$. To compute the join of two AG matrices, stack the two matrices vertically and Howellize the result.

The King/Søndergaard Domain. An element in the King/Søndergaard domain (KS) is a two-vocabulary matrix whose rows represent constraints on a two-vocabulary relation. A KS element is an r-by-$(2k+1)$ matrix X, with $0 \leq r \leq 2k+1$. The concretization of a KS element is

$$\gamma_{\text{KS}}(X) \stackrel{\text{def}}{=} \left\{(x, x') \mid x, x' \in \mathbb{Z}_{2^w}^k \wedge [x\ x' | 1] \in \text{null}^t\, G\right\}.$$

Like the AG domain, the KS domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states. The original KS paper [11] gives polynomial-time algorithms for join and projection; projection can be used to implement composition.

It is easy to read off affine relations from a KS element: if $[a_1\ \cdots\ a_k\ a'_1\ \cdots\ a'_k\ |{-}b]$ is a row of X, then $\sum_i a_i x_i + \sum_i a'_i x'_i = b$ is a constraint on $\gamma_{\text{KS}}(X)$. The conjunction of these constraints describes $\gamma_{\text{KS}}(X)$ precisely.

The bottom element of the KS domain is the matrix $[0\ 0|1]$, and the KS element that represents the identity relation is the matrix $[I\ {-I}|0]$.

A Howell-form KS element can easily be checked for emptiness: it is empty if and only if it contains a row whose leading entry is in its last column. In that sense, an implementation of the KS domain in which all elements are kept in Howell form has redundant representations of bottom (whose concretization is $\emptyset$). However, such KS elements can always be detected during HOWELLIZE and replaced by the canonical representation of bottom, $[0\ 0|1]$.

The Müller-Olm/Seidl Domain. An element in the Müller-Olm/Seidl domain (MOS) is an affine set of affine transformers, as detailed in [21]. An MOS element represents 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 $x' := x \cdot M + b$, or, equivalently, $[1|x'] := [1|x]\, T$.

An MOS element $\mathcal{B}$ consists of a set of $(k+1)$-by-$(k+1)$ matrices, and represents the affine span of the set, denoted by $\langle\mathcal{B}\rangle$ and defined as follows:

$$\langle\mathcal{B}\rangle \stackrel{\text{def}}{=} \left\{ T \;\middle|\; \exists w \in \mathbb{Z}_{2^w}^{|\mathcal{B}|} : T = \sum_{B \in \mathcal{B}} w_B B \wedge T_{1,1} = 1 \right\}.$$

The meaning of $\mathcal{B}$ is the union of the graphs of the affine transformers in $\langle B\rangle$

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

The bottom element of the MOS domain is $\emptyset$, and the MOS element that represents the identity relation is the singleton set $\{I\}$.

The operations join and compose can be performed in polynomial time. If $\mathcal{B}$ and $\mathcal{C}$ are MOS elements, then $\mathcal{B} \sqcup \mathcal{C} = \textsc{Howellize}\,(\mathcal{B} \cup \mathcal{C})$ and $\mathcal{B} \circ \mathcal{C} = \textsc{Howellize}\,\{BC \mid B \in \mathcal{B} \wedge C \in \mathcal{C}\}$. In this setting, Howellize of a set of $(k+1)$-by-$(k+1)$ matrices $\{M_1, \ldots, M_n\}$ means "Apply Alg. 1 to a larger, n-by-$(k+1)^2$ matrix, each of whose rows is the linearization (e.g., in row-major order) of one of the M_i."

3 Relating AG and KS Elements

AG and KS are equivalent domains. One can convert an AG element to an equivalent KS element with no loss of precision, and vice versa. In essence, these are a single abstract domain with two representations: constraint form (KS) and generator form (AG).

We use an operation similar to singular value decomposition, called diagonal decomposition:

Definition 2. *The* **diagonal decomposition** *of a square matrix M is a triple of matrices, L, D, R, such that $M = LDR$; L and R are invertible matrices; and D is a diagonal matrix in which all entries are either 0 or a power of 2.*

Müller-Olm and Seidl give a decomposition algorithm that nearly performs diagonal decomposition [21, Lemma 2.9], except that the entries in their D might not be powers of 2. We can easily adapt that algorithm. Suppose that their method yields LDR (where L and R are invertible). Pick u and r so that $u_i 2^{r_i} = D_{i,i}$ with each u_i odd, and define U as the diagonal matrix where $U_{i,i} = u_i$. (If $D_{i,i} = 0$, then $u_i = 1$.) It is easy to show that U is invertible. Let $L' = LU$ and $D' = U^{-1}D$. Consequently, $L'D'R = LDR = M$, and $L'D'R$ is a diagonal decomposition.

From diagonal decomposition we derive the dual operation, denoted by $\cdot^{\perp}$, such that the rows of $M^{\perp}$ generate the null space of M, and vice versa.

Definition 3. *The* **dualization** *of M is $M^{\perp}$, where:*
- $\textsc{Pad}(M)$ *is the $(2k+1)$-by-$(2k+1)$ matrix* $\left[\begin{smallmatrix} M \\ \mathbf{0} \end{smallmatrix}\right]$,
- *L, D, R is the diagonal decomposition of* $\textsc{Pad}(M)$,
- *T is the diagonal matrix with $T_{i,i} \stackrel{def}{=} 2^{w - \mathrm{rank}(D_{i,i})}$, and*
- $M^{\perp} \stackrel{def}{=} \left(L^{-1}\right)^t T \left(R^{-1}\right)^t$

This definition of dualization has the following useful property:

Theorem 1. *For any matrix M,* $\mathrm{null}^t M = \mathrm{row}\, M^{\perp}$ *and* $\mathrm{row}\, M = \mathrm{null}^t M^{\perp}$.

We can therefore use dualization to convert between equivalent KS and AG elements. For a given (padded, square) AG matrix $G = [c|Y\ Y']$, we seek a KS matrix Z of the form $[X\ X'|b]$ such that $\gamma_{\mathrm{KS}}(Z) = \gamma_{\mathrm{AG}}(G)$. We construct Z by letting $[b|X\ X'] = G^{\perp}$ and permuting those columns to $Z \stackrel{\mathrm{def}}{=} [X\ X'|b]$. This works by Thm. 1, and because

$$\begin{aligned}
\gamma_{\mathrm{AG}}(G) &= \{(x, x') \mid [1|x\ x'] \in \mathrm{row}\, G\} \\
&= \left\{(x, x') \;\middle|\; [1|x\ x'] \in \mathrm{null}^t G^{\perp}\right\} \\
&= \left\{(x, x') \;\middle|\; [x\ x'|1] \in \mathrm{null}^t Z\right\} = \gamma_{\mathrm{KS}}(Z)\,.
\end{aligned}$$

Furthermore, to convert from any KS matrix to an equivalent AG matrix, we reverse the process. Reversal is possible because dualization is an involution: for any matrix M, $\left(M^{\perp}\right)^{\perp} = M$.

4 Relating KS and MOS

4.1 MOS and KS are Incomparable

The MOS and KS 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 KS is a domain of *relations*.

KS can capture restrictions on both the pre-state and post-state vocabularies while MOS captures restrictions only on its post-state vocabulary. For example, when $k = 1$, the KS element for "assume $x = 2$" is $\left\{\left[\begin{array}{cc|c} 1 & 0 & -2 \\ 1 & -1 & 0 \end{array}\right]\right\}$, i.e., "$x = 2 \wedge x' = x$". An MOS element cannot encode an assume statement. For "assume $x = 2$", the best MOS element is the identity transformer $\left\{\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right]\right\}$. In general, an MOS element cannot encode a non-trivial condition on the pre-state. If an MOS element contains a single transition, it encodes that transition for every possible pre-state. Therefore, KS can encode relations that MOS cannot encode.

On the other hand, an MOS element can encode two-vocabulary relations that are not affine. One example is the matrix basis $\mathcal{B} = \left\{\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right]\right\}$. The set that $\mathcal{B}$ encodes is

$$\begin{aligned}
\gamma_{\text{MOS}}(\mathcal{B}) &= \left\{ \left[x\ y\ x'\ y'\right] \;\middle|\; \begin{array}{c} \exists w_0, w_1 \colon \left[1 \middle| x\ y\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & w_0 & w_0 \\ 0 & w_1 & w_1 \end{array}\right] = \left[1 \middle| x'\ y'\right] \\ \wedge\ w_0 + w_1 = 1 \end{array} \right\} \\
&= \left\{ \left[x\ y\ x'\ y'\right] \;\middle|\; \exists w_0 \colon x' = y' = w_0 x + (1 - w_0) y \right\} \\
&= \left\{ \left[x\ y\ x'\ y'\right] \;\middle|\; \exists w_0 \colon x' = y' = x + (1 - w_0)(y - x) \right\} \\
&= \left\{ \left[x\ y\ x'\ y'\right] \;\middle|\; \exists p \colon x' = y' = x + p(y - x) \right\} \qquad (1)
\end{aligned}$$

Affine spaces are closed under affine combinations of their elements. Thus, $\gamma_{\text{MOS}}(\mathcal{B})$ is not an affine space because some affine combinations of its elements are not in $\gamma_{\text{MOS}}(\mathcal{B})$. 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 $a \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = 0$ in Eqn. (1), $b \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = -1$, and $c \notin \gamma_{\text{MOS}}(\mathcal{B})$ (the equation "$-4 = 0 + p(0 - 0)$" has no solution for p). Moreover, $2a - b = c$, so c is an affine combination of a and b. Thus, $\gamma_{\text{MOS}}(\mathcal{B})$ is not closed under affine combinations of its elements, and so $\gamma_{\text{MOS}}(\mathcal{B})$ is not an affine space. Because every KS element encodes a two-vocabulary affine space, MOS can represent $\gamma_{\text{MOS}}(\mathcal{B})$ but KS cannot.

4.2 Converting MOS Elements to KS

Soundly converting an MOS element to a KS element is equivalent to stating two-vocabulary affine constraints satisfied by that MOS element. To convert an MOS element $\mathcal{B}$ to a KS element, we

1. build a two-vocabulary AG matrix from each one-vocabulary matrix in $\mathcal{B}$,
2. compute the join of all the AG matrices from Step 1, and
3. convert the resulting AG matrix to a KS element.

For Step 1, assume that $\mathcal{B} = \left\{ \left[\begin{array}{c|c} 1 & c_i \\ \hline 0 & N_i \end{array}\right] \,\middle|\, 0 < i \right\}$, $c_i \in \mathbb{Z}_{2^w}^{1\times k}$, and $N_i \in \mathbb{Z}_{2^w}^{k\times k}$. If the original MOS element $\mathcal{B}_0$ fails to satisfy this property, let $\mathcal{C} = \textsc{Basis}(\mathcal{B}_0)$, pick the unique $B \in \mathcal{C}$ such that $B_{1,1} = 1$, and let $\mathcal{B} = \{B\} \cup \{B + C \,|\, C \in \mathcal{C} \setminus \{B\}\}$. $\mathcal{B}$ now satisfies the property, and $\langle \mathcal{B} \rangle = \langle \mathcal{B}_0 \rangle$.

From $\mathcal{B}$, we construct the matrices $G_i = \left[\begin{array}{c|cc} 1 & 0 & c_i \\ \hline 0 & I & N_i \end{array}\right]$. Note that, for each matrix $B_i \in \mathcal{B}$ with corresponding matrix G_i, $\gamma_{\text{MOS}}(\{B_i\}) = \gamma_{\text{AG}}(G_i)$. In Step 2, we join the G_i matrices in the AG domain to yield one matrix G. Thm. 2 proves the soundness of this transformation from MOS to AG, i.e., $\gamma_{\text{AG}}(G) \supseteq \gamma_{\text{MOS}}(\mathcal{B})$. Finally, G is converted in Step 3 to an equivalent KS element by the method given in §3.

Theorem 2. *Suppose that $\mathcal{B}$ is an MOS element such that, for every $B \in \mathcal{B}$, $B = \left[\begin{array}{c|c} 1 & c_B \\ \hline 0 & M_B \end{array}\right]$ for some $c_B \in \mathbb{Z}_{2^w}^{1\times k}$ and $M_B \in \mathbb{Z}_{2^w}^{k\times k}$. Define $G_B = \left[\begin{array}{c|cc} 1 & 0 & c_B \\ \hline 0 & I & M_B \end{array}\right]$ and $G = \bigsqcup_{AG} \{G_B \,|\, B \in \mathcal{B}\}$. Then, $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$.*

4.3 Converting KS Without Pre-State Guards to MOS

If a KS element is total with respect to pre-state inputs, then we can convert it to an equivalent MOS element. First, convert the KS element to an AG element G. When G expresses no restrictions on its pre-state, it has the form $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, where $b \in \mathbb{Z}_{2^w}^{1\times k}$; $I, M \in \mathbb{Z}_{2^w}^{k\times k}$; and $R \in \mathbb{Z}_{2^w}^{k\times r}$ with $0 \le r \le k$.

Definition 4. *An AG matrix of the form $\left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \end{array}\right]$, such as the G_i matrices discussed in §4.2, is said to be in* **explicit form** *because it represents the state transformation $x' := x \cdot M + b$.*

Explicit form is desirable because we can read off the MOS element $\left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\}$ from the AG matrix of Defn. 4.

G is not in explicit form because of the rows $[0|0\ R]$; however, G is quite close to being in explicit form, and we can read off a *set* of matrices to create an appropriate MOS element. We produce this set of matrices via the Shatter operation, where

$$\textsc{Shatter}(G) \stackrel{\text{def}}{=} \left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\} \cup \left\{ \left[\begin{array}{c|c} 0 & R_{j,*} \\ \hline 0 & 0 \end{array}\right] \,\middle|\, 0 < j \le r \right\}, \text{ where } R_{j,*} \text{ is row } j \text{ of } R.$$

As shown in Thm. 3, $\gamma_{\text{AG}}(G) = \gamma_{\text{MOS}}(\textsc{Shatter}(G))$. Intuitively, this holds because coefficients in an affine combination of the rows of G correspond cleanly to coefficients in an affine combination of the $R_{j,*}$ matrices in $\textsc{Shatter}(G)$.

Theorem 3. *When $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, then $\gamma_{AG}(G) = \gamma_{MOS}(\textsc{Shatter}(G))$.*

Algorithm 2 MakeExplicit: Transform an AG matrix G in Howell form to near-explicit form.

Require: G is an AG matrix in Howell form

```
 1: procedure MakeExplicit(G)
 2:     for all i from 2 to k + 1 do            ▷ Consider each col. of the pre-state voc.
 3:         if there is a row r of G with leading index i then
 4:             if rank r_i > 0 then
 5:                 for all j from 1 to 2k + 1 do          ▷ Build s from r, with s_i = 1
 6:                     s_j ← r_j ≫ rank r_i
 7:                 Append s to G
 8:                 G ← Howellize(G)
 9:     for all i from 2 to k + 1 do
10:         if there is no row r of G with leading index i then
11:             Insert, as the i^th row of G, a new row of all zeroes
```

4.4 Converting KS With Pre-State Guards to MOS

If a KS element is not total with respect to pre-state inputs, then there is no MOS element with the same concretization. However, we can find sound over-approximations within MOS for such KS elements.

We convert the KS element into an AG matrix G as in §4.3 and put G in Howell form. There are two ways that G can enforce guards on the pre-state vocabulary: it might contain one or more rows whose leading value is even, or it might skip some leading indexes in row-echelon form.

While we cannot put G in explicit form, we can run MakeExplicit to coarsen G so that it is close enough to the form that arose in §4.3. Adding extra rows to an AG element can only enlarge its concretization. Thus, to handle a leading value $2^p, p > 0$ in the pre-state vocabulary, MakeExplicit introduces an extra, over-approximating row constructed by copying the row with leading value 2^p and right-shifting each value in the copied row by p bits (lines 4–8). After the loop on lines 2–8 finishes, every leading value in a row that generates pre-state-vocabulary values is 1. MakeExplicit then introduces all-zero rows so that each leading element from the pre-state vocabulary lies on the diagonal (lines 9–11).

Example 1. Suppose that $k = 3$, $w = 4$, and $G = \left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 4 & 0 & 12 & 2 & 4 & 0 \\ & & & & 4 & 0 & 8 \end{array}\right]$. After line 11 of MakeExplicit, all pre-state vocabulary leading values of G have been made ones, and the resulting G' has $\operatorname{row} G' \supseteq \operatorname{row} G$. In our case, $G' = \left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$. To handle "skipped" indexes, lines 9–11 insert all-zero rows into G' so that each leading element from the pre-state vocabulary lies on the diagonal. The resulting matrix is $\left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & 0 & 0 & 0 & 0 & 0 \\ & & & 0 & 0 & 0 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$.

Theorem 4. *For any G, $\gamma_{AG}(G) \subseteq \gamma_{MOS}(\textsc{Shatter}(\textsc{MakeExplicit}(G)))$.*

Thus, we can use Shatter, MakeExplicit, and the AG–to–KS conversion of §3 to obtain an over-approximation of a KS element in MOS.

Example 2. The final MOS value for Ex. 1 is $\left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 0 & 8 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\}$.

4.5 Symbolic Implementation of the α Function for MOS

As mentioned in the Introduction, it was not previously known how to perform symbolic abstraction for MOS. Using $\widehat{\alpha}_{\mathrm{KS}}$ [12, Fig. 2] in conjunction with the algorithms from §3 and §4.4, we can soundly define $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi) \stackrel{\text{def}}{=}$ **let** $G =$ $\textsc{ConvertKStoAG}(\widehat{\alpha}_{\mathrm{KS}}(\varphi))$ **in** $\textsc{Shatter}(\textsc{MakeExplicit}(G))$.

5 Using KS for Interprocedural Analysis

This section describes how to use the KS domain in interprocedural-analysis algorithms in the style of Sharir and Pnueli [24], Knoop and Steffen [13], Müller-Olm and Seidl [18], and Lal et al. [15].

Project. In [11, §3], King and Søndergaard describe a way to project a KS element X onto a suffix $x_i, \ldots, x_k$ of its vocabulary: (i) put X in row-echelon form, and (ii) remove every row a in which any of $a_1, \ldots, a_{i-1}$ is nonzero. However, when the leading values of X are not all 1, step (ii) is not guaranteed to produce the most-precise projection of X onto $x_i, \ldots, x_k$ (although the value obtained is always sound). Instead, we put X in Howell form, and by Thm. 5, step (ii) returns the most-precise projection.

Theorem 5. *Suppose that M has c columns. If matrix M is in Howell form, $x \in \mathrm{null}^t M$ if and only if $\forall i \colon \forall y_1, \ldots y_{i-1} \colon \left[y_1 \ \cdots y_{i-1} \ x_i \ \cdots \ x_c\right] \in \mathrm{null}^t [M]_i$.*

Example 3. Suppose that $X = [4\ 2|6]$, with $w = 4$, and the goal is to project away the first column (for x_1). King and Søndergaard obtain the empty matrix, which represents no constraints on x_2. The Howell form of X is $\left[\begin{array}{cc|c} 4 & 2 & 6 \\ 0 & 8 & 8 \end{array}\right]$, and thus the most precise projection of X onto x_2 is $[8|8]$, which represents $x_2 \in \{1, 3, \ldots, 15\}$.

Compose. In [12, §5.2], King and Søndergaard present a technique to compose two-vocabulary affine relations. For completeness, that algorithm follows. Suppose that we have KS elements $Y = \left[\, Y_{\mathrm{pre}}\ Y_{\mathrm{post}} \middle| y \,\right]$ and $Z = \left[\, Z_{\mathrm{pre}}\ Z_{\mathrm{post}} \middle| z \,\right]$, where Y_* and Z_* are k-column matrices, and y and z are column vectors. We want to compute $Y \circ Z$; i.e., some X such that $(x, x'') \in \gamma_{\mathrm{KS}}(X)$ if and only if $\exists x' \colon (x, x') \in \gamma_{\mathrm{KS}}(Y) \wedge (x', x'') \in \gamma_{\mathrm{KS}}(Z)$.

Because the KS domain has a projection operation, we can create $Y \circ Z$ by first constructing the three-vocabulary matrix $W = \left[\begin{array}{ccc|c} Y_{\mathrm{post}} & Y_{\mathrm{pre}} & 0 & y \\ Z_{\mathrm{pre}} & 0 & Z_{\mathrm{post}} & z \end{array}\right]$. Any element

$(x', x, x'') \in \gamma_{\text{KS}}(W)$ has $(x, x') \in \gamma_{\text{KS}}(Y)$ and $(x', x'') \in \gamma_{\text{KS}}(Z)$. Consequently, projecting away the first vocabulary of W yields a matrix X such that $\gamma_{\text{KS}}(X) = \gamma_{\text{KS}}(Y) \circ \gamma_{\text{KS}}(Z)$, as required.

Join. In [11, §3], King and Søndergaard give a method to compute the join of two KS elements by building a $(6k+3)$-column matrix and projecting onto its last $2k+1$ variables. We improve their approach slightly, building a $(4k+2)$-column matrix and then projecting onto its last $2k+1$ variables. That is, to join two KS elements $Y = \left[\, Y_{\text{pre}}\ Y_{\text{post}} \middle| y \,\right]$ and $Z = \left[\, Z_{\text{pre}}\ Z_{\text{post}} \middle| z \,\right]$, we first construct the matrix $\left[\begin{smallmatrix} -Y_{\text{pre}} & -Y_{\text{post}} & -y & Y_{\text{pre}} & Y_{\text{post}} & y \\ Z_{\text{pre}} & Z_{\text{post}} & z & 0 & 0 & 0 \end{smallmatrix}\right]$, and then project onto the last $2k+1$ columns. Roughly speaking, this works because $\left[\begin{smallmatrix} -Y & Y \\ Z & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right] = 0$ if and only if $Y(v-u) = 0 \wedge Zu = 0$.

Because $(v-u) \in \operatorname{null} Y$, and $u \in \operatorname{null} Z$, $v = ((v-u)+u) \in \operatorname{null} Y + \operatorname{null} Z$. The correctness of this join algorithm can be proved by starting from the King and Søndergaard join matrix [11, §3], applying row-reductions, permuting columns due to be projected away, and partially performing the projection.

Merge Functions. Knoop and Steffen [13] extended the Sharir and Pnueli algorithm [24] for interprocedural dataflow analysis to handle local variables. At a call site at which procedure P calls procedure Q, the local variables of P are modeled as if the current incarnations of P's locals are stored in locations that are inaccessible to Q and to procedures transitively called by Q. Because the contents of P's locals cannot be affected by the call to Q, a *merge function* is used to combine them with the value returned by Q to create the state after Q returns. Other work using merge functions includes Müller-Olm and Seidl [18] and Lal et al. [15].

To simplify the discussion, assume that all scopes have the same number of locals L. Each merge function is of the form

$$\textsc{Merge}(a, b) \stackrel{\text{def}}{=} \textsc{ReplaceLocals}(b) \circ a.$$

Suppose that vocabulary i consists of sub-vocabularies g_i and l_i. The operation $\textsc{ReplaceLocals}(b)$ is defined as follows:

1. Project away vocabulary l_2 from b.
2. Insert L columns for l_2 in which all entries are 0.
3. Append L rows, $[0, I, 0, -I|0]$, so that in $\textsc{ReplaceLocals}(b)$ each variable in vocabulary l_2 is constrained to have the value of the corresponding variable in vocabulary l_1.

The $\widehat{\alpha}$ Operation. King and Søndergaard give an algorithm for $\widehat{\alpha}$ [12, Fig. 2]. That algorithm needs the minor correction of using Howell form instead of row-echelon form for the projections that take place in its join operations, as discussed above.

6 Experiments

Our experiments were run on a single core of a single-processor quad-core 3.0 GHz Xeon computer running 64-bit Windows XP (Service Pack 2), configured so that a user process has 4 GB of memory. To implement $\widehat{\alpha}_{\text{KS}}$, we used the Yices

Prog.	Measures of size				Performance (x86)							Better KS precision
					MOS			KS				
name	instrs	CFGs	BBs	branches	WPDS	post*	query	WPDS	t/o	post*	query	1-voc.
finger	532	18	298	48	0.969	0.281	0.094	121.0	5	0.297	0.016	11/48 (23%)
subst	1093	16	609	74	1.422	0.266	0.031	199.0	4	0.328	0.094	13/74 (18%)
label	1167	16	573	103	1.359	0.282	0.046	154.6	2	0.375	0.032	50/103 (49%)
chkdsk	1468	18	787	119	1.797	0.172	0.031	397.2	16	0.203	0.047	3/119 (2.5%)
logoff	2470	46	1145	306	3.047	2.078	0.610	817.8	19	1.906	0.094	37/306 (12%)
setup	4751	67	1862	589	5.578	1.406	0.484	1917.8	64	1.157	0.063	34/589 (5.8%)

Fig. 1. WPDS experiments. The columns show the number of instructions (instrs); the number of procedures (CFGs); the number of basic blocks (BBs); the number of branch instructions (branches); the times, in seconds, for MOS and KS WPDS construction, running post*, and finding one-vocabulary affine relations at blocks that end with branch instructions, as well as the number of WPDS rules for which KS-weight generation timed out (t/o); and the degree of improvement gained by using $\widehat{\alpha}_{\mathrm{KS}}$-generated KS weights rather than TSL-generated MOS weights (measured as the number of basic blocks that (i) end with a branch instruction, and (ii) begin with a node whose inferred one-vocabulary affine relation was strictly more precise under KS-based analysis).

solver [3], with the timeout for each invocation set to 3 seconds. The experiments were designed to answer the following questions:

1. Is it faster to use MOS or KS?
2. Does MOS or KS yield more precise answers? This question actually has several versions, depending on whether we are interested in
   - the two-vocabulary transformers for individual statements (or basic blocks)
   - the one-vocabulary affine relations that hold at program points

We ran each experiment on x86 machine code, computing affine relations over the x86 registers.

To address question 1, we ran ARA on a corpus of Windows utilities using the WALi [10] system for weighted pushdown systems (WPDSs) [23, 2]. We used two weight domains: (i) a weight domain of TSL-generated MOS transformers, and (ii) a weight domain of $\widehat{\alpha}_{\mathrm{KS}}$-generated KS transformers. The weight on each WPDS rule encoded the MOS/KS transformer for a basic block $B = [I_1, \ldots, I_k]$ of the program, including a jump or branch to a successor block.

- In the case of MOS, the semantic specification of each instruction $I_j \in B$ is evaluated according to the MOS reinterpretation of the operations of the TSL meta-language to obtain $[\![I_j]\!]_{\mathrm{MOS}}$. ($[\![\cdot]\!]_{\mathrm{MOS}}$ denotes the MOS semantics for an instruction.) The single-instruction transformers are then composed: $w^B_{\mathrm{MOS}} := [\![I_k]\!]_{\mathrm{MOS}} \circ \ldots \circ [\![I_1]\!]_{\mathrm{MOS}}$.
- In the case of KS, a formula φ_B is created that captures the concrete semantics of B, and then the KS weight for B is obtained by performing $w^B_{\mathrm{KS}} := \widehat{\alpha}_{\mathrm{KS}}(\varphi_{\mathcal{B}})$.

We used EWPDS merge functions [15] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDS algorithm [14].

Fig. 1 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches) along with the times for constructing abstract transformers and running post*.[4] Column 10 of Fig. 1 shows the number of WPDS rules for which KS-weight generation timed out. During WPDS construction, if Yices times out during $\widehat{\alpha}_{\mathrm{KS}}$, the implementation creates the MOS weight for the rule instead, and then converts it to an over-approximating KS weight (§4.2). The number of rules equals the number of basic blocks plus the number of branches, so a timeout occurred for about 0.3–2.6% of the rules (geometric mean: 1.1%).

The experiment showed that the cost of constructing transformers via an SMT solver is high: creating the KS weights via $\widehat{\alpha}_{\mathrm{KS}}$ is about 185 times slower than creating MOS weights using TSL (computed as the geometric mean of the construction-time ratios).

To address question 2, we performed two experiments:

- On a corpus of 11,144 instances of x86 instructions, we compared (i) the KS transformer created by applying $\widehat{\alpha}_{\mathrm{KS}}$ to a quantifier-free bit-vector (QFBV) formula that captures the precise bit-level semantics of an instruction against (ii) the MOS transformer created for the instruction by the operator-by-operator reinterpretation method supported by TSL [16].
- We compared the precision improvement gained by using $\widehat{\alpha}_{\mathrm{KS}}$-generated KS weights rather than TSL-generated MOS weights in the WPDS-based analyses used to answer question 1. In particular, column 13 of Fig. 1 reports the number of basic blocks that (i) end with a branch instruction, and (ii) begin with a node whose inferred one-vocabulary affine relation was strictly more precise under KS-based analysis.

The first precision experiment showed that the $\widehat{\alpha}_{\mathrm{KS}}$ method is strictly more precise for about 8.3% of the instructions—910 out of the 11,022 instructions for which a comparison was possible. There were 122 Yices timeouts: 105 during $\widehat{\alpha}_{\mathrm{KS}}$ and 17 during the weight-comparison check.

		Undetermined		
Identical precision	KS more precise	Timeout during KS construction	Timeout during KS/MOS comparison	Total
10,112	910	105	17	11,144

Example 4. One instruction for which the $\widehat{\alpha}_{\mathrm{KS}}$-created transformer is better than the MOS transformer is "`add bh,al`", which adds the low-order byte of register `eax` to the second-to-lowest byte of register `ebx`. The transformer created by the TSL-based operator-by-operator reinterpretation method corresponds to `havoc(ebx)`. All other registers are unchanged in both transformers—i.e.,

[4] Due to the high cost of the KS-based WPDS construction, we ran all analyses without the code for libraries. Values are returned from x86 procedure calls in register `eax`, and thus library functions were modeled approximately (albeit unsoundly, in general) by "`eax := ?`", where "`?`" denotes an unknown value [18] (sometimes written as "`havoc(eax)`").

"$(\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx}) \wedge \ldots$". In contrast, the transformer obtained via $\widehat{\alpha}_{\mathrm{KS}}$ is

$$(2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx}) \wedge \ldots$$

Both transformers are over-approximations of the instruction's semantics, but the latter captures a relationship between the low-order two bytes of `ebx` and the low-order byte of `eax`, and hence is more precise.

The second precision experiment was based on the WPDS-based analyses used to answer question 1. The experiment showed that in our WPDS runs, the KS weights identified more precise one-vocabulary affine relations at about 12.3% of the basic blocks that end with a branch instruction (computed as the geometric mean of the precision ratios); see column 13 of Fig. 1.[5] In addition to the phenomenon illustrated in Ex. 4, two other factors contribute to the improved precision obtained via the KS domain:

- As discussed in §4.1 and §4.4, an MOS weight cannot express a transformation involving a guard on the pre-state vocabulary, whereas a KS weight can capture affine equality guards.
- To construct a KS weight w^B_{KS}, $\widehat{\alpha}_{\mathrm{KS}}$ is applied to $\varphi_{\mathcal{B}}$, which not only is bit-level precise, but also includes *all memory-access/update and flag-access/update operations*. Consequently, even though the KS weights we used in our experiments are designed to capture only transformations on registers, w^B_{KS} can account for transformations of register values that involve a sequence of memory and/or flag operations within a basic block.

The fact that $\varphi_{\mathcal{B}}$ can express dependences among registers that are mediated by one or more flag updates and flag accesses by instructions of a block, can allow a KS weight generated by $\widehat{\alpha}_{\mathrm{KS}}(\varphi_{\mathcal{B}})$ to sometimes capture `NULL`-pointer or return-code checks (as affine equality guards). For instance, the `test` instruction sets the zero flag `ZF` to *true* if the bitwise-and of its arguments is zero; the `jnz` instruction jumps if `ZF` is *false*. Thus,

- "`test esi, esi`; . . . `jnz xxx`" is an idiom for a `NULL`-pointer check: "`if(p == NULL)...`"
- "`call foo`; `test eax, eax`; . . . `jnz yyy`" is an idiom for checking whether the return value is zero: "`if(foo(...) == 0)...`".

The KS weights for the fall-through branches include the constraints "$\texttt{esi} = 0 \wedge \texttt{esi}' = 0$" and "$\texttt{eax} = 0 \wedge \texttt{eax}' = 0$", respectively, which both contain guards on the pre-state (i.e., "$\texttt{esi} = 0$" and "$\texttt{eax} = 0$", respectively). In contrast, the corresponding MOS weights—"$\texttt{esi}' = \texttt{esi}$" and "$\texttt{eax}' = \texttt{eax}$"—impose no constraints on the pre-state.

If a block $B = [I_1, \ldots, I_k]$ contains a spill to memory of register R_1 and a subsequent reload into R_2, the fact that w^B_{KS} is created from φ_B, which has

[5] Register `eip` is the x86 instruction pointer. There are some situations that cause the MOS weights to fail to capture the value of `eip` at a successor. Therefore, before comparing the affine relations computed via MOS and KS, we performed `havoc(eip)` so as to avoid biasing the results in favor of KS merely because of trivial affine relations of the form "`eip` = *constant*".

a "global perspective" on the semantics of B, can—in principle—allow w^B_{KS} to capture the transformation $R'_2 = R_1$. The corresponding MOS weight w^B_{MOS} would not capture $R'_2 = R_1$ because the TSL-generated MOS weights are created by evaluating the semantic specifications of the individual instructions of B (over a domain of MOS values) and composing the results. Because each MOS weight $[\![I_j]\!]_{\mathrm{MOS}}$ in the composition sequence $w^B_{\mathrm{MOS}} := [\![I_k]\!]_{\mathrm{MOS}} \circ \ldots \circ [\![I_1]\!]_{\mathrm{MOS}}$ discards all information about how memory is transformed, the net effect on R'_2 in w^B_{MOS} is $\mathtt{havoc}(R'_2)$. A second type of example involving a memory update followed by a memory access within a basic block is a sequence of the form "`push` *constant*; `pop esi`"; such sequences occur in several of the programs listed in Fig. 1.

Unfortunately, in our experiments Yices timed out on the formulas that arose in both kinds of examples, even with the timeout value set to 100 seconds.

7 Related Work

The original work on affine-relation analysis (ARA) was an intraprocedural ARA algorithm due to Karr [9]. Müller-Olm and Seidl introduced the MOS domain for affine relations, and gave an algorithm for interprocedural ARA [19, 21]. King and Søndergaard defined the KS domain, and used it to create implementations of best abstract ARA transformers for the individual bits of a bit-blasted concrete semantics [11, 12]. They used bit-blasting to express a bit-precise concrete semantics for a statement or basic block. The use of bit-blasting let them track the effect of non-linear bit-twiddling operations, such as shift and xor.

In this paper, we also work with a bit-precise concrete semantics; however, we avoid the need for bit-blasting by working with QFBV formulas expressed in terms of word-level operations; such formulas also capture the precise bit-level semantics of each instruction or basic block. We take advantage of the ability of an SMT solver to decide the satisfiability of such formulas, and use $\widehat{\alpha}_{\mathrm{KS}}$ to create best word-level transformers.

In contrast with both the Müller-Olm/Seidl and King/Søndergaard work, we take advantage of the Howell form of matrices. For each of the domains KS, AG, and MOS, because Howell form is canonical for non-empty sets of basis vectors, it provides a way to test pairs of elements for equality of their concretizations—an operation needed by analysis algorithms to determine when a fixed point is reached.

The algorithms given by Müller-Olm and Seidl avoid computing multiplicative inverses, which are needed to put a matrix in Howell form (line 8 of Alg. 1). However, their preference for algorithms that avoid inverses was originally motivated by the fact that at the time of their original 2005 work they were unaware [20] of Warren's $O(\log w)$ algorithms [26, §10-15] for computing the inverse of an odd element, and only knew of an $O(w)$ algorithm [19, Lemma 1].

Gulwani and Necula introduced the technique of random interpretation and applied it to identifying both intraprocedural [5] and interprocedural [6] affine relations. The fact that random interpretation involves collecting samples—which are similar to rows of AG elements—suggests that the AG domain might be used as an efficient abstract datatype for storing and manipulating data during

random interpretation. Because the AG domain is equivalent to the KS domain (see §3), the KS domain would be an alternative abstract datatype for storing and manipulating data during random interpretation.

References

1. E. Bach. Linear algebra modulo n. Unpublished manuscript, Dec. 1992.
2. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
3. B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.
4. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. Tech. Rep. TR-1691, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, June 2011.
5. S. Gulwani and G. Necula. Discovering affine equalities using random interpretation. In *POPL*, 2003.
6. S. Gulwani and G. Necula. Precise interprocedural analysis using random interpretation. In *POPL*, 2005.
7. J. Hafner and K. McCurley. Asymptotically fast triangularization of matrices over rings. *SIAM J. Comput.*, 20(6), 1991.
8. J. Howell. Spans in the module $(\mathbb{Z}_m)^s$. *Linear and Multilinear Algebra*, 19, 1986.
9. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6:133–151, 1976.
10. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
11. A. King and H. Søndergaard. Inferring congruence equations with SAT. In *CAV*, 2008.
12. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.
13. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
14. A. Lal and T. Reps. Improving pushdown system model checking. In *CAV*, 2006.
15. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
16. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
17. C. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, Philadelphia, PA, 2000.
18. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
19. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
20. M. Müller-Olm and H. Seidl. Personal communication, Apr. 2005.
21. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *TOPLAS*, 29(5), 2007.
22. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.
23. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58(1–2), Oct. 2005.
24. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
25. A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, ETH Zurich, Zurich, Switzerland, 2000. Diss. ETH No. 13922.
26. H. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2003.