Abstract Domains of Affine Relations

MATT ELDER, University of Wisconsin
JUNGHEE LIM, University of Wisconsin
TUSHAR SHARMA, University of Wisconsin
TYCHO ANDERSEN, University of Wisconsin
THOMAS REPS, University of Wisconsin and GrammaTech, Inc.

This paper considers some known abstract domains for affine-relation analysis, along with several variants, and studies how they relate to each other. The various 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 (in particular, arithmetic performed modulo 2^w, for some machine-integer width w).

We show that the abstract domains of Müller-Olm/Seidl (MOS) and King/Søndergaard (KS) are, in general, incomparable. However, we give sound interconversion methods. That is, we give an algorithm to convert a KS element v_{KS} to an over-approximating MOS element v_{MOS}—i.e., $\gamma(v_{\mathrm{KS}}) \subseteq \gamma(v_{\mathrm{MOS}})$—as well as an algorithm to convert an MOS element w_{MOS} to an over-approximating KS element w_{KS}—i.e., $\gamma(w_{\mathrm{MOS}}) \subseteq \gamma(w_{\mathrm{KS}})$.

The paper provides insight on the range of options that one has for performing affine-relation analysis in a program analyzer.

—We describe how to perform a greedy, operator-by-operator abstraction method to obtain KS abstract transformers.

—We also describe a more global approach to obtaining KS abstract transformers that considers the semantics of an entire instruction, basic block or other loop-free program fragment.

The latter method can yield *best* abstract transformers, and hence can be more precise than the former method. However, the latter method is more expensive.

We also explain how to use the KS domain for interprocedural program analysis using a bit-precise concrete semantics, but without bit-blasting.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers; formal methods; validation*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants; mechanical verification*

General Terms: Algorithms, Theory, Verification, Experimentation, Performance

Additional Key Words and Phrases: abstract domain, abstract interpretation, affine relation, static analysis, modular arithmetic, Howell form, symbolic abstraction

---

M. Elder is currently affiliated with Quixey. J. Lim is currently affiliated with GrammaTech, Inc. T. Andersen is currently affiliated with Consolidated Court Automation Programs of Wisconsin.
Portions of this work appeared in the 2011 Static Analysis Symposium [Elder et al. 2011].
This work was supported in part by NSF under grants CCF-{0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251, 11-C-0447}, by ARL under grant W911NF-09-1-0413, by AFRL under contracts FA9550-09-1-0279 and FA8650-10-C-7088, and by DARPA under cooperative agreement HR0011-12-2-0012. 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.
Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.
Corresponding author's address: Thomas Reps, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53703, and GrammaTech, Inc., 3310 Univ. Ave., Madison, WI 53705; e-mail: reps@cs.wisc.edu.

1. INTRODUCTION

The work reported in this paper was motivated by our work on TSL [Lim and Reps 2008; 2013], 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.

TSL advances the state of the art in program analysis by providing a YACC-like mechanism for creating the key components of machine-code analyzers. From a description of (a) the concrete operational semantics of a given instruction set, and (b) the operations of a desired abstract domain $\mathcal{A}$, TSL automatically creates an implementation of an *abstract-transformer generator*: the abstract-transformer generator maps a given instruction I to a sound abstract $\mathcal{A}$-transformer for I.

However, the following characteristics of the reinterpretation method used in TSL can cause a given abstract $\mathcal{A}$-transformer to be not as precise as possible:

(1) The reinterpretation method abstracts each TSL meta-language operation in isolation, and is therefore rather myopic.

(2) The operations 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. Unfortunately, few abstract domains retain precision over the full gamut of such operations.

Moreover, the myopia of item (1) can amplify the deficiencies of item (2) because of cascade effects when reinterpretation is applied to a large expression (e.g., one that captures the semantics of a complex machine-code instruction).

In contrast, a more global approach that considers the semantics of an entire instruction—or, even better, an entire basic block or other loop-free program fragment—can yield a more precise abstract transformer. In particular, Cousot and Cousot [1979] gave a *specification* of the most-precise abstract interpretation of a concrete transformer τ that is possible in a given abstract domain $\mathcal{A}$:

> Given a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$, the *best abstract transformer*, $\tau^{\#} : \mathcal{A} \rightarrow \mathcal{A}$, is the most precise abstract operator possible that over-approximates τ. $\tau^{\#}$ can be expressed as follows: $\tau^{\#} = \alpha \circ \tau \circ \gamma$.

The latter equation defines the limit of precision obtainable using abstraction $\mathcal{A}$. However, the definition does not provide a useful *algorithm*, either for applying $\tau^{\#}$ or for finding a representation of the function $\tau^{\#}$. In particular, in many cases, the explicit application of γ to an abstract value would yield an intermediate result—a set of concrete states—that is either infinite or too large to fit in computer memory.

The notion of *symbolic abstraction* [Reps et al. 2004] both (i) adopts a global outlook, and (ii) provides an algorithm for obtaining abstract transformers.

—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 element $\widetilde{\alpha}(\psi) \in \mathcal{A}$ 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 $\widehat{\alpha}$ [Reps et al. 2004]. For every ψ, $\widehat{\alpha}$ finds the best element in $\mathcal{A}$ that over-approximates $[\![\psi]\!]$.

Using symbolic abstraction, the issue of "myopia" can be addressed by first creating a logical formula φ_I that captures the concrete semantics of each instruction I (or basic block, or loop-free program 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 [Lim and Reps 2008, §3.4].

We explored these precision issues in the context of abstract domains for affine-relation analysis (ARA) for modular arithmetic. In this setting, an affine relation is a linear-equality constraint over a given set of variables that hold machine integers. An abstract-domain element represents a set of states that satisfy a conjunction of affine relations. ARA finds, for each point in the program, a domain element that over-approximates the set of states that can arise at that point. ARA generalizes such analyses as linear-constant propagation [Sagiv et al. 1996] and induction-variable analysis.

In our work, we made use of two existing ARA domains—one defined by Müller-Olm and Seidl [2005a], [2007] (MOS) and one defined by King and Søndergaard [2008], [2010] (KS). Both MOS and KS are based on an extension of linear algebra to modules over a ring [Howell 1986; Hafner and McCurley 1991; Bach 1992; Storjohann 2000; Müller-Olm and Seidl 2005a; 2007]. In this paper, we describe our own variant of the KS domain, which is inspired by—but different from, and arguably easier to use than—the version of KS developed by King and Søndergaard. Our version is presented in §5 and §6.

The contributions of our work fall into two broad categories: "Comparing MOS and KS" and "Employing KS".

Comparing MOS and KS. For MOS, it was not previously known how to perform $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ in a non-trivial fashion (i.e., other than defining $\widetilde{\alpha}_{\mathrm{MOS}}$ to be $\lambda f.\top$). In contrast, King and Søndergaard [2010, Fig. 2] gave an algorithm for $\widehat{\alpha}_{\mathrm{KS}}$, which led us to examine more closely how MOS and KS are related.

With respect to the question of how the two abstract domains relate to each other, our contributions include the following:

—We introduce a third domain for representing affine relations, called AG, which stands for *affine generators* (§2.2). Whereas an element in the KS domain consists of a set of *constraints* on the values of variables, AG represents a collection of allowed values of variables via a set of *generators*. We show that AG is the

generator counterpart of KS: a KS element can be converted to an AG element, and vice versa, with no loss of precision (§3).

—We show that MOS and KS/AG are, in general, *incomparable* (§4.1). In particular, we show that KS and AG can express transformers with affine guards, which MOS cannot express.

—We give sound interconversion methods between MOS and KS/AG (§4.2–§4.4):
  —We show that an AG element v_{AG} can be converted to an over-approximating MOS element v_{MOS}—i.e., $\gamma(v_{\mathrm{AG}}) \subseteq \gamma(v_{\mathrm{MOS}})$.
  —We show that an MOS element w_{MOS} can be converted to an over-approximating AG element w_{AG}—i.e., $\gamma(w_{\mathrm{MOS}}) \subseteq \gamma(w_{\mathrm{AG}})$.

—Consequently, by means of the conversion path $\varphi \rightarrow \mathrm{KS} \rightarrow \mathrm{AG} \rightarrow \mathrm{MOS}$, we obtain a method for performing $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ (§4.5).

Employing KS. The version of the KS domain that we work with is inspired by the techniques described in two papers by King and Søndergaard [2008], [2010]. In both this paper and their papers, the goal is to be able to create abstract transformers automatically that are bit-precise, modulo the inherent limitation on precision that stems from having to work with affine-closed sets of values. In the approach described by King and Søndergaard, it is necessary to perform bit-blasting to express a bit-precise concrete semantics for a program's statements or basic blocks. A major drawback of bit-blasting is the huge number of variables that it introduces (e.g., 32 or 64 Boolean-valued variables for each `int`-valued program variable). Given that one needs to perform 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 [2010, §7], although they state that there is room for improvement by, e.g., using sparse matrices.

In this paper, we avoid the use of bit-blasting, and work directly with representations of w-bit affine-closed sets. The motivation for bit-blasting in King and Søndergaard's work was to track the effects of non-linear bit-twiddling operations, such as shift operations, masking operations, and bitwise-complementation. However, symbolic abstraction provides a fully-automatic method for tracking the effects of bit-twiddling operations (§5.8 and §8.3). In §6.2.5, we show that, in some circumstances, the operator-reinterpretation paradigm can also track some of the effects of bit-twiddling operations.

With respect to the topic of employing KS, our contributions include the following:

—We show that best KS transformers can be obtained without resorting to bit-blasting. In place of bit-blasting, 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 $\widehat{\alpha}_{\mathrm{KS}}$—which invokes an SMT solver rather than a Boolean SAT solver—to create best word-level transformers.[1]

[1]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.

—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 (§5).

—In §5.2, we show that the algorithm for projection given by King and Søndergaard [2008, §3] does not always find answers that are as precise as the domain is capable of representing. One consequence is that their algorithm for join does not always find the least upper bound of its two arguments. In §5.2 and §5.4, these issues are corrected by employing the Howell form of matrices [Howell 1986; Storjohann 2000] to normalize KS elements.

—We describe a greedy, operator-by-operator abstraction method for obtaining KS abstract transformers (§6). (The material presented in §6 also serves as a model for how an operator-by-operator abstraction method can be developed for almost any relational numeric abstract domain.)

Experiments. §7 presents an experimental study with the Intel IA32 (x86) instruction set in which the $\widehat{\alpha}_{\mathrm{KS}}$ method and two greedy, operator-by-operator reinterpretation methods—KS-reinterpretation (§6) and MOS-reinterpretation [Lim and Reps 2013, §4.1.2]—are compared in terms of their performance and precision. The precision comparison is done by comparing the affine invariants obtained at branch points, as well as the affine procedure summaries obtained for procedures. For KS-reinterpretation and MOS-reinterpretation, we also compare the abstract transformers generated for individual x86 instructions. The experiments were designed to answer the following questions:

—Which method of obtaining abstract transformers is fastest: $\widehat{\alpha}_{\mathrm{KS}}$, KS-reinterpretation, or MOS-reinterpretation?

—Does MOS-reinterpretation or KS-reinterpretation yield more precise abstract transformers for machine instructions?

—For what percentage of branch points and procedures does $\widehat{\alpha}_{\mathrm{KS}}$ produce more precise answers than KS-reinterpretation?

Organization. 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 element can be converted to a KS element, and vice versa. §4 presents our results on the incomparability of the MOS and KS domains, but gives sound methods to convert a KS element into an over-approximating MOS element, and vice versa. §5 explains how to use the KS domain for interprocedural analysis. §6 describes a greedy, operator-by-operator abstraction method for obtaining KS abstract transformers. §7 presents experimental results. §8 discusses related work. §9 concludes. Proofs can be found in the appendices.

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 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, and thus we must be cautious about carrying over intuition from standard linear algebra. In particular, each odd

element in $\mathbb{Z}_{2^w}$ has a multiplicative inverse (which may be found in time $O(\log w)$ [Warren 2003, Fig. 10-5]), but no even element has a multiplicative inverse. The *rank* of a value $x \in \mathbb{Z}_{2^w}$ is the maximum integer $p \leq w$ such that 2^p divides evenly into x [Müller-Olm and Seidl 2005a; 2007]. For example, $\text{rank}(1) = 0$, $\text{rank}(24) = 3$, 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 transition relation between values of variables in the *pre-state* vocabulary and values of variables in the *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). The rows of a matrix M are numbered from 1 to $\texttt{rows}(M)$; the columns of M are numbered starting from 1.

Because the MOS domain inherently involves pre-state-vocabulary to post-state-vocabulary transformers (see §2.4), our definitions of the AG and KS domains (§2.2 and §2.3, respectively) are also two-vocabulary domains. Technically, AG and KS can have an arbitrary number of vocabularies, including just a single vocabulary. To be able to give simpler examples, some of the AG and KS examples use one-vocabulary domain elements.

States in the various abstract domains are represented by row vectors of length $k+1$. The *row space* of a matrix M is defined by $\text{row}\, M \stackrel{\text{def}}{=} \{x \mid \exists w\colon wM = x\}$. 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 $\text{null}^t\, M \stackrel{\text{def}}{=} \{x \mid Mx^t = 0\}$.

2.1 Matrices in Howell Form

One way to appreciate how linear algebra in rings differs from linear algebra in fields is to see how certain issues are finessed when converting a matrix to *Howell form* [Howell 1986]. The Howell form of a matrix is an extension of reduced row-echelon form [Meyer 2000] suitable for matrices over $\mathbb{Z}_n$. Because Howell form is *canonical* for matrices over principal ideal rings [Howell 1986; Storjohann 2000], it provides a way to test whether two abstract-domain elements are equal—i.e., whether they represent the same set of concrete values. Such an equality test is needed during program analysis to determine whether a fixed point has been reached.

Definition 2.1. The leftmost nonzero value in a row vector is its **leading value**. The leading value's index is the **leading index**. A matrix M is in **row-echelon form** if

—All all-zero rows are at the bottom.

—Each row's leading index is greater than that of the row above it.

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** if

(1) M is in row-echelon form and has no all-zero rows,

(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 $2^p r \in \text{row}([M]_i)$.

□

In Defn. 2.1, item (4) may be confusing, and thus warrants an example.

Example 2.2. Suppose that $w = 4$, so that we are working in $\mathbb{Z}_{16}$. Consider the following two matrices and their Howellizations:

$$M_1 \stackrel{\text{def}}{=} \begin{bmatrix} 4 & 2 & 4 \end{bmatrix} \qquad \textsc{Howellize}(M_1) = \begin{bmatrix} 4 & 2 & 4 \\ 0 & 8 & 0 \end{bmatrix}$$

$$M_2 \stackrel{\text{def}}{=} \begin{bmatrix} 4 & 2 & 4 \\ 0 & 4 & 0 \end{bmatrix} \qquad \textsc{Howellize}(M_2) = \begin{bmatrix} 4 & 2 & 4 \\ 0 & 4 & 0 \end{bmatrix}$$

First, notice that M_1 does not satisfy item (4). M_1 has only one row, [4 2 4], and consider what happens when this row is multiplied by powers of 2:

$$\begin{aligned} 2^1 \cdot [4\ 2\ 4] &= [8\ 4\ 8] \\ 2^2 \cdot [4\ 2\ 4] &= [0\ 8\ 0] \\ 2^3 \cdot [4\ 2\ 4] &= [0\ 0\ 0] \end{aligned}$$

In particular, the leading index of $2^2 \cdot [4\ 2\ 4] = [0\ 8\ 0]$ is 2; however, because $\text{row}([M]_2) = \emptyset$, $[0\ 8\ 0] \notin \text{row}([M]_2)$. Consequently, [0 8 0] must be included in $\textsc{Howellize}(M_1)$. We say that a row like [0 8 0] is a *logical consequence of* [4 2 4] *that is added to satisfy item (4) of Defn. 2.1.*

In contrast, matrix M_2 satisfies item (4) (and, in fact, M_2 is already in Howell form). For matrix M_2 to fail to satisfy item (4), there would have to be some row r and power p for which (a) the leading index i of $2^p r$ is strictly greater than the leading index of r, (b) $2^p r \neq 0$, and (c) $2^p r \notin \text{row}([M]_i)$. In this example, the only interesting quantity of the form $2^p r$ is $2^2 \cdot [4\ 2\ 4] = [0\ 8\ 0]$. The leading index of [0 8 0] is 2, but $[0\ 8\ 0] = 2 \cdot [0\ 4\ 0]$, and so $[0\ 8\ 0] \in \text{row}([M]_2)$. Consequently, M_2 satisfies item (4). □

The Howell form of a matrix is unique among all matrices with the same row space (or null space) [Howell 1986]. As mentioned earlier, this property of Howell form provides a way to test two MOS elements, two KS elements, or two AG elements for equality.

The notion of a *saturated set of generators* used by Müller-Olm and Seidl [2007] is closely related to Howell form, but is defined for an unordered set of matrices 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 §8.2), 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 the 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)

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
```

of Defn. 2.1) with the leading value of every row a power of two. (Line 8 enforces item (2) of Defn. 2.1.) 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. 2.1, and lines 18–19 enforce item (4). Lines 12–13 remove all-zero rows, which is needed for Howell form to be canonical.

Alg. 1 is simple and easy to implement. For analyses over large vocabularies, one should replace Alg. 1, which has cubic-time complexity with, say, the algorithm of Storjohann [2000], which has the same asymptotic complexity as matrix multiplication.

2.2 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 $\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & I \end{array}\right]$. The AG element

$$\left[\begin{array}{c|cccc} 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right] \tag{1}$$

represents the transition relation in which $x'_1 = x_1$, x_2 can have any value, and x'_2 can have any even value.

To compute the join of two AG elements, stack the two matrices vertically and Howellize the result.

2.3 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 M, with $0 \leq r \leq 2k+1$. The concretization of a KS element M is

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

Like the AG domain, the KS domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states.

It is easy to read out affine equalities from a KS element M (regardless of whether M is in Howell form): if

$$\begin{array}{c} \begin{array}{cccccc} x_1 & \cdots & x_k & x'_1 & \cdots & x'_k \end{array}\ 1 \\ \left[\begin{array}{cccccc|c} a_1 & \ldots & a_k & a'_1 & \ldots & a'_k & b \end{array}\right] \end{array}$$

is a row of M, then $\sum_i a_i x_i + \sum_i a'_i x'_i = -b$ is a constraint on $\gamma_{\text{KS}}(M)$. The conjunction of these constraints describes $\gamma_{\text{KS}}(M)$ exactly.

The bottom element of the KS domain is the matrix $\begin{array}{c} \vec{X}\ \ \vec{X}'\ \ 1 \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array}$, and the KS element that represents the identity relation is the matrix $\begin{array}{c} \vec{X}\ \ \vec{X}'\ \ 1 \\ \left[\begin{array}{cc|c} I & -I & 0 \end{array}\right] \end{array}$. Suppose that $w = 4$, so that we are working in $\mathbb{Z}_{16}$. The KS element

$$\begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x'_1 & x'_2 \end{array}\ 1 \\ \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{array}\right] \end{array} \tag{2}$$

represents the transition relation in which $x'_1 = x_1$, x_2 can have any value, and x'_2 can have any even value. Thus, Eqns. (1) and (2) represent the same transition relation in AG and KS, respectively.

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, namely, $\begin{array}{c} \vec{X}\ \ \vec{X}'\ \ 1 \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array}$.

The original King and Søndergaard paper [2008] gives polynomial-time algorithms for join and projection; projection can be used to implement composition (see §5.3).

2.4 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 [Müller-Olm and Seidl 2007]. 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 $x' := x \cdot M + b$, 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 $\langle\mathcal{B}\rangle$

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

The bottom element of the MOS domain is $\emptyset$, and the MOS element that represents the identity relation is the singleton set $\{I\}$. If $w = 4$, the MOS element $\mathcal{B} = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 0 & 0 & 2 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\}$ represents the affine span

$$\langle\mathcal{B}\rangle = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 2 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 4 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \ldots, \left[\begin{array}{c|cc} 1 & 0 & 14 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\},$$

which corresponds to the transition relation in which $x_1' = x_1$, x_2 can have any value, and x_2' can have any even value—i.e., $\mathcal{B}$ represents the same transition relation as Eqns. (1) and (2).

The operations join and compose can be performed in polynomial time. If $\mathcal{B}$ and $\mathcal{C}$ are MOS elements, $\mathcal{B} \sqcup \mathcal{C} = \textsc{Howellize}(\mathcal{B} \cup \mathcal{C})$ and $\mathcal{B};\mathcal{C} = \textsc{Howellize}\{BC \mid B \in \mathcal{B} \wedge C \in \mathcal{C}\}$. 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."

2.5 Domain Heights

In all three domains, an element can be represented via an appropriate matrix in Howell form (where in the case of the MOS domain, we mean a matrix in the extended sense discussed in §2.4). For a fixed bit width and a fixed number of columns, there are only a constant number of Howell-form matrices. Consequently, the KS, AG, and MOS domains are all finite domains, and hence of finite height.

Domain elements need not necessarily be maintained in Howell form; instead, they could be Howellized on demand when it is necessary to check containment (see §5.6). Our implementation maintains domain elements in Howell form using essentially the "list of lists" sparse-matrix representation: each matrix is represented via a C++ vector of rows; each row is a vector of (column-index, nonzero-value) pairs.

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 3.1. 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 [2007, Lemma 2.9] give a decomposition algorithm that nearly performs diagonal decomposition, 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 dualization operation, denoted by $\cdot^{\perp}$, such that the rows of $M^{\perp}$ generate the null space of M, and vice versa.

Definition 3.2. The **dualization** of M, denoted by $M^{\perp}$, is defined as follows:

—$\textsc{Pad}(M)$ is the $(2k+1)$-by-$(2k+1)$ matrix $\left[\begin{smallmatrix} M \\ \vec{0} \end{smallmatrix}\right]$,
—L, D, R is the diagonal decomposition of $\textsc{Pad}(M)$,
—T is the diagonal matrix with $T_{i,i} \stackrel{\text{def}}{=} 2^{w-\text{rank}(D_{i,i})}$, and
—$M^{\perp} \stackrel{\text{def}}{=} \left(L^{-1}\right)^t T \left(R^{-1}\right)^t$

□

This definition of dualization has the following useful property:

THEOREM 3.3. *For any matrix* M, $\text{null}^t M = \text{row}\, M^{\perp}$ *and* $\text{row}\, M = \text{null}^t M^{\perp}$.

PROOF. See App. A. □

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 $\gamma_{\text{KS}}(Z) = \gamma_{\text{AG}}(G)$. We construct Z by letting $[b|X\ X'] = G^{\perp}$ and permuting those columns to $Z \stackrel{\text{def}}{=} [X\ X'|b]$. This works by Thm. 3.3, and because

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

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 can capture restrictions only on its post-state vocabulary. For example, when $k = 1$, the KS element for "assume $x = 2$" (in un-Howellized form) is $\left[\begin{array}{cc|c} {\scriptstyle x} & {\scriptstyle x'} & {\scriptstyle 1} \\ 1 & 0 & -2 \\ 1 & -1 & 0 \end{array}\right]$, i.e., "$x = 2 \wedge x = x'$".

In contrast, an MOS element cannot encode an assume statement. The smallest MOS element that over-approximates "assume $x = 2$" is the identity transformer $\left\{\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right]\right\}$. 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

$$\begin{aligned}
\gamma_{\text{MOS}}(\mathcal{B}) &= \left\{ \begin{bmatrix} x & y & x' & y' \end{bmatrix} \;\middle|\; \exists w_0, w_1 : \left[\begin{array}{c|cc} 1 & x & y \end{array}\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & w_0 & w_0 \\ 0 & w_1 & w_1 \end{array}\right] = \left[\begin{array}{c|cc} 1 & x' & y' \end{array}\right] \right. \\
&\qquad\qquad \left. \wedge\, w_0 + w_1 = 1 \right\} \\
&= \left\{ \begin{bmatrix} x & y & x' & y' \end{bmatrix} \;\middle|\; \exists w_0 : x' = y' = w_0 x + (1 - w_0) y \right\} \\
&= \left\{ \begin{bmatrix} x & y & x' & y' \end{bmatrix} \;\middle|\; \exists w_0 : x' = y' = x + (1 - w_0)(y - x) \right\} \\
&= \left\{ \begin{bmatrix} x & y & x' & y' \end{bmatrix} \;\middle|\; \exists p : x' = y' = x + p(y - x) \right\} \qquad (3)
\end{aligned}$$

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 $a = \begin{bmatrix} 1 & -1 & 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 2 & -2 & 6 & 6 \end{bmatrix}$, and $c = \begin{bmatrix} 0 & 0 & -4 & -4 \end{bmatrix}$. By Eqn. (3), we have $a \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = 0$ in Eqn. (3), $b \in \gamma_{\text{MOS}}(\mathcal{B})$ 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 $\mathcal{B}$ to a KS element is equivalent to stating two-vocabulary affine constraints satisfied by $\mathcal{B}$.

To convert an MOS element $\mathcal{B}$ to a KS element, we

(1) rewrite $\mathcal{B}$ so that every matrix it contains has a 1 in its top-left corner,

(2) build a two-vocabulary AG matrix from each one-vocabulary matrix in $\mathcal{B}$,

(3) join the resulting AG matrices, and

(4) convert the joined AG matrix to a KS element.

For Step (1), we rewrite $\mathcal{B}$ so that

$$\mathcal{B} = \left\{ \left[\begin{array}{c|c} 1 & c_i \\ \hline 0 & N_i \end{array}\right] \right\}, \text{ where } c_i \in \mathbb{Z}_{2^w}^{1\times k} \text{ and } N_i \in \mathbb{Z}_{2^w}^{k\times k}.$$

If our original MOS element $\mathcal{B}_0$ fails to satisfy this property, we can construct an equivalent $\mathcal{B}$ that does. Let $\mathcal{C} = \textsc{Howellize}(\mathcal{B}_0)$; 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 $\langle\mathcal{B}\rangle = \langle\mathcal{B}_0\rangle$.

In Step (2), 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 $B_i \in \mathcal{B}$, $\gamma_{\text{MOS}}(\{B_i\}) = \gamma_{\text{AG}}(G_i)$. In Step (3), we join the G_i matrices in the AG domain to yield one matrix G. Thm. 4.1 states the soundness of this transformation from MOS to AG, i.e., $\gamma_{\text{MOS}}(\mathcal{B}) \subseteq \gamma_{\text{AG}}(G)$. Finally, G is converted in Step (4) to an equivalent KS element by the method given in §3.

Theorem 4.1. *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, $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$.*

Proof. See App. B. □

Because we can easily read affine relations from KS elements (§2.3), this conversion method also gives an easy way to create a quantifier-free formula that over-approximates the meaning of an MOS element. In particular, the formula read out of the KS element obtained from MOS-to-KS conversion captures affine relations implied by the MOS element.

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

$$G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right], \tag{4}$$

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 4.2. 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**. An AG matrix in this form represents the transition relation $x' = x \cdot M + b$. □

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

Require: G is an AG matrix in Howell form

procedure MakeExplicit(G)
  for all i from 2 to $k+1$ **do** ▷ Consider each col. of the pre-state voc.
    if there is a row r of G with leading index i **then**
      if $\operatorname{rank} r_i > 0$ **then**
        for all j from 1 to $2k+1$ **do** ▷ Build s from r, with $s_i = 1$
          $s_j \leftarrow r_j \gg \operatorname{rank} r_i$
        Append s to G
        $G \leftarrow \text{Howellize}(G)$
  for all i from 2 to $k+1$ **do**
    if there is no row r of G with leading index i **then**
      Insert, as the i^{th} row of G, a new row of all zeroes

Explicit form is desirable because we can immediately convert the AG matrix of Defn. 4.2 into the MOS element

$$\left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\}.$$

The matrix G in Eqn. (4) 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

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

As shown in Thm. 4.3, $\gamma_{\text{AG}}(G) = \gamma_{\text{MOS}}(\text{Shatter}(G))$. Intuitively, this property holds because the coefficients of the $[0|0\ R_{j,*}]$ rows in an affine combination of the rows of G correspond to coefficients of the $\left\{ \left[\begin{array}{c|c} 0 & R_{j,*} \\ \hline 0 & 0 \end{array}\right] \right\}$ matrices in an affine combination of the matrices in $\text{Shatter}(G)$.

Theorem 4.3. *When* $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, *then* $\gamma_{AG}(G) = \gamma_{MOS}(\text{Shatter}(G))$.

Proof. See App. B. □

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 4.4. 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 4.5. *For $G \in AG$, $\gamma_{AG}(G) \subseteq \gamma_{MOS}(\textsc{Shatter}(\textsc{MakeExplicit}(G)))$.*

Proof. See App. B. □

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

Example 4.6. The final MOS element for Ex. 4.4 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 Abstraction for the MOS Domain

As mentioned in the introduction, it was not previously known how to perform symbolic abstraction for MOS. Using $\widehat{\alpha}_{\mathrm{KS}}$ (see §5.8) in conjunction with the algorithms from §3 and §4.4, we can soundly define $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ as

$$\textbf{let } G = \textsc{ConvertKSToAG}\left(\widehat{\alpha}_{\mathrm{KS}}(\varphi)\right) \textbf{ in } \textsc{Shatter}\left(\textsc{MakeExplicit}\left(G\right)\right).$$

5. USING KS FOR INTERPROCEDURAL ANALYSIS

This section presents a two-vocabulary version of the KS abstract domain, focusing on the operations that are useful in a program analyzer. Unlike previous work by King and Søndergaard [2008], [2010], it is not necessary to perform bit-blasting to use the version of KS presented here. §5.1–§5.7 describe the suite of operations needed to use the KS domain in an interprocedural-analysis algorithm in the style of Sharir and Pnueli [1981] or Knoop and Steffen [1992], or to use the KS domain as a weight domain in a weighted pushdown system (WPDS) [Bouajjani et al. 2003; Reps et al. 2005; Lal et al. 2005; Kidd et al. 2007].

§5.8 discusses symbolic abstraction for the KS domain, which provides one way to create two-vocabulary KS elements that represent abstract transformers needed by a program analyzer. (§6 provides an alternative method for creating KS abstract transformers, based on the operator-reinterpretation paradigm.)

§5.9 shows how to compute the number of tuples that satisfy a KS element.

5.1 Meet

As discussed in §2.3, the meaning of a KS matrix X can be expressed as a formula by forming a conjunction that consists of one equality for each row of X. We can obtain a KS element that precisely represents the conjunction of any number of such formulas by stacking the rows that represent the equalities, and putting the resulting matrix in Howell form. Consequently, we can compute the meet $X \sqcap Y$ of any two KS elements X and Y by putting the block matrix $\left[\begin{smallmatrix} X \\ Y \end{smallmatrix}\right]$ into Howell form. The resulting matrix exactly represents the intersection of the meanings of X and Y:

$$\gamma_{\text{KS}}(X \sqcap Y) = \gamma_{\text{KS}}(X) \cap \gamma_{\text{KS}}(Y).$$

5.2 Project and Havoc

King and Søndergaard [2008, §3] 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 to create X'; (ii) create X'' by removing from X' every row a in which any of $a_1, \ldots, a_{i-1}$ is nonzero (i.e., $X'' = [X']_i$); and (iii) remove columns $1, \ldots, i-1$. (Note that the resulting matrix has only a portion of the original vocabulary; we have projected away $\{x_1, \ldots, x_{i-1}\}$.) However, although their method works for Boolean-valued KS elements (i.e., KS elements over $\mathbb{Z}_2^k$), when the leading values of X are not all 1, as can occur in KS elements over $\mathbb{Z}_{2^w}^k$ for $w > 1$, step (ii) is not guaranteed to produce the most-precise projection of X onto $x_i, \ldots, x_k$, although the KS element obtained is always sound.

Example 5.1. Suppose that $X = \begin{array}{c} \begin{matrix} x_1 & x_2 & 1 \end{matrix} \\ \left[\begin{array}{cc|c} 4 & 2 & 6 \end{array}\right] \end{array}$, with $w = 4$, and the goal is to project away the first column (for x_1). When the King/Søndergaard projection algorithm is applied to X, we obtain the empty matrix, which represents no constraints on x_2—i.e., $x_2 \in \{0, 1, \ldots, 15\}$. However, closer inspection reveals that x_2 cannot be even; if x_2 were even, then both of the terms $4x_1$ and $2x_2$ would both be divisible by 4, and hence both values would have at least two zeros as their least-significant bits. Such a pair of values could not sum to a value congruent to 6 because the binary representation of 6 ends with $\ldots 10$. □

Instead, we put X in Howell form before removing rows. By Thm. 5.2, step (ii) above returns the exact projection of the original KS element onto the smaller vocabulary.

Theorem 5.2. *Suppose that M has c columns. If matrix M is in Howell form, $x \in \text{null}^t M$ if and only if $\forall i \colon \forall y_1, \ldots y_{i-1} \colon \begin{bmatrix} y_1 & \cdots y_{i-1} & x_i & \cdots & x_c \end{bmatrix} \in \text{null}^t([M]_i)$.*

Proof. See App. C. □

Example 5.3. The Howell form of X from Ex. 5.1 is $\begin{bmatrix} \overset{x_1}{4} & \overset{x_2}{2} & \overset{1}{6} \\ 0 & 8 & 8 \end{bmatrix}$, and thus we obtain the following answer for the projection of X onto x_2: $\begin{bmatrix} \overset{x_2}{8} & \overset{1}{8} \end{bmatrix}$, which represents $x_2 \in \{1, 3, \ldots, 15\}$.

This example illustrates that while the answer produced in Ex. 5.1 by the King/Søndergaard projection algorithm is a sound over-approximation, it is not as precise as the most-precise answer that can be represented in the KS domain. □

Given KS element M, it is also possible to project away a set of variables V that does not constitute a prefix of the vocabulary: create M' by permuting the columns of M so that the columns for the variables in V come first—the order chosen for the V columns themselves is unimportant—and then project away V from M' as described earlier.

The *havoc* operation removes all constraints on a set of variables V. To havoc V from KS element M, project away V and then (i) add back an all-0 column for each variable in V, and (ii) permute columns to restore the original variable order. Because of the all-0 columns, the resulting KS element has no constraints on the values of the variables in V.

Example 5.4. Suppose that we wish to havoc x_2 from the KS value $\begin{bmatrix} \overset{x_1}{2} & \overset{x_2}{4} & \overset{1}{6} \end{bmatrix}$. We permute columns and Howellize to create $\begin{bmatrix} \overset{x_2}{4} & \overset{x_1}{2} & \overset{1}{6} \\ 0 & 8 & 8 \end{bmatrix}$, project onto the vocabulary suffix x_1, obtaining $\begin{bmatrix} \overset{x_1}{8} & \overset{1}{8} \end{bmatrix}$, add back an all-0 column for x_2, $\begin{bmatrix} \overset{x_2}{0} & \overset{x_1}{8} & \overset{1}{8} \end{bmatrix}$, and permute columns back to the original order to obtain $\begin{bmatrix} \overset{x_1}{8} & \overset{x_2}{0} & \overset{1}{8} \end{bmatrix}$. □

5.3 Compose

King and Søndergaard [2010, §5.2] present a technique to compose two-vocabulary affine relations. For completeness, that algorithm follows. Suppose that we have KS elements $Y = \begin{bmatrix} Y_{\text{pre}} & Y_{\text{post}} & y \end{bmatrix}$ and $Z = \begin{bmatrix} Z_{\text{pre}} & Z_{\text{post}} & z \end{bmatrix}$, where Y_{pre}, Y_{post}, Z_{pre}, and Z_{post} are k-column matrices, and y and z are column vectors. We want to compute the relational composition "$Y\,;Z$"; i.e., find some X such that $(x, x'') \in \gamma_{\text{KS}}(X)$ if and only if $\exists x' \colon (x, x') \in \gamma_{\text{KS}}(Y) \wedge (x', x'') \in \gamma_{\text{KS}}(Z)$.

Because the KS domain has a projection operation, we can create $Y\,;Z$ by first constructing the three-vocabulary matrix W,

$$W = \left[\begin{array}{ccc|c} Y_{\text{post}} & Y_{\text{pre}} & 0 & y \\ Z_{\text{pre}} & 0 & Z_{\text{post}} & z \end{array}\right],$$

and then projecting away the first vocabulary of W. 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, the projection yields a matrix X such that $\gamma_{\text{KS}}(X) = \gamma_{\text{KS}}(Y)\,;\gamma_{\text{KS}}(Z)$, as required.

Alternatively, we can think of abstract composition as happening in three steps: (i) adding 0-columns and reordering vocabularies in Y and Z; (ii) computing the meet W of the resulting matrices; and (iii) projecting onto the initial and final

vocabulary of W. Thus, because reordering, meet, and projection are all exact operations, abstract composition is also an exact operation:

$$\gamma_{\text{KS}}(Y\,;Z) = \gamma_{\text{KS}}(Y)\,;\gamma_{\text{KS}}(Z).$$

Note that the steps of the abstract-composition algorithm mimic a standard way to express the composition of concrete relations, i.e.,

$$Y\,;Z = \exists U : Y[\text{pre}, \text{post}] \wedge Z[\text{pre}, \text{post}] \wedge U = Y.\text{post} \wedge U = Z.\text{pre}.$$

5.4 Join

To join two KS elements Y and Z, we first construct the matrix $\left[\begin{smallmatrix} -Y & Y \\ Z & 0 \end{smallmatrix}\right]$ and then project onto the last $2k+1$ columns.

King and Søndergaard [2008, §3] 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.

If Y and Z are considered as representing *linear* spaces, rather than affine spaces, this approach works because $\left[\begin{smallmatrix} -Y & Y \\ Z & 0 \end{smallmatrix}\right]\left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right] = 0$ is true just if $(Y(v-u) = 0) \wedge (Zu = 0)$. Because $(v-u) \in \text{null}\, Y$, and $u \in \text{null}\, Z$, we know that v is the sum of values in $\text{null}\, Y$ and $\text{null}\, Z$, and so v is in their linear closure. In App. D, Thm. D.1 demonstrates the correctness of the same algorithm in affine spaces; that proof is driven by roughly the same intuition.

Join is not exact in the same sense that meet, project, and compose are above: affine spaces are not closed under union. However, this algorithm does return the least upper bound of Y and Z in the space of KS elements.

Neither meet nor compose distribute over join, as illustrated in the following examples:

Meet over join. [2]

---

[2] In this example, we use the fact that $\top = \left[\begin{smallmatrix} x & x' & 1 \\ 1 & 0 & 0 \end{smallmatrix}\right] \sqcup \left[\begin{smallmatrix} x & x' & 1 \\ 0 & 1 & 0 \end{smallmatrix}\right]$. Although technically we are not working with a vector space over a field, the intuition is that the KS element $\left[\begin{smallmatrix} x & x' & 1 \\ 1 & 0 & 0 \end{smallmatrix}\right]$ represents the "line" $x = 0$, the KS element $\left[\begin{smallmatrix} x & x' & 1 \\ 0 & 1 & 0 \end{smallmatrix}\right]$ represents the "line" $x' = 0$, and their affine closure is the whole "plane" (i.e., $\top$).

$$
\begin{aligned}
\text{“}(x = x')\text{”} = \; & \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} = \top \sqcap \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} \\
= \; & \left(\overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]} \sqcup \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]} \right) \sqcap \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} \\
\neq \; & \left(\overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]} \sqcap \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} \right) \sqcup \left(\overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]} \sqcap \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} \right) \\
= \; & \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]} \sqcup \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]} = \overset{x\;\; x'\;\; 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]} = \text{“}(x = 0) \wedge (x' = 0)\text{”}
\end{aligned}
$$

Compose over join.

$$
\begin{aligned}
\text{(i)} \quad \text{“}(x_1 = x_2)\text{”} = \; & \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right]} = \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]} ; \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right]} \\
= \; & \left(\overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right]} \sqcup \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} \right) ; \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right]} \\
\neq \; & \left(\overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right]} ; \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right]} \right) \\
& \sqcup \left(\overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} ; \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right]} \right) \\
= \; & \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]} \sqcup \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]} \\
= \; & \overset{x_1\;\; x_2\;\; x'_1\;\; x'_2\;\; 1}{\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]} = \text{“}(x_1 = 0) \wedge (x_2 = 0)\text{”}
\end{aligned}
$$

$$
\begin{aligned}
\text{(ii)}\quad \text{“}(x_1' = x_2')\text{”} &= \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} = \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} ; \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]} \\
&= \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} ; \left(\overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right]} \sqcup \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]} \right) \\
&\neq \begin{array}{l} \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} ; \left(\overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right]} \right) \\ \sqcup \ \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} ; \left(\overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]} \right) \end{array} \\
&= \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} \sqcup \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} \\
&= \overset{x_1\ x_2\ x_1'\ x_2'\ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} = \text{“}(x_1' = 0) \wedge (x_2' = 0)\text{”}
\end{aligned}
$$

Similarly, join does not distribute over either meet or compose, as illustrated in the following examples:

Join over meet.

$$
\begin{aligned}
\text{“}(x = x')\text{”} &= \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & -1 & 0 \end{array}\right]} = \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]} \sqcup \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & -1 & 0 \end{array}\right]} \\
&= \left(\overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & 0 & 0 \end{array}\right]} \sqcap \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 0 & 1 & 0 \end{array}\right]} \right) \sqcup \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & -1 & 0 \end{array}\right]} \\
&\neq \left(\overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & 0 & 0 \end{array}\right]} \sqcup \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & -1 & 0 \end{array}\right]} \right) \sqcap \left(\overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 0 & 1 & 0 \end{array}\right]} \sqcup \overset{x\ x'\ 1}{\left[\begin{array}{ccc|c} 1 & -1 & 0 \end{array}\right]} \right) = \top \sqcap \top = \text{“true”}
\end{aligned}
$$

Join over compose.

$$
\begin{aligned}
\text{“}(x' = x + 1)\text{”} &= \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} = \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \sqcup \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \\
&= \left(\mathrm{Id}\,;\ \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \right) \sqcup \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \\
&= \left(\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]^{x\ x'\ 1}\ ;\ \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \right) \sqcup \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \\
&\neq \left(\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]^{x\ x'\ 1} \sqcup \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \right) ; \left(\left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \sqcup \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} \right) \\
&= \top\,;\ \left[\begin{array}{cc|c} 1 & -1 & 1 \end{array}\right]^{x\ x'\ 1} = \top = \text{“true”}
\end{aligned}
$$

5.5 Assuming Conditions

By “assuming” a condition φ on a KS element X, we mean to compute a minimal KS element Y such that

$$\gamma_{\mathrm{KS}}(Y) \supseteq \gamma_{\mathrm{KS}}(X) \cap \{v \,|\, \varphi(v)\}\,.$$

This operation is needed to compute the transformer for an assume edge in a program graph (i.e., the true-branch or false-branch of an if-then-else statement). It can also be used to create transformers for assignments; for instance, the transformer for the assignment $x \leftarrow 3u + 2v$ can be created by starting with the KS element for the identity relation on vocabulary $\vec{V}$, $\left[\begin{array}{cc|c} I & -I & 0 \end{array}\right]^{\vec{V}\ \vec{V}'\ 1}$, havocking $x' \in \vec{V}'$, and assuming the equality $x' = 3u + 2v$.

Assuming a w-bit affine constraint is straightforward: rewrite the constraint to isolate 0 on one side; form a matrix row from resulting constraint's coefficients; append the row to the KS element X; and Howellize. In other words, when φ is an affine constraint, we create a one-row KS element that represents φ exactly, and take the meet with X.

It is also possible to perform an assume with respect to an affine congruence of the form “lhs = rhs (mod 2^h)”, with $h < w$. (Examples in which we need to assume such congruences are discussed §6.2.5.2.) We rewrite the congruence as an equivalent congruence modulo 2^w, by multiplying the modulus 2^h and all of the coefficients by 2^{w-h}, to obtain the w-bit affine constraint “2^{w-h}lhs = 2^{w-h}rhs”. We then proceed as before.

5.6 Containment

Two KS elements X and Y are equal if their concretizations are equal: $\gamma(X) = \gamma(Y)$. However, when each KS element is in Howell form, equality checking is trivial because Howell form is unique among all matrices with the same row space (or null space) [Howell 1986]. Consequently, containment can be checked using meet and equality: $X \sqsubseteq Y$ iff $X = X \sqcap Y$.

Require: φ: a QFBV formula
Ensure: $\widehat{\alpha}(\varphi)$ for the KS domain
1: $lower \leftarrow \bot$
2: $i \leftarrow 1$
3:
4: **while** $i \leq \texttt{rows}(lower)$ **do**
5: $p \leftarrow lower[\texttt{rows}(lower) - i + 1]$
$\{p \sqsupseteq lower\}$
6: $S \leftarrow \texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(p))$
7: **if** S is `TimeOut` **then return** $\top$
8: **else if** S is `None` **then** $\{\varphi \Rightarrow \widehat{\gamma}(p)\}$
9: $i \leftarrow i + 1$
10:
11: **else** $\{S \not\models \widehat{\gamma}(p)\}$
12: $lower \leftarrow lower \sqcup \beta(S)$
13: $ans \leftarrow lower$
14: **return** ans

(a)

Require: φ: a QFBV formula
Ensure: $\widehat{\alpha}(\varphi)$ for the KS domain
1: $lower \leftarrow \bot$
2: $i \leftarrow 1$
3: $upper \leftarrow \top$
4: **while** $i \leq \texttt{rows}(lower)$ **do**
5: $p \leftarrow lower[\texttt{rows}(lower) - i + 1]$
$\{p \sqsupseteq lower,\ p \not\sqsupseteq upper\}$
6: $S \leftarrow \texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(p))$
7: **if** S is `TimeOut` **then return** $upper$
8: **else if** S is `None` **then** $\{\varphi \Rightarrow \widehat{\gamma}(p)\}$
9: $i \leftarrow i + 1$
10: $upper \leftarrow upper \sqcap p$
11: **else** $\{S \not\models \widehat{\gamma}(p)\}$
12: $lower \leftarrow lower \sqcup \beta(S)$
13: $ans \leftarrow lower$
14: **return** ans

(b)

Fig. 1. (a) The King-Søndergaard algorithm for symbolic abstraction ($\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$). (b) The Thakur-Elder-Reps bilateral algorithm for symbolic abstraction, instantiated for the KS domain: $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}(\varphi)$. In both algorithms, *lower* is maintained in Howell form throughout.

5.7 Merge Functions

Knoop and Steffen [1992] extended the Sharir and Pnueli algorithm [1981] for interprocedural dataflow analysis to handle local variables. At a site where 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 element returned by Q to create the state in P after the call to Q has finished. Other work using merge functions includes Müller-Olm and Seidl [2004] and Lal et al. [2005].

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

$$\textsc{Merge}(\text{CallSiteVal}, \text{CalleeExitVal}) \stackrel{\text{def}}{=} \text{CallSiteVal}\,;\textsc{ReplaceLocals}(\text{CalleeExitVal}).$$

Suppose that the i^{th} vocabulary consists of sub-vocabularies $\vec{g_i}$ and $\vec{l_i}$. The operation ReplaceLocals(CalleeExitVal) is defined as follows:

(1) Project away vocabulary $\vec{l_2}$ from CalleeExitVal.

(2) Insert L all-0 columns for vocabulary $\vec{l_2}$. The KS element now has no constraints on the variables in $\vec{l_2}$.

(3) Append L rows, $\overset{\vec{g_1}}{[\,0}\ \overset{\vec{l_1}}{I}\ \overset{\vec{g_2}}{0}\ \overset{\vec{l_2}}{-I}\,|\,\overset{1}{0}\,]$, so that in ReplaceLocals(CalleeExitVal) each variable in vocabulary $\vec{l_2}$ is constrained to have the value of the corresponding variable in vocabulary $\vec{l_1}$.

5.8 Symbolic Abstraction ($\widehat{\alpha}(\varphi)$)

King and Søndergaard [2010, Fig. 2] gave an algorithm for the problem of *symbolic abstraction* with respect to the KS domain: given a quantifier-free bit-vector

(QFBV) formula φ, the algorithm returns the best element in KS that over-approximates $[\![\varphi]\!]$. In §1, we denoted such an algorithm generically as $\widehat{\alpha}(\varphi)$. The algorithm given by King and Søndergaard, which we will denote by $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$, 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 in §5.4.

Pseudo-code for $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$ is shown in Fig. 1(a). The matrix *lower* is maintained in Howell form throughout. In line 6, $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$ uses an operation to convert an element p of the KS domain to a logical formula, called the *symbolic concretization* of p. In general,

> For all $A \in \mathcal{A}$, the *symbolic concretization* of A, denoted by $\widehat{\gamma}(A)$, maps A to a formula $\widehat{\gamma}(A)$ such that A and $\widehat{\gamma}(A)$ represent the same set of concrete states (i.e., $\gamma(A) = [\![\widehat{\gamma}(A)]\!]$) [Reps et al. 2004].

For most abstract domains, including KS, it is easy to write a $\widehat{\gamma}$ function. As mentioned in §2.3, affine equalities can be read out from a KS element M (regardless of whether M is in Howell form) as follows:

> If $\begin{bmatrix} \overset{x_1}{a_1} & \overset{\dots}{\dots} & \overset{x_k}{a_k} & \overset{x'_1}{a'_1} & \overset{\dots}{\dots} & \overset{x'_k}{a'_k} & \overset{1}{b} \end{bmatrix}$ is a row of M, then $\sum_i a_i x_i + \sum_i a'_i x'_i = -b$ is a constraint on $\gamma_{\mathrm{KS}}(M)$.

The conjunction of these constraints describes $\gamma_{\mathrm{KS}}(M)$ exactly. Consequently, $\widehat{\gamma}(M)$ can be defined as follows:

$$\widehat{\gamma}(M) \stackrel{\text{def}}{=} \bigwedge_{\substack{[a_1 \ \cdots \ a_k \ a'_1 \ \cdots \ a'_k \ |b] \\ \text{is a row of } M}} \sum_i a_i x_i + \sum_i a'_i x'_i = -b$$

The algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$ is a successive-approximation algorithm: it computes a sequence of successively larger approximations to $[\![\varphi]\!]$. It maintains an under-approximation of the final answer in the variable "*lower*", which is initialized to $\bot$ on line 1. On each iteration, the algorithm selects p, a single row (constraint) of *lower* (line 5), and calls a decision procedure to determine whether there is a model that satisfies the formula "$\varphi \wedge \neg\widehat{\gamma}(p)$" (line 6). When $\varphi \wedge \neg\widehat{\gamma}(p)$ is unsatisfiable, φ implies $\widehat{\gamma}(p)$. In this case, p cannot be used to figure out how to make *lower* larger, so variable i is incremented (line 9), which means that on the next iteration of the loop, the algorithm selects the row immediately above p (line 5).

On the other hand, if the decision procedure returns a model S, the under-approximation *lower* is updated to make it larger via the join performed on the right-hand side of the assignment in line 12

$$lower \leftarrow lower \sqcup \beta(S). \tag{5}$$

Because KS elements represent two-vocabulary relations, S is an assignment of concrete values to both the pre-state and post-state variables:

$$S = [\dots, x_i \mapsto v_i, \dots, x'_i \mapsto v'_i, \dots],$$

or, equivalently,

$$S = [\vec{X} \mapsto \vec{v}, \vec{X}' \mapsto \vec{v}']. \tag{6}$$

The notation $\beta(S)$ in line 12 denotes the abstraction of the singleton state-set $\{S\}$ to a KS element. $\{S\}$ can always be represented exactly in the KS domain as follows (where the superscript t denotes the operation of vector transpose):

$$\beta(S) \stackrel{\text{def}}{=} \begin{array}{c} \begin{array}{cc|c} \vec{X} & \vec{X}' & 1 \end{array} \\ \left[\begin{array}{cc|c} I & 0 & (-\vec{v})^t \\ 0 & I & (-\vec{v}')^t \end{array}\right] \end{array} \tag{7}$$

5.8.1 *Correctness of* $\widehat{\alpha}^{\uparrow}_{KS}$. The argument that Fig. 1(a) is correct is somewhat subtle. In particular, one of the tricky aspects of Fig. 1(a) is the indexing into matrix *lower* via variable i: i is used to index rows of *lower* relative to the *final* row of *lower*. Initially, *lower* is the one-row matrix $\begin{array}{c} \begin{array}{cc|c} \vec{X} & \vec{X}' & 1 \end{array} \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array}$, which represents $\bot_{\text{KS}}$, and $i = 1$ indexes the final row of *lower*. However, assuming that φ is satisfiable, the first iteration of the while loop finds an assignment S that satisfies "$\varphi \wedge \neg(0 = 1)$" (line 6), and performs the assignment "$\mathit{lower} \leftarrow \bot_{\text{KS}} \sqcup \beta(S)$" (line 12), after which *lower* holds the value $\beta(S)$. Thus, as can be seen from Eqn. (7), after the first iteration *lower* has $2k$ rows.

Thereafter, each iteration of the while loop considers a single row p, selected by the assignment $p \leftarrow \mathit{lower}[\texttt{rows}(\mathit{lower}) - i + 1]$ on line 5. During each iteration, either i is incremented and *lower* is left unchanged (line 9), or an update $\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$ is performed (line 12 and Eqn. (5)). The latter step seems problematic because, in general, the join operation will cause the number of rows in *lower* to change. Fortunately, as we show in Lem. 5.8, the join on line 12 leaves the bottommost $i - 1$ rows of *lower unchanged*—whereas the topmost $(\texttt{rows}(\mathit{lower}) - i + 1)$ rows can be changed by the join. The fact that the bottommost $i - 1$ rows are not changed by "$\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$" is what makes it possible to index rows of *lower* relative to the *final* row of *lower*.

Algorithm $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ maintains two invariants:

(1) $\mathit{lower} \sqsubseteq \widehat{\alpha}(\varphi)$

(2) $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})] \sqsupseteq \widehat{\alpha}(\varphi)$

Note that both invariants are established before the loop is entered on line 4: (i) the assignment "$\mathit{lower} \leftarrow \bot$" on line 1 sets $\mathit{lower} = \bot \sqsubseteq \widehat{\alpha}(\varphi)$; and (ii) the assignment "$i \leftarrow 1$" on line 2 sets $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})] = \top \sqsupseteq \widehat{\alpha}(\varphi)$.[3]

Henceforth, we abbreviate $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})]$ as "*upper*", and restate invariant (2) as $\mathit{upper} \sqsupseteq \widehat{\alpha}(\varphi)$. The structure of *lower* is depicted in Fig. 2.

Lemma 5.5. The assignment "$i \leftarrow i + 1$" on line 9 of Fig. 1(a) maintains invariant (1).

Proof. Assume that invariant (1) holds before the assignment on line 9. The assignment does not change *lower*; hence invariant (1) continues to hold after the assignment. □

[3] When $i = 1$, the range $(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})$ is empty, and $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})]$ denotes the empty set of constraints, which equals $\top$.

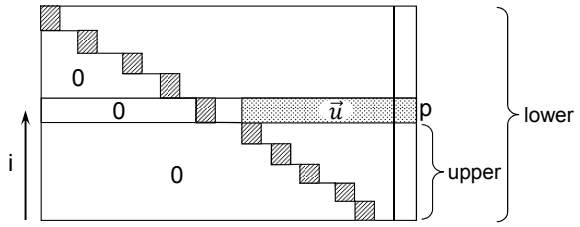


Fig. 2. Depiction of the structure of *lower* during $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$. The shaded blocks running diagonally represent the leading values of the different rows. The dotted block labeled $\vec{u}$ represents a portion of row p.

Lemma 5.6. The assignment "$i \leftarrow i + 1$" on line 9 of Fig. 1(a) maintains invariant (2).

Proof. Assume that invariant (2) holds before the assignment on line 9. By line 6, $\varphi \wedge \neg\widehat{\gamma}(p)$ is unsatisfiable, and hence $\varphi \Rightarrow \widehat{\gamma}(p)$, or equivalently, $[\![\varphi]\!] \subseteq [\![p]\!]$. Moreover, because abstraction function α is monotonic,

$$\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!]) \sqsubseteq \alpha([\![p]\!]) = \widehat{\alpha}(\widehat{\gamma}(p)). \tag{8}$$

Row $p = \mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 1)]$ is a single constraint from the Howell-form matrix *lower*. p by itself is a KS element, although it might not be in Howell form; the Howell-form matrix for $\widehat{\alpha}(\widehat{\gamma}(p))$ equals p, together with any additional rows needed to satisfy Defn. 2.1. (Recall that additional rows are introduced by Defn. 2.1(4).) However, because *lower* is in Howell form, so is the matrix that consists of p and *upper* (i.e., $\mathit{lower}[(\texttt{rows}(\mathit{lower})-i+1) \ldots \texttt{rows}(\mathit{lower})]$). In particular, by Defn. 2.1(4), the row space of *upper* already contains constraints that are equal to or stronger than all of the logical consequences of p. The logical consequences of p—which are all of the form $2^m\vec{u}$ for some m that is sufficiently large to zero out all entries of p to the left of the region labeled $\vec{u}$ in Fig. 2—are exactly the ones with which p is augmented when $\widehat{\alpha}(\widehat{\gamma}(p))$ is put in Howell form. Consequently, by Eqn. (8) and invariant (2),

$$\begin{aligned}\widehat{\alpha}(\varphi) &\sqsubseteq \widehat{\alpha}(\widehat{\gamma}(p)) \sqcap \mathit{upper} \\ &= \mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 1) \ldots \texttt{rows}(\mathit{lower})].\end{aligned}$$

Hence, after the assignment "$i \leftarrow i + 1$" on line 9 of Fig. 1(a), invariant (2) again holds: $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})] \sqsupseteq \widehat{\alpha}(\varphi)$. □

Lemma 5.7. The assignment "$\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$" on line 12 of Fig. 1(a) maintains invariant (1).

Proof. Assume that invariant (1) holds before the assignment on line 12. $S \models \varphi$ implies $\beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$. This property, together with invariant (1), implies that $\mathit{lower} \sqcup \beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$, so invariant (1) holds after the assignment on line 12. □

Lemma 5.8. The assignment "$\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S)$" on line 12 of Fig. 1(a) does not change *upper*.

Proof. See App. E. □

From Lem. 5.8, we conclude the following:

Corollary 5.9. The assignment "$lower \leftarrow lower \sqcup \beta(S)$" on line 12 of Fig. 1(a) maintains invariant (2).

THEOREM 5.10. *If algorithm $\widehat{\alpha}^{\uparrow}_{KS}$ from Fig. 1(a) does not encounter a timeout, (i) the algorithm terminates, and (ii) the element returned is $\widehat{\alpha}(\varphi)$ with respect to the KS domain.*

PROOF. Property (i) follows from the observation that algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ makes progress on each iteration of the while loop, as we now show:

(1) The assignment "$i \leftarrow i + 1$" on line 9 increments i, which is used as an index on rows. Because a Howell-form KS element with $2k + 1$ columns can have at most $2k$ rows, the increment of i on line 9 can be executed no more than $2k$ times.

(2) Row p is a single constraint from the Howell-form matrix *lower*. Thus, just before line 12, we know that $[\![lower]\!] \subseteq [\![p]\!]$. We also know that $S \models \varphi \wedge \neg\widehat{\gamma}(p)$. These two observations imply that $S \not\models \widehat{\gamma}(lower)$.

Consequently, the value of *lower* after the assignment "$lower \leftarrow lower \sqcup \beta(S)$" is strictly greater than (i.e., $\sqsupset$ in the KS lattice) the value of *lower* before the assignment. Because the KS domain is finite-height, line 12 can be performed at most a finite number of times.

Consequently, the algorithm must eventually terminate.

Property (ii) follows from invariants (1) and (2), which were shown to hold by Lemmas 5.5–5.7 and Cor. 5.9. Thus, if algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ from Fig. 1(a) does not encounter a timeout, it reaches line 14 with $i \geq \texttt{rows}(lower) + 1$, in which case

$$lower = \overbrace{lower[(\texttt{rows}(1) \ \ldots \ \texttt{rows}(lower)] \sqsupseteq \widehat{\alpha}}^{invariant\ (2)}\underbrace{(\varphi) \sqsupseteq lower}_{invariant\ (1)},$$

and hence $ans = lower = \widehat{\alpha}(\varphi)$. $\square$

5.8.2 *An Improvement to $\widehat{\alpha}^{\uparrow}_{KS}$.* Algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ from Fig. 1(a) is related to, but distinct from, an earlier $\widehat{\alpha}$ algorithm, due to Reps et al. [2004] (RSY), which applies not just to the KS domain, but to all abstract domains that meet a certain interface. (In other words, $\widehat{\alpha}_{\mathrm{RSY}}$ is the cornerstone of a *framework* for symbolic abstraction.) The two algorithms resemble one another in that they both find $\widehat{\alpha}(\varphi)$ via successive approximation from below. However, there is a key difference in the nature of the satisfiability queries that are passed to the decision procedure by the two algorithms. Compared to $\widehat{\alpha}_{\mathrm{RSY}}$, $\widehat{\alpha}_{\mathrm{KS}}$ issues comparatively inexpensive satisfiability queries in which only a single affine equality is negated[4]—i.e., line 6 of Fig. 1(a) calls $\texttt{Model}(\varphi \wedge \neg\widehat{\gamma}(p))$, where p is a single constraint from *lower*.

This difference—together with the observation that in practice $\widehat{\alpha}_{\mathrm{KS}}$ was about ten times faster than $\widehat{\alpha}_{\mathrm{RSY}}$ when the latter was instantiated for the KS domain—led

[4]See [Thakur et al. 2012, §3] for a more extensive explanation of the differences between $\widehat{\alpha}_{\mathrm{KS}}$ and $\widehat{\alpha}_{\mathrm{RSY}}$.

Thakur, Elder, and Reps [2012] (TER) to investigate the fundamental principles underlying $\widehat{\alpha}_{\mathrm{RSY}}$ and $\widehat{\alpha}_{\mathrm{KS}}$. They developed a new framework, $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$, that transfers $\widehat{\alpha}_{\mathrm{KS}}$'s advantages from the KS domain to other abstract domains [Thakur et al. 2012].

In addition to generating less expensive satisfiability queries, the second benefit of $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$ is that $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$ generally returns a more precise answer than $\widehat{\alpha}_{\mathrm{RSY}}$ and $\widehat{\alpha}_{\mathrm{KS}}$ when a timeout occurs. Because $\widehat{\alpha}_{\mathrm{RSY}}$ and $\widehat{\alpha}_{\mathrm{KS}}$ maintain only under-approximations of the desired answer, if the successive-approximation process takes too much time and needs to be stopped, they must return $\top$ to be sound. In contrast, $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$ is *bilateral*, and can generally return a nontrivial (non-$\top$) element in case of a timeout. That is, $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$ maintains both an under-approximation and a (nontrivial) over-approximation of the desired answer, and hence is resilient to timeouts: $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$ returns the over-approximation if it is stopped at any point.

Fig. 1(b) shows the $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$ algorithm instantiated for the KS domain, which we call $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$. The differences between Fig. 1(a) and (b) are highlighted in gray. In particular, the over-approximation of $\widehat{\alpha}(\varphi)$ is obtained by materializing the ghost variable *upper* from invariant (2) of §5.8.1 as an actual variable. $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ initializes *upper* to $\top$ on line 3. At any stage in the algorithm $\mathit{upper} \sqsupseteq \widehat{\alpha}(\varphi)$. By exactly the same argument given in Lem. 5.6, it is sound to update *upper* on line 10 by performing a meet with the row p that was selected in line 5. Because p's leading index, $LI\,(p)$, is less than the leading index of every row in *upper*, p constrains the value of variable $x_{LI(p)}$, whereas *upper* places no constraints on variable $x_{LI(p)}$. Therefore, $p \not\sqsupseteq \mathit{upper}$, which guarantees progress because $p \sqcap \mathit{upper} \sqsubset \mathit{upper}$. Termination is guaranteed by the same argument used in Thm. 5.10.

In case of a decision-procedure timeout (line 7), $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ returns *upper* as the answer (line 7). If the algorithm finishes without a timeout, then $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ computes $\widehat{\alpha}(\varphi)$; on the other hand, if a timeout occurs, the element returned is generally an over-approximation of $\widehat{\alpha}(\varphi)$—i.e., $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ computes $\widetilde{\alpha}(\varphi)$.

In the KS instantiation of $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$, *upper* can actually be represented implicitly. By invariant (2), we know that $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})] \sqsupseteq \widehat{\alpha}(\varphi)$ always holds. Consequently, the assignment $\mathit{upper} \leftarrow \mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})]$ need only be performed if line 7 is reached, and neither of the assignments on lines 3 and 10 need to be performed explicitly.

5.9 Number of Satisfying Solutions

The *size* of a KS element X with k variables over $\mathbb{Z}_{2^w}$ is the number of k-tuples that satisfy X. The size computation is inexpensive; the size of X depends on the leading values in X, and the number of rows in X. (X is assumed to be in Howell form.)

—If X is bottom, then $\textsc{Size}(X) = 0$.

—Otherwise, we can derive how to compute $\textsc{Size}(X)$ by imagining that we are building up a partial assignment for the variables, from right to left. (In what follows, for simplicity we assume that we have a one-vocabulary KS element and "right to left" means from higher-indexed variables to lower-indexed variables.)

In this case, each variable v_i is constrained by the current partial assignment to the variables $\{v_j \mid i < j\}$, and by the row with leading index i:

—If the leading value of that row is 1, then for every partial assignment to the variables $\{v_j \mid i < j\}$, there is exactly one consistent value for v_i, namely, whatever value for v_i satisfies the equation for the row when the values in the partial assignment are used for the higher-indexed variables.

—If the leading value of a row is 2^m for some value m, then for every partial assignment to the variables $\{v_j \mid i < j\}$, there is exactly one consistent value for $2^m v_i$, namely, whatever value y for $2^m v_i$ satisfies the equation for the row when the values in the partial assignment are used for the higher-indexed variables.

However, there are 2^m different ways to choose v_i to obtain the needed value y. That is, if $\overline{v}$ is a value such that $2^m \overline{v} = y$, then so are all 2^m values in the set

$$\{(\overline{v} + 2^{w-m}p) \pmod{2^w} \mid 0 \leq p \leq 2^m - 1\}.$$

—Finally, if there is no row with leading index i, then v_i is fully unconstrained, and can take on any of the 2^w available values.

Altogether, the product of these counts is the number of satisfying solutions of KS element X. In particular, let u be the number of indices that are not the leading index of any row of X. Then $\textsc{Size}(X)$ is the product of the leading values in X, times $(2^w)^u$.

Example 5.11. Consider again the KS element from Eqn. (2)

$$X_0 = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{array}\right] \end{array},$$

where $w = 4$, so that we are working in $\mathbb{Z}_{16}$. Then $\textsc{Size}(X_0)$ equals $1 \times 8 \times (2^4)^2 = 2,048$.

6. USING KS FOR REINTERPRETATION

Most program analyses that use abstract domains must compute an abstract transformer $\tau^\#$ for each concrete program transformer τ. There are actually two slightly different but related problems that arise:

(1) Applying $\tau^\#$: This process can be viewed as a function ApplyAbsTrans that takes an abstract state and a concrete transformer τ as input, and yields a new abstract state as output. For instance, the effect of the assignment $z \leftarrow x + 2y$ on a state with current abstract value A would be computed as

$$\textsc{ApplyAbsTrans}(\text{“}z \leftarrow x + 2y\text{”}, A).$$

ApplyAbsTrans may be computed in an ad-hoc, analysis-specific way, as long as the resulting abstract state is a sound over-approximation of applying the assignment “$z \leftarrow x + 2y$” to all concrete states in $\gamma(A)$.

(2) Creating a representation of $\tau^\#$: This process can be viewed as a function CreateAbsTrans that takes a concrete transformer τ as input, and yields an abstract transformer as output. In this case, the abstract version of the

assignment $z \leftarrow x + 2y$ would be computed as

$$\textsc{CreateAbsTrans}(\text{“}z \leftarrow x + 2y\text{”}).$$

CreateAbsTrans may also be computed in an ad-hoc, analysis-specific way, as long as the resulting abstract transformer is a sound over-approximation of the concrete semantics of “$z \leftarrow x + 2y$”, viewed as a function from sets of concrete states to sets of concrete states.

Semantic reinterpretation [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993; Lim and Reps 2008] is a principled method for implementing ApplyAbsTrans and CreateAbsTrans. Semantic reinterpretation is 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* [Schmidt 1986]), 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.

To use semantic reinterpretation to implement ApplyAbsTrans, in addition to an abstract type to represent sets of *states*, we also need an abstract type to represents sets of *values*. We will refer to these two types as “abstract states” and “abstract values”, respectively. To reinterpret the assignment “$z \leftarrow x + 2y$” on abstract state A, we

(1) Use the abstract state A to compute abstract-integer values for x and y.
(2) Use abstract multiplication and addition to compute abstract-integer values for the expressions $2y$ and $x + 2y$, respectively.
(3) Use abstract assignment to create a new abstract state A' in which z is constrained to have the abstract value that was computed for $x + 2y$ in step (2).

Here the semantic core consists of integers, states, operations like multiplication and addition, and operations to lookup a variable's value in a state and to create a variant of a given state in which a variable is bound to a new value. The reinterpretation of the core consists 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.

In the remainder of this section, we describe the algorithms needed to create a semantic reinterpretation based on the KS domain. The KS reinterpretation that we present creates an abstract transformer for an assignment statement (§6.2.6); in other words, our KS reinterpretation implements CreateAbsTrans. It would also be possible to define a different KS reinterpretation that implements ApplyAbsTrans—and the reader will see that some of the sub-pieces of the KS CreateAbsTrans have the flavor of ApplyAbsTrans.

The material presented in this section also serves as a model for how an operator-by-operator abstraction method can be developed for almost any relational numeric abstract domain.

TSL [Lim and Reps 2008; 2013] is a system that produces semantic reinterpretations of machine-code instructions, given the abstract domains (in the form of

types and operators) needed for a specific semantic reinterpretation. The needs of TSL motivated the work described in this section, and we use two TSL examples to illustrate our methods (Exs. 6.1 and 6.3); however, our presentation is intended to be understandable without any prior knowledge of TSL.

For a reader interested solely in implementing a "traditional" abstract interpretation using the KS domain, this section should still provide insight on details that are useful in any implementation of ApplyAbsTrans and CreateAbsTrans for the KS domain. Such a reader may think of semantic reinterpretation as just a particular way to organize the implementation of ApplyAbsTrans and CreateAbsTrans.

6.1 Types for Reinterpretation

The discussion of KS values in §4 and §5 focused primarily on abstract transformers. In that discussion, we assumed that (i) a KS value describes a transition relation between states, where each state is an assignment to some set of variables, and (ii) both states have the same of variables. In this section, we use such abstract values to represent state-to-*state* transformations; however, to describe the KS reinterpretation we will also introduce a way to represent state-to-*value* transformations.

Notation for Varying Vocabularies. For a set of variables V, the type KS[V] denotes the set of affine-closed sets of assignments to V. When V and V' are disjoint sets of variables, the type KS[$V; V'$] denotes the set of KS values over variables $V \cup V'$. KS[$V; V'$] could also be written as KS[$V \cup V'$], but because V and V' generally denote the pre-state and post-state variables, respectively, the notation KS[$V; V'$] emphasizes the different roles of V and V'. We extend this notation to cover singletons: if i is a single variable not in V, then the type KS[$V; i$] denotes the set of KS values over the variables $V \cup \{i\}$. (Operations sometimes introduce additional temporary variables, in which case we have types like KS[$V; i, i'$] and KS[$V; i, i', i''$].)

For example, our implementation tracks affine relationships between processor registers at different program points, e.g., `p` and `p'`. For analyzing machine code, the abstraction of instructions is KS[$R; R'$], where R is the set of register names in the processor (e.g., for Intel `IA32` machine code, {`eax`, `ebx`, ...}), and R' is the same set of names, distinguished by primes ({`eax'`, `ebx'`, ...}). R and R' denote the pre-state and post-state registers for executions that start at `p` and finish at `p'`. (Typically, we think of the states at `p` as "initial states" and those at `p'` as "current states".)

In a reinterpretation that yields abstractions of concrete transition relations, the state type becomes a KS[$R; R'$] relation on pre-states to post-states, and a machine-integer type becomes a relation on pre-states to machine integers. Thus, for machine-integer types, we introduce a fresh variable i to hold the "current value" of a reinterpreted machine integer. Because R still refers to the pre-state registers, we write the type of a KS-reinterpreted machine integer as KS[$R; i$]. Although technically we are working with relations, for a KS[$R; i$] value it is often useful to think of R as a set of *independent variables* and i as the *dependent variable*.

Example 6.1. Fig. 3 shows a TSL specification for the `MOV` and `ADD` instructions of the Intel `IA32` instruction set. Consider the instruction "`add eax, ebx`", which

```
(1)   // Abstract-syntax declarations
(2)   reg: EAX() | EBX() | . . . ;
(3)   flag: ZF() | SF() | . . . ;
(4)   operand: Indirect(reg reg INT8 INT32) | DirectReg(reg) | Immediate(INT32) | ...;
(5)   instruction: MOV(operand operand) | ADD(operand operand) | . . . ;
(6)   state: State(MAP[INT32,INT8]   // memory-map
(7)                MAP[reg,INT32]  // register-map
(8)                MAP[flag,BOOL]);   // flag-map
(9)   // Interpretation functions
(10)  INT32 interpOp(state S, operand op) { . . . };
(11)  state updateFlag(state S, INT32 v1, INT32 v2, INT32 v3) { . . . };
(12)  state updateState(state S, operand op, INT32 val) { . . . };
(13)  state interpInstr(instruction I, state S) {
(14)    with(I) (
(15)      MOV(dstOp, srcOp):
(16)        let srcVal = interpOp(S, srcOp);
(17)        in ( updateState(S, dstOp, srcVal) ),
(18)      ADD(dstOp, srcOp):
(19)        let dstVal = interpOp(S, dstOp);
(20)            srcVal = interpOp(S, srcOp);
(21)            result = dstVal + srcVal;
(22)            S2 = updateFlag(S, dstVal, srcVal, result);
(23)        in ( updateState(S2, dstOp, result) ),
(24)      . . .
(25)    );
(26)  };
```

Fig. 3. A fragment of the TSL specification of the concrete semantics of the Intel IA32 instruction set.

(i) adds the value of register `ebx` to that of `eax`, (ii) stores the result in `eax`, and (iii) sets the processor's flags according to the result. The instruction would be represented as the TSL term "$\texttt{ADD}(\texttt{DirectReg}(\texttt{EAX}()), \texttt{DirectReg}(\texttt{EBX}()))$". The semantics of $\texttt{ADD}(\cdot, \cdot)$ terms is specified on lines (18)–(23) of Fig. 3.

To simplify the example, assume that R, the set of register names, is $\{\texttt{eax}, \texttt{ebx}\}$. To translate "$\texttt{add eax}, \texttt{ebx}$" to a KS$[\{\texttt{eax}, \texttt{ebx}\}; \{\texttt{eax}', \texttt{ebx}'\}]$ value, we would evaluate

$$\texttt{interpInstr}(\texttt{ADD}(\texttt{DirectReg}(\texttt{EAX}()), \texttt{DirectReg}(\texttt{EBX}())), \mathtt{S}_{\mathrm{Id}})$$

under the KS-reinterpretation, where $\mathtt{S}_{\mathrm{Id}}$ is the identity relation for KS$[\{\texttt{eax}, \texttt{ebx}\}; \{\texttt{eax}', \texttt{ebx}'\}]$, namely,

$$\left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right].$$

As will be discussed in §6.2.1, §6.2.2, and §6.2.4, the variables used in lines (19)–

(21) would have the following values of type KS[{eax, ebx}; i]:

Line	Variable	Value of type KS[{eax, ebx}; i]	Meaning
(19)	dstVal	$\overset{i}{1}\ \overset{\texttt{eax}}{-1}\ \overset{\texttt{ebx}}{0}\ \big\vert\ \overset{1}{0}$	$i = \texttt{eax}$
(20)	srcVal	$\overset{i}{1}\ \overset{\texttt{eax}}{0}\ \overset{\texttt{ebx}}{-1}\ \big\vert\ \overset{1}{0}$	$i = \texttt{ebx}$
(21)	result	$\overset{i}{1}\ \overset{\texttt{eax}}{-1}\ \overset{\texttt{ebx}}{-1}\ \big\vert\ \overset{1}{0}$	$i = \texttt{eax} + \texttt{ebx}$

□

6.2 Operations on KS-Reinterpreted Integers

Semantic reinterpretation computes the value of an expression by first evaluating the constants and variables at the expression's leaves, then evaluating the operation at each internal node, until it yields an abstract value for the entire expression. Semantic reinterpretation is compositional, in the same way that the concrete interpretation of expressions is compositional. Therefore, a reinterpretation is fully defined by how it handles constants, variable accesses, operations in expressions, and assignments. In this section, we show how these operations are carried out for the KS domain.

6.2.1 *Constants.* The KS reinterpretation of a constant c is the KS[V; i] value that encodes the equation $i = c$: $\overset{i}{1}\ \overset{V}{0 \ldots 0}\ \big\vert\ \overset{1}{-c}$.

6.2.2 *Variable-Access Expressions.* The KS reinterpretation of a variable-access expression "access(S, v_x)", where S's value is a KS state-transformer of type KS[V; V'] and $v_x \in V$, is the KS[V; i] value obtained as follows:

(1) Extend S to be a KS[V; V'; i] value, adding an all-0 column for i. (An all-0 column for i means that there is no constraint on the value of i.)

(2) Assume the constraint $i = v'_x$ on the extended S value. (We wish to obtain the value of variable v_x from the "current state", which corresponds to vocabulary V'.)

(3) Project away V', yielding a KS[V; i] value, as desired.

Assuming the constraint $i = v'_x$ is straightforward, because it is represented exactly by the KS value $\overset{i}{1}\ \overset{V}{0}\ \overset{v'_1}{0}\ \overset{\ldots}{\ldots}\ \overset{v'_x}{-1}\ \overset{\ldots}{\ldots}\ \overset{v'_k}{0}\ \big\vert\ \overset{1}{0}$.

6.2.3 *Multiplication by a Constant.* Suppose that we have the KS[V; i] value x, and wish to compute the KS[V; i] value for the expression $c * x$. We proceed as follows:

(1) Extend x to be a KS[V; i, i'] value, adding an all-0 column for i'.

(2) Assume the constraint $i' = ci$ on the extended x value.

(3) Project away i, yielding a KS[V; i'] value.

(4) Rename i' to i, yielding a KS[V; i] value, as desired.

The constraint $i' = ci$ is represented exactly by the KS value $\begin{bmatrix} \overset{i'}{1} & \overset{i}{-c} & \overset{V}{0} & \overset{1}{0} \end{bmatrix}$. Because projection and the added constraint are both exact, the resulting value is the most precise value that the KS domain can represent.

6.2.4 *Addition.* Suppose that we have two KS$[V; i]$ values x and y, and wish to compute the KS$[V; i]$ value for the expression $x + y$. We proceed as follows:

(1) Rename y's i variable to i'; this makes y a KS$[V; i']$ value.
(2) Extend both x and y to be KS$[V; i, i', i'']$ values, adding all-0 columns for i' and i'' to x, and all-0 columns for i and i'' to y. This step causes i' and i'' to be unconstrained in x, and i and i'' to be unconstrained in y.
(3) Compute $x \sqcap y$.
(4) Assume the constraint $i'' = i + i'$ on the KS$[V; i, i', i'']$ value computed in step (3).
(5) Project away i and i', yielding a KS$[V; i'']$ value.
(6) Rename i'' to i, yielding a KS$[V; i]$ value, as desired.

The vocabulary manipulations in the first two steps put the values into comparable form (i.e., KS$[V; i, i', i'']$), and are easy to perform. The constraint $i'' = i + i'$ is represented exactly by the KS value $\begin{bmatrix} \overset{i''}{1} & \overset{i'}{-1} & \overset{i}{-1} & \overset{V}{0} & \overset{1}{0} \end{bmatrix}$. Because projection, meet, and the added constraint are all exact, the resulting value is the most precise value that the KS domain can represent.

Remark 6.2. Multiplication by a constant and addition are both examples of linear operations. The KS domain can precisely compute any linear combination of KS-value types (e.g., given two KS$[V; i]$ values x and y, we can compute a KS$[V; i]$ value for the expression $3x + 8y$). The steps are similar to those used for addition except that step (4) would be, e.g., "Assume the constraint $i'' = 3i + 8i'$ on $x \sqcap y$".

However, semantic reinterpretation is compositional at the level of individual operators. It creates an over-approximating abstract transformer for a state-transformation expression via an over-approximating reinterpretation for *each individual operator*, and thus $3x+8y$ would be treated as two instances of multiplication-by-a-constant and one instance of addition. While in some cases this approach is myopic—i.e., one could obtain a more precise transformer by considering the semantics of an entire instruction (or, even better, an entire basic block or other loop-free program fragment)—in the case of compositions of linear operators there is no loss of precision. For instance, when we treat the expression $3x + 8y$ as two multiplication-by-a-constant operators and an addition, we obtain the same KS$[V; i]$ value that we would obtain by treating $3x + 8y$ as a single linear operator. □

6.2.5 *Non-Linear Operations.* The KS domain cannot interpret most instances of non-linear operations precisely. However, when a KS$[V; i]$ value has the form $\begin{bmatrix} \overset{i}{1} & \overset{V}{0 \ldots 0} & \overset{1}{-c} \end{bmatrix}$, the dependent variable i of a KS$[V; i]$ value can have only a single concrete value c, regardless of the values of the independent variables; i.e., the KS$[V; i]$ value represents the constant c. When an operation's input KS$[V; i]$

values each denote a constant, the operation can be performed precisely by performing it in concrete arithmetic on the identified constants. In essence, this approach uses special-case handling to identify constants, and then performs constant propagation—including constant propagation over non-linear operators.

6.2.5.1 *Identifying Partially-Constant Values.* We can generalize the notion of "constant value" to a class of *partially-constant* values. A variable is partially constant if some bits of the dependent variable are constant across all valid assignments to the independent variables, even though overall the dependent variable might take on multiple values.

For instance, consider the following KS$[\{x, y\}; i]$ value, in $\mathbb{Z}_{16}$:

$$X = \begin{array}{c} \overset{i}{} \quad \overset{x}{} \quad \overset{y}{} \quad \overset{1}{} \\ \left[\begin{array}{cccc|c} 1 & 12 & 8 & 13 \end{array}\right] \end{array}$$

This value captures the congruence $i + 12x + 8y = 3$. If we consider these values modulo 4, we would have $i + 0x + 0y = 3 \pmod 4$, which means that the rightmost two bits of i must both be `1`, even though the leftmost two bits of i depend on the values of x and y. Consequently, i is partially constant.

Using projection, we can locate the right-hand constant portion of a partially-constant KS variable, and determine the value of those constant bits:

> Given a non-bottom KS$[V; i]$ value X, project away the variables V and Howellize the result. Call the Howellized matrix X'. If X' is the empty matrix, i is fully non-constant; it may be any value in $\mathbb{Z}_{2^w}$. Otherwise, for some m and c, Howellization leaves X' in the following form:
>
> $$X' = \begin{array}{c} \overset{i}{} \quad \overset{1}{} \\ \left[\begin{array}{c|c} 2^m & -2^m c \end{array}\right]. \end{array}$$

X' denotes the congruence $2^m i = 2^m c \pmod{2^w}$, which may be expressed equivalently as $i = c \pmod{2^{w-m}}$.[5] Consequently, the rightmost $w - m$ bits of i are constant, and their value is c.

Notice that i is (fully) constant exactly when $m = 0$.

6.2.5.2 *Bitwise Operations on Partially-Constant Values.* In this section, we consider the case of binary bitwise operations, such as bitwise-and, bitwise-or, and bitwise-xor, when we have information that the argument KS$[V; i]$ values are partially constant. In such a case, we can obtain a non-trivial over-approximation of the result of a bitwise operation by the method described below.

First, the partially-constant values for the two arguments are computed using the technique described in §6.2.5.1. For the moment, assume that for each argument exactly m bits are known to be partially constant. Let a denote the rightmost m bits of the first argument, and b denote the rightmost m bits of the second argument. Let $\bowtie$ denote the operation to be performed (i.e., bitwise-and, bitwise-or, or bitwise-xor). The exact value for the rightmost m bits of the answer is $c = a \bowtie b$. We then proceed as in §6.2.4, except that step (4) assumes the congruence $i'' =$

[5] As discussed in §5.5, a congruence of the form "lhs = rhs (mod 2^h)" can be expressed as the w-bit affine constraint "2^{w-h}lhs = 2^{w-h}rhs". In the example above, $m = w - h$, and thus $h = w - m$.

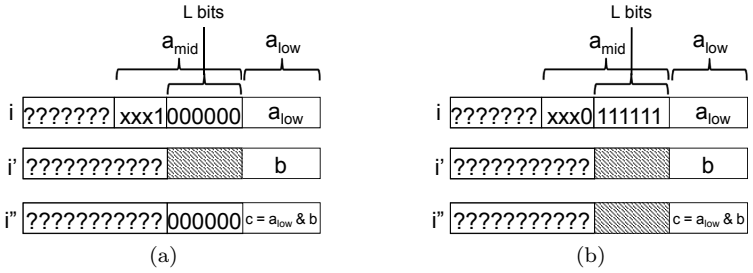


Fig. 4. Bitwise-and when more bits are known to be partially constant for the first argument i than for the second argument i'. (a) a_{mid} ends in L zeros; (b) a_{mid} ends in L ones.

$c \pmod{2^m}$ (as explained in §5.5). The latter constraint expresses the condition that the rightmost m bits of i'' are c.

When different numbers of bits are known for the two arguments, we can also manage to retain precision for some of the bit-locations where only *one* of the two operands is known to be partially constant. For example, suppose that we wish to perform a bitwise-and operation, and we know the rightmost $m' + m$ bits of the first argument and the rightmost m bits of the second argument. In particular, suppose that the first argument's partially-constant value is $2^m a_{\text{mid}} + a_{\text{low}}$, where $0 \leq a_{\text{low}} < 2^m$ and $0 \leq a_{\text{mid}} < 2^{m'}$. (That is, a_{low} is the value of the m "low" bits of the first argument, and a_{mid} is the value of the m' "middle" bits of the first argument.) Let b denote the rightmost m bits of the second argument, and let $c = a_{\text{low}} \,\&\, b$.

The binary representation of a_{mid} must either end in a string of zeros or ones. If a_{mid} ends in L zeros, then we know that the corresponding L bits in the answer are also 0 (see Fig. 4(a)). Thus, we can capture $L + m$ constant bits in the answer by using the method from §6.2.4, except that step (4) assumes the congruence

$$i'' = c \pmod{2^{L+m}}.$$

On the other hand, if a_{mid} ends in a string of L ones, then we know that the corresponding L bits of the answer are equal to the corresponding L bits of the second argument—which may vary. Usually, linear congruences cannot describe a bit region that does not stretch to the least-significant end of the value. In this case, though, the rightmost m bits of the second argument are known to be the constant value b (see Fig. 4(b)). If i' represents the value of the second argument, then the rightmost m bits of $i' - b$ are zeros, and the remaining bits are the bits of i'. Thus, we can capture $L + m$ bits in the answer using the method from §6.2.4, except that step (4) assumes the congruence

$$i'' = i' - b + c \pmod{2^{L+m}}. \tag{9}$$

Both bitwise-or and bitwise-xor can be handled in a similar fashion. The only subtlety is that for bitwise-xor, when a_{mid} ends in a string of L ones, the analogue of Eqn. (9) must *complement* the L middle-region bits of the second argument. The bitwise-complement of a value v is $-v - 1$, and hence can be expressed as an affine constraint. Using i' to represent the second argument, $i' - b$ is i' with its rightmost m bits replaced by zeros. Thus, $-(i' - b) - 1$ is the bitwise complement of i' with

its rightmost m bits replaced by ones. To eliminate those ones, we subtract $2^m - 1$; that is, $-(i' - b) - 2^m$ is the bitwise complement of i' with its rightmost m bits replaced by zeros. Consequently, we can capture $L + m$ bits in the answer using the method from §6.2.4, except that step (4) assumes the congruence

$$i'' = -(i' - b) - 2^m + c \pmod{2^{L+m}},$$

where here $c = a_{\text{low}} \operatorname{xor} b$.

6.2.6 *Assignments.* We are now left with the question of how to reinterpret the assignment of the value of an expression e to a variable (in a reinterpreted state). Suppose that (i) we have the abstract transition relation $S \in \text{KS}[V; V']$, (ii) the reinterpretation of e produces the reinterpreted value $X \in \text{KS}[V; i]$, and (iii) we want to create an abstract transition relation $S'' \in \text{KS}[V; V']$ that acts like S, except that the post-state ("current-state") variable $v' \in V'$ holds the value $X \in \text{KS}[V; i]$. In other words, we want to perform the abstract state update

$$S'' \leftarrow \texttt{updateState}(S, v', X).$$

This operation can be carried out by the following sequence of steps:

(1) Let S' be the result of havocking v' from S. (As discussed in §5.2, to havoc v', project away v' and then add back an all-0 column for v'.)

(2) Let X' be the result of starting with X, renaming i to v', and then adding an all-0 column corresponding to every variable in the set $V' \setminus \{v'\}$. Note that $X' \in \text{KS}[V; V']$.

(3) Return $S'' \stackrel{\text{def}}{=} S' \sqcap X'$.

In this method, S' captures the state in which we "forget" the previous value of v', and X' captures the assertion that v' equals the value of the assignment's expression.

Example 6.3. Returning to Ex. 6.1, consider the "assignment" operation in the interpretation of the instruction "`add eax, ebx`". To create the abstract transformer for "`add eax, ebx`", the initial value of `state` S supplied to `interpInstr` in line (13) of Fig. 3 would be the identity element of $\text{KS}[\{\texttt{eax}, \texttt{ebx}\}; \{\texttt{eax}', \texttt{ebx}'\}]$, namely,

$$\begin{array}{c} \begin{array}{cccc} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' \end{array} \quad 1 \\ \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] \end{array}.$$

In the KS reinterpretation of TSL, the function `updateState` would be reinterpreted to perform an abstract assignment, using the method described above. As described in Ex. 6.1, variable `result` would have the $\text{KS}[V; i]$ value $\begin{array}{c} \begin{array}{cccc} i & \texttt{eax} & \texttt{ebx} & \end{array} \quad 1 \\ \left[\begin{array}{cccc|c} 1 & -1 & -1 & 0 \end{array}\right] \end{array}$. The function call "`updateState(S2, dstOp, result)`" on line (23) of Fig. 3 would return (the Howellization of) the following abstract transformer:

$$\begin{array}{c} \begin{array}{cccc} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' \end{array} \quad 1 \\ \left[\begin{array}{cccc|c} 1 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] \end{array},$$

which corresponds to the transition relation

$$(\texttt{eax}' = \texttt{eax} + \texttt{ebx}) \wedge (\texttt{ebx}' = \texttt{ebx}).$$

Instruction Characteristics		
Kind	# instruction instances	# different opcodes
ordinary	12,734	164
lock prefix	2,048	147
rep prefix	2,143	158
repne prefix	2,141	154
full corpus	19,066	179

Fig. 5. Some of the characteristics of the corpus of 19,066 (non-privileged, non-floating point, non-mmx) instructions.

7. EXPERIMENTS

In this section, we present the results of experiments to evaluate the costs and benefits—in terms of time and precision—of the methods described in earlier sections. The experiments were designed to shed light on the following questions:

(1) Which method of obtaining abstract transformers is fastest: $\widehat{\alpha}_{\mathrm{KS}}$ (§5.8), KS-reinterpretation (§6), or MOS-reinterpretation ([Lim and Reps 2013, §4.1.2])?

(2) Does MOS-reinterpretation or KS-reinterpretation yield more precise abstract transformers for machine instructions?

(3) For what percentage of program points does $\widehat{\alpha}_{\mathrm{KS}}$ produce more precise answers than KS-reinterpretation and MOS-reinterpretation? This question actually has two versions, depending on whether we are interested in
—one-vocabulary affine relations that hold at branch points
—two-vocabulary procedure summaries obtained at procedure-exit points.

As shown in §4.1, the MOS and KS domains are incomparable. To compare the final results obtained using the two domains, we converted each MOS element to a KS element, using the algorithm from §4.2, and then checked for equality, containment (§5.6), or incomparability. It might be argued that this approach biases the results in favor of KS. However, if we have run an MOS-based analysis and are interested in using affine relations in a client application, we must extract an affine relation from each computed MOS element. In §4.1, we showed that, in general, an MOS element $\mathcal{B}$ does not represent an affine relation; thus, a client application needs to obtain an affine relation that over-approximates $\gamma_{\mathrm{MOS}}(\mathcal{B})$. Consequently, the comparison method that we used *is* sensible, because it compares the precision of the affine relations that would be seen by a client application.

7.1 Experimental Setup

To address these questions, we performed two experiments. Both experiments were run on a single core of a single-processor 16-core 2.27 GHz Xeon computer running 64-bit Windows 7 Enterprise (Service Pack 1), configured so that a user process has 4 GB of memory.

Per-Instruction Experiment. On a corpus of 19,066 instances of x86 instructions, we measured (i) the time taken to create MOS and KS transformers via the operator-by-operator reinterpretation method supported by TSL [Lim and Reps

Program Name	Measures of Size						
					WPDS Rules		
	Instrs	Procs	BBs	Branches	Δ_0	Δ_1	Δ_2
write	232	10	134	26	10	151	5
finger	532	18	298	48	18	353	20
subst	1093	16	609	74	16	728	13
chkdsk	1468	18	787	119	18	887	32
convert	1927	38	1013	161	38	1266	22
route	1982	40	931	243	40	1368	63
comp	2377	35	1261	224	35	1528	30
logoff	2470	46	1145	306	46	1648	72
setup	4751	67	1862	589	67	2847	121

Fig. 6. Program information. All nine utilities are from Microsoft Windows version 5.1.2600.0, except setup, which is from version 5.1.2600.5512. The columns show the number of instructions (Instrs); the number of procedures (Procs); the number of basic blocks (BBs); the number of branch instructions (Branches); and the number of Δ_0, Δ_1, and Δ_2 rules in the WPDS encoding (WPDS Rules).

2008; 2013], and (ii) the relative precision of the abstract transformers obtained by the two methods.

This corpus was created using the ISAL instruction-decoder generator [Lim and Reps 2013, §2.1] in a mode in which the input specification of the concrete syntax of the x86 instruction set was used to create a randomized instruction *generator*—instead of the standard mode in which ISAL creates an instruction *recognizer*. By this means, we are assured that the corpus has substantial coverage of the syntactic features of the x86 instruction set (including opcodes, addressing modes, and prefixes, such as "lock", "rep", and "repne"); see Fig. 5.

Interprocedural-Analysis Experiment. We performed flow-sensitive, context-sensitive, interprocedural affine-relation analysis on the executables of nine Windows utilities, using four different sets of abstract transformers:

(1) MOS transformers for basic blocks, created by performing operator-by-operator MOS-reinterpretation.

(2) KS transformers for basic blocks, created by performing operator-by-operator KS-reinterpretation.

(3) KS transformers for basic blocks, created by symbolic abstraction of quantifier-free bit-vector (QFBV) formulas that capture the precise bit-level semantics of register-access/update operations in the different basic blocks. We denote this symbolic-abstraction method by $\widehat{\alpha}_{\mathrm{KS}}$.

(4) KS transformers for basic blocks, created by symbolic abstraction of quantifier-free bit-vector (QFBV) formulas that, in addition to register-access/update operations, also capture the precise bit-level semantics of all *memory-access/update* and *flag-access/update* operations. We denote this symbolic-abstraction method by $\widehat{\alpha}^{+}_{\mathrm{KS}}$.

For these programs, the generated abstract transformers were used as "weights" in a weighted pushdown system (WPDS). WPDSs are a modern formalism for solving flow-sensitive, context-sensitive interprocedural dataflow-analysis problems

[Bouajjani et al. 2003; Reps et al. 2005]. The weight on each WPDS rule is the MOS/KS transformer for a basic block of the program, including a jump or branch to a successor block. The asymptotic cost of weight generation is linear in the size of the program: to generate the weights, each basic block in the program is visited once, and a weight is generated by the relevant method.

Fig. 6 lists several size parameters of the executables (number of instructions, procedures, basic blocks, branches, and number of WPDS rules). WPDS rules can be divided into three categories, called Δ_0, Δ_1, and Δ_2 rules [Bouajjani et al. 2003; Reps et al. 2005]. The number of Δ_1 rules corresponds roughly to the total number of edges in a program's intraprocedural control-flow graphs; the number of Δ_2 rules corresponds to the number of call sites in the program; the number of Δ_0 rules corresponds to the number of procedure-exit sites.

$\widehat{\alpha}^{+}_{\mathrm{KS}}$ has the potential to create more precise KS weights than $\widehat{\alpha}_{\mathrm{KS}}$ because $\widehat{\alpha}^{+}_{\mathrm{KS}}$ can account for transformations of register values that involve a sequence of memory-access/update and/or flag-access/update operations within a basic block $\mathcal{B}$. For example, suppose that $\mathcal{B}$ contains a store to memory of register `eax`'s value, and a subsequent load from memory of that value into `ebx`. Because $\widehat{\alpha}^{+}_{\mathrm{KS}}$ uses a formula that captures the two memory operations, it can find a weight that captures the transformation `ebx' = eax`. A second type of example involving a store to memory followed by a load from memory within a basic block involves a sequence of the form "`push` *constant*; ... `pop edi`", and thus represents the transformation `edi'` = *constant*. Such sequences occur in several of the programs listed in Fig. 6.

As illustrated in line 13 of Fig. 3, the top-level function that is reinterpreted in TSL is interpInstr, which is of type

$$\mathrm{interpInstr} : \mathrm{instruction} \times \mathrm{state} \to \mathrm{state}.$$

To use semantic reinterpretation to implement CreateAbsTrans for the KS domain, interpInstr is reinterpreted as a $\mathrm{KS}[V; V']$ transformer; that is, $\mathrm{interpInstr}_{\mathrm{KS}}$ has the type

$$\mathrm{interpInstr}_{\mathrm{KS}} : \mathrm{instruction} \times \mathrm{KS}[V; V'] \to \mathrm{KS}[V; V'].$$

Let Id denote the $\mathrm{KS}[V; V']$ identity relation, $\left[\begin{array}{cc|c} \vec{V} & \vec{V}' & 1 \\ I & -I & 0 \end{array}\right]$. To reinterpret an individual instruction ι, one invokes $\mathrm{interpInstr}_{\mathrm{KS}}(\iota, \mathrm{Id})$.

For a basic block $B = [\iota_1, \ldots, \iota_m]$, there are two approaches to performing $\mathrm{KS}[V; V']$ reinterpretation:

—*Composed reinterpretation*:

$$w^{B}_{\mathrm{KS}} \leftarrow \mathrm{interpInstr}_{\mathrm{KS}}(\iota_1, \mathrm{Id}) ; \mathrm{interpInstr}_{\mathrm{KS}}(\iota_2, \mathrm{Id}) ; \ldots ; \mathrm{interpInstr}_{\mathrm{KS}}(\iota_m, \mathrm{Id}).$$

—*Chained reinterpretation*:

$$w^{B}_{\mathrm{KS}} \leftarrow \mathrm{interpInstr}_{\mathrm{KS}}(\iota_m, \ldots \mathrm{interpInstr}_{\mathrm{KS}}(\iota_2, \mathrm{interpInstr}_{\mathrm{KS}}(\iota_1, \mathrm{Id})) \ldots).$$

Our experiments use chained reinterpretation for two reasons:

(1) There are cases in which chained reinterpretation creates a more precise $\mathrm{KS}[V; V']$ element. For instance, consider the following code fragment, which

zeros the two low-order bytes of register `eax` and does a bitwise-or of `eax` into `ebx` (`ax` denotes the two low-order bytes of register `eax`):

$$\begin{array}{l} \iota_1 : \texttt{xor ax, ax} \\ \iota_2 : \texttt{or ebx, eax} \end{array}$$

The semantics of this code fragment can be expressed as follows:

$$\texttt{ebx'} = (\texttt{ebx} \mid (\texttt{eax} \,\&\, \texttt{xFFFF0000})) \wedge \texttt{eax'} = (\texttt{eax} \,\&\, \texttt{xFFFF0000}),$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively. $\text{interpInstr}_{\text{KS}}(\iota_1, \text{Id})$ creates the KS element $\left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right]$, which captures $(\texttt{ebx'} = \texttt{ebx}) \wedge (2^{16}\texttt{eax'} = 0)$. The two approaches to reinterpretation produce the following answers:

—*Composed reinterpretation*:

$$\begin{aligned} &\text{interpInstr}_{\text{KS}}(\iota_1, \text{Id}) \,;\, \text{interpInstr}_{\text{KS}}(\iota_2, \text{Id}) \\ &= \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] ; \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 1 & 0 & -1 & 0 & 0 \end{array}\right] \\ &= \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\ &= (2^{16}\texttt{eax}' = 0). \end{aligned}$$

—*Chained reinterpretation*:

$$\begin{aligned} &\text{interpInstr}_{\text{KS}}(\iota_2, \text{interpInstr}_{\text{KS}}(\iota_1, \text{Id})) \\ &= \text{interpInstr}_{\text{KS}}(\iota_2, \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right]) \\ &= \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 2^{16} & 0 & -2^{16} & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\ &= (2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx}) \wedge (2^{16}\texttt{eax}' = 0). \end{aligned}$$

In particular, the reinterpretation of "ι_2: `or ebx, eax`" takes place in a context in which the two low-order bytes of `eax` are partially constant ($2^{16}\texttt{eax} = 0$). Because of this additional piece of information, the reinterpretation technique described in §6.2.5.2 recovers the additional conjunct "$2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx}$".

(2) In the case of $\widehat{\alpha}_{\text{KS}}$, a formula φ_B is created that captures the concrete semantics of B (via symbolic execution), and then the KS weight for B is obtained by performing $w^B_{\text{KS}} \leftarrow \widehat{\alpha}_{\text{KS}}(\varphi_{\mathcal{B}})$. Letting QFBV denote the type of quantifier-free bit-vector formulas, the QFBV reinterpretation of interpInstr has the type

$$\text{interpInstr}_{\text{QFBV}} : \text{instruction} \times \text{QFBV} \to \text{QFBV}.$$

Symbolic execution is performed by chained reinterpretation:

$$\varphi_B \leftarrow \text{interpInstr}_{\text{QFBV}}(\iota_m, \ldots \text{interpInstr}_{\text{QFBV}}(\iota_2, \text{interpInstr}_{\text{QFBV}}(\iota_1, \text{Id})) \ldots).$$

Total instructions	MOS-reinterpretation time (seconds)	KS-reinterpretation time (seconds)
19,066	23.3	348.2

Fig. 7. Comparison of the performance of MOS-reinterpretation and KS-reinterpretation for x86 instructions.

Identical precision	MOS-reinterpretation more precise	KS-reinterpretation more precise	Total
18,158	0	908	19,066

Fig. 8. Comparison of the precision of MOS-reinterpretation and KS-reinterpretation for x86 instructions.

The $\widehat{\alpha}^{+}_{\mathrm{KS}}$ weight for B is created similarly, except that we also arrange for $\varphi_{\mathcal{B}}$ to encode all memory-access/update and flag-access/update operations.

For our experiments, we wanted to control for any precision improvements that might be due solely to the use of chained reinterpretation; thus, we use chained reinterpretation for all of the weight-generation methods.[6]

The interprocedural-analysis experiments used the WALi system [Kidd et al. 2007] for WPDSs. EWPDS merge functions [Lal et al. 2005] were used to preserve caller-save and callee-save registers across call sites. Running a WPDS-based analysis to find the join-over-all-paths value for a given set of program points involves calling two operations, "post*" and "path summary", as detailed in [Reps et al. 2005]. The post* queries used the FWPDS algorithm [Lal and Reps 2006].

Due to the high cost in §7.3 of constructing WPDS weights via $\widehat{\alpha}_{\mathrm{KS}}$ and $\widehat{\alpha}^{+}_{\mathrm{KS}}$, we ran all WPDS analyses without the code for libraries. Values are returned from x86 procedure calls in register `eax`, and thus in our experiments library functions were modeled approximately (albeit unsoundly, in general) by "`havoc(eax')`".

To implement $\widehat{\alpha}_{\mathrm{KS}}$ and $\widehat{\alpha}^{+}_{\mathrm{KS}}$, we used the Yices solver [Dutertre and de Moura 2006], version 1.0.19, with the timeout for each invocation set to three seconds.

We compared the precision of the one-vocabulary affine relations at branch points, as well as two-vocabulary affine relations at procedure exits, which can be used as procedure summaries.

7.2 Reinterpretation of Individual Instructions

Figs. 7 and 8 summarize the results of the per-instruction experiment. They answer questions (1) and (2) posed at the beginning of §7.

—KS-reinterpretation created an abstract transformer that was more precise than the one created by MOS-reinterpretation for about 4.76% of the instructions. MOS-reinterpretation never created an abstract transformer that was more precise than the one created by KS-reinterpretation.

—However, MOS-reinterpretation is much faster: to generate abstract transformers for the entire corpus of instructions, MOS-reinterpretation is about 14.9 times

[6]MOS-reinterpretation of a basic block is performed by chained reinterpretation, using $\mathrm{interpInstr}_{\mathrm{MOS}} : \mathrm{instruction} \times \mathrm{MOS} \to \mathrm{MOS}$.

faster than KS-reinterpretation.

Example 7.1. One instruction for which the abstract transformer created by KS-reinterpretation is more precise than the transformer created by MOS-reinterpretation is $\iota \stackrel{\text{def}}{=}$ "`add bh,al`". This instruction adds the value of `al`, the low-order byte of register `eax`, to the value of `bh`, the second-to-lowest byte of register `ebx`, and stores the result in `bh`. The semantics of this instruction can be expressed as a QFBV formula as follows:

$$\varphi_\iota \stackrel{\text{def}}{=} \texttt{ebx'} = \begin{pmatrix} (\texttt{ebx} \;\&\; \texttt{0xFFFF00FF}) \\ |\; ((\texttt{ebx} + 256 * (\texttt{eax} \;\&\; \texttt{0xFF})) \;\&\; \texttt{0xFF00}) \end{pmatrix} \wedge (\texttt{eax'} = \texttt{eax}). \quad (10)$$

Eqn. (10) shows that the semantics of the instruction involves non-linear bit-masking operations.

The abstract transformer created via MOS-reinterpretation corresponds to `havoc(ebx')`; all other registers are unchanged. That is, if we only had the three registers `eax`, `ebx`, and `ecx`, the abstract transformer created via MOS-reinterpretation would be

$$\left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\},$$

which captures the affine transformation "$(\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx})$". In contrast, the transformer created via KS-reinterpretation is

$$\begin{array}{c} \texttt{eax} \;\; \texttt{ebx} \;\; \texttt{ecx} \;\; \texttt{eax}' \;\; \texttt{ebx}' \;\; \texttt{ecx}' \;\; 1 \\ \left[\begin{array}{cccccc|c} 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2^{24} & 0 & 0 & -2^{24} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right], \end{array}$$

which corresponds to "$(\texttt{eax}' = \texttt{eax}) \wedge (2^{24}\texttt{ebx}' = 2^{24}\texttt{ebx}) \wedge (\texttt{ecx}' = \texttt{ecx})$". Both transformers are over-approximations of the instruction's semantics, but the extra conjunct $(2^{24}\texttt{ebx}' = 2^{24}\texttt{ebx})$ in the KS element captures the fact that the low-order byte of `ebx` is not changed by executing "`add bh,al`".

In contrast, $\widehat{\alpha}_{\text{KS}}(\varphi_\iota)$, the most-precise over-approximation of φ_ι that can be expressed as a KS element is (the Howellization of)

$$\begin{array}{c} \texttt{eax} \;\; \texttt{ebx} \;\; \texttt{ecx} \;\; \texttt{eax}' \;\; \texttt{ebx}' \;\; \texttt{ecx}' \;\; 1 \\ \left[\begin{array}{cccccc|c} 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 2^{24} & 2^{16} & 0 & 0 & -2^{16} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right], \end{array}$$

which corresponds to "$(\texttt{eax'} = \texttt{eax}) \wedge (2^{16}\texttt{ebx'} = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{ecx'} = \texttt{ecx})$". Multiplying by a power of 2 serves to shift bits to the left; because it is performed in arithmetic mod 2^{32}, bits shifted off the left end are unconstrained. Thus, the second conjunct captures the relationship between the low-order two bytes of `ebx'`, the low-order two bytes of `ebx`, and the low-order byte of `eax`. □

Prog. name	Performance of Interprocedural Analysis													
	MOS-reinterp			KS-reinterp			$\widehat{\alpha}_{\mathrm{KS}}$				$\widehat{\alpha}^{+}_{\mathrm{KS}}$			
	WPDS	post*	query at branch points	WPDS	post*	query at branch points	WPDS	post*	query at branch points	t/o	WPDS	post*	query at branch points	t/o
write	0.088	0.073	0.091	0.58	0.063	0.257	59.167	0.067	2.681	2(1.17%)	97.732	0.066	6.047	2(1.17%)
finger	0.25	0.889	0.275	1.405	0.192	0.366	80.294	0.208	3.61	3(0.78%)	272.939	0.178	6.217	7(1.82%)
subst	0.38	0.895	0.257	2.045	0.412	0.535	164.033	0.427	5.266	2(0.29%)	261.865	0.433	8.775	4(0.57%)
chkdsk	0.472	0.187	0.314	2.577	0.261	0.831	359.949	0.267	61.907	8(0.81%)	349.519	0.275	26.828	11(1.11%)
convert	0.672	1.643	0.7	3.481	0.647	1.091	206.25	0.629	22.409	17(1.43%)	317.142	0.627	30.959	19(1.60%)
route	1.144	3.581	1.413	6.393	0.82	2.073	396.732	0.959	87.864	4(0.32%)	704.816	0.902	75.348	28(2.27%)
comp	0.8	2.916	1.112	4.55	0.738	1.606	599.272	0.835	41.013	6(0.39%)	912.319	0.806	54.349	9(0.59%)
logoff	1.111	4.858	1.936	7.843	1.01	2.908	449.851	1.061	57.276	24(1.58%)	915.493	1.033	141.981	32(2.10%)
setup	2.054	3.259	1.907	13.598	0.521	5.731	1191.37	0.591	277.8	58(2.28%)	1852.79	0.599	464.781	93(3.65%)

Fig. 9. Performance of WPDS-based interprocedural analysis. The times, in seconds, for WPDS construction, performing interprocedural dataflow analysis (i.e., running post* and performing path-summary) and finding one-vocabulary affine relations at branch instructions, using MOS-reinterpretation, KS-reinterpretation, $\widehat{\alpha}_{\mathrm{KS}}$, and $\widehat{\alpha}^{+}_{\mathrm{KS}}$ to generate weights. The columns labeled "t/o" report the number of WPDS rules for which weight generation timed out during symbolic abstraction.

7.3 Interprocedural Analysis

Fig. 9 shows the times for WPDS construction (including constructing the weights that serve as abstract transformers) and performing interprocedural dataflow analysis by performing post* and path summary. Columns 11 and 15 of Fig. 9 show the number of $\widehat{\alpha}$ calls for which weight generation timed out during $\widehat{\alpha}_{\mathrm{KS}}$ and $\widehat{\alpha}^{+}_{\mathrm{KS}}$, respectively. During WPDS construction, if Yices times out during $\widehat{\alpha}_{\mathrm{KS}}$ or $\widehat{\alpha}^{+}_{\mathrm{KS}}$, the implementation uses a weight that is less precise than the best transformer, but it always uses a weight that is at least as precise as the weight obtained using KS-reinterpretation.[7] The number of WPDS rules is given in Fig. 6; a timeout occurred for about 1.0% of the $\widehat{\alpha}_{\mathrm{KS}}$ calls (computed as a geometric mean[8]), and for about 1.65% of the $\widehat{\alpha}^{+}_{\mathrm{KS}}$ calls.

The experiment showed that the cost of constructing weights via $\widehat{\alpha}_{\mathrm{KS}}$ is high, which was to be expected because $\widehat{\alpha}_{\mathrm{KS}}$ repeatedly calls an SMT solver. Creating KS weights via $\widehat{\alpha}_{\mathrm{KS}}$ is about 81.5 times slower than creating them via KS-reinterpretation (computed as the geometric mean of the construction-time ratios).

Moreover, creating KS weights via KS-reinterpretation is itself 5.9 times slower than creating MOS weights using MOS-reinterpretation. The latter number is different from the 14.9-fold slowdown reported in §7.2 for two reasons: (i) §7.2 reported the cost of creating KS and MOS abstract transformers for individual

---

[7] This footnote explains more precisely how weights were constructed in the $\widehat{\alpha}_{\mathrm{KS}}$ runs. We used the following "chained" method for generating weights:

(1) KS-reinterpretation is the method of §6.

(2) "Stålmarck" is the generalized-Stålmarck algorithm of Thakur and Reps [2012], *starting with the element obtained via KS-reinterpretation.* The generalized-Stålmarck algorithm successively over-approximates the best transformer from above. By starting the algorithm with the element obtained via KS-reinterpretation, the generalized-Stålmarck algorithm does not have to work its way down from $\top$; it merely continues to work its way down from the over-approximation already obtained via KS-reinterpretation.

(3) $\widehat{\alpha}_{\mathrm{KS}}$ is actually $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$, from Fig. 1(b), which maintains both an under-approximation and a (nontrivial) over-approximation of the desired answer, and hence is resilient to timeouts—i.e., it returns the over-approximation if a timeout occurs. In the chained method for generating weights, $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ is *started with the element obtained via the Stålmarck method as an over-approximation* as a way to accelerate its performance.

The generalized-Stålmarck algorithm is a faster algorithm than $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$, but is not guaranteed to find the best abstract transformer [Thakur and Reps 2012]. $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ *is* guaranteed to obtain the best abstract transformer, except for cases in which an SMT solver timeout is reported. The use of KS-reinterpretation accelerates Stålmarck, and the use of Stålmarck accelerates $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$. Moreover, $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}} \sqsubseteq$ KS-reinterpretation is always guaranteed to hold for the weights that are computed.

[8] We use "computed as a geometric mean" as a shorthand for "computed by converting the data to ratios; finding the geometric mean of the ratios; and converting the result back to a percentage". For instance, suppose that you have improvements of 3%, 17%, 29% (i.e., .03, .17, and .29). The geometric mean of the values .03, .17, and .29 is .113. Instead, we express the original improvements as ratios and take the geometric mean of 1.03, 1.17, and 1.29, obtaining 1.158. We subtract 1, convert to a percentage, and report "15.8% improvement (computed as a geometric mean)".

The advantage of this approach is that it handles datasets that include one or more instances of 0% improvement, as well as negative percentage improvements.

Prog. name	Δ_1 rules Rules	WPDS Weights MOS-reinterp < KS-reinterp	KS-reinterp < MOS-reinterp	$\widehat{\alpha}_{\mathrm{KS}}$ < KS-reinterp	$\widehat{\alpha}^{+}_{\mathrm{KS}} < \widehat{\alpha}_{\mathrm{KS}}$
write	151	0(0.00%)	0(0.00%)	11(7.28%)	0(0.00%)
finger	353	0(0.00%)	0(0.00%)	29(8.22%)	1(0.28%)
subst	728	0(0.00%)	0(0.00%)	59(8.10%)	0(0.00%)
chkdsk	887	0(0.00%)	0(0.00%)	86(9.70%)	1(0.11%)
convert	1266	0(0.00%)	2(0.16%)	131(10.35%)	0(0.00%)
route	1368	0(0.00%)	3(0.22%)	142(10.38%)	0(0.00%)
comp	1528	0(0.00%)	1(0.07%)	163(10.67%)	0(0.00%)
logoff	1648	0(0.00%)	4(0.24%)	191(11.59%)	1(0.06%)
setup	2847	0(0.00%)	20(0.70%)	432(15.17%)	8(0.28%)

Fig. 10. Comparison of the precision of the WPDS weights computed using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (E.g., KS-reinterp < MOS-reinterp reports the number of rules for which the KS-reinterp weight was more precise than the MOS-reinterp weight.)

instructions, whereas in Fig. 9 the transformers are for basic blocks, and (ii) the WPDS construction times in Fig. 9 include the cost of creating merge functions for use at procedure-exit sites, which was about the same for KS-reinterpretation and MOS-reinterpretation.

A comparison of the $\widehat{\alpha}_{\mathrm{KS}}$ columns of Fig. 9 against the $\widehat{\alpha}^{+}_{\mathrm{KS}}$ columns reveals that

—Creating KS weights via $\widehat{\alpha}^{+}_{\mathrm{KS}}$ is about 1.7 times slower than creating weights via $\widehat{\alpha}_{\mathrm{KS}}$ (computed as the geometric mean of the construction-time ratios). The slowdown occurs because the formula created for use by $\widehat{\alpha}^{+}_{\mathrm{KS}}$ is more complicated than the one created for use by $\widehat{\alpha}_{\mathrm{KS}}$: the former contains additional conjuncts that capture the effects of memory-access/update and flag-access/update operations.

—The times for performing "post*" and "path summary" are almost the same for both methods, because these phases do not involve any calls to the respective $\widehat{\alpha}$ procedures.

—Answering queries at branch points was 1.4 times slower for $\widehat{\alpha}^{+}_{\mathrm{KS}}$ compared to $\widehat{\alpha}_{\mathrm{KS}}$. The reason for the slowdown is that this phase must call the respective $\widehat{\alpha}$ procedures once for each branch point: "post*" and "path summary" return the weight that holds at the *beginning* of a basic block $B = [\iota_1, \ldots, \iota_m]$. To obtain the one-vocabulary affine relation that hold just before branch point ι_m at the end of B, we need to perform an additional $\widehat{\alpha}$ computation for $[\iota_1, \ldots, \iota_{m-1}]$ (i.e., for B, but *without* the branch instruction at the end of B).

Figs. 10, 11, and 12 present three studies that compare the precision obtained via MOS-reinterpretation, KS-reinterpretation, $\widehat{\alpha}_{\mathrm{KS}}$, and $\widehat{\alpha}^{+}_{\mathrm{KS}}$.

Fig. 10 compares the precision of the WPDS weights computed by the different methods for each of the example programs. It shows that $\widehat{\alpha}_{\mathrm{KS}}$ creates strictly more precise weights than KS-reinterpretation for about 10.14% of the WPDS rules (computed as a geometric mean). The "$\widehat{\alpha}_{\mathrm{KS}}$ < KS-reinterp" column of Fig. 10 is particularly interesting in light of the fact that a study of relative precision of abstract transformers created for *individual* instructions via KS-reinterpretation and $\widehat{\alpha}_{\mathrm{KS}}$ [Lim and Reps 2013, §5.4.1], reported that $\widehat{\alpha}_{\mathrm{KS}}$ creates strictly more precise transformers than KS-reinterpretation for only about 3.2% of the instructions that occur in the corpus of 19,066 instructions from §7.2. The numbers in Fig. 10

Prog. name	Branches	1-Vocabulary Affine Relations at Branch Points			
		MOS-reinterp < KS-reinterp	KS-reinterp < MOS-reinterp	$\widehat{\alpha}_{\mathrm{KS}}$ < KS-reinterp	$\widehat{\alpha}^{+}_{\mathrm{KS}} < \widehat{\alpha}_{\mathrm{KS}}$
write	26	0(0.00%)	0(0.00%)	4(15.38%)	0(0.00%)
finger	48	0(0.00%)	0(0.00%)	14(29.17%)	32(66.67%)
subst	74	1(1.35%)	0(0.00%)	15(20.27%)	0(0.00%)
chkdsk	119	0(0.00%)	0(0.00%)	13(10.92%)	0(0.00%)
convert	161	1(0.62%)	0(0.00%)	49(30.43%)	0(0.00%)
route	243	0(0.00%)	4(1.65%)	63(25.93%)	0(0.00%)
comp	224	0(0.00%)	0(0.00%)	7(3.12%)	0(0.00%)
logoff	306	0(0.00%)	0(0.00%)	91(29.74%)	20(6.54%)
setup	589	0(0.00%)	0(0.00%)	39(6.62%)	0(0.00%)

Fig. 11. Comparison of the precision of the one-vocabulary affine relations identified to hold at branch points via interprocedural analysis, using weights created using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (E.g., KS-reinterp < MOS-reinterp reports the number of branch points at which the KS-reinterp results were more precise than the MOS-reinterp results.)

Prog. name	Procs	2-Vocabulary Procedure Summaries			
		MOS-reinterp < KS-reinterp	KS-reinterp < MOS-reinterp	$\widehat{\alpha}_{\mathrm{KS}}$ < KS-reinterp	$\widehat{\alpha}^{+}_{\mathrm{KS}} < \widehat{\alpha}_{\mathrm{KS}}$
write	10	0(0.00%)	0(0.00%)	5(50.00%)	0(0.00%)
finger	18	0(0.00%)	0(0.00%)	10(55.56%)	2(11.11%)
subst	16	0(0.00%)	0(0.00%)	6(37.50%)	0(0.00%)
chkdsk	18	0(0.00%)	0(0.00%)	9(50.00%)	0(0.00%)
convert	38	0(0.00%)	0(0.00%)	8(21.05%)	0(0.00%)
route	40	0(0.00%)	0(0.00%)	18(45.00%)	0(0.00%)
comp	35	1(2.86%)	0(0.00%)	13(37.14%)	0(0.00%)
logoff	46	0(0.00%)	0(0.00%)	14(30.43%)	1(2.17%)
setup	67	0(0.00%)	1(1.49%)	40(59.70%)	0(0.00%)

Fig. 12. Comparison of the precision of the two-vocabulary affine relations identified to hold at procedure-exit points via interprocedural analysis, using weights created using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (E.g., KS-reinterp < MOS-reinterp reports the number of procedure-exit points at which the KS-reinterp results were more precise than the MOS-reinterp results.)

differ from that study in two ways: (i) Fig. 10 compares the precision of abstract transformers for basic blocks rather than for individual instructions; and (ii) Fig. 10 is a comparison for the instructions that appear in specific programs, whereas the corpus of 19,066 instructions used in the per-instruction study from [Lim and Reps 2013, §5.4.1] was created using a randomized instruction generator.

$\widehat{\alpha}^{+}_{\mathrm{KS}}$ creates strictly more precise weights than $\widehat{\alpha}_{\mathrm{KS}}$ for only about 0.1% of the WPDS rules (computed as a geometric mean). Improvements are obtained in only four of the nine programs (`finger`, `chkdsk`, `logoff`, and `setup`).

Figs. 11 and 12 answer question (3) posed at the beginning of this section:

> For what percentage of program points does $\widehat{\alpha}_{\mathrm{KS}}$ produce more precise answers than KS-reinterpretation and MOS-reinterpretation?

Figs. 11 and 12 summarize the results obtained from comparing the precision of the

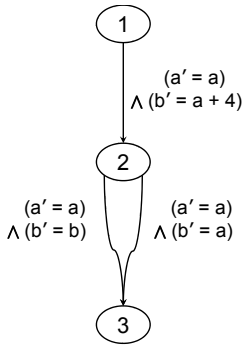


Fig. 13. Simplified version of an example that caused KS results to be less precise than MOS results, due to compose not distributing over join in the KS domain.

affine relations identified via interprocedural analysis using the different weights.[9]

Compared to runs based on either KS-reinterpretation or MOS-reinterpretation, the analysis runs based on $\widehat{\alpha}_{\mathrm{KS}}$ weights identified more precise affine relations at a substantial number of points (for both one-vocabulary affine relations that hold at branch points—Fig. 11, col. 5—and two-vocabulary affine relations that hold at procedure-exit points—Fig. 12, col. 5). For one-vocabulary affine relations, the $\widehat{\alpha}_{\mathrm{KS}}$ analysis results are strictly better than the KS-reinterpretation results at 18.6% of all branch points (computed as a geometric mean). For two-vocabulary affine relations describing procedure summaries, the $\widehat{\alpha}_{\mathrm{KS}}$ analysis results are strictly better than the KS-reinterpretation results at 42% of all procedures (computed as a geometric mean).

For one-vocabulary affine relations, the $\widehat{\alpha}^{+}_{\mathrm{KS}}$ analysis results are strictly better than the $\widehat{\alpha}_{\mathrm{KS}}$ results at 7.3% of all branch points (computed as a geometric mean). For two-vocabulary affine relations describing procedure summaries, the $\widehat{\alpha}^{+}_{\mathrm{KS}}$ analysis results are strictly better than the $\widehat{\alpha}_{\mathrm{KS}}$ results at 1.4% of all procedures (computed as a geometric mean). However, in both cases improvements are obtained in only two of the nine programs (`finger` and `logoff`).

7.4 Imprecision Due to Non-Distributivity of KS

Fig. 11 shows that in a couple of cases, one in `subst` and the other in `convert`, the MOS-reinterpretation results were better than the KS-reinterpretation results. (Although not shown in Fig. 11, the MOS-reinterpretation results were also better than the $\widehat{\alpha}_{\mathrm{KS}}$ results in the two cases.) We examined these cases, and found that they were an artifact of (i) the evaluation order chosen, and (ii) compose failing to distribute over join in the KS domain (see §5.4).

Fig. 13 is a simplified version of the actual transformers in `subst` and `convert` that caused the KS-based analyses to return a less-precise element than the MOS-

[9]Register `eip` is the x86 instruction pointer. There are some situations that cause the MOS-reinterpretation weights and KS-reinterpretation weights to fail to capture the value of the post-state `eip` value. Therefore, before comparing affine relations, we performed `havoc(eip')`. This adjustment avoids biasing the results merely because of trivial affine relations of the form "`eip'` = *constant*".

based analysis. In particular, if the join of the transformers on the two edges from node 2 to node 3 is performed before the composition of the individual $2 \to 3$ transformers with the $1 \to 2$ transformer, the combined $2 \to 3$ KS transformer is "$a' = a$" (i.e., b and b' are unconstrained). The loss of information about b and b' cascades to the $1 \to 3$ KS transformer, which is also "$a' = a$". In particular, it fails to contain the conjunct $2^{30}\mathtt{b}' = 2^{30}\mathtt{a}$, which expresses that the two low-order bits of $\mathtt{b}'$ at node 3 are the same as the two low-order bits of $\mathtt{a}$ at node 1.

In contrast, in the MOS domain, the combined $2 \to 3$ transformer is the affine closure of the transformers "$a' = a \wedge b' = b$" and "$a' = a \wedge b' = a$", which avoids the complete loss of information about b and b', and hence the $1 \to 3$ MOS transformer is able to capture the relation "$a' = a \wedge 2^{30}\mathtt{b}' = 2^{30}\mathtt{a}$".

8. RELATED WORK

8.1 Abstract Domains for Affine-Relation Analysis

The original work on affine-relation analysis (ARA) was an intraprocedural ARA algorithm due to Karr [1976]. In Karr's work, a domain element represents a set of points that satisfy affine relations over variables that hold elements of a *field*. Karr's algorithms are based on linear algebra (i.e., vector spaces).

Müller-Olm and Seidl [2004] gave an algorithm for interprocedural ARA, again for vector spaces over a field. Later [2005a; 2007], they generalized their techniques to work for modular arithmetic: they introduced the MOS domain, in which an element represents an affine-closed set of affine transformers over variables that hold machine integers, and gave an algorithm for interprocedural ARA. The algorithms for operations of the MOS domain are based on an extension of linear algebra to modules over a ring.

The version of the KS domain presented in this paper was inspired by, but is somewhat different from, the techniques described in two papers by King and Søndergaard [2008], [2010]. Our goals and theirs are similar, namely, to be able to create abstract transformers automatically that are bit-precise, modulo the inherent limitation on precision that stems from having to work with affine-closed sets of values. Compared with their work, we avoid the use of bit-blasting, and work directly with representations of w-bit affine-closed sets. The greatly reduced number of variables that comes from working at word level opened up the possibility of applying our methods to much larger problems, and as discussed in §5 and §7, we were able to apply our methods to interprocedural program analysis.

As shown in §5.2, the algorithm for projection given by King and Søndergaard [2008, §3] does not always find answers that are as precise as the domain is capable of representing. One consequence is that their join algorithm does not always find the least upper bound of its two arguments. In this paper, these issues have been corrected by employing the Howell form of matrices to normalize KS elements (§2.1, §5.2, and §5.4; see also §8.2 below).

King and Søndergaard introduced another interesting technique that we did not explore, which is to use affine relations over m-bit numbers, for $m > 1$, to represent sets of 1-bit numbers. To make this approach sensible, their concretization function intersects the "natural" concretization, which yields an affine-closed set of tuples of m-bit numbers, with the set $\{\langle v_1, \ldots, v_k\rangle \mid v_1, \ldots, v_k \in \{0, 1\}\}$. In essence,

this approach restricts the concretization to tuples of 1-bit numbers [King and Søndergaard 2010, Defn. 2]. The advantage of the approach is that KS elements over $\mathbb{Z}^k_{2^m}$ can then represent sets of 1-bit numbers that can only be over-approximated using KS elements over $\mathbb{Z}^k_2$.

Example 8.1. Suppose that we have three variables $\{x_1, x_2, x_3\}$, and want to represent the set of assignments $\{\langle 001\rangle, \langle 010\rangle, \langle 100\rangle\}$. The best KS element over $\mathbb{Z}^3_2$ is

$$\begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & 1 \end{array} \\ \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \end{array}\right], \end{array} \tag{11}$$

which corresponds to the affine relation $x_1 + x_2 + x_3 + 1 = 0$. The set of satisfying assignments is $\{\langle 001\rangle, \langle 010\rangle, \langle 100\rangle, \langle 111\rangle\}$, which includes the extra tuple $\langle 111\rangle$.

Now consider what sets can be represented when $m = 2$, so that arithmetic is performed mod 4. In particular, instead of the matrix in Eqn. (11) we use the following KS element over $\mathbb{Z}^3_4$:

$$\begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & 1 \end{array} \\ \left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \end{array}\right], \end{array} \tag{12}$$

which corresponds to the affine relation $x_1 + x_2 + x_3 + 3 = 0$. The matrix in Eqn. (12) has sixteen satisfying assignments:

$$\begin{array}{cccccccc} \langle 001\rangle & \langle 010\rangle & \langle 100\rangle & \langle 122\rangle & \langle 212\rangle & \langle 221\rangle & \langle 032\rangle & \langle 023\rangle \\ \langle 203\rangle & \langle 230\rangle & \langle 302\rangle & \langle 320\rangle & \langle 113\rangle & \langle 131\rangle & \langle 311\rangle & \langle 333\rangle \end{array}$$

However, only three of the assignments are in the restricted concretization, namely, the desired set $\{\langle 001\rangle, \langle 010\rangle, \langle 100\rangle\}$. □

To represent sets of tuples of w-bit numbers, as considered in this paper, the analogous technique would use a y-bit KS domain, $y > w$, in a similar fashion. That is, the "natural" concretization would be intersected with the set $\{\langle v_1, \ldots, v_k\rangle \mid v_1, \ldots, v_k \in \{0, 1, \ldots, 2^w - 1\}\}$. We leave the exploration of these issues for possible future research.

Like some other relational domains, including polyhedra [Cousot and Halbwachs 1978; PPL ; Jeannet] and grids [Bagnara et al. 2006], KS/AG fits the dual-representation paradigm of having both a constraint representation (KS) and a generator representation (AG). MOS is based on a generator representation. Whereas many implementations of domains with a dual representation perform some operations in one representation and other operations in the other representation, converting between representations as necessary, one of the clever aspects of both MOS and KS is that they avoid the need to convert between representations.

Granger [1989] describes congruence lattices, where the lattice elements are cosets in the group $\mathbb{Z}^k$. The generator form for a congruence-lattice element is defined by a point and a basis. The basis is used to describe the coset. The corresponding constraint form for a domain element is a system of Diophantine linear congruence equations. Conversion from the "normalized representation" (generator form) to an equation system is done by an elimination algorithm. The reverse direction is carried out by solving a set of equations. Granger's normalized representation can be used as a domain for representing affine relations over machine integers.

However, Granger's approach does not have unique normalized representations, because a coset space can have multiple bases. As a result, his method for checking that two domain elements are equal is to check containment in both directions (i.e., to perform two containment checks). Moreover, checking containment is costly because a representation conversion is required: one has to compare the cosets, which involves converting one coset into equational form and then checking if the other coset (in generator form) satisfies the constraints of equational form. In contrast, for the KS domain, one can easily check containment using the KS meet and equality operations:

$$\gamma(X) \subseteq \gamma(Y) \quad \text{iff} \quad X \sqsubseteq Y \quad \text{iff} \quad X = (X \sqcap Y).$$

Similarly, containment checking in AG can be performed using the AG join and equality operations. Thus, in both KS and AG, the costly step of converting between generator form and constraint form (or vice versa) is avoided.

Gulwani and Necula introduced the technique of random interpretation (for vector spaces over a field), and applied it to identifying both intraprocedural [2003] and interprocedural [2005] 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.

8.2 Howell Form

In contrast with both the Müller-Olm/Seidl work and the King/Søndergaard work, our work takes advantage of the Howell form of matrices. Howell form can be used with each of the domains KS, AG, and MOS defined in §2. 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. In contrast, Müller-Olm and Seidl [2007, §2] and King and Søndergaard [2008, Fig. 1], [2010, Fig. 2] use "echelon form" (called "triangular form" by King and Søndergaard), which is not canonical.

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 [Müller-Olm and Seidl 2005b] of Warren's $O(\log w)$ algorithms [Warren 2003, §10-15] for computing the inverse of an odd element, and only knew of an $O(w)$ algorithm [Müller-Olm and Seidl 2005a, Lemma 1].

8.3 Symbolic Abstraction for Affine-Relation Analysis

King and Søndergaard [2008], [2010] defined the KS domain, and used it to create implementations of best KS transformers for the individual bits of a bit-blasted concrete semantics. 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.

Prior to our SAS 2011 paper [Elder et al.], it was not known how to perform $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ in a non-trivial fashion (other than defining $\widetilde{\alpha}_{\mathrm{MOS}}$ to be $\lambda f.\top$). The fact that King and Søndergaard [2010, Fig. 2] had been able to devise an algorithm for $\widehat{\alpha}_{\mathrm{KS}}$ caused us to look more closely at the relationship between MOS and KS. The results presented in §4.1 establish that MOS and KS are different, incomparable abstract domains. We were able to give sound interconversion methods (§4.2–§4.4), and thereby obtained a method for performing $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ (§4.5).

9. CONCLUSION

This paper has explored a variety of issues pertaining to the MOS and KS domains for affine-relation analysis over variables that hold machine integers. What is particularly interesting about these domains is that they are based on arithmetic performed modulo 2^w, for some bit width w, which allows them to track machine arithmetic exactly for linear transformations.

We showed that, in general, MOS and KS are incomparable abstract domains. That is, some relations are expressible in each domain that are not expressible in the other. The central difference is that MOS is a domain of sets of functions, while KS is a domain of relations. However, we gave sound methods to convert a KS element v_{KS} to an over-approximating MOS element v_{MOS}—i.e., $\gamma(v_{\mathrm{KS}}) \subseteq \gamma(v_{\mathrm{MOS}})$—and to convert an MOS element w_{MOS} to an over-approximating KS element w_{KS}—i.e., $\gamma(w_{\mathrm{MOS}}) \subseteq \gamma(w_{\mathrm{KS}})$.

The paper also contributes to a broader research objective of ours—namely, the development of techniques to raise the level of automation in abstract interpreters. §6 presents an approach for obtaining KS abstract transformers based on semantic reinterpretation—i.e., by a greedy, operator-by-operator approach. §5.8 discusses symbolic abstraction for the KS domain, which provides not only a more global approach to creating KS abstract transformers, but one that attains the fundamental limits on precision that abstract-interpretation theory establishes. §7 presents the results of experiments to evaluate the costs and benefits of these methods. The experiments showed that, compared with the semantic-reinterpretation approach, it is considerably more costly to obtain KS abstract transformers via symbolic abstraction ($\widehat{\alpha}_{\mathrm{KS}}$ and $\widehat{\alpha}^{+}_{\mathrm{KS}}$). However, the transformers obtained via symbolic abstraction, as well as the one-vocabulary affine relations discovered to hold at branch points and the two-vocabulary affine relations discovered as procedure summaries, are often more precise than the ones obtained via semantic reinterpretation. More precisely, 10.14% of the $\widehat{\alpha}_{\mathrm{KS}}$ transformers, 18.6% of the one-vocabulary affine relations discovered to hold at branch points using the $\widehat{\alpha}_{\mathrm{KS}}$ transformers, and 42% the two-vocabulary affine relations discovered as procedure summaries using the $\widehat{\alpha}_{\mathrm{KS}}$ transformers are more precise than their KS-reinterpretation counterparts. (All three numbers are computed as geometric means.)

Acknowledgments. We thank Evan Driscoll and Aditya Thakur for their comments on a draft of this paper.

REFERENCES

Bach, E. 1992. Linear algebra modulo n. Unpublished manuscript.

Bagnara, R., Dobson, K., Hill, P., Mundell, M., and Zaffanella, E. 2006. Grids: A domain for analyzing the distribution of numerical values. In *Int. Workshop on Logic Based Prog. Dev. and Transformation*.

Bouajjani, A., Esparza, J., and Touili, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *POPL*. 269–282.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear constraints among variables of a program. In *POPL*. 84–96.

Dutertre, B. and de Moura, L. 2006. Yices: An SMT solver. http://yices.csl.sri.com/.

Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T. 2011. Abstract domains of affine relations. In *SAS*.

Granger, P. 1989. Static analysis of arithmetic congruences. *Int. J. of Comp. Math.*.

Gulwani, S. and Necula, G. 2003. Discovering affine equalities using random interpretation. In *POPL*.

Gulwani, S. and Necula, G. 2005. Precise interprocedural analysis using random interpretation. In *POPL*.

Hafner, J. and McCurley, K. 1991. Asymptotically fast triangularization of matrices over rings. *SIAM J. Comput. 20,* 6.

Howell, J. 1986. Spans in the module $(\mathbb{Z}_m)^s$. *Linear and Multilinear Algebra 19*.

Jeannet, B. New Polka. www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html.

Jones, N. and Mycroft, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *POPL*. 296–306.

Karr, M. 1976. Affine relationship among variables of a program. *Acta Inf. 6*, 133–151.

Kidd, N., Lal, A., and Reps, T. 2007. WALi: The Weighted Automaton Library. www.cs.wisc.edu/wpis/wpds/download.php.

King, A. and Søndergaard, H. 2008. Inferring congruence equations with SAT. In *CAV*.

King, A. and Søndergaard, H. 2010. Automatic abstraction for congruences. In *VMCAI*.

Knoop, J. and Steffen, B. 1992. The interprocedural coincidence theorem. In *CC*.

Lal, A. and Reps, T. 2006. Improving pushdown system model checking. In *CAV*.

Lal, A., Reps, T., and Balakrishnan, G. 2005. Extended weighted pushdown systems. In *CAV*.

Lim, J. and Reps, T. 2008. A system for generating static analyzers for machine instructions. In *CC*.

Lim, J. and Reps, T. 2013. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *TOPLAS 35,* 1 (Apr.).

Malmkjær, K. 1993. Abstract interpretation of partial-evaluation algorithms. Ph.D. thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas.

Meyer, C. 2000. *Matrix Analysis and Applied Linear Algebra*. SIAM, Philadelphia, PA.

Müller-Olm, M. and Seidl, H. 2004. Precise interprocedural analysis through linear algebra. In *POPL*.

Müller-Olm, M. and Seidl, H. 2005a. Analysis of modular arithmetic. In *ESOP*.

Müller-Olm, M. and Seidl, H. 2005b. Personal communication.

Müller-Olm, M. and Seidl, H. 2007. Analysis of modular arithmetic. *TOPLAS 29,* 5.

Mycroft, A. and Jones, N. 1985. A relational framework for abstract interpretation. In *Programs as Data Objects*.

Nielson, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comp. Sci. 69*, 117–242.

PPL. PPL: The Parma polyhedra library. www.cs.unipr.it/ppl/.

Reps, T., Sagiv, M., and Yorsh, G. 2004. Symbolic implementation of the best transformer. In *VMCAI*.

Reps, T., Schwoon, S., Jha, S., and Melski, D. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP 58,* 1–2 (Oct.).

Sagiv, M., Reps, T., and Horwitz, S. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci. 167*, 131–170.

Schmidt, D. 1986. *Denotational Semantics*. Allyn and Bacon, Inc., Boston, MA.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

Storjohann, A. 2000. Algorithms for matrix canonical forms. Ph.D. thesis, ETH Zurich, Zurich, Switzerland. Diss. ETH No. 13922.

Thakur, A., Elder, M., and Reps, T. 2012. Bilateral algorithms for symbolic abstraction. In *SAS*.

Thakur, A. and Reps, T. 2012. A method for symbolic computation of abstract operations. In *CAV*.

Warren, Jr., H. 2003. *Hacker's Delight*. Addison-Wesley.

A. DUALIZATION

For any matrix M, it is a common lemma that $\left(M^{-1}\right)^t = (M^t)^{-1}$. Thus, the notation M^{-t} denotes $\left(M^{-1}\right)^t$.

Lemma A.1. Let D and T be square, diagonal matrices, where $D_{ii} = 2^{p_i}$ and $T_{ii} = 2^{w-p_i}$ for all i. Then, $\text{null}^t T = \text{row}\, D$ and $\text{null}^t D = \text{row}\, T$.

PROOF. Let z be any row vector. To see that $\text{null}^t T = \text{row}\, D$:

$$\begin{aligned} z \in \text{null}^t T &\iff Tz^t = 0 \iff \forall i\colon z_i 2^{w-p_i} = 0 \\ &\iff \forall i\colon 2^{p_i} | z_i \iff \exists v\colon \forall i\colon v_i 2^{p_i} = z_i \\ &\iff \exists v\colon vD = z \iff z \in \text{row}\, D. \end{aligned}$$

One can show that $\text{null}^t D = \text{row}\, T$ by essentially the same reasoning. □

Theorem 3.3 *For any matrix M, $\text{null}^t M = \text{row}\, M^\perp$ and $\text{row}\, M = \text{null}^t M^\perp$.* □

PROOF. Again, let L, D, and R be the diagonal decomposition of M (see Defn. 3.1, and construct T from D as in Lem. A.1. Recall that L is invertible. To see that $\text{row}\, M = \text{null}^t M^\perp$,

$$\begin{aligned} &\text{row}\, M = \text{row}\, LDR = \text{row}\, DR, \text{ so } x \in \text{row}\, DR \iff xR^{-1} \in \text{row}\, D \\ &\iff xR^{-1} \in \text{null}^t T \iff TR^{-t}x^t = 0 \iff x \in \text{null}^t TR^{-t}. \end{aligned}$$

We know that L^{-t} is also invertible, so

$$\text{null}^t TR^{-t} = \text{null}^t L^{-t}TR^{-t} = \text{null}^t M^\perp.$$

Thus, $\text{row}\, M = \text{null}^t M^\perp$. One can show that $\text{null}^t M = \text{row}\, M^\perp$ by essentially the same reasoning. □

B. DOMAIN CONVERSIONS

Thm. 4.1 states that the transformation from MOS to AG given in §4.2 is sound.

Theorem 4.1 *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 $G = \bigsqcup_{AG}\{G_B \mid B \in \mathcal{B}\}$. Then, $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$.* □

PROOF. First, recall that for any two AG elements E and F, $E \sqcup_{\text{AG}} F$ equals $\text{HOWELLIZE}\left(\left[\begin{smallmatrix} E \\ F \end{smallmatrix}\right]\right)$. Because HOWELLIZE does not change the row space of a matrix, $\gamma_{\text{AG}}\left(E \sqcup_{\text{AG}} F\right)$ equals $\gamma_{\text{AG}}\left(\left[\begin{smallmatrix} E \\ F \end{smallmatrix}\right]\right)$. By the definition of G, we know that $\gamma_{\text{AG}}(G) = \gamma_{\text{AG}}(\mathbb{G})$, where $\mathbb{G}$ is all of the matrices G_B stacked vertically. Therefore, to show that $\gamma_{\text{MOS}}(\mathcal{B}) \subseteq \gamma_{\text{AG}}(G)$, we show that $\gamma_{\text{MOS}}(\mathcal{B}) \subseteq \gamma_{\text{AG}}(\mathbb{G})$.

Suppose that $(x, x') \in \gamma_{\text{MOS}}(\mathcal{B})$. Then, for some vector v,

$$\left[\begin{array}{c|c} 1 & x \end{array}\right] \left(\sum_{B\in\mathcal{B}} v_B B\right) = \left[\begin{array}{c|c} 1 & x' \end{array}\right].$$

If we break this equation apart, we see that

$$\sum_{B\in\mathcal{B}} v_B = 1 \quad \text{and} \quad \sum_{B\in\mathcal{B}} v_B c_B + x\left(\sum_{B\in\mathcal{B}} v_B M_B\right) = x'.$$

Let $\otimes$ denote Kronecker product. Now consider the following product, which uses $(v \otimes [\,1 \,|\, x\,])$ as a vector of coefficients for the rows of $\mathbb{G}$:

$$\begin{aligned}
\big(v \otimes [\,1 \,|\, x\,]\big)\,\mathbb{G} &= \sum_{B \in \mathcal{B}} [\,v_B \,|\, v_B x I \quad v_B c_B + v_B x M_B\,] \\
&= \left[\, \sum_{B \in \mathcal{B}} v_B \;\middle|\; x\left(\sum_{B \in \mathcal{B}} v_B\right) \quad \sum_{B \in \mathcal{B}} v_B c_B + x\left(\sum_{B \in \mathcal{B}} v_B M_B\right) \right] \\
&= [\,1 \,|\, x \;\; x'\,].
\end{aligned}$$

Thus, $[\,1 \,|\, x \;\; x'\,]$ is a linear combination of the rows of $\mathbb{G}$, and so $(x, x') \in \gamma_{\text{AG}}(\mathbb{G})$. Therefore, $\gamma_{\text{MOS}}(\mathcal{B}) \subseteq \gamma_{\text{AG}}(G)$. $\square$

Theorem 4.3 *When* $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, *then* $\gamma_{AG}(G) = \gamma_{MOS}(\textsc{Shatter}(G))$. $\square$

Proof.

$$\begin{aligned}
(x, x') \in \gamma_{\text{AG}}(G) &\iff \exists v\colon [\,1 \,|\, x \;\; v\,]\, G = [\,1 \,|\, x \;\; x'\,] \\
&\iff \exists v\colon b + xM + vR = x' \\
&\iff \exists v\colon [\,1 \,|\, x\,] \left(\left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + \sum_i v_i \left[\begin{array}{c|c} 0 & R_i \\ \hline 0 & 0 \end{array}\right] \right) = [\,1 \,|\, x'\,] \\
&\iff (x, x') \in \gamma_{\text{MOS}}(\textsc{Shatter}(G))
\end{aligned}$$

$\square$

Lemma B.1. Suppose that M and N are square matrices of equal dimension such that

(1) M has only ones and zeroes on its diagonal,
(2) if $M_{i,i} = 1$, then $M_{h,i} = 0$ for all $h \neq i$, and
(3) if $M_{i,i} = 0$, then $N_{i,h} = 0$ for all h.

Then, $MN = N$.

Proof. We know $(MN)_{i,j} = \sum_h M_{i,h} N_{h,j}$. By Items 2 and 3, if $h \neq i$ then either $M_{i,h} = 0$ or $N_{h,j} = 0$, so $(MN)_{i,j} = M_{i,i} N_{i,j}$. If $M_{i,i} = 0$, then by Item 2, $N_{i,j} = 0$; otherwise, $M_{i,i} = 1$. In either case, $(MN)_{i,j} = N_{i,j}$, as we require. $\square$

Lemma B.2. When $G = \left[\begin{array}{c|cc} 1 & a & b \\ \hline 0 & J & M \\ 0 & 0 & R \end{array}\right]$, such that $\left[\begin{array}{c|c} 1 & a \\ \hline 0 & J \end{array}\right]$ and $\left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$ satisfy the conditions of Lem. B.1, then $\gamma_{\text{AG}}(G) \subseteq \gamma_{\text{MOS}}(\textsc{Shatter}(G))$.

Proof.

$$\begin{aligned}
(x, x') \in \gamma_{\text{AG}}(G) &\implies \exists v, v'\colon [\,1 \,|\, v \;\; v'\,]\, G = [\,1 \,|\, x \;\; x'\,] \\
&\implies \exists v, v'\colon [\,1 \,|\, v\,] \left[\begin{array}{c|c} 1 & a \\ \hline 0 & J \end{array}\right] = [\,1 \,|\, x\,] \\
&\qquad\qquad \wedge [\,1 \,|\, v\,] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + v' [\,0 \,|\, R\,] = [\,1 \,|\, x'\,]
\end{aligned}$$

By Lem. B.1, $\left[\begin{array}{c|c} 1 & v \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] = \left[\begin{array}{c|c} 1 & v \end{array}\right] \left[\begin{array}{c|c} 1 & a \\ \hline 0 & J \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] = \left[\begin{array}{c|c} 1 & x \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$, so

$$\begin{aligned}(x, x') \in \gamma_{\text{AG}}(G) &\implies \exists v' \colon \left[\begin{array}{c|c} 1 & x \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + v' \left[\begin{array}{c|c} 0 & R \end{array}\right] = \left[\begin{array}{c|c} 1 & x' \end{array}\right] \\ &\implies \exists v' \colon \left[\begin{array}{c|c} 1 & x \end{array}\right] \left(\left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + \sum_i v'_i \left[\begin{array}{c|c} 0 & R_i \\ \hline 0 & 0 \end{array}\right] \right) = \left[\begin{array}{c|c} 1 & x' \end{array}\right] \\ &\implies (x, x') \in \gamma_{\text{MOS}}(\textsc{Shatter}(G))\end{aligned}$$

□

Theorem 4.5 *For $G \in AG$, $\gamma_{AG}(G) \subseteq \gamma_{MOS}(\textsc{Shatter}(\textsc{MakeExplicit}(G)))$.* s□

Proof. Without loss of generality, assume that G has $2k+1$ columns and is in Howell form.

MakeExplicit(G) consists of two loops. In the first loop, every row r with leading index $i \leq k+1$ for which the rank of the leading value is greater than 0 is generalized by creating from r a row s, which is added to G, such that s's leading index is also i, but its leading value is 1. Consequently, after the call on Howellize(G) in line 8 of MakeExplicit, the leading value of the row with leading index i is 1.

In the second loop, the matrix is expanded by all-zero rows so that any row with leading index $i \leq k+1$ is placed in row i.

Thus, for any AG element G, we can decompose MakeExplicit(G) into the matrix $\left[\begin{array}{c|cc} 1 & c & b \\ \hline 0 & J & M \\ 0 & 0 & R \end{array}\right]$, where $c, b \in \mathbb{Z}_{2^w}^{1\times k}$; $J, M \in \mathbb{Z}_{2^w}^{k\times k}$; and $R \in \mathbb{Z}_{2^w}^{r\times k}$ for some $r \leq k$. Moreover, we know that

—J is upper-triangular,
—J has only ones and zeroes on its diagonal,
—if $J_{j,j} = 1$, then column j of J is zero everywhere else, and
—if $J_{j,j} = 0$, then row j of J and row j of M are all zeroes.

By these properties, Lem. B.2 holds, so we know that

$$\gamma_{\text{AG}}(G) \subseteq \gamma_{\text{MOS}}(\textsc{Shatter}(\textsc{MakeExplicit}(G))).$$

□

C. HOWELL PROPERTIES

Definition C.1. Two module spaces R and S are **perpendicular** (denoted by $R \perp S$) if

(1) $r \in R \wedge s \in S \implies rs^t = 0$,
(2) $(\forall r \in R \colon rs^t = 0) \implies s \in S$, and
(3) $(\forall s \in S \colon rs^t = 0) \implies r \in R$.

□

Lemma C.2. If $R \perp S$ and $R \perp S'$, then $S = S'$.

Lemma C.3. For any matrix M, $\operatorname{row} M \perp \operatorname{null}^t M$.

These are standard facts in linear algebra; their standard proofs essentially carry over for module spaces.

Lemma C.4. If $R \perp R'$ and $S \perp S'$, then $R + S \perp R' \cap S'$.

PROOF. Pick G_R and G_S so that $\operatorname{row} G_R = R$ and $\operatorname{row} G_S = S$. Because the rows of a matrix are linear generators of its row space,

$$R + S = \operatorname{row} \left[\begin{smallmatrix} G_R \\ G_S \end{smallmatrix}\right], \qquad \text{so, by Lem. C.3,} \qquad R + S \perp \operatorname{null}^t \left[\begin{smallmatrix} G_R \\ G_S \end{smallmatrix}\right].$$

Because each row of a matrix acts as a constraint on its null space,

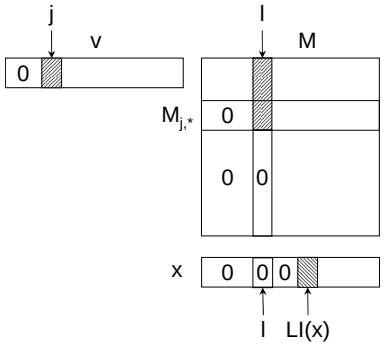
$$R + S \perp \operatorname{null}^t G_R \cap \operatorname{null}^t G_S.$$

By Lem. C.3 again, we know that $\operatorname{row} G_R \perp \operatorname{null}^t G_R = R \perp R'$, so $\operatorname{null}^t G_R = R'$ by Lem. C.2. Similarly, $\operatorname{null}^t G_S = S'$. Thus, $R + S \perp R' \cap S'$. □

Note. Recall from §2 that $[M]_i$ is the matrix that consists of all rows of M whose leading index is i or greater. For any row r, define $LI(r)$ to be the leading index of r. Define e_i to be the vector with 1 at index i and 0 everywhere else.

THEOREM C.5. *If matrix M is in Howell form, and $x \in \operatorname{row} M$, then $x \in \operatorname{row}([M]_{LI(x)})$.*

PROOF. Pick v so that $x = vM$, let $j \stackrel{\text{def}}{=} LI\,(v)$, and let $\ell \stackrel{\text{def}}{=} LI\,(M_{j,*})$. If $\ell \geq LI\,(x)$, then we already know that $x \in \operatorname{row}([M]_{LI(x)})$. Otherwise, assume $\ell < LI\,(x)$. Under these conditions, as depicted in the diagram below,



—$(vM)_\ell = 0$, because $LI\,(vM) = LI\,(x) > \ell$,
—$M_{h,\ell} = 0$ for any $h > j$, by Rule 1 of Defn. 2.1, and
—$v_h = 0$ for any $h < j$, because $j = LI\,(v)$.

Therefore, $0 = (vM)_\ell = \sum_h v_h M_{h,\ell} = v_j M_{j,\ell}$. Thus, because $j = LI\,(v)$, we know that $LI\,(v_j M_{j,*})$ is strictly greater than $\ell = LI\,(M_{j,*})$.

Because multiplication by invertible values can never change nonzero values to zero, we have $LI\,(v_j M_{j,*}) = LI\left(2^{\operatorname{rank}(v_j)} M_{j,*}\right)$. Thus, by Rule 4 of Defn. 2.1, we know that $v_j M_{j,*}$ can be stated as a linear combination of rows $j+1$ and greater. That is, $v_j M_{j,*} \in \operatorname{row}([M]_{j+1})$, or equivalently, $v_j M_{j,*} = uM$ with $LI\,(u) \geq j+1$. We can thus construct $v' = v - v_j e_j + u$ for which $x = v'M$ and $LI\,(v') \geq j+1$.

By employing this construction iteratively for increasing values of j, we can construct $x = yM$ with $LI\left(M_{LI(y),*}\right) \geq LI\left(x\right)$. Consequently, x can be stated as a linear combination of rows with leading indexes $LI\left(x\right)$ or greater; i.e., $x \in \text{row}([M]_{LI(x)})$. $\square$

Theorem 5.2 *Suppose that M has c columns. If matrix M is in Howell form, $x \in \text{null}^t\, M$ if and only if $\forall i\colon \forall y_1, \ldots y_{i-1}\colon \begin{bmatrix} y_1 & \cdots y_{i-1} & x_i & \cdots & x_c \end{bmatrix} \in \text{null}^t([M]_i)$.* $\square$

Proof. We know that $\text{row}\, M \perp \text{null}^t\, M$, and that $\text{row}([M]_i) \perp \text{null}^t([M]_i)$. Let E_i be the module space generated by $\{e_j \,|\, j < i\}$, and let F_i be the module space generated by $\{e_j \,|\, j \geq i\}$. Clearly, $E_i \perp F_i$. By Thm. C.5, we have that $\text{row}([M]_i) = \text{row}\, M \cap F_i$. Thus,

$$\text{null}^t([M]_i) \perp \text{row}\, M \cap F_i.$$

By Lem. C.4, we therefore have

$$\text{null}^t([M]_i) = \text{null}^t\, M + E_i, \tag{13}$$

Because $(\text{null}^t\, M + E_i)$ is the set $\left\{x + y \,\middle|\, x \in \text{null}^t\, M \wedge \forall h \geq i\colon y_h = 0\right\}$, Eqn. (13) is an equivalent way of stating the property to be shown. $\square$

D. CORRECTNESS OF KS JOIN

Theorem D.1. *If Y and Z are both $N+1$-column KS matrices, and $\gamma_{KS}(Y)$ and $\gamma_{KS}(Z)$ are both non-empty sets, then $Y \sqcup Z$ is the projection of $\left[\begin{smallmatrix} -Y & Y \\ Z & 0 \end{smallmatrix}\right]$ onto its right-most $N+1$ columns.*

Proof. $\gamma_{\text{KS}}\left(Y \sqcup Z\right)$ is the affine closure of $\gamma_{\text{KS}}(Y) \cup \gamma_{\text{KS}}(Z)$. Thus, we need to show that, for all $x \in \mathbb{Z}_{2^w}^N$,

$$\exists u \in \mathbb{Z}_{2^w}^N, \sigma \in \mathbb{Z}_{2^w}\colon \begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} u \\ \sigma \\ x \\ 1 \end{bmatrix} = 0$$

if and only if

x is an affine combination of values in $\gamma_{\text{KS}}(Y)$ and $\gamma_{\text{KS}}(Z)$.

Recall that an affine combination is a linear combination whose coefficients sum to 1.

Proof of the "if" direction: Fix a particular $x \in \mathbb{Z}_{2^w}^N$, and suppose that we have specific values for $\lambda \in \mathbb{Z}_{2^w}$, $y \in \gamma_{\text{KS}}(Y)$, and $z \in \gamma_{\text{KS}}(Z)$, such that $x = \lambda y + z(1-\lambda)$.

Pick $\sigma = 1 - \lambda$, and $u = (1 - \lambda)z$. Then,

$$\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} (1-\lambda)z \\ 1-\lambda \\ x \\ 1 \end{bmatrix} = 0$$

$$\text{if and only if } -Y \begin{bmatrix} (1-\lambda)z \\ 1-\lambda \end{bmatrix} + Y \begin{bmatrix} x \\ 1 \end{bmatrix} = 0 \text{ and } Z \begin{bmatrix} (1-\lambda)z \\ 1-\lambda \end{bmatrix} = 0$$

$$\text{if and only if } Y \begin{bmatrix} -(1-\lambda)z + x \\ \lambda \end{bmatrix} = 0 \text{ and } (1-\lambda)Z \begin{bmatrix} z \\ 1 \end{bmatrix} = 0$$

$$\text{if and only if } \lambda Y \begin{bmatrix} y \\ 1 \end{bmatrix} = 0 \text{ and } (1-\lambda)Z \begin{bmatrix} z \\ 1 \end{bmatrix} = 0.$$

These last equations are true because $y \in \gamma_{\text{KS}}(Y)$ and $z \in \gamma_{\text{KS}}(Z)$. Thus, if x is in the affine closure of $\gamma_{\text{KS}}(Y) \cup \gamma_{\text{KS}}(Z)$, then $\begin{bmatrix} x \\ 1 \end{bmatrix}$ is in the null space of the projected matrix.

Proof of the "only if" direction: Suppose that x is in the null space of the projected matrix. Pick $u \in \mathbb{Z}_{2^w}^N$ and $\sigma \in \mathbb{Z}_{2^w}$ such that

$$\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} u \\ \sigma \\ x \\ 1 \end{bmatrix} = 0.$$

We must show that x is in the affine closure of $\gamma_{\text{KS}}(Y) \cup \gamma_{\text{KS}}(Z)$.

Immediately, we know that $Z \begin{bmatrix} u \\ \sigma \end{bmatrix} = 0$ and $Y \begin{bmatrix} x-u \\ 1-\sigma \end{bmatrix} = 0$. Because $\gamma_{\text{KS}}(Y)$ and $\gamma_{\text{KS}}(Z)$ are nonempty, we can select an arbitrary $y_0 \in \gamma_{\text{KS}}(Y)$ and $z_0 \in \gamma_{\text{KS}}(Z)$. Thus,

$$0 = Y \begin{bmatrix} x-u \\ 1-\sigma \end{bmatrix} + \sigma Y \begin{bmatrix} y_0 \\ 1 \end{bmatrix} = Y \begin{bmatrix} x-u+\sigma y_0 \\ 1 \end{bmatrix}, \text{ and}$$

$$0 = Z \begin{bmatrix} u \\ \sigma \end{bmatrix} + (1-\sigma) Z \begin{bmatrix} z_0 \\ 1 \end{bmatrix} = Z \begin{bmatrix} u + (1-\sigma)z_0 \\ 1 \end{bmatrix}.$$

Now define y and z to be the values that we have just shown to be in $\gamma_{\text{KS}}(Y)$ and $\gamma_{\text{KS}}(Z)$:

$$y \stackrel{\text{def}}{=} x - u + \sigma y_0 \text{ and } z \stackrel{\text{def}}{=} u + (1-\sigma)z_0.$$

If we solve for x and eliminate u in these equations, we get:

$$x = y - \sigma y_0 + z + (\sigma - 1)z_0.$$

Because $y, y_0 \in \gamma_{\text{KS}}(Y)$, $z, z_0 \in \gamma_{\text{KS}}(Z)$, and $1 - \sigma + 1 + (\sigma - 1) = 1$, we have now stated x as an affine combination of values in $\gamma_{\text{KS}}(Y)$ and $\gamma_{\text{KS}}(Z)$, as required. □

E. JOIN WITH A KS ELEMENT THAT REPRESENTS A SINGLE STATE

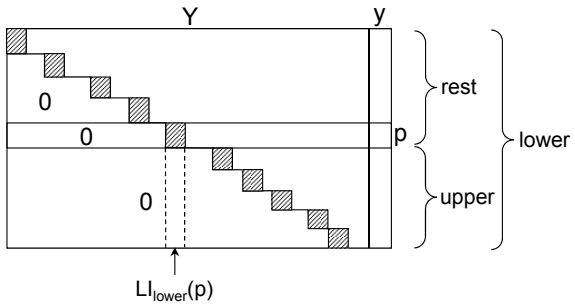
Lemma 5.8 *The assignment "lower ← lower ⊔ β(S)" on line 12 of Fig. 1(a) does not change upper.* □

PROOF. S is a state that satisfies $\varphi \wedge \neg\widehat{\gamma}(p)$. From Eqns. (6) and (7), we have $S = [\vec{X} \mapsto \vec{v}, \vec{X}' \mapsto \vec{v}']$ and

$$\beta(S) = \begin{array}{c} \begin{matrix} \vec{X} & \vec{X}' & 1 \end{matrix} \\ \left[\begin{array}{cc|c} I & 0 & (-\vec{v})^t \\ 0 & I & (-\vec{v}')^t \end{array}\right] \end{array}.$$

Because the distinction between one-vocabulary and two-vocabulary KS elements is unimportant for this proof, we will consider one-vocabulary KS elements with $k+1$ columns, abbreviating S as $[\vec{X} \mapsto \vec{v}]$ and $\beta(S)$ as $\begin{array}{c}\vec{X}\quad 1\\ [\,I\,|\,-v\,]\end{array}$.

Let $\mathit{lower} = [\,Y\,|\,y\,]$ and $\mathit{upper} = \mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})]$. We will refer to portions of *lower* by the names shown in the diagram below (where the shaded blocks represent the leading values of the different rows):



For instance, $\mathit{lower} = \left[\begin{array}{c|c} Y_{rest} & y_{rest} \\ Y_{upper} & y_{upper} \end{array}\right]$.

To perform $\mathit{lower} \sqcup \beta(S) = [\,Y\,|\,y\,] \sqcup [\,I\,|\,-v\,]$, we create the $2k+2$-column matrix

$$M \stackrel{\text{def}}{=} \left[\begin{array}{cccc} -Y & -y & Y & y \\ I & -v & 0 & 0 \end{array}\right]$$

and Howellize M. Because I is already in Howell form, we rearrange rows to form

$$\left[\begin{array}{cccc} I & -v & 0 & 0 \\ -Y & -y & Y & y \end{array}\right],$$

and then cancel the $-Y$ block by multiplying $[\,I\ -v\ 0\ 0\,]$ on the left by Y, and adding the result to $[\,-Y\ -y\ Y\ y\,]$, which produces

$$\left[\begin{array}{cccc} I & -v & 0 & 0 \\ 0 & (-Yv - y) & Y & y \end{array}\right]. \tag{14}$$

Note that the column $\left[\begin{array}{c} -v \\ (-Yv - y) \end{array}\right]$ is the only column that causes matrix (14) to fail to be in Howell form. We can factor this column as follows:

$$[\,(-Yv - y)\,] = [\,-Y\ -y\,] \left[\begin{array}{c} v \\ 1 \end{array}\right] = -\mathit{lower} \left[\begin{array}{c} v \\ 1 \end{array}\right] = -\left[\begin{array}{c|c} Y_{rest} & y_{rest} \\ Y_{upper} & y_{upper} \end{array}\right] \left[\begin{array}{c} v \\ 1 \end{array}\right].$$

By invariant (2) from §5.8.1, we have $\mathit{upper} \sqsupseteq \widehat{\alpha}(\varphi)$, and hence $[\![\varphi]\!] \subseteq [\![\widehat{\alpha}(\varphi)]\!] \subseteq \gamma(\mathit{upper})$. Moreover, $S \models \varphi \wedge \neg\widehat{\gamma}(p)$ means that $S \in [\![\varphi]\!]$, which implies that

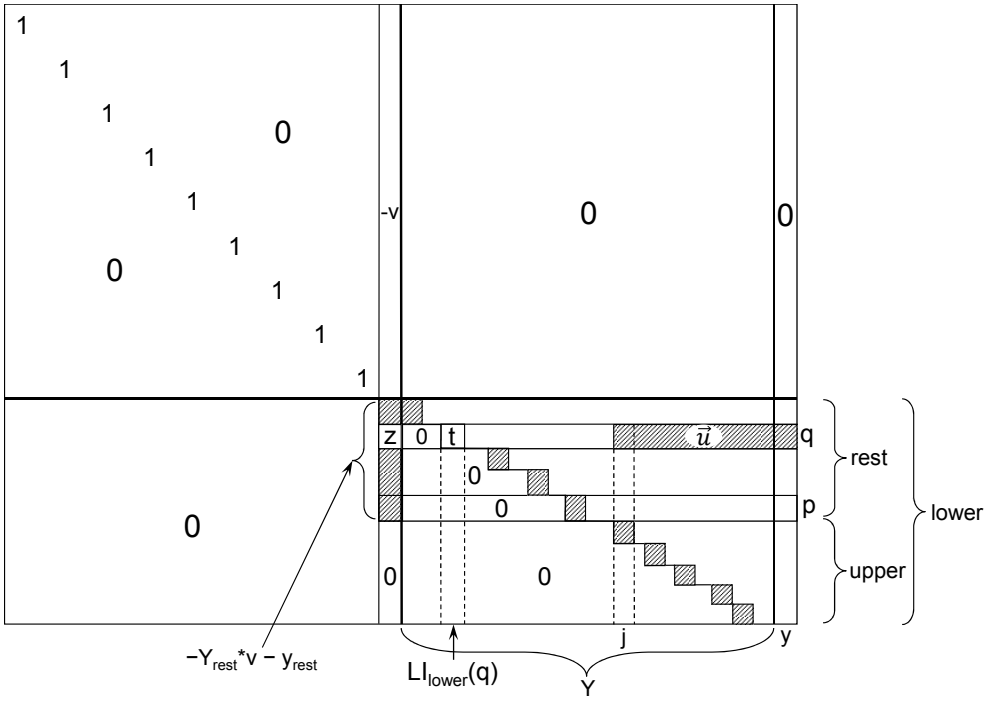


Fig. 14. Depiction of the structure of the partially Howellized matrix (15) that arises during the operation $lower \sqcup \beta(S)$.

$S \in \gamma(upper)$. The latter fact can be expressed as $S \models \widehat{\gamma}(upper)$, or in matrix terms, as $\left[\, Y_{upper} \,|\, y_{upper} \,\right] \begin{bmatrix} v \\ 1 \end{bmatrix} = 0$. Therefore, matrix (14) has the following structure

$$\begin{bmatrix} I & -v & 0 & 0 \\ 0 & (-Y_{rest} * v - y_{rest}) & Y_{rest} & y_{rest} \\ 0 & 0 & Y_{upper} & y_{upper} \end{bmatrix}, \tag{15}$$

which is depicted in more detail in Fig. 14.

We now want to argue that none of the remaining steps carried out to finish putting matrix (15) into Howell form can affect a row of *upper*. Obviously, none of the steps of HOWELLIZE that resemble Gaussian elimination—used to enforce items (1) and (2) of Defn. 2.1—can affect a row of *upper*, because all of the entries in column $k+1$ of matrix (15) for rows of *upper* are 0. Nor can *upper* be affected by the steps of HOWELLIZE that resemble back-substitution, which are used to enforce Defn. 2.1(3).

More surprisingly, the logical-consequence rows added to matrix (15) to enforce Defn. 2.1(4) cannot change *upper* either. For instance, consider a row such as row q in Fig. 14, which has the form $\left[\, 0 \;\ldots\; 0 \;\; z \;\; 0 \;\ldots\; 0 \;\; t \;\ldots\; \vec{u} \,\right]$. Let j be the leading index (with respect to matrix *lower*) of the first row of *upper*. Suppose that s is the smallest number such that when the portion of row q that is in *lower*, namely, $\left[\, 0 \;\ldots\; 0 \;\; t \;\ldots\; \vec{u} \,\right]$, is multiplied by 2^s, all entries in column positions $< j$ are 0, and

we are left with $\left[\begin{array}{cccc} 0 & \ldots & 0 & 2^s\vec{u} \end{array}\right]$. Because *lower* is in Howell form, by Defn. 2.1(4) row($[upper]_j$) includes constraints that are equal to or stronger than all multiples of $\left[\begin{array}{cccc} 0 & \ldots & 0 & 2^s\vec{u} \end{array}\right]$.

Now consider again the full row q of matrix (15), $\left[\begin{array}{ccccccccc} 0 & \ldots & 0 & z & 0 & \ldots & 0 & t & \ldots & \vec{u} \end{array}\right]$. For a logical consequence of row q to have non-zero entries only at positions $k+1+j$ or greater, we must multiply q by at least a power of 2 that is sufficient to zero out z *and* all elements at positions $k+1+LI_{lower}(q)$ to $k+1+j-1$. Consequently, the multiplicand must be a multiple of 2^s; however, in that case the result is a vector that is a multiple of $\left[\begin{array}{cccc} 0 & \ldots & 0 & 2^s\vec{u} \end{array}\right]$. As observed above, such a constraint cannot change *upper* because row($[upper]_j$) already includes constraints that are equal to or stronger than all multiples of $\left[\begin{array}{cccc} 0 & \ldots & 0 & 2^s\vec{u} \end{array}\right]$. □