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.; and T. Andersen is currently affiliated with Canonical.

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

An affine relation is a linear-equality constraint between numeric-valued variables of the form $\Sigma_{i=1}^{n} a_i x_i + b = 0$. For a given set of variables $\{x_i\}$, affine-relation analysis (ARA) identifies affine relations that are invariants of a program. For instance, in the program shown below $i - 2j - 5 = 0$ always holds at the head of the loop.

```
int main() {
   unsigned int i = 5;
   for (unsigned int j = 0; j < 100; j++) {
      i = i + 2;
   }
}
```

An *induction variable* is a variable that is incremented or decremented by a constant amount on every iteration of a loop. The results of ARA can be used to identify induction variables by establishing that an affine relation holds between a given variable and a loop-control variable. For instance, because `j` is the loop-control variable in the above example, and $i - 2j - 5 = 0$ holds at the head of the loop, `i` is an induction variable of the loop.

The results of ARA can also be used to determine a more precise abstract value for a variable via *semantic reduction* [Cousot and Cousot 1979]. For instance, interval analysis would reveal that the value of `j` at the loop head is always in the interval $[0, 100]$ (i.e., $\texttt{j} \mapsto [0, 100]$ or, equivalently, $0 \leq j \wedge j \leq 100$), but would not identify an upper bound on the value of `i` ($\texttt{i} \mapsto \top$). Semantic reduction of the interval invariant $\{\texttt{i} \mapsto \top, \texttt{j} \mapsto [0, 100]\}$ with respect to the affine relation $i - 2j - 5 = 0$ allows the known inequality fact $j \leq 100$ to be propagated to `i` to obtain $i \leq 205$.

ARA over integers, or $\text{ARA}_{\mathbb{Z}}$, was first studied by Karr [1976]. The versions of ARA in which we are interested are based on machine arithmetic, e.g., arithmetic modulo 2^8, 2^{16}, 2^{32}, or 2^{64}, and are able to take care of arithmetic overflow. In this paper, an element of an abstract domain of affine relations represents the set of points that satisfy affine relations over variables that hold machine integers.

Operations on values in $\text{ARA}_{\mathbb{Z}}$ are based on linear algebra. In contrast, operations on values in ARA_{2^w} (for some machine-integer width w, such as $w = 32$ or $w = 64$) are based on an extension of linear algebra to modules over a ring—in particular, arithmetic performed modulo 2^w. Compared with $\text{ARA}_{\mathbb{Z}}$, ARA_{2^w} has a number of advantages:

—*ARA_{2^w} matches the arithmetic used on actual machines.* Because arithmetic overflow is not accounted for in $\text{ARA}_{\mathbb{Z}}$, the results of an $\text{ARA}_{\mathbb{Z}}$ analysis can be unsound: the analysis may report that properties are invariants of the program when, in fact, they do not always hold.

—*ARA_{2^w} is more expressive than $ARA_{\mathbb{Z}}$.* In arithmetic mod 2^w, multiplying by a power of 2 serves to shift bits to the left, with the effect that bits shifted off the left end are unconstrained. For example, in $\text{ARA}_{2^{32}}$ the affine relation $2^{31}x = 2^{31}$ places a constraint solely on the least-significant bit of x; consequently, $2^{31}x = 2^{31}$ is satisfied by all states in which x is odd. Similarly, $2^{31}x = 0$ is satisfied by all states in which x is even. In contrast, with affine relations over integers it is not possible to express information about the parity of a variable.

This paper addresses a wide variety of issues that arise when performing ARA over machine arithmetic, and presents new results in each of the topics considered. The overall contribution of our work is a new abstract domain for ARA, for which we present

—algorithms for creating sound abstract transformers, which are needed to set up the ARA equation system that models a given program (§6 and §7),

—algorithms needed inside a program analyzer for solving a set of ARA equations (§5),

—results that show how our abstract domain relates to previous work (§4), and

—results of experiments with an implementation to evaluate the costs and benefits of our methods (§8).

Our work was inspired by techniques described in two papers by King and Søndergaard [2008], [2010]. Despite the potential for confusion between our work and that of King and Søndergaard, we have adopted the name "KS_{2^w}" (or simply "KS" when the value of w is understood) for our ARA domain, in homage to their work.

The version of the KS domain that King and Søndergaard used for program analysis [2010] is the same as our KS domain *restricted to Booleans* (i.e., affine relations over variables that hold 1-bit values: KS_{2^1}). The authors of the present paper improved upon the King/Søndergaard work in [Elder et al. 2011], which introduced a normal form for representing KS_{2^w} elements for arbitrary machine-integer width w, and used the normal form to give polynomial-time algorithms for "best" versions of join, meet, equality, and projection, as well as for finding best KS_{2^w} transformers (replacing the over-approximating methods that had been given by King and Søndergaard).

Because the abstraction of King and Søndergaard abstracts sets of points in $\mathbb{B}^n$, they model program variables by using, e.g., 32 or 64 Boolean variables, and express all operations via bit-blasting. 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. As discussed in §5 and §8, we are able to apply our methods to interprocedural program analysis.

The remainder of this section introduces the issues addressed in the individual sections of the paper.

Linear Algebra for Arithmetic Modulo 2^w (§2). To perform ARA for machine arithmetic, we must work in the ring $\mathbb{Z}_{2^w}$, rather than in a field, as in ARA for integers [Karr 1976; Müller-Olm and Seidl 2004]. Some aspects of standard linear algebra do not apply in $\mathbb{Z}_{2^w}$, and thus §2 discusses what can and cannot be carried over from standard linear algebra. In §2, we give the technical definitions of the abstract domains for ARA with which we work. In particular, we introduce a useful normal form (so-called "Howell form" for matrices). Howell form has not been used in past work on modular-arithmetic ARA, and turns out to simplify some of the issues that come up in later sections.

Comparing Domains for ARA (§3 and §4). Prior to our work, Müller-Olm and Seidl [2005a], [2007] and King and Søndergaard [2008], [2010] proposed two different

abstract domains for modular-arithmetic ARA (which we call the MOS domain and the KS domain, respectively). However, as mentioned above, the KS domain was only fully worked out for the Boolean case, KS_{2^1}. In addition, the relationship between the MOS and KS domains was unclear.

§3 and §4 considers these domains, along with several variants, and shows how they relate to each other. In particular, we show that MOS_{2^w} and KS_{2^w} are, in general, incomparable; however, we give sound interconversion methods. That is, we give an algorithm to convert an MOS_{2^w} element u_{MOS} to an over-approximating KS_{2^w} element u_{KS}—i.e., $\gamma_{\mathrm{MOS}}(u_{\mathrm{MOS}}) \subseteq \gamma_{\mathrm{KS}}(u_{\mathrm{KS}})$—(see §4.2), as well as an algorithm to convert a KS_{2^w} element v_{KS} to an over-approximating MOS_{2^w} element v_{MOS}—i.e., $\gamma_{\mathrm{KS}}(v_{\mathrm{KS}}) \subseteq \gamma_{\mathrm{MOS}}(v_{\mathrm{MOS}})$—(see §4.3 and §4.4).

Operations on KS Domain Elements (§5). In §5, we describe the details of KS_{2^w}. §5 covers two related issues:

—How to perform all of the standard abstract-domain operations—join, meet, assume, checking containment, etc.—needed for solving KS_{2^w} equations for an intraprocedural analysis problem (§5.1).

—How to perform the additional domain operations needed for solving KS_{2^w} equations for an interprocedural analysis problem (§5.2).

Creating Abstract Transformers in the KS Domain (§6 and §7). Past work on ARA has assumed that the analysis is to be applied to source-code programs, and that one was only interested in creating abstract transformers for statements that perform affine transformations—e.g., $x = 3 * x + 4 * y + 6$. More precisely, past work has been targeted at supporting a *modeling language* of affine programs, in which one can define non-deterministic flow graphs that can call each other recursively [Müller-Olm and Seidl 2004]. To model program actions, the modeling language provides a restricted language of (i) affine transformations, and (ii) non-deterministic assignments of the form "x =?," for which the semantics is that x may hold any possible value afterward. An analyzer would create an abstraction of a concrete program by modeling affine transformations exactly, but modeling read statements of the form "read(x)" and assignment statements of the form "$x = t$," where t is a non-affine expression, with "x =?."

The material presented in §5 is sufficient for creating a similar modeling language, based on the KS domain, for affine programs using modular arithmetic. However, there are situations in which using such a modeling language may be inconvenient. In particular, one might have a formal specification of the concrete semantics of the programming language of interest, e.g., as an operational semantics or an axiomatic semantics. Such a specification of the semantics is not likely to use only affine transformations; on the contrary, the specification is likely to include arithmetic, logical, and "bit-twiddling" operations (such as left-shift; arithmetic and logical right-shift; bitwise-and, bitwise-or, and bitwise-xor; etc.). To address this problem, §6 and §7 describe two techniques that can be used to create sound abstract transformers for the KS domain.

—§6 describes a greedy, operator-by-operator abstraction method for obtaining KS abstract transformers.

—§7 describes how a solver for satisfiability modulo a theory (SMT) can be used to create *best* KS abstract transformers—i.e., ones that attain the fundamental limits on precision that abstract-interpretation theory establishes.

Experiments (§8). §8 presents an experimental study with the Intel IA32 (x86) instruction set in which best KS abstract transformers 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.

Other Material. §9 discusses related work. §10 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 x [Müller-Olm and Seidl 2005a; 2007]. For example, $\text{rank}(1) = 0$, $\text{rank}(24) = 3$, and $\text{rank}(0) = w$. Every element $x \in \mathbb{Z}_{2^w}$ has a unique factoring of the form $u2^{\text{rank}(x)}$, where u is odd and $1 \leq u \leq 2^{w-\text{rank}(x)} - 1$ (e.g., $24 = 3 * 2^3$). Elements of $\mathbb{Z}_{2^w}$ are still ordered by less-than ($<$); i.e., $0 < 1 < 2 < \ldots < 2^w - 1$; however, because of arithmetic wrap-around, we no longer have the usual laws that connect $+$ with $<$, and $\times$ with $<$. As a shorthand, we sometimes write a value in $\mathbb{Z}_{2^w}$ as a negative number "$-m$" rather than as $2^w - m$ (e.g., "-1" really denotes $2^w - 1$).

Throughout the paper, k is the size of the *vocabulary* V—i.e., the variable-set under analysis. A *concrete state* is an assignment to the variables of V—i.e., a function in $V \to \mathbb{Z}_{2^w}$ (which is isomorphic to $\mathbb{Z}_{2^w}^k$). Assignment$[V]$ denotes the set of assignments over V.

A *two-vocabulary* relation $R[V; V']$ is a transition relation between assignments in the *pre-state* vocabulary V (i.e., Assignment$[V]$) and assignments in the *post-state* vocabulary V' (Assignment$[V']$). Thus, a transition relation $R[V; V']$ in the concrete collecting semantics is a subset of $\mathbb{Z}_{2^w}^k \times \mathbb{Z}_{2^w}^k$ (which is isomorphic to $\mathbb{Z}_{2^w}^{2k}$).

Matrix addition and multiplication are defined as usual, forming a matrix ring. We denote the transpose of a matrix M by M^t. A *one-vocabulary matrix* is a matrix with $k + 1$ columns. A *two-vocabulary matrix* is a matrix with $2k + 1$ columns. In each case, the "+1" is related to the fact that we capture affine rather than linear relations (however, the technical reasons for the extra column vary according to what kind of matrix we are dealing with). In conjunction with such matrices, a concrete state is represented by a row vector of length $k + 1$, where the entry in

the "extra" position is 1. 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 are numbered from 1 to $\texttt{columns}(M)$.

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 $\text{null}^t\, M \stackrel{\text{def}}{=} \{x \mid Mx^t = 0\}$.

Notation for Varying Vocabularies. 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. Thus, to be able to distinguish between one-vocabulary and two-vocabulary instances of AG and KS, we introduce the following notation. For a set of variables V, KS$[V]$ denotes the set of KS values over V; when V and V' are disjoint sets of variables, KS$[V; V']$ denotes the set of KS values over $V \cup V'$. 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'. AG$[V]$, AG$[V; V']$, and Assignment$[V; V']$ are defined similarly.

For use in §6, we extend this notation to cover singletons: if i is a single variable not in V, then 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 domains such as KS$[V; \{i, i'\}]$, KS$[V; \{i, i', i''\}]$, and KS$[V; V'; \{i\}]$.

As will become clear in §2.2–§2.4, each element of AG$[V]$ and KS$[V]$ represents a set in $\mathcal{P}$(Assignment$[V]$), and each element of MOS, AG$[V; V']$, and KS$[V; V']$ represents a set in $\mathcal{P}$(Assignment$[V; V']$).

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 strictly greater than that of the row above it.

If j is not the leading index of any row in a matrix in row-echelon form, we say that column j is a **skipped column**.

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 $\vec{r}$ of M, for any $p \in \mathbb{Z}$, if i is the leading index of $2^p\vec{r}$, then $2^p\vec{r} \in \text{row}([M]_i)$.

□

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 $\vec{r}$ and power p for which (a) the leading index i of $2^p\vec{r}$ is strictly greater than the leading index of $\vec{r}$, (b) $2^p\vec{r} \neq 0$, and (c) $2^p\vec{r} \notin \text{row}([M]_i)$. In this example, the only interesting quantity of the form $2^p\vec{r}$ 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). □

As we show shortly, a matrix can be put in Howell form using the same elementary row operations familiar from ordinary linear algebra: (i) the addition of a multiple of one row to a different row, (ii) interchanging two rows, and (iii) multiplying all entries in a row by a nonzero constant. As in ordinary linear algebra, operations (i) and (ii) leave a matrix's row space and null space unchanged. Unlike ordinary linear algebra, operation (iii) preserves the row space and null space only if the constant is odd. Moreover, the Howell form of a matrix is unique among all matrices with the same row space (or null space) [Howell 1986].[1] As mentioned earlier, this property

[1] In this respect, Howell form plays the same role for matrices over $\mathbb{Z}_{2^w}$ that reduced row-echelon form plays for matrices over rationals. A matrix M is in *reduced row-echelon form* if

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 columns(G) do
 4:         Let R = {all rows of G with leading index i}
 5:         if R ≠ ∅ then
 6:             Pick an r⃗ ∈ R that minimizes rank r_i
 7:             Pick the odd u and rank p so that u2^p = r_i
 8:             r⃗ ← u^{-1} r⃗                    ▷ Adjust r⃗, leaving r_i = 2^p
 9:             for all s⃗ in R \ {r⃗} do
10:                 Pick the odd v and rank t so that v2^t = s_i
11:                 s⃗ ← s⃗ − (v2^{t−p}) r⃗                    ▷ Zero out s_i
12:                 if row s⃗ contains only zeros then
13:                     Remove s⃗ from G
14:             In G, swap r⃗ with G_{j+1}          ▷ Place r⃗ 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 − dr⃗          ▷ Adjust row G_h, leaving 0 ≤ G_{h,i} < r_i
18:             if r_i ≠ 1 then              ▷ Add logical consequences of r⃗ to G
19:                 Add 2^{w−p} r⃗ as last row of G     ▷ New row has leading index > i
20:             j ← j + 1
```

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 §9.2), 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) 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.

---

(1) It is in row-echelon form.

(2) Every leading value is 1 and is the only nonzero entry in its column.

In $\mathbb{Z}_{2^w}$, each odd element has a multiplicative inverse, but no even element has one; consequently, one cannot divide a row by a power of 2 when working in $\mathbb{Z}_{2^w}$. Given this restriction, items (2) and (3) of Defn. 2.1 are a natural generalization of item (2) above.

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

The following properties of matrices in Howell form will be used in §2.5 to bound the heights of the KS and MOS domains:

Lemma 2.3. Suppose that $(n \times m)$-matrix M is in Howell form. Suppose that we append a new row $\vec{r} \notin row(M)$ to M and Howellize the result, to obtain matrix M' such that $M \neq M'$. Then this change affects at least one of the columns. In particular, if you think of a skipped column as having rank w, no column's rank increases, and for at least one column the rank decreases:

(1) for every column position j, either
   (a) j is the leading index of some row in M, with leading value $v = 2^p$, and j is the leading index of some row in M' with leading value $v' = 2^q$, where $0 \leq q \leq p < w$,
   (b) j is a skipped column in both M and M', or
   (c) j is a skipped column in M, and j is the leading index of some row in M' with leading value $v' = 2^q$, where $0 \leq q < w$.

(2) for at least one column position j, either
   (a) j is the leading index of some row in M, with leading value $v = 2^p$, and j is the leading index of some row in M' with leading value $v' = 2^q$, where $0 \leq q < p < w$, or
   (b) j is a skipped column in M, and j is the leading index of some row in M' with leading value $v' = 2^q$, where $0 \leq q < w$.

Note that the condition on q and p in (1a) is $0 \leq q \leq p < w$, whereas in (2a) it is $0 \leq q < p < w$.

Proof. For a matrix M in Howell form with n columns, we can create a "counter" vector c, defined by

$$c_i = \begin{cases} w & \text{if } i \text{ is a skipped column} \\ \text{rank}(l_i) & \text{if } l_i \text{ is the leading value of column } i \end{cases}$$

The lemma asserts that if $M' = \text{Howellize}(M \cup \{\vec{r}\}) \neq M$, then no entry in c increases (properties (1a)–(1c)), and at least one entry in c decreases (properties (2a)–(2b)).

We use the fact that the Howell form of a matrix is canonical. Thus, when a minimal-rank pivot element is chosen in line (6) of Alg. 1, if one of the possibilities is a row $\vec{s}$ from M, we can always pick $\vec{s}$. Consequently, for the result of $\text{Howellize}(M \cup \{\vec{r}\})$ to be different from M, either (i) a row is generated for which the rank of the leading value is smaller than the rank of the leading value of some row of M, or (ii) a row $\vec{t}$ is generated for which the leading index j corresponds to a skipped column in M. In the latter case, $\text{rank}(t_j) < w$, and thus in both situations, (1) no entry in c increases, and (2) the value of at least one entry in c decreases. □

Corollary 2.4. For a matrix M in Howell form with n columns, the longest sequence of row-append operations that can be made in which each operation satisfies the conditions of Lem. 2.3 is wn.

PROOF. Consider again the vector c used in the proof of Lem. 2.3. As successive row-append operations are performed, c acts as a kind of "counter." For the empty matrix, the counter has the maximum possible value: $(w, w, \ldots, w)$. Each slot of c can only be pushed down to 0; once c reaches $(0, 0, \ldots, 0)$, any row-append operation, followed by Howellization, will not change M. By Lem. 2.3, each row-append operation decreases the value of at least one entry in c; consequently, at most wn row-append operations are possible.

Moreover, it is possible to have a sequence of row-append operations of length wn that each change M. Start with the empty matrix and $c = (w, w, \ldots, w)$. Then, repeatedly (i) chose a column j for which $0 < c_j \leq w$, and (ii) append to M a row with all zeros, except for $2^{c_j - 1}$ in the j^{th} position. After M is Howellized, c is unchanged except in position j, where the value of c_j has been decreased by 1. □

2.2 The Affine-Generator Domain

An element in the Affine Generator domain (AG$[\vec{X}; \vec{X}']$) is a two-vocabulary matrix whose rows are the affine generators of a two-vocabulary relation over variables $\vec{X}$.

An AG$[\vec{X}; \vec{X}']$ element is an r-by-$(2k+1)$ matrix G, with $0 < r \leq 2k+1$. The concretization of an AG$[\vec{X}; \vec{X}']$ element is the set in $\mathcal{P}(\text{Assignment}[\vec{X}; \vec{X}'])$ defined as follows:

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

The AG$[\vec{X}; \vec{X}']$ 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$[\vec{X}; \vec{X}']$ element that represents the identity relation is the matrix $\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & I \end{array}\right]$ (with columns labeled 1, $\vec{X}$, $\vec{X}'$). The AG$[\{x_1, x_2\}; \{x_1', x_2'\}]$ element

$$\left[\begin{array}{c|cccc} 1 & x_1 & x_2 & x_1' & x_2' \\ \hline 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right] \tag{1}$$

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 affine constraints on a two-vocabulary relation. A KS$[\vec{X}; \vec{X}']$ element is an r-by-$(2k+1)$ matrix M, with $0 \leq r \leq 2k+1$. The concretization of a KS$[\vec{X}; \vec{X}']$ element M is the set in $\mathcal{P}(\text{Assignment}[\vec{X}; \vec{X}'])$ defined as follows:

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

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

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

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.

KS elements can be ordered by containment ($\subseteq$) of their concretizations. The bottom element of the KS domain is the matrix $\begin{array}{c} \begin{array}{ccc} \vec{X} & \vec{X}' & 1 \end{array} \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array}$; it consists of one unsatisfiable constraint, and thus its concretization is the empty set of tuples. The KS element that represents the identity relation is the matrix $\begin{array}{c} \begin{array}{ccc} \vec{X} & \vec{X}' & 1 \end{array} \\ \left[\begin{array}{cc|c} I & -I & 0 \end{array}\right] \end{array}$. Suppose that $w = 4$, so that we are working in $\mathbb{Z}_{16}$. The KS element

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

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: $\gamma_{\text{KS}}(M) = \emptyset$ 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, $\begin{array}{c} \begin{array}{ccc} \vec{X} & \vec{X}' & 1 \end{array} \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array}$.

Polynomial-time algorithms for domain operations, such as join and projection, are discussed in §5. To compute the meet of two KS elements, stack the two matrices vertically and Howellize the result. Meet and projection can be used to implement composition (see §5.2.1).

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 concretization of $\mathcal{B}$ is the union of the graphs of the affine transformers in

$\langle\mathcal{B}\rangle$—i.e., the set in $\mathcal{P}(\text{Assignment}[\vec{X};\vec{X}'])$ defined as follows:

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

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

Example 2.5. 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

$$\langle\mathcal{B}\rangle = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 2 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 4 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \ldots, \left[\begin{array}{c|cc} 1 & 0 & 12 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 14 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\},$$

which corresponds to the transition relation in which $x'_1 = x_1$, 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 $\mathcal{C} \circ \mathcal{B} = \textsc{Howellize}\{BC \mid B \in \mathcal{B} \wedge C \in \mathcal{C}\}$. 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, in which row i consists of the elements of matrix M_i (e.g., arranged in row-major order)."

Example 2.6. Consider the set of matrices $\langle\mathcal{B}\rangle$ from Ex. 2.5. They can be arranged as the (8×9)-matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ \cdots & & & & & & & & \\ 1 & 0 & 12 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 14 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

If we apply HOWELLIZE to M, we obtain $\left[\begin{array}{ccccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$, which is the MOS element B from Ex. 2.5 arranged as a (2×9)-matrix. □

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 Ex. 2.6). 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.

THEOREM 2.7. *(1) The height of the KS[V] domain is $wk+1$. The height of the KS[$V;V'$] domain is $2wk+1$.*

(2) The height of the MOS domain is $wk(k+1)+1$.

PROOF. KS[V], KS[$V;V'$], and MOS are all finite-height domains. Thus, the length of the longest ascending chain equals the length of the longest descending chain.

Proof of (1). For KS[V] and KS[$V;V'$], Lem. 2.3 and Cor. 2.4 describe properties of descending chains. By a slight variation on the argument given in Cor. 2.4, the respective lengths of the longest descending chains in the two domains are $wk+1$ and $2wk+1$. The values are $wk+1$ and $2wk+1$ rather than $w(k+1)$ and $w(2k+1)$ because any KS value that has a row in which the leading index is the last column represents $\bot$; thus, only the counter values in the first k and $2k$ columns, respectively, can be decreased in the manner of Cor. 2.4. The counter value for the rightmost column can be pushed down exactly once: from w (when the rightmost column is a skipped column) to 0 (when the KS value is $\begin{array}{c} \vec{x} \quad \vec{x}' \quad 1 \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array} = \bot$), which accounts for the "+1."

Proof of (2). For MOS, Lem. 2.3 and Cor. 2.4 describe properties of ascending chains. Again, by a slight variation on Cor. 2.4, the length of the longest ascending chain is $wk(k+1)$. The value is $wk(k+1)+1$ rather than $w(k+1)^2+1$ because for each matrix in an MOS value, the first column is either a 1 followed by all zeros, or all zeros, so there are only $k(k+1)$ rather than $(k+1)^2$ counter values that can be decreased in the manner of Cor. 2.4. The final "+1" is for $\bot$ (the empty set of $(k+1)$-by-$(k+1)$ matrices).

□

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 §5.1.5). 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, there is a single abstract domain that has two representations: constraint form (KS) and generator form (AG). The situation is similar to some other relational domains, including polyhedra [Cousot and Halbwachs 1978; Bagnara et al. 2008; Jeannet] and grids [Bagnara et al. 2006], which also have dual representations. 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. In contrast, one of the interesting aspects of KS is that all operations needed by an analyzer can be performed on the KS representation, without converting to the AG representation.

To convert between KS and AG (and vice versa), 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 $\texttt{columns}(M)$-by-$\texttt{columns}(M)$ 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

—$M^{\perp} \stackrel{\text{def}}{=} \left(L^{-1}\right)^t T \left(R^{-1}\right)^t$

□

Matrices D and T in Defn. 3.2 are related by the following property:

Lemma 3.3. Let D and T be square, diagonal matrices, where $D_{i,i} = 2^{p_i}$ and $T_{i,i} = 2^{w-\text{rank}(D_{i,i})} = 2^{w-p_i}$ for all i. Then, $\text{null}^t T = \text{row}\, D$ and $\text{null}^t D = \text{row}\, T$.

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

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

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

Matrix dualization has the following useful property:

Theorem 3.4. *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. 3.4, 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 $x = 2$" (before Howellization) 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, there is no MOS element that represents $x = 2 \wedge x = x'$. 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 M cannot represent transitions with a pre-condition guard because every element in the concretization of M represents a concrete transformer that is *total*. Therefore, KS can encode transition 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

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

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

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

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

$$\left[\begin{array}{c|cc} {\scriptstyle 1} & {\scriptstyle V} & {\scriptstyle V'} \\ \hline 1 & 0 & b \\ \hline 0 & I & M \end{array}\right], \tag{5}$$

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 $x' = x \cdot M + b$. □

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 $\begin{bmatrix} 1 & V & V' \\ 0 & 0 & R \end{bmatrix}$; however, G is quite close to being in explicit form, and we can read off a *set* of matrices to create an appropriate MOS element. We produce this set of matrices via the SHATTER operation, where

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

As shown in Thm. 4.3, $\gamma_{\text{AG}}(G) = \gamma_{\text{MOS}}(\textsc{Shatter}(G))$. Intuitively, this property holds because the coefficients of the $\begin{bmatrix} 1 & V & V' \\ 0 & 0 & R_{j,*} \end{bmatrix}$ 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 $\textsc{Shatter}(G)$.

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

PROOF.

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

□

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 an over-approximating element in the MOS domain for such a KS element. This section describes two different over-approximation methods. As in §4.3, KS-to-MOS conversion starts by converting the KS element into an AG matrix G using the algorithm from §3 and HOWELLIZE.

There are two ways in which 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. Either feature is enough to prevent us from being able to put G in either (i) explicit form (Eqn. (5)), or (ii) the form shown in Eqn. (4). In §4.4.1 and §4.4.2, we describe methods to find over-approximating elements G' and G'' such that

—G' has the form shown in Eqn. (4) (§4.4.1)

—G'' is in a form that is a close relative of Eqn. (4) (§4.4.2).

As we show in §4.4.3, the G' and G'' that we find are, in general, incomparable. The intuition to keep in mind is that G' captures only a trivial pre-state/post-state relation—i.e., the pre-state is completely unconstrained. In contrast, G'' retains some aspects of G's pre-state/post-state relation. We will illustrate both methods on the AG element $G = \left[\begin{array}{c|cccccc} & 1 & x_1 & x_2 & x_3 & x_1' & x_2' & x_3' \\ \hline 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 4 & 0 & 12 & 2 & 4 & 0 \\ & & & & 4 & 0 & 8 \end{array}\right]$, where $k = 3$ and $w = 4$. Note that all tuples in $\gamma(G)$ satisfy

$$(x_1 = x_2') \wedge (4x_1 = 0). \tag{6}$$

4.4.1 *KS-to-MOS Conversion Method 1.* G' is created by havocking the pre-state variables of G, which can be performed via $G' := G \sqcup \left[\begin{array}{c|cc} & 1 & \vec{X} & \vec{X}' \\ \hline 1 & 0 & 0 \\ \hline 0 & I & 0 \end{array}\right]$. For instance, for our running example this operation yields

$$G' = \left[\begin{array}{c|cccccc} & 1 & x_1 & x_2 & x_3 & x_1' & x_2' & x_3' \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 8 \end{array}\right].$$

The HAVOC operation causes the pre-state to be completely unconstrained in $\gamma_{\text{AG}}(G')$.

G' has the form shown in Eqn. (4), and hence we can apply the SHATTER operation from §4.3 to G' to obtain an equivalent MOS element M'. In our example,

$$M' = \textsc{Shatter}(G') = \left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 2 & 4 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 8 & 8 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\}. \tag{7}$$

4.4.2 *KS-to-MOS Conversion Method 2.* The second conversion method uses procedure MAKEEXPLICIT (see Alg. 2) to retain some aspects of G's pre-state/post-state relation in the matrix $G'' := \textsc{MakeExplicit}(G)$. MAKEEXPLICIT uses transformations on G that ensure that $\gamma_{\text{AG}}(G'') \supseteq \gamma_{\text{AG}}(G)$. Moreover, G'' has a form that is a close relative of Eqn. (4)—close enough that we can apply a variant of SHATTER to it to obtain the desired MOS element.

The idea used in MAKEEXPLICIT is as follows:

> For each row $\vec{r}$ in which (i) the leading value is in the pre-state, and (ii) the leading value is even (and equals 2^p), split $\vec{r}$ "bitwise" to create two rows, $\vec{r}_{\text{high}}$ and $\vec{r}_{\text{low}}$, made up, respectively, of the high-order bits ($p + 1 \leq h \leq w$) and low-order bits ($1 \leq l \leq p$) of each entry of $\vec{r}$.

Because it is possible to reconstruct $\vec{r}$ from $\vec{r}_{\text{high}}$ and $\vec{r}_{\text{low}}$, this transformation never loses elements from the concretization of G, although it may add elements to it.

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

Require: G is an AG matrix in Howell form

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

(Appending extra rows to an element $G \in \mathrm{AG}$—or in this case, replacing a row $\vec{r}$ by two rows from which $\vec{r}$ can be reconstructed—can only enlarge the concretization of G.)

For instance, consider $\vec{r} = \begin{bmatrix} \overset{1}{0} \mid \overset{x_1}{4} & \overset{x_2}{0} & \overset{x_3}{12} & \overset{x'_1}{2} & \overset{x'_2}{4} & \overset{x'_3}{0} \end{bmatrix}$. Assuming that $w = 4$, $\vec{r}$ in binary notation is $\begin{bmatrix} \overset{1}{0000} \mid \overset{x_1}{0100} & \overset{x_2}{0000} & \overset{x_3}{1100} & \overset{x'_1}{0010} & \overset{x'_2}{0100} & \overset{x'_3}{0000} \end{bmatrix}$. The leading value is $4 = 2^2$, so $\vec{r}$ is split into $\vec{r}_{\text{high}}$, with bits 3 and 4 of each entry (shifted to the right), and $\vec{r}_{\text{low}}$, with bits 1 and 2 of each entry:

$$\vec{r}_{\text{high}} = \begin{bmatrix} \overset{1}{00} \mid \overset{x_1}{01} & \overset{x_2}{00} & \overset{x_3}{11} & \overset{x'_1}{00} & \overset{x'_2}{01} & \overset{x'_3}{00} \end{bmatrix} = \begin{bmatrix} \overset{1}{0} \mid \overset{x_1}{1} & \overset{x_2}{0} & \overset{x_3}{3} & \overset{x'_1}{0} & \overset{x'_2}{1} & \overset{x'_3}{0} \end{bmatrix} \tag{8}$$

$$\vec{r}_{\text{low}} = \begin{bmatrix} \overset{1}{00} \mid \overset{x_1}{00} & \overset{x_2}{00} & \overset{x_3}{00} & \overset{x'_1}{10} & \overset{x'_2}{00} & \overset{x'_3}{00} \end{bmatrix} = \begin{bmatrix} \overset{1}{0} \mid \overset{x_1}{0} & \overset{x_2}{0} & \overset{x_3}{0} & \overset{x'_1}{2} & \overset{x'_2}{0} & \overset{x'_3}{0} \end{bmatrix}$$

Note that $\vec{r} = 4\vec{r}_{\text{high}} + \vec{r}_{\text{low}}$. (Because the leading value of $\vec{r}_{\text{high}}$ is always 1, in general, $\vec{r} = \text{leading-value}(\vec{r}) * \vec{r}_{\text{high}} + \vec{r}_{\text{low}}$.)

The effect of the bitwise split of $\vec{r}$ (and right-shift of the bits in $\vec{r}_{\text{high}}$) is two-fold:

(1) Whereas $\vec{r} \in G$ has some value 2^p, $p > 1$, at the leading-index position i of $\vec{r}$, G'' has a row whose leading value is 1 at position i.

(2) G'' retains from row $\vec{r} \in G$ (i) some relation derived from the high-order bits of $\vec{r}$, and (ii) some relation derived from the low-order bits of $\vec{r}$.

For instance, in $\vec{r}_{\text{high}}$ shown in Eqn. (8), (i) the leading value is 1, and (ii) there is a non-trivial relation on x_1, x_3, and x'_2, namely, $(x_1 = x'_2) \wedge (x_3 = 3x'_2)$.

Lines (4)–(8) of Alg. 2 only append $\vec{r}_{\text{high}}$ (i.e., $\vec{s}$ in line (7)), relying on the call to Howellize to replace $\vec{r}$ by $\vec{r} - \text{leading-value}(\vec{r}) * \vec{r}_{\text{high}}$ $(= \vec{r}_{\text{low}})$, as well as to perform the remaining steps of Howellization.

As mentioned earlier, there are two ways in which G can enforce guards on the pre-state vocabulary: (i) it might contain one or more rows whose leading value is even, or (ii) it might skip some leading indexes in row-echelon form. The

bitwise-split/right-shift transformation deals with (i): after the loop on lines (2)–(8) finishes, for each row that generates a non-zero pre-state-vocabulary value, the leading value is 1. To handle (ii), the loop on lines (9)–(11) inserts all-zero rows into G so that each leading element from the pre-state vocabulary lies on the diagonal.

The row space of the final matrix G'' may be larger than the original row space of G, but, as shown above, the row space of G'' can retain some non-trivial pre-state/post-state relationships from the original G.

Example 4.4. On the first iteration of the loop on lines (2)–(8) of MakeExplicit, one additional row, $\vec{s} = [\,0\,|\,1\;\;0\;\;3\;\;0\;\;1\;\;0\,]$, is appended to G. Note that this operation makes the row space strictly greater than the original row space of G: $\mathrm{row}(G \cup \{\vec{s}\}) \supsetneq \mathrm{row}\, G$. After the call to Howellize on line (8), we have $\left[\begin{array}{c|cccccc} & 1 & x_1 & x_2 & x_3 & x'_1 & x'_2 & x'_3 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$. No additional rows are added during the rest of the loop on lines (2)–(8), and after the loop on lines (9)–(11), the final matrix returned is $G'' = \left[\begin{array}{c|cccccc} & 1 & x_1 & x_2 & x_3 & x'_1 & x'_2 & x'_3 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & 0 & 0 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0 & 0 \\ & & & & & 2 & 0 & 0 \\ & & & & & & & 8 \end{array}\right]$. □

The effect of MakeExplicit is to convert G into a matrix G'' that has the form $\left[\begin{array}{c|cc} 1 & V & V' \\ \hline 0 & J & M \\ 0 & 0 & R \end{array}\right]$, such that

—J and M are square matrices, where J is upper-triangular and 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.

Although G'' does not have the form shown in Eqn. (4), the properties possessed by G'' make it a close relative of Eqn. (4). We can define a suitable variant of Shatter as follows:

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

Note that a and J, which may represent a pre-state guard, do not contribute explicit values to the result of $\textsc{Shatter}(G'')$. Consequently, as illustrated by the following example, we can have $\gamma_{\mathrm{MOS}}(\textsc{Shatter}(G'')) \supsetneq \gamma_{\mathrm{AG}}(G'')$:

Example 4.5. The final MOS element for Ex. 4.4 is

$$M'' = \textsc{Shatter}(G'') = \left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 0 & 8 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\}. \tag{9}$$

Note that, because of the first matrix in M'', all transformers generated by M'' in Eqn. (9) must satisfy $(x_1 = x'_2)$. That property is weaker than the property

$(x_1 = x_2') \wedge (4x_1 = 0)$ that all tuples in $\gamma(G)$ satisfy (see Eqn. (6)). However, $(x_1 = x_2') \wedge (4x_1 = 0)$ implies $(x_1 = x_2')$, and thus M'' retains *some* non-trivial pre-state/post-state relationship from G. In contrast, the transformers generated by M' in Eqn. (7) do not capture any relationship between x_1 and x_2'.

For this example, $\gamma_{\mathrm{MOS}}(M'') \supsetneq \gamma_{\mathrm{AG}}(G'') \supsetneq \gamma_{\mathrm{AG}}(G)$: for instance, $\begin{bmatrix} & 1 & x_1 & x_2 & x_3 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} \times \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] = \begin{bmatrix} & 1 & x_1' & x_2' & x_3' \\ 1 & 0 & 1 & 0 \end{bmatrix}$, so we have $\gamma_{\mathrm{MOS}}(M'') \ni \begin{bmatrix} x_1 & x_2 & x_3 & x_1' & x_2' & x_3' \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \notin \gamma_{\mathrm{AG}}(G'')$. Moreover, $\gamma_{\mathrm{AG}}(G'') \ni \begin{bmatrix} x_1 & x_2 & x_3 & x_1' & x_2' & x_3' \\ 1 & 0 & 3 & 0 & 1 & 0 \end{bmatrix} \notin \gamma_{\mathrm{AG}}(G)$. □

THEOREM 4.6. *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.

4.4.3 *Incomparability of the Two Conversion Methods.* In general, the methods presented in §4.4.1 and §4.4.2 lead to incomparable results, as shown by the following example:

Example 4.7. Let $k = 1$ and $w = 4$, and consider the matrix G shown in the middle below, along with G' and G'' produced by the methods presented in §4.4.1 and §4.4.2.

$$G' = \left[\begin{array}{c|cc} & 1 & x & x' \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{array}\right] \xleftarrow{\texttt{havoc } x} \left[\begin{array}{c|cc} & 1 & x & x' \\ \hline 1 & 0 & 0 \\ 0 & 4 & 12 \end{array}\right] \xrightarrow{\textsc{MakeExplicit}} \left[\begin{array}{c|cc} & 1 & x & x' \\ \hline 1 & 0 & 0 \\ 0 & 1 & 3 \end{array}\right] = G.''$$

G' and G'' are incomparable:

$\gamma_{\mathrm{AG}}(G') \ni \begin{bmatrix} & 1 & x & x' \\ 1 & 0 & 4 \end{bmatrix} \notin \gamma_{\mathrm{AG}}(G'')$, whereas $\gamma_{\mathrm{AG}}(G') \not\ni \begin{bmatrix} & 1 & x & x' \\ 1 & 1 & 3 \end{bmatrix} \in \gamma_{\mathrm{AG}}(G'')$.

After the calls on the respective variants of SHATTER, we end up with two incomparable MOS values:

$$M' = \textsc{Shatter}(G') = \left\{ \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 0 \end{array}\right], \left[\begin{array}{c|c} 0 & 4 \\ \hline 0 & 0 \end{array}\right] \right\} \qquad M'' = \textsc{Shatter}(G'') = \left\{ \left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 3 \end{array}\right] \right\}.$$

M' cannot generate the transformer $\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 3 \end{array}\right] \in \langle M'' \rangle$, and M'' cannot generate the transformer $\left[\begin{array}{c|c} 1 & 4 \\ \hline 0 & 0 \end{array}\right] \in \langle M' \rangle$. □

5. DOMAIN OPERATIONS FOR THE KS DOMAIN

This section describes algorithms for performing all of the domain operations for the KS domain that would be used inside a program analyzer for solving a set of equations over KS values. This section actually covers two related, but somewhat different, sets of operations:

(1) All of the basic abstract-domain operations—meet, project, join, assume, checking containment, etc.—needed for solving KS equations for an intraprocedural analysis problem (§5.1).

(2) The additional domain operations needed for solving KS equations for an interprocedural analysis problem (§5.2).

The algorithms described in §5.1 apply to KS values of types KS[V] and KS[$V; V'$] (although the examples used in §5.1 all use KS[V] values). The algorithms in §5.2 apply to KS values of type KS[$V; V'$]

5.1 Basic KS Domain Operations

5.1.1 *Meet.* We define $X \sqcap Y$ to be the result of creating the block matrix $\left[\begin{smallmatrix} X \\ Y \end{smallmatrix}\right]$ and then Howellizing. 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. Thus, $\left[\begin{smallmatrix} X \\ Y \end{smallmatrix}\right]$—and hence HOWELLIZE($\left[\begin{smallmatrix} X \\ Y \end{smallmatrix}\right]$)—precisely represents the conjunction of the formulas for X and Y. Consequently, the resulting matrix exactly represents the intersection of the meanings of X and Y:

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

5.1.2 *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 $X = \begin{bmatrix} \overset{x_1}{4} & \overset{x_2}{2} & \overset{1}{6} \end{bmatrix}$, 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 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 "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 $\left[\begin{array}{cc|c} \overset{x_1}{4} & \overset{x_2}{2} & \overset{1}{6} \\ 0 & 8 & 8 \end{array}\right]$, and thus we

obtain the following answer for the projection of X onto x_2: $\left[\begin{array}{c|c} \overset{x_2}{8} & \overset{1}{8} \end{array}\right]$, which represents $x_2 \in \{1, 3, \ldots, 15\}$.

This example illustrates how the logical-consequence step of Howellization (lines (18)–(19) of Alg. 1) introduces a row that is (i) not deleted by the column-removal step of projection, and (ii) represents a constraint that is not in the answer produced in Ex. 5.1 by the King/Søndergaard projection algorithm. □

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 $\left[\begin{array}{cc|c} \overset{x_1}{2} & \overset{x_2}{4} & \overset{1}{6} \end{array}\right]$. We permute columns and Howellize to create $\left[\begin{array}{cc|c} \overset{x_2}{4} & \overset{x_1}{2} & \overset{1}{6} \\ 0 & 8 & 8 \end{array}\right]$, project onto the vocabulary suffix x_1, obtaining $\left[\begin{array}{c|c} \overset{x_1}{8} & \overset{1}{8} \end{array}\right]$, add back an all-0 column for x_2, $\left[\begin{array}{cc|c} \overset{x_2}{0} & \overset{x_1}{8} & \overset{1}{8} \end{array}\right]$, and permute columns back to the original order to obtain $\left[\begin{array}{cc|c} \overset{x_1}{8} & \overset{x_2}{0} & \overset{1}{8} \end{array}\right]$. □

5.1.3 *Join.* To join two KS elements Y and Z, we first construct the matrix $\left[\begin{array}{cc} -Y & Y \\ Z & 0 \end{array}\right]$ 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 $\left[\begin{array}{cc} -Y & Y \\ Z & 0 \end{array}\right]\left[\begin{array}{c} u \\ v \end{array}\right] = 0$ is true just if $(Y(v-u) = 0) \wedge (Zu = 0)$. Because $(v-u) \in \operatorname{null} Y$, and $u \in \operatorname{null} Z$, we know that v is the sum of values in $\operatorname{null} Y$ and $\operatorname{null} Z$, 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.

Meet and join do not distribute, as illustrated in the following examples:

Meet over join. In this example, we use the fact that $\top = \left[\begin{array}{cc|c} \overset{x}{1} & \overset{x'}{0} & \overset{1}{0} \end{array}\right] \sqcup \left[\begin{array}{cc|c} \overset{x}{0} & \overset{x'}{1} & \overset{1}{0} \end{array}\right]$. Although technically we are not working with a vector space over a field, the intu-

ition is that the KS element $\overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]}$ represents the "line" $x = 0$, the KS element $\overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]}$ represents the "line" $x' = 0$, and their affine closure is the whole "plane" (i.e., $\top$).

$(a \sqcup b) \sqcap c \neq (a \sqcap c) \sqcup (b \sqcap c)$ for $a = \overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]}$, $b = \overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]}$, and $c = \overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]}$, because

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

Join over meet. Similarly, $(a \sqcap b) \sqcup c \neq (a \sqcup c) \sqcap (b \sqcup c)$ for $a = \overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]}$, $b = \overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]}$, and $c = \overset{x\ \ x'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]}$, because

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

5.1.4 *Assuming Conditions.* By "assuming" a condition φ on a KS element X, we mean to compute a minimal KS element Y such that

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

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 $\vec{V}$, $\overset{\vec{V}\ \ \vec{V}'\ \ 1}{\left[\begin{array}{cc|c} I & -I & 0 \end{array}\right]}$, 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 §6.6.4.2.) 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.

5.1.5 *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 $X = X \sqcap Y$.

5.1.6 *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.[2] 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]A size operation is not often described in presentations of abstract domains; however, it can be useful in some situations. It has received attention in the related context of applying linear and non-linear constraint solvers to support automatic loop parallelization [Tawbi 1994; Pugh 1994; Clauss 1996; Fahringer 1998]. Constraints are generated in which each integer solution corresponds to a distinct pattern in which computational resources are consumed in a program, such as the memory locations or cache lines touched by a loop, the operations executed by a loop, or the memory locations whose contents need to be transferred from one process to another. The *number* of solutions can reveal such information as the computation-to-memory balance of a computation; whether a loop is load-balanced (as well as whether a loop never iterates at all); estimates of statement-execution counts, branch probabilities, and message traffic; or how large message buffers need to be. We also have colleagues who are using the number of solutions to a set of constraints in an information-flow analysis to bound the likelihood of disclosing certain properties of private variables during the execution of a program [Fredrikson and Jha 2013]. (All such work has been for contexts other than the KS domain.)

$2^m v_i$, 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 5.5. Consider again the KS$[V; V']$ element from Eqn. (2)

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

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

5.2 Operations for Interprocedural KS Analysis

This section presents a two-vocabulary version of the KS abstract domain for use in interprocedural dataflow analysis. Unlike previous work by King and Søndergaard [2008], [2010], it is not necessary to perform bit-blasting to perform interprocedural dataflow analysis using the version of KS presented here.

An interprocedural analyzer would use KS$[V; V']$ versions of all the operations discussed in §5.1. §5.2.1 and §5.2.2 describe the additional operations needed to use the KS$[V; V']$ domain with 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].

To perform interprocedural analysis, one needs to create two-vocabulary KS elements that represent abstract transformers. §6 and §7 discuss two methods for that task.

5.2.1 *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 $Y = \left[\begin{array}{cc|c} Y_{\text{pre}} & Y_{\text{post}} & y \end{array}\right]$ and $Z = \left[\begin{array}{cc|c} Z_{\text{pre}} & Z_{\text{post}} & z \end{array}\right]$, 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 "$Z \circ Y$"; 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 $Z \circ Y$ 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 $\gamma_{\text{KS}}(X) = \gamma_{\text{KS}}(Z) \circ \gamma_{\text{KS}}(Y)$, as required. Note that the steps of the abstract-composition algorithm mimic a standard way to express the composition of concrete relations, i.e.,

$$(Z \circ Y)[V; V''] = \exists V' \colon Y[V; V'] \wedge Z[V'; V''].$$

Compose and join do not distribute, as illustrated in the following examples:

Compose over join.

i. $a \circ (b \sqcup c) \neq (a \circ c) \sqcup (b \circ c)$ for $a = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array}$, $b = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array}$,

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

$$\text{“}(x_1 = x_2)\text{”} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array} = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array} \circ \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] \end{array}$$

$$= \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array} \circ \left(\begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array} \sqcup \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \right)$$

$$\neq \left(\begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array} \circ \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array} \right) \sqcup \left(\begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{ccccc|c} 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array} \circ \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array} \right)$$

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

$$= \begin{array}{c} \begin{matrix} x_1 & x_2 & x_1' & x_2' & 1 \end{matrix} \\ \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array} = \text{“}(x_1 = 0) \wedge (x_2 = 0)\text{”}$$

ii. $(a \sqcup b) \circ c \neq (a \circ c) \sqcup (b \circ c)$ for $a = \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right]$, $b = \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]$, and $c = \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right]$, because

$$\text{“}(x_1' = x_2')\text{”} = \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right] = \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] \circ \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right]$$

$$= \left(\left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \right) \circ \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right]$$

$$\neq \left(\left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right] \circ \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right] \right) \sqcup \left(\left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \circ \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right] \right)$$

$$= \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]$$

$$= \left[\begin{array}{cccc|c} x_1 & x_2 & x_1' & x_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] = \text{“}(x_1' = 0) \wedge (x_2' = 0)\text{”}$$

Join over compose. $(a \circ b) \sqcup c \neq (a \sqcup c) \circ (b \sqcup c)$ for $a = \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right]$, $b = \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 0 \end{smallmatrix}\right]$, and $c = \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right]$, because

$$
\begin{aligned}
\text{“}(x' = x + 1)\text{”} &= \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \sqcup \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \\
&= \left(\left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \circ \mathit{Id} \right) \sqcup \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \\
&= \left(\left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \circ \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 0 \end{smallmatrix}\right] \right) \sqcup \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \\
&\neq \left(\left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \sqcup \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \right) \circ \left(\left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 0 \end{smallmatrix}\right] \sqcup \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \right) \\
&= \left[\begin{smallmatrix} x & x' & 1 \\ 1 & -1 & 1 \end{smallmatrix}\right] \circ \top = \top = \text{“true”}
\end{aligned}
$$

5.2.2 *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].

Let us start by presenting the desired concrete collecting semantics of merge functions. To simplify the discussion, assume that all scopes have the same number of locals, and that each vocabulary V consists of sub-vocabularies G and L so that we have relations of the form $R[G, L; G', L']$. We still use k to denote $|V| = |G| + |L|$.

Suppose that we have two relations, $R_1[G, L; G', L']$ and $R_2[G, L; G', L']$, each of which is a subset of $\mathbb{Z}_{2^w}^k \times \mathbb{Z}_{2^w}^k$ Operationally, we want $\textsc{Merge}(R_1[G, L; G', L'], R_2[G, L; G', L'])$ to act as a modified relational composition in which R_2 acts like the identity function on locals so that L' values from R_1 are passed through R_2 unchanged to become the L' values of the result. This semantics can be specified as follows:

$$
\begin{aligned}
&\textsc{Merge}(R_1[G, L; G', L'], R_2[G, L; G', L']) \\
&\qquad = ((\exists L' : R_2[G, L; G', L']) \wedge (L = L')) \circ R_1[G, L; G', L'] \qquad (10)
\end{aligned}
$$

In an interprocedural analysis, the use case for Eqn. (10) is to apply Merge to the relation $\text{CallSiteVal}[G, L; G', L']$ that arises at a call-site that calls procedure Q with the relation $\text{CalleeExitVal}[G, L; G', L']$ that arises at the exit of Q:

$$\textsc{Merge}(\text{CallSiteVal}[G, L; G', L'], \text{CalleeExitVal}[G, L; G', L']).$$

In the KS abstract semantics, the implementation of $(\exists L' : R_2[G, L; G', L']) \wedge (L = L')$ is straightforward:

(1) Havoc vocabulary L' in R_2.

(2) Append L rows, $\begin{bmatrix} \overset{G}{0} & \overset{L}{I} & \overset{G'}{0} & \overset{L'}{-I} & \overset{1}{0} \end{bmatrix}$, so that each variable in vocabulary L' is constrained to have the value of the corresponding variable in vocabulary L.

Example 5.6. Suppose that $G = \{g_1, g_2\}$ and $L = \{l\}$, and that the CallSiteVal and CalleeExitVal transformations are as follows:

$$\begin{array}{rl} \text{CallSiteVal}[G, L; G', L'] : & g_1' = g_1 + 7l \ \wedge \ g_2' = g_2 \qquad \wedge \ l' = g_1 + 3l \\ \text{CalleeExitVal}[G, L; G', L'] : & g_1' = g_1 + l \ \ \wedge \ g_2' = g_1 + g_2 \ \wedge \ l' = g_1 + g_2 + l \end{array}$$

The (Howellized) KS values for CallSiteVal and CalleeExitVal are

$$\text{CallSiteVal} = \begin{array}{c} \begin{matrix} g_1 & g_2 & l & g_1' & g_2' & l' & 1 \end{matrix} \\ \left[\begin{array}{cccccc|c} 1 & 0 & 3 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 4 & -1 & 0 & 1 & 0 \end{array}\right] \end{array} \qquad \text{CalleeExitVal} = \begin{array}{c} \begin{matrix} g_1 & g_2 & l & g_1' & g_2' & l' & 1 \end{matrix} \\ \left[\begin{array}{cccccc|c} 1 & 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -1 & 0 \end{array}\right] \end{array}$$

Steps (1) and (2) produce

$$\begin{array}{c} \begin{matrix} g_1 & g_2 & l & g_1' & g_2' & l' & 1 \end{matrix} \\ \left[\begin{array}{ccccccc} 1 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 \end{array}\right] \end{array} \qquad \begin{array}{c} \begin{matrix} g_1 & g_2 & l & g_1' & g_2' & l' & 1 \end{matrix} \\ \left[\begin{array}{ccccccc} 1 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right] \end{array}$$

Step (1) Step (2)

Finally, the composition of the Step (2) value with CallSiteVal produces

$$\begin{array}{c} \begin{matrix} g_1 & g_2 & l & g_1' & g_2' & l' & 1 \end{matrix} \\ \left[\begin{array}{ccccccc} 1 & 0 & 3 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 4 & -1 & 0 & 2 & 0 \end{array}\right] \end{array} \tag{11}$$

It is easy to check this result: letting double-primed symbols denote the state-variables after the call, the desired transformation is expressed symbolically as

$$\begin{array}{rcl} g_1'' = g_1' + l' & = & 2g_1 + 10l \\ g_2'' = g_1' + g_2' & = & g_1 + g_2 + 7l, \\ l'' = l' & = & g_1 + 3l \end{array}$$

or as the matrix

$$\begin{array}{c} \begin{matrix} g_1 & g_2 & l & g_1'' & g_2'' & l'' & 1 \end{matrix} \\ \left[\begin{array}{ccccccc} 2 & 0 & 10 & -1 & 0 & 0 & 0 \\ 1 & 1 & 7 & 0 & -1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & -1 & 0 \end{array}\right] \end{array}$$

which becomes Eqn. (11) when put in Howell form. □

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

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

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

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

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.

In the remainder of this section, we describe the algorithms needed to create a semantic reinterpretation, based on the KS domain, that implements CreateAbsTrans: the reinterpretation creates an abstract transformer—as a KS$[V; V']$ value—for an assignment statement or sequence of assignment statements. 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.

Consider an assignment statement “$z \leftarrow x + 2y$.” The semantic core of a language that supports such statements 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 must consist of a domain of abstract integers, a domain of abstract states, abstract multiplication and abstract addition of abstract integers, and operations to lookup a variable's value in an abstract state and to create an updated version of a given abstract state.

At first glance, one might think that semantic reinterpretation is limited to non-relational domains (such as intervals)—e.g., abstract integers would represent sets of concrete integers, and abstract states would represent sets of concrete states. The key insight behind the approach presented in §6.3 is as follows:

Abstract integers are values in a *relational* domain (KS$[V;\{i\}]$). Because a relationship is kept at all times between the pre-state and the computed value, the reinterpretation of a state-transformer expression is a KS$[V;V']$ value that can have nontrivial relations between pre-state and post-state values.

The work reported in this section was motivated by our work on TSL [Lim and Reps 2008; 2013], which is a system that uses semantic reinterpretation to generate abstract interpreters for machine code. The presentation is intended to be understandable without any prior knowledge of TSL. TSL is summarized in §6.1, and a few TSL examples are used in §6.3–§6.7 to illustrate how to create a suitable CreateAbsTrans function for the KS domain. §6.2.2 presents the outlines of a relational concrete collecting semantics, which is used in §6.4, §6.5, and §6.6.3 to sketch the core parts of a proof of soundness of semantic reinterpretation for KS.

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. 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. (See [Lim and Reps 2013, §3.1.4.2] for an abbreviated, domain-neutral presentation of the approach used in this section.)

6.1 Semantic Reinterpretation in TSL

With TSL, one specifies an instruction set's concrete operational semantics by defining an interpreter for the instruction set

$$\texttt{interpInstr} : \texttt{instruction} \times \texttt{state} \to \texttt{state}$$

using a first-order functional language. TSL provides features that are common to many functional languages, such as (i) a datatype-definition mechanism for defining recursive datatypes, (ii) data-construction expressions, and (iii) data deconstruction by means of pattern matching (in the style pioneered by Burstall [1969]).

Example 6.1. Fig. 1(a) shows a TSL specification for the `MOV` and `ADD` instructions of the Intel `IA32` instruction set. Consider the instruction "`add eax, ebx`," which (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 (22)–(27) of Fig. 1(a).

Given a state `S`, to obtain the state $\texttt{S}'$ that holds after the execution of the instruction "`add eax, ebx`," one performs

$$\texttt{S}' := \texttt{interpInstr}(\texttt{ADD}(\texttt{DirectReg}(\texttt{EAX}()), \texttt{DirectReg}(\texttt{EBX}())), \texttt{S}).$$

□

From the concrete operational semantics and a specification of a specific abstract domain, the TSL compiler produces semantic reinterpretations of machine-code instructions.

```
(1)  // Abstract-syntax declarations
(2)  reg: EAX() | EBX() | . . . ;
(3)  flag: ZF() | SF() | . . . ;
(4)  operand: Indirect(reg reg INT8 INT32)
(5)           | DirectReg(reg)
(6)           | Immediate(INT32) | . . .;
(7)  instruction
(8)      : MOV(operand operand)
(9)      | ADD(operand operand) | . . . ;
(10) state: State(MAP[INT32,INT8]  // memory-map
(11)              MAP[reg,INT32]  // register-map
(12)              MAP[flag,BOOL]);  // flag-map
(13) // Interpretation functions
(14) INT32 interpOp(state S, operand op) { . . . };
(15) state updateFlag(state S, . . .) { . . . };
(16) state updateState(state S, . . .) { . . . };
(17) state interpInstr(instruction I, state S) {
(18)   with(I) (
(19)     MOV(dstOp, srcOp):
(20)       let srcVal = interpOp(S, srcOp);
(21)       in ( updateState(S, dstOp, srcVal) ),
(22)     ADD(dstOp, srcOp):
(23)       let dstVal = interpOp(S, dstOp);
(24)           srcVal = interpOp(S, srcOp);
(25)           res = dstVal + srcVal;
(26)           S2 = updateFlag(S, dstVal, srcVal, res);
(27)       in ( updateState(S2, dstOp, res) ),
(28)     . . .
(29)   );
(30) };
```

(a)

```
(1)  template <class INTERP> class CIR {
(2)    class reg { . . . };
(3)    class EAX : public reg { . . . }; . . .
(4)    class flag { . . . };
(5)    class ZF : public flag { . . . }; . . .
(6)    class operand { . . . };
(7)    class Indirect: public operand { . . . }; . . .
(8)    class instruction { . . . };
(9)    class MOV : public instruction { . . .
(10)     operand op1; operand op2; . . .
(11)   };
(12)   class ADD : public instruction { . . . }; . . .
(13)   class state { . . . };
(14)   class State: public state { . . . };
(15)   INTERP::INT32 interpOp(state S, operand op) { . . . };
(16)   state updateFlag(state S, . . . ) { . . . };
(17)   state updateState(state S, . . .) { . . . };
(18)   state interpInstr(instruction I, state S) {
(19)    switch(I.id) {
(20)     case ID_MOV: . . .
(21)     case ID_ADD:
(22)       operand dstOp = I.get_child1();
(23)       operand srcOp = I.get_child2();
(24)       INTERP::INT32 dstVal = interpOp(S, dstOp);
(25)       INTERP::INT32 srcVal = interpOp(S, srcOp);
(26)       INTERP::INT32 res = INTERP::Plus32(dstVal, srcVal);
(27)       state S2 = updateFlag(S, dstVal, srcVal, res);
(28)       ans = updateState( S2, dstOp, res );
(29)       break;
(30)     . . .
(31)   }
(32)  }};
```

(b)

Fig. 1. (a) A fragment of the TSL specification of the concrete semantics of the Intel IA32 instruction set. (b) A part of the reinterpretation template generated from (a) (simplified for presentational purposes). ("CIR" stands for "common intermediate representation.")

—The compiler creates a *reinterpretation template* that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the TSL base-types, the map-types used in the semantic specification, and operations on base-type and map-type values.

—An abstract domain $\mathcal{A}$ is given in the form of a C++ class for representing and operating on abstract values.

—By instantiating the reinterpretation template with an abstract domain for each TSL base-type and map-type used in the instruction-set specification, the reinterpretation of the instruction-set specification is extended to TSL expressions and functions.

—As long as the reinterpreted operators over-approximate the semantics of each operation of the TSL meta-language, the reinterpretation of `interpInstr` is an abstract transformer that over-approximates the semantics of each instruction of the instruction set.

Example 6.2. Fig. 1(b) shows part of the reinterpretation template generated from Fig. 1(a). (The template has been simplified for the presentation in this paper.)

A reinterpretation template is a C++ template class that is parameterized on

class INTERP, which specifies an abstract domain for an analysis of interest (line (1) of Fig. 1(b)). The user-defined abstract syntax (lines (2)–(9) of Fig. 1(a)) is translated to a set of C++ abstract-syntax classes (lines (2)–(12) of Fig. 1(b)). The user-defined types, such as reg, operand, and instruction, are translated to abstract C++ classes, and the constructors, such as EAX(), Indirect(_,_,_,_), and ADD(_,_), are subclasses of the appropriate parent abstract C++ classes.

Each user-defined function in Fig. 1(a) is translated to a C++ function (lines (15)–(32) of Fig. 1(b)). Each TSL base-type name and base-type operator name is prepended with the template parameter name INTERP. To instantiate the reinterpretation template, a class INTERP that is appropriate for the analysis of interest is supplied by an analysis developer. □

6.2 Concrete Semantics and Concrete Collecting Semantics

6.2.1 *Concrete Semantics.* Throughout the remainder of §6, as an alternative to `interpInstr` : `instruction` × `state` → `state`, it will often be convenient to denote the standard concrete semantics of a deterministic state-to-state transformer τ by $[\![\tau]\!]$: `state` → `state`. That is, $(x'_1, \ldots, x'_k) = [\![\tau]\!]((x_1, \ldots, x_k))$ is the output state from applying τ to input state $(x_1, \ldots, x_k)$. Similarly, for an expression e, $[\![e]\!]((w_1, \ldots, w_k))$ denotes the output value obtained when e is evaluated according to the concrete semantics of expressions on input state $(w_1, \ldots, w_k)$.

6.2.2 *Concrete Collecting Semantics.* The *concrete collecting semantics of a state-to-state transformer* τ, denoted by $[\![\tau]\!]_c$, is the relation in $\mathcal{P}(\text{Assignment}[V; V'])$ defined as follows:

$$[\![\tau]\!]_c = \{(x_1, \ldots, x_k, x'_1, \ldots, x'_k) \mid (x'_1, \ldots, x'_k) = [\![\tau]\!]((x_1, \ldots, x_k))\}.$$

Our definition of the concrete collecting semantics of an expression is somewhat non-standard. It is motivated by the need to represent the values that a given (sub)expression can take on in the context of a particular state-to-state transformation. The *concrete collecting semantics of e with respect to a given state-to-state transformation* $\rho \in \mathcal{P}(\mathit{Assignment}[V; V'])$, denoted by $[\![e]\!]_c(\rho)$, is the set of assignments in $\mathcal{P}(\text{Assignment}[V; \{i\}])$ defined as follows:

$$[\![e]\!]_c(\rho) = \{(v, x_1, \ldots, x_k) \mid (x_1, \ldots, x_k, x'_1, \ldots, x'_k) \in \rho \wedge v = [\![e]\!]((x'_1, \ldots, x'_k))\}.$$

Note that $[\![e]\!]_c(\rho)$ is an element of $\mathcal{P}(\text{Assignment}[V; \{i\}])$, not $\mathcal{P}(\text{Assignment}[V'; \{i\}])$: the concrete collecting semantics of a subexpression with respect to a given prestate-to-poststate transformation is a relation that represents a *prestate-to-value transformation.*

6.3 Abstract Domains for Reinterpretation

The discussion in §4 and §5.2 focused on the abstract domain $\text{KS}[V; V']$. Each value in $\text{KS}[V; V']$ describes an affine-closed relation between pre-states and post-states, where (i) each state is an assignment to some set of variables V, and (ii) both states have the same number of variables. In this section, we use $\text{KS}[V; V']$ values to over-approximate the concrete collecting semantics of state-to-state transformers outlined in §6.2.2. Typically, we think of V states as "initial states" and V' states as "current states." However, to describe semantic reinterpretation for KS, we also

need a way to represent state-to-*value* transformations, which are needed to over-approximate the concrete collecting semantics of a subexpression with respect to a given state-to-state transformation (§6.2.2).

Recall from §2 our notation for the addition of an individual variable $i \notin V$: the domain $\mathrm{KS}[V; \{i\}]$ is the set of KS values over the variables $V \cup \{i\}$. Operations sometimes introduce additional temporary variables, in which case we have domains like $\mathrm{KS}[V; \{i, i'\}]$ and $\mathrm{KS}[V; \{i, i', i''\}]$.

To over-approximate the concrete collecting semantics of a subexpression with respect to a given state-to-state transformation (which represents the evaluation context of the subexpression), we use the domain $\mathrm{KS}[V; \{i\}]$, and reinterpret machine-integer values as affine-closed relations on pre-states to machine integers. In particular, we introduce a fresh variable i to hold the "current value" of a subexpression in its evaluation context. Because the evaluation context refers to the pre-state V, we write the abstract domain as "$\mathrm{KS}[V; \{i\}]$." Although technically we are working with relations, for a $\mathrm{KS}[V; \{i\}]$ value it is often useful to think of V as a set of *independent variables* and i as the *dependent variable*.

The *KS reinterpretation of a state-to-state transformer* τ, denoted by $[\![\tau]\!]_{\mathrm{KS}} : \mathrm{KS}[V; V']$, is defined in §6.4 and §6.5.2. The *KS reinterpretation of expression* e *with respect to* $K \in KS[V; V']$, denoted by $[\![e]\!]_{\mathrm{KS}} : \mathrm{KS}[V; V'] \to \mathrm{KS}[V; \{i\}]$, is defined in §6.5.1 and §6.6.

As illustrated in line (17) of Fig. 1, the top-level function that is reinterpreted in TSL is $\texttt{interpInstr} : \texttt{instruction} \times \texttt{state} \to \texttt{state}$. To use semantic reinterpretation to implement CreateAbsTrans for the KS domain, `state` is reinterpreted as a $\mathrm{KS}[V; V']$ value, and `interpInstr` is reinterpreted as a $\mathrm{KS}[V; V']$ transformer; that is, $\texttt{interpInstr}_{\mathrm{KS}}$ has the type

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

Let Id_{KS} 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 $\texttt{interpInstr}_{\mathrm{KS}}(\iota, Id_{\mathrm{KS}})$. Thus, for a TSL-based KS reinterpretation, $[\![\iota]\!]_{\mathrm{KS}} \stackrel{\text{def}}{=} \texttt{interpInstr}_{\mathrm{KS}}(\iota, Id_{\mathrm{KS}})$.

Example 6.3. Our implementation of KS for IA32 tracks affine relationships between the processor registers at different program points. It uses an abstraction in which the abstraction of an instruction's semantics is a value in $\mathrm{KS}[R; R']$, where R is the set of register names in the processor. To simplify the example, assume that R is $\{\texttt{eax}, \texttt{ebx}\}$.

Consider again the IA32 instruction "`add eax, ebx`" from Ex. 6.2, which is represented as the term "$\iota_1 = \texttt{ADD}(\texttt{DirectReg}(\texttt{EAX}()), \texttt{DirectReg}(\texttt{EBX}()))$." To translate "`add eax, ebx`" to a $\mathrm{KS}[\{\texttt{eax}, \texttt{ebx}\}; \{\texttt{eax}', \texttt{ebx}'\}]$ value, we would evaluate $\texttt{interpInstr}_{\mathrm{KS}}(\iota_1, Id_{\mathrm{KS}})$. As discussed in §6.5.1 and §6.6.3, the variables used in lines (23)–(25) of Fig. 1(a) would have the following values in the domain

KS[{eax, ebx}; i]:

Line	Variable	Value in KS[{eax, ebx}; i]	Meaning
(23)	dstVal	$\left[\begin{array}{cccc\|c} i & \texttt{eax} & \texttt{ebx} & 1 \\ 1 & -1 & 0 & 0 \end{array}\right]$	$i = \texttt{eax}$
(24)	srcVal	$\left[\begin{array}{cccc\|c} i & \texttt{eax} & \texttt{ebx} & 1 \\ 1 & 0 & -1 & 0 \end{array}\right]$	$i = \texttt{ebx}$
(25)	res	$\left[\begin{array}{cccc\|c} i & \texttt{eax} & \texttt{ebx} & 1 \\ 1 & -1 & -1 & 0 \end{array}\right]$	$i = \texttt{eax} + \texttt{ebx}$

As discussed in §6.5.2, the function call "updateState(S2, dstOp, res)" on line (27) of Fig. 1(a) would return (the Howellization of) $K_1 = \left[\begin{array}{cccc\|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 1 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]$, whose meaning is "$(\texttt{eax}' = \texttt{eax} + \texttt{ebx}) \wedge (\texttt{ebx}' = \texttt{ebx})$." □

6.4 Basic Blocks

For a basic block $B = \iota_1; \iota_2; \ldots; \iota_m$, there are two approaches to performing KS[$V; V'$] reinterpretation:

—*Composed reinterpretation*:

$$\texttt{interpInstr}_{\mathrm{KS}}(\iota_m, Id_{\mathrm{KS}}) \circ_{\mathrm{KS}} \ldots \circ_{\mathrm{KS}} \texttt{interpInstr}_{\mathrm{KS}}(\iota_2, Id_{\mathrm{KS}}) \circ_{\mathrm{KS}} \texttt{interpInstr}_{\mathrm{KS}}(\iota_1, Id_{\mathrm{KS}}).$$

—*Cascaded reinterpretation*:

$$\texttt{interpInstr}_{\mathrm{KS}}(\iota_m, \ldots \texttt{interpInstr}_{\mathrm{KS}}(\iota_2, \texttt{interpInstr}_{\mathrm{KS}}(\iota_1, Id_{\mathrm{KS}})) \ldots).$$

Example 6.4. To illustrate cascaded reinterpretation, suppose that we want the KS[{eax, ebx}; {eax′, ebx′}] value for the instruction sequence "add eax, ebx; add ebx, eax." Let ι_1 be the TSL term that represents "add eax, ebx," as in Ex. 6.3; let ι_2 be the TSL term that represents "add ebx, eax": "ADD(DirectReg(EBX()), DirectReg(EAX()))." Using cascaded reinterpretation, we would evaluate

$$K = \texttt{interpInstr}_{\mathrm{KS}}(\iota_2, \texttt{interpInstr}_{\mathrm{KS}}(\iota_1, Id_{\mathrm{KS}})) = \texttt{interpInstr}_{\mathrm{KS}}(\iota_2, K_1).$$

During the course of evaluating $\texttt{interpInstr}_{\mathrm{KS}}(\iota_2, K_1)$, we would have

Line	Variable	Value in KS[{eax, ebx}; i]	Meaning
(23)	dstVal	$\left[\begin{array}{cccc\|c} i & \texttt{eax} & \texttt{ebx} & 1 \\ 1 & 0 & -1 & 0 \end{array}\right]$	$i = \texttt{ebx}$
(24)	srcVal	$\left[\begin{array}{cccc\|c} i & \texttt{eax} & \texttt{ebx} & 1 \\ 1 & -1 & -1 & 0 \end{array}\right]$	$i = \texttt{eax} + \texttt{ebx}$
(25)	res	$\left[\begin{array}{cccc\|c} i & \texttt{eax} & \texttt{ebx} & 1 \\ 1 & -1 & -2 & 0 \end{array}\right]$	$i = \texttt{eax} + 2\texttt{ebx}$

As discussed in §6.5.2, the function call "updateState(S2, dstOp, res)" on line (27) of Fig. 1(a) would return (the Howellization of) $K = \left[\begin{array}{cccc\|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 1 & 1 & -1 & 0 & 0 \\ 1 & 2 & 0 & -1 & 0 \end{array}\right]$, whose meaning is "$(\texttt{eax}' = \texttt{eax} + \texttt{ebx}) \wedge (\texttt{ebx}' = \texttt{eax} + 2\texttt{ebx})$."

Alternatively we could use composed reinterpretation, which would perform two independent calls $K_1 = \texttt{interpInstr}_{\mathrm{KS}}(\iota_1, Id_{\mathrm{KS}})$ and $K_2 =$

$\texttt{interpInstr}_{\text{KS}}(\iota_2, Id_{\text{KS}})$, and then perform $K = K_2 \circ_{\text{KS}} K_1$. In this example, K_2 equals (the Howellization of) $\left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 1 & 0 & -1 & 0 & 0 \\ 1 & 1 & 0 & -1 & 0 \end{array}\right]$ and $K = K_2 \circ K_1$ equals $\left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 1 & 0 & -2 & 1 & 0 \\ 0 & 1 & 1 & -1 & 0 \end{array}\right]$, which again corresponds to the transition relation $(\texttt{eax}' = \texttt{eax} + \texttt{ebx}) \wedge (\texttt{ebx}' = \texttt{eax} + 2\texttt{ebx})$. □

In §6.7, we give examples that demonstrate that neither technique is strictly better than the other. For reasons discussed in §8.2, the technique used in our experiments is cascaded reinterpretation. Thus, we assume henceforth that the reinterpretation of each basic block is performed via cascaded reinterpretation. The abstraction that we obtain of $[\![\iota_1; \iota_2; \ldots; \iota_m]\!]_c$, the concrete collecting semantics from §6.2.2, is thus

$$[\![\iota_1; \iota_2; \ldots; \iota_m]\!]_{\text{KS}} \stackrel{\text{def}}{=} \texttt{interpInstr}_{\text{KS}}(\iota_m, \ldots, \texttt{interpInstr}_{\text{KS}}(\iota_2, \texttt{interpInstr}_{\text{KS}}(\iota_1, Id_{\text{KS}})) \ldots). \tag{12}$$

In the degenerate case when $m = 0$, $[\![\iota_1; \iota_2; \ldots; \iota_m]\!]_{\text{KS}} = [\![\epsilon]\!]_{\text{KS}} = Id_{\text{KS}}$. Note that $[\![\iota]\!]_{\text{KS}} = \texttt{interpInstr}_{\text{KS}}(\iota, Id_{\text{KS}})$.

Soundness of the Abstraction. We now describe the correctness requirements on $\texttt{interpInstr}_{\text{KS}}$ and $[\![e]\!]_{\text{KS}}$ to ensure that cascaded reinterpretation of a basic block over-approximates the concrete collecting semantics.

Property 6.5. For all $\iota \in \texttt{instruction}$, $e \in \texttt{expression}$, and $K \in \text{KS}[V; V']$,

(1) $\gamma(\texttt{interpInstr}_{\text{KS}}(\iota, K)) \supseteq [\![\iota]\!]_c \circ \gamma(K)$, and
(2) $\gamma([\![e]\!]_{\text{KS}}(K)) \supseteq [\![e]\!]_c(\gamma(K))$.

□

Lemma 6.6. (Soundness of Cascaded Reinterpretation.) If Property 6.5(1) holds, then for all n,

$$\gamma([\![\iota_1; \ldots; \iota_n]\!]_{\text{KS}}) \supseteq [\![\iota_1; \ldots; \iota_n]\!]_c. \tag{13}$$

Proof. By induction on n. Let K_m denote $[\![\iota_1; \ldots; \iota_m]\!]_{\text{KS}}$.

Base case. When $n = 0$, $\gamma([\![\epsilon]\!]_{\text{KS}}) = \gamma(Id_{\text{KS}}) = Id = [\![\epsilon]\!]_c$.

Induction step. Assume that Eqn. (13) holds for all natural numbers $\leq m$.

$$\begin{array}{lll} \gamma([\![\iota_1; \ldots; \iota_m; \iota_{m+1}]\!]_{\text{KS}}) & & \\ \quad = \gamma(\texttt{interpInstr}_{\text{KS}}(\iota_{m+1}), K_m) & \text{by Eqn. (12)} \\ \quad \supseteq [\![\iota_{m+1}]\!]_c \circ \gamma(K_m) & \text{by the hypothesis of the lemma} \\ \quad = [\![\iota_{m+1}]\!]_c \circ \gamma([\![\iota_1; \ldots; \iota_m]\!]_{\text{KS}}) & \\ \quad \supseteq [\![\iota_{m+1}]\!]_c \circ [\![\iota_1; \ldots; \iota_m]\!]_c & \text{by the induction hypothesis} \\ \quad = [\![\iota_1; \ldots; \iota_m; \iota_{m+1}]\!]_c & \end{array}$$

□

6.5 Access and Update Operations

Let S be a state, $\texttt{x}_j$ be a variable name, and X be a value. In the concrete semantics, the operations $\texttt{access}(S, \texttt{x}_j)$ and $\texttt{update}(S, \texttt{x}_j, X)$ on states have the usual semantics:

—An access on state S with respect to $\mathtt{x}_j$ returns the value of $\mathtt{x}_j$ in S. That is,

$$[\![\mathtt{x}_j]\!](S) \stackrel{\text{def}}{=} \mathtt{access}(S, \mathtt{x}_j),$$

and hence,

$$\begin{aligned}[\![\mathtt{x}_j]\!]_c(\rho) &= \{(v, x_1, \ldots, x_k) \mid (x_1, \ldots, x_k, x'_1, \ldots, x'_k) \in \rho \wedge v = \mathtt{access}((x'_1, \ldots, x'_k), \mathtt{x}_j)\} \\ &= \{(x'_j, x_1, \ldots, x_k) \mid (x_1, \ldots, x_k, x'_1, \ldots, x'_k) \in \rho\}.\end{aligned}$$

—The update of state S with respect to $\mathtt{x}_j$ and value X results in an updated state that acts just like S, except that $\mathtt{x}_j$ is mapped to X:

$$\mathtt{access}(\mathtt{update}(S, \mathtt{x}_j, X), \mathtt{x}_i) = ((\mathtt{x}_i = \mathtt{x}_j)\ ?\ X : \mathtt{access}(S, \mathtt{x}_i)),$$

where $(exp_1\ ?\ exp_2 : exp_3)$ denotes an "if-then-else" expression.

A typical use of $\mathtt{update}(S, \mathtt{x}_j, X)$ is the final state-creation operation of an instruction with operand e that evaluates to X in state S. For brevity, we sometimes leave S implicit, and write "$\mathtt{x}_j \leftarrow e$," where

$$[\![\mathtt{x}_j \leftarrow e]\!](S) \stackrel{\text{def}}{=} \mathtt{update}(S, \mathtt{x}_j, [\![e]\!](S)),$$

and hence,

$$\begin{aligned}&[\![\mathtt{x}_j \leftarrow e]\!]_c \\ &\quad= \{(x_1, \ldots, x_k, x'_1, \ldots, x'_k) \mid (x'_1, \ldots, x'_k) = [\![\mathtt{x}_j \leftarrow e]\!]((x_1, \ldots, x_k))\} \\ &\quad= \{(x_1, \ldots, x_k, x'_1, \ldots, x'_k) \mid (x'_1, \ldots, x'_k) = \mathtt{update}((x_1, \ldots, x_k), \mathtt{x}_j, [\![e]\!]((x_1, \ldots, x_k)))\} \\ &\quad= \{(x_1, \ldots, x_k, x_1, \ldots, x_{j-1}, v, x_{j+1}, \ldots x_k) \mid v = [\![e]\!]((x_1, \ldots, x_k))\}.\end{aligned}$$

In the KS reinterpretation, $\mathtt{access}_{\mathrm{KS}}$ and $\mathtt{update}_{\mathrm{KS}}$ are operations that transform values from $\mathrm{KS}[V; V']$ to $\mathrm{KS}[V; \{i\}]$ and vice versa:

$$\begin{aligned}&\mathtt{access}_{\mathrm{KS}} : \mathrm{KS}[V; V'] \times \mathtt{identifier} \to \mathrm{KS}[V; \{i\}] \\ &\mathtt{update}_{\mathrm{KS}} : \mathrm{KS}[V; V'] \times \mathtt{identifier} \times \mathrm{KS}[V; \{i\}] \to \mathrm{KS}[V; V'].\end{aligned}$$

In a slight abuse of notation, we pretend that "$\mathtt{x}_j \leftarrow e$" is an instruction, and write

$$\mathtt{interpInstr}_{\mathrm{KS}}(\mathtt{x}_j \leftarrow e, K) \stackrel{\text{def}}{=} \mathtt{update}_{\mathrm{KS}}(K, \mathtt{x}_j, [\![e]\!]_{\mathrm{KS}}(K)).$$

6.5.1 *Access Operations.* We sometimes denote the operation $\mathtt{access}_{\mathrm{KS}}(K, \mathtt{x}_j)$, where $K \in \mathrm{KS}[V; V']$, by $[\![\mathtt{x}_j]\!]_{\mathrm{KS}}(K)$. For $\mathtt{access}_{\mathrm{KS}}(K, \mathtt{x}_j)$, the following method is used to create the $\mathrm{KS}[V; \{i\}]$ result:

(1) Extend K to be a $\mathrm{KS}[V; V'; \{i\}]$ 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 $i = \mathtt{x}'_j$ on the extended K value. (We wish to obtain the value of variable $\mathtt{x}_j$ from the "current state," which corresponds to vocabulary V'.)

(3) Project away V', yielding a $\mathrm{KS}[V; \{i\}]$ value, as desired.

Assuming the constraint $i = \mathtt{x}'_j$ is straightforward, because it is represented exactly by the KS value $\begin{bmatrix} \overset{i}{1} & \overset{V}{0} & \overset{x'_1}{0} & \overset{\ldots}{\ldots} & \overset{x'_j}{-1} & \overset{\ldots}{\ldots} & \overset{x'_k}{0} & \overset{1}{0} \end{bmatrix}$.

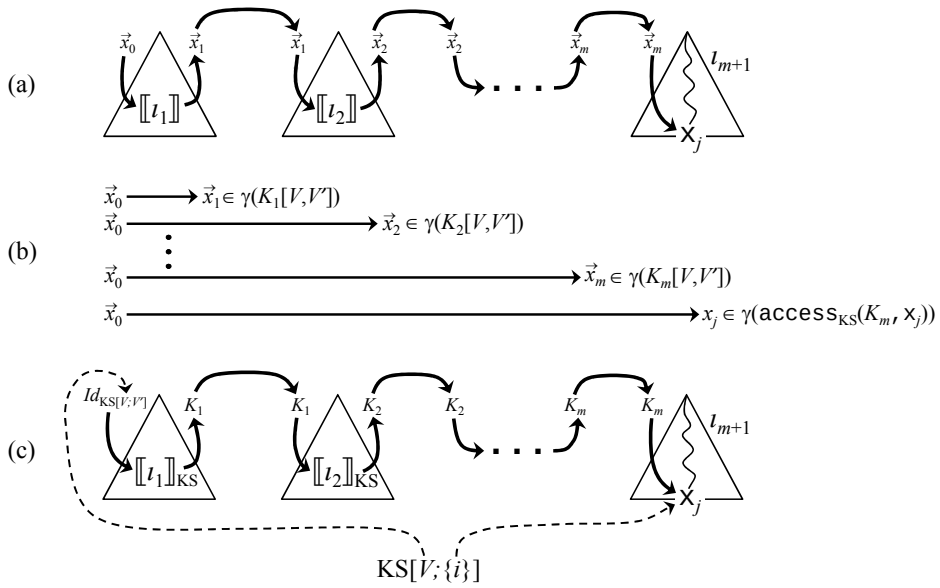


Fig. 2. (a) Concrete semantics of $[\![\mathtt{x}_j]\!]_c([\![\iota_1;\iota_2;\ldots;\iota_m]\!]_c)$. (b) State pairs that are over-approximated by the successive abstract values $K_1, K_2, \ldots, K_m \in \mathrm{KS}[V, V']$ that arise during cascaded interpretation, and the state-to-value tuples that are over-approximated by $\mathtt{access}_{\mathrm{KS}}(K_m, \mathtt{x}_j) \in \mathrm{KS}[V;\{i\}]$. (c) Depiction of the vocabularies involved in the $\mathrm{KS}[V;\{i\}]$ relation returned by $\mathtt{access}_{\mathrm{KS}}(K_m, \mathtt{x}_j)$.

Soundness of the Abstraction. We now use the concrete collecting semantics from §6.2.2 to argue that steps (1)–(3) above over-approximate the concrete collecting semantics of a variable access $[\![\mathtt{x}_j]\!]_c(\gamma(K)) : \mathcal{P}(\mathrm{Assignment}[V;\{i\}])$. We can assume that Property 6.5 holds on proper subterms. (In a formal proof, the derivation given below, along with those in §6.4, §6.5.2, and §6.6.3, would be part of an inductive argument in which Property 6.5 would be one of the properties proven.)

Fig. 2 depicts the concrete semantics of $[\![\mathtt{x}_j]\!]_c([\![\iota_1;\iota_2;\ldots;\iota_m]\!]_c)$, and how it is over-approximated by the $\mathrm{KS}[V;\{i\}]$ relation returned by cascaded reinterpretation. Let K_m denote $[\![\iota_1;\iota_2;\ldots;\iota_m]\!]_{\mathrm{KS}}$. We have

$$
\begin{aligned}
&[\![\mathtt{x}_j]\!]_c([\![\iota_1;\iota_2;\ldots;\iota_m]\!]_c)\\
&\quad\subseteq [\![\mathtt{x}_j]\!]_c(\gamma([\![\iota_1;\iota_2;\ldots;\iota_m]\!]_{\mathrm{KS}})) \qquad \text{(by Lem. 6.6)}\\
&\quad= [\![\mathtt{x}_j]\!]_c(\gamma(K_m))\\
&\quad= \{(v, x_1, \ldots, x_k) \mid (x_1, \ldots, x_k, x'_1, \ldots, x'_k) \in \gamma(K_m) \wedge v = [\![\mathtt{x}_j]\!](x'_1, \ldots, x'_k)\}\\
&\quad= \{(x'_j, x_1, \ldots, x_k) \mid (x_1, \ldots, x_k, x'_1, \ldots, x'_k) \in \gamma(K_m)\}\\
&\quad= \gamma(\mathtt{access}_{\mathrm{KS}}(K_m, \mathtt{x}_j)) \quad \text{(by steps (1)–(3) of "}\mathtt{access}_{\mathrm{KS}}(K_m, \mathtt{x}_j)\text{" and Thm. 5.2)}\\
&\quad= \gamma([\![\mathtt{x}_j]\!]_{\mathrm{KS}}(K_m)).
\end{aligned}
$$

In particular, lines 3–7 in the derivation above show that for $\gamma([\![\mathtt{x}_j]\!]_{\mathrm{KS}}(K_m))$ and $[\![\mathtt{x}_j]\!]_c(\gamma(K_m))$, Property 6.5(2) holds with equality (rather than with $\supseteq$).

6.5.2 *Update Operations.* In the KS reinterpretation, suppose that (i) we have the abstract transformer $K \in \mathrm{KS}[V;V']$, and (ii) the reinterpretation of some expression e with respect to K has produced the reinterpreted value $J = [\![e]\!]_{\mathrm{KS}}(K) \in$

$\mathrm{KS}[V;\{i\}]$. Intuitively, we want to create an abstract transformer $K'' \in \mathrm{KS}[V;V']$ that acts like K, except that the post-state variable $\mathtt{x}' \in V'$ satisfies the constraints in $J \in \mathrm{KS}[V;\{i\}]$.

The operation $\mathtt{update}_{\mathrm{KS}}(K, \mathtt{x}_j, J)$ is carried out by the following sequence of steps:

(1) Let K' be the result of havocking $\mathtt{x}'$ from K. (As discussed in §5.1.2, to havoc $\mathtt{x}'$, project away $\mathtt{x}'$ and then add back an all-0 column for $\mathtt{x}'$.)

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

(3) Return $K'' \stackrel{\text{def}}{=} K' \sqcap \tilde{J}$.

In this method, K' captures the state in which we "forget" the previous value of $\mathtt{x}'$, and $\tilde{J}$ captures the assertion that $\mathtt{x}'$ equals the value of the assignment's expression.

Example 6.7. Returning to Ex. 6.3, consider the call to $\mathtt{updateState}(\mathtt{S2}, \mathtt{dstOp}, \mathtt{res})$ in line (27) of Fig. 1(a) during the interpretation of the instruction "$\mathtt{add\ eax, ebx}$" under the KS reinterpretation of TSL. To create the abstract transformer for "$\mathtt{add\ eax, ebx}$," the initial value of $\mathtt{state}$ S supplied to $\mathtt{interpInstr}$ line (17) would be Id_{KS}, the identity element of $\mathrm{KS}[\{\mathtt{eax}, \mathtt{ebx}\}; \{\mathtt{eax}', \mathtt{ebx}'\}]$.

Because the KS reinterpretation does not track values of IA32 flags, $\mathtt{updateFlag}$ is the identity function, and thus on line (26), $\mathtt{S2}$ would have the value Id_{KS}. As described in Ex. 6.3, variable $\mathtt{res}$ would have the $\mathrm{KS}[V;\{i\}]$ value $\left[\begin{array}{cccc|c} \overset{i}{1} & \overset{\mathtt{eax}}{-1} & \overset{\mathtt{ebx}}{-1} & \overset{1}{0} \end{array}\right]$. The call "$\mathtt{updateState}(\mathtt{S2}, \mathtt{dstOp}, \mathtt{res})$" would return the matrix

$$
\begin{aligned}
&\mathtt{havoc}\left(\left[\begin{array}{cccc|c} \overset{\mathtt{eax}}{1} & \overset{\mathtt{ebx}}{0} & \overset{\mathtt{eax}'}{-1} & \overset{\mathtt{ebx}'}{0} & \overset{1}{0} \\ 0 & 1 & 0 & -1 & 0 \end{array}\right], \mathtt{eax}'\right) \sqcap \left[\begin{array}{cccc|c} \overset{\mathtt{eax}}{1} & \overset{\mathtt{ebx}}{1} & \overset{\mathtt{eax}'}{-1} & \overset{\mathtt{ebx}'}{0} & \overset{1}{0} \end{array}\right] \\
&\quad = \left[\begin{array}{cccc|c} \overset{\mathtt{eax}}{0} & \overset{\mathtt{ebx}}{1} & \overset{\mathtt{eax}'}{0} & \overset{\mathtt{ebx}'}{-1} & \overset{1}{0} \end{array}\right] \sqcap \left[\begin{array}{cccc|c} \overset{\mathtt{eax}}{1} & \overset{\mathtt{ebx}}{1} & \overset{\mathtt{eax}'}{-1} & \overset{\mathtt{ebx}'}{0} & \overset{1}{0} \end{array}\right] \\
&\quad = \mathtt{Howellize}\left(\left[\begin{array}{cccc|c} \overset{\mathtt{eax}}{1} & \overset{\mathtt{ebx}}{1} & \overset{\mathtt{eax}'}{-1} & \overset{\mathtt{ebx}'}{0} & \overset{1}{0} \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]\right) = \left[\begin{array}{cccc|c} \overset{\mathtt{eax}}{1} & \overset{\mathtt{ebx}}{0} & \overset{\mathtt{eax}'}{-1} & \overset{\mathtt{ebx}'}{1} & \overset{1}{0} \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] = K_1,
\end{aligned}
$$

which corresponds to the transition relation $(\mathtt{eax}' = \mathtt{eax} + \mathtt{ebx}) \wedge (\mathtt{ebx}' = \mathtt{ebx})$.

To create the abstract transformer for the sequence "$\mathtt{add\ eax, ebx; add\ ebx, eax}$," a second call on $\mathtt{interpInstr}$ would be performed with $\mathtt{instruction}$ I being the TSL term for "$\mathtt{add\ ebx, eax}$," and $\mathtt{state}$ S having the value K_1. As described in Ex. 6.4, variable $\mathtt{res}$ would have the $\mathrm{KS}[V;\{i\}]$ value $\left[\begin{array}{cccc|c} \overset{i}{1} & \overset{\mathtt{eax}}{-1} & \overset{\mathtt{ebx}}{-2} & \overset{1}{0} \end{array}\right]$. The call

"updateState(S2, dstOp, res)" on line (27) of Fig. 1(a) would return the matrix

$$\texttt{havoc}\left(\begin{array}{c}\begin{array}{cccc|c}\texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1\end{array}\\ \left[\begin{array}{cccc|c}1 & 0 & -1 & 1 & 0\\ 0 & 1 & 0 & -1 & 0\end{array}\right], \texttt{ebx}'\end{array}\right) \sqcap \begin{array}{c}\begin{array}{cccc|c}\texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1\end{array}\\ \left[\begin{array}{cccc|c}1 & 2 & 0 & -1 & 0\end{array}\right]\end{array}$$

$$= \begin{array}{c}\begin{array}{cccc|c}\texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1\end{array}\\ \left[\begin{array}{cccc|c}1 & 1 & -1 & 0 & 0\end{array}\right]\end{array} \sqcap \begin{array}{c}\begin{array}{cccc|c}\texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1\end{array}\\ \left[\begin{array}{cccc|c}1 & 2 & 0 & -1 & 0\end{array}\right]\end{array}$$

$$= \texttt{Howellize}\left(\begin{array}{c}\begin{array}{cccc|c}\texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1\end{array}\\ \left[\begin{array}{cccc|c}1 & 1 & -1 & 0 & 0\\ 1 & 2 & 0 & -1 & 0\end{array}\right]\end{array}\right) = \begin{array}{c}\begin{array}{cccc|c}\texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1\end{array}\\ \left[\begin{array}{cccc|c}1 & 0 & -2 & 1 & 0\\ 0 & 1 & 1 & -1 & 0\end{array}\right],\end{array}$$

which corresponds to the transition relation $(\texttt{eax}' = \texttt{eax} + \texttt{ebx}) \wedge (\texttt{ebx}' = \texttt{eax} + 2\texttt{ebx})$.
□

Soundness of the Abstraction. We now use the concrete collecting semantics from §6.2.2 to argue that steps (1)–(3) above over-approximate the concrete semantics of "$\texttt{update}(S, \texttt{x}_j, X)$." In particular, we wish to show that with KS reinterpretation, $\texttt{interpInstr}_{\text{KS}}(\texttt{x}_j \leftarrow e, K) \stackrel{\text{def}}{=} \texttt{update}_{\text{KS}}(K, \texttt{x}_j, [\![e]\!]_{\text{KS}}(K))$ satisfies Property 6.5(1).

As in the proof sketch in §6.5.1, we can assume that Property 6.5 holds on proper subterms. (In particular, see §6.5.1 and §6.6.3 for the justification of Property 6.5(2).) Let K_m denote $[\![\iota_1; \iota_2; \ldots; \iota_m]\!]_{\text{KS}}$. We now show that Property 6.5(1) holds for $\gamma(\texttt{interpInstr}_{\text{KS}}(\texttt{x}_j \leftarrow e, K_m))$, as follows:

$$\begin{aligned}
&[\![\texttt{x}_j \leftarrow e]\!]_c \circ \gamma(K_m)\\
&= \left\{(x_1, \ldots, x_k, x_1'', \ldots, x_k'') \,\middle|\, \begin{pmatrix}(x_1, \ldots, x_k, x_1', \ldots, x_k') \in \gamma(K_m)\\ \wedge\, (x_1', \ldots, x_k', x_1'', \ldots, x_k'') \in [\![\texttt{x}_j \leftarrow e]\!]_c\end{pmatrix}\right\}\\
&= \left\{(x_1, \ldots, x_k, x_1', \ldots, x_{j-1}', v, x_{j+1}', \ldots x_k') \,\middle|\, \begin{pmatrix}(x_1, \ldots, x_k, x_1', \ldots, x_k') \in \gamma(K_m)\\ \wedge\, v = [\![e]\!]((x_1', \ldots, x_k'))\end{pmatrix}\right\}\\
&= \left\{(x_1, \ldots, x_k, x_1', \ldots, x_{j-1}', v, x_{j+1}', \ldots x_k') \,\middle|\, \begin{pmatrix}(x_1, \ldots, x_k, x_1', \ldots, x_k') \in \gamma(K_m)\\ \wedge\, (v, x_1, \ldots, x_k) \in [\![e]\!]_c(\gamma(K_m))\end{pmatrix}\right\}\\
&\subseteq \left\{(x_1, \ldots, x_k, x_1', \ldots, x_{j-1}', v, x_{j+1}', \ldots x_k') \,\middle|\, \begin{pmatrix}(x_1, \ldots, x_k, x_1', \ldots, x_k') \in \gamma(K_m)\\ \wedge\, (v, x_1, \ldots, x_k) \in \gamma([\![e]\!]_{\text{KS}}(K_m))\end{pmatrix}\right\}\\
&\qquad \text{(by Property 6.5(2))}\\
&= \begin{pmatrix}\{(x_1, \ldots, x_k, x_1', \ldots, x_{j-1}', w_j, x_{j+1}', \ldots, x_k') \mid (x_1, \ldots, x_k, x_1', \ldots, x_k') \in \gamma(K_m)\}\\ \cap \{(x_1, \ldots, x_k, w_1, \ldots, w_{j-1}, v, w_{j+1}, \ldots, w_k) \mid (v, x_1, \ldots, x_k) \in \gamma([\![e]\!]_{\text{KS}}(K_m))\}\end{pmatrix}\\
&= \begin{pmatrix}\texttt{havoc}(K_m, \texttt{v}_j')\\ \cap \{(x_1, \ldots, x_k, w_1, \ldots, w_{j-1}, v, w_{j+1}, \ldots, w_k) \mid (v, x_1, \ldots, x_k) \in \gamma([\![e]\!]_{\text{KS}}(K_m))\}\end{pmatrix}\\
&= \gamma(\texttt{update}_{\text{KS}}(K_m, \texttt{x}_j, [\![e]\!]_{\text{KS}}(K_m))) \quad \text{(by steps (1)–(3) of "}\texttt{update}_{\text{KS}}(K_m, \texttt{x}_j, [\![e]\!]_{\text{KS}}(K_m))\text{")}\\
&= \gamma(\texttt{interpInstr}_{\text{KS}}(\texttt{x}_j \leftarrow e, K_m)).
\end{aligned}$$

6.6 Operations on Reinterpreted Integers in KS$[V; \{i\}]$

With semantic reinterpretation, the value of an expression is a function of the values of the expression's proper constituents[3]—just as in the concrete semantics.

[3]This principle is often referred to as "compositionality." However, we avoid this term because in §6.4 we defined "composed reinterpretation" of a basic block to be a specific reinterpretation

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.

6.6.1 *Constants.* The KS reinterpretation of a constant c, $[\![c]\!]_{\mathrm{KS}} : \mathrm{KS}[V; V'] \to \mathrm{KS}[V; \{i\}]$, ignores the input argument and returns the $\mathrm{KS}[V; \{i\}]$ value that encodes the equation $i = c$: $\left[\begin{array}{c|c} \overset{i}{1} & \overset{V}{0 \ldots 0} \end{array} \middle| \begin{array}{c} \overset{1}{-c} \end{array}\right]$.

6.6.2 *Multiplication by a Constant.* Suppose that we have the $\mathrm{KS}[V; \{i\}]$ value K_e for subexpression e, and wish to compute the $\mathrm{KS}[V; \{i\}]$ value for the expression $c * e$. We proceed as follows:

(1) Extend K_e to be a $\mathrm{KS}[V; \{i, i'\}]$ value, adding an all-0 column for i'.

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

(3) Project away i, yielding a $\mathrm{KS}[V; \{i'\}]$ value.

(4) Rename i' to i, yielding a $\mathrm{KS}[V; \{i\}]$ value, as desired.

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

6.6.3 *Addition.* Suppose that we have the $\mathrm{KS}[V; \{i\}]$ values K_x and K_y for subexpressions x and y, respectively, and wish to compute the $\mathrm{KS}[V; \{i\}]$ value for the expression $x + y$. We proceed as follows:

(1) Rename K_y's i variable to i'; this makes K_y a $\mathrm{KS}[V; \{i'\}]$ value.

(2) Extend both K_x and K_y to be $\mathrm{KS}[V; \{i, i', i''\}]$ values, adding all-0 columns for i' and i'' to x, and all-0 columns for i and i'' to K_y. This step causes i' and i'' to be unconstrained in K_x, and i and i'' to be unconstrained in K_y.

(3) Compute $K_x \sqcap K_y$.

(4) Assume the constraint $i'' = i + i'$ on the $\mathrm{KS}[V; \{i, i', i''\}]$ value computed in step (3).

(5) Project away i and i', yielding a $\mathrm{KS}[V; \{i''\}]$ value.

(6) Rename i'' to i, yielding a $\mathrm{KS}[V; \{i\}]$ value, as desired.

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

Soundness of the Abstraction. To justify abstract addition in $\mathrm{KS}[V; \{i\}]$ with respect to the concrete collecting semantics defined in §6.2.2, suppose that $e = e_1 + e_2$, and that $\rho : \mathcal{P}(\mathrm{Assignment}[V; V'])$ is some state-to-state transformation.

---

pattern (different from the "cascaded reinterpretation" of a basic block).

Then the concrete collecting semantics of addition in the evaluation context ρ can be expressed as follows:

$$[\![e_1 + e_2]\!]_c(\rho) = \left\{ (v, x_1, \ldots, x_k) \;\middle|\; \begin{array}{l} (v_1, x_1, \ldots, x_k) \in [\![e_1]\!]_c(\rho) \\ \wedge\, (v_2, x_1, \ldots, x_k) \in [\![e_2]\!]_c(\rho) \\ \wedge\, v = v_1 + v_2 \end{array} \right\}.$$

In an inductive proof of soundness, we can assume that Property 6.5(2) holds for e_1 and e_2. That is, for all $K \in \mathrm{KS}[V; V']$ and $i \in \{1, 2\}$, $\gamma([\![e_i]\!]_{\mathrm{KS}}(K)) \supseteq [\![e_i]\!]_c(\gamma(K))$. Then,

$$\begin{aligned} [\![e_1 + e_2]\!]_c(\gamma(K)) \;&=\; \left\{ (v, x_1, \ldots, x_k) \;\middle|\; \begin{array}{l} (v_1, x_1, \ldots, x_k) \in [\![e_1]\!]_c(\gamma(K)) \\ \wedge\, (v_2, x_1, \ldots, x_k) \in [\![e_2]\!]_c(\gamma(K)) \\ \wedge\, v = v_1 + v_2 \end{array} \right\} \\ &\subseteq\; \left\{ (v, x_1, \ldots, x_k) \;\middle|\; \begin{array}{l} (v_1, x_1, \ldots, x_k) \in \gamma([\![e_1]\!]_{\mathrm{KS}}(K)) \\ \wedge\, (v_2, x_1, \ldots, x_k) \in \gamma([\![e_2]\!]_{\mathrm{KS}}(K)) \\ \wedge\, v = v_1 + v_2 \end{array} \right\} \quad (14) \\ &=\; \gamma([\![e_1 + e_2]\!]_{\mathrm{KS}}(K)). \quad (15) \end{aligned}$$

Line (15) follows from line (14) by steps (1)–(6) above, which perform $\mathrm{KS}[V; \{i\}]$, $\mathrm{KS}[V; \{i'\}]$, and $\mathrm{KS}[V; \{i, i', i''\}]$ operations that implement the set-former expression in line (14).

Remark 6.8. Multiplication by a constant and addition are both examples of linear operations. The KS domain can precisely compute any linear combination of KS values. For instance, given $\mathrm{KS}[V; \{i\}]$ values K_x and K_y for the subexpressions x and y, respectively, we can compute a $\mathrm{KS}[V; \{i\}]$ value for the expression $3x+8y$. The steps are similar to those used for addition, except that step (4) would be "Assume the constraint $i'' = 3i + 8i'$ on $K_x \sqcap K_y$."

However, semantic reinterpretation 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 combinations 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 $\mathrm{KS}[V; \{i\}]$ value that we would obtain by treating $3x + 8y$ as a single linear operator. □

6.6.4 *Non-Linear Operations.* The KS domain cannot interpret most instances of non-linear operations precisely. However, when a $\mathrm{KS}[V; \{i\}]$ value has the form

$$\begin{array}{c} \scriptstyle i \quad \scriptstyle V \quad\; \scriptstyle 1 \\ [\,1 \;\; 0 \ldots 0 \,|\, -c\,] \end{array},$$

the dependent variable i of a $\mathrm{KS}[V; \{i\}]$ value can have only a single concrete value c, regardless of the values of the independent variables; i.e., the $\mathrm{KS}[V; \{i\}]$ value represents the constant c. When an operation's input $\mathrm{KS}[V; \{i\}]$ values each denote a constant, the operation can be performed precisely by performing it in concrete arithmetic on the identified constants. In essence, this ap-

proach uses special-case handling to identify constants, and then performs constant propagation—including constant propagation over non-linear operators.

6.6.4.1 *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. As with constants, when an operation's input KS$[V;\{i\}]$ values denote partially-constant values, special-case handling in the abstract operation can be used to compute a more-precise answer than would otherwise be obtained. (see §6.6.4.2).

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

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

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:
>
> $$X' = \begin{array}{c} \begin{array}{cc} \scriptstyle i & \scriptstyle 1 \end{array} \\ \left[\begin{array}{c|c} 2^m & -2^m c \end{array}\right] \end{array}.$$

X' denotes the congruence $2^m i = 2^m c \pmod{2^w}$, which may be expressed equivalently as $i = c \pmod{2^{w-m}}$.[4] Consequently, the rightmost $w - m$ bits of i are constant, and their value is c.

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

6.6.4.2 *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 §6.6.4.1. 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

[4] As discussed in §5.1.4, a congruence of the form "lhs = rhs $\pmod{2^h}$" 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 $h = w - m$.

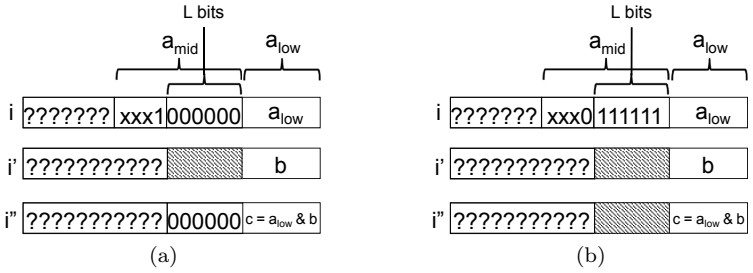


Fig. 3. Bitwise-and when more bits are known to be partially constant for the first argument i than for the second argument i'. (a) a_{mid} ends in L zeros; (b) a_{mid} ends in L ones.

proceed as in §6.6.3, except that step (4) assumes the congruence $i'' = c \pmod{2^m}$ (as explained in §5.1.4). 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 $0 \le a_{\text{low}} < 2^m$ and $0 \le a_{\text{mid}} < 2^{m'}$. (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. 3(a)). Thus, we can capture $L + m$ constant bits in the answer by using the method from §6.6.3, 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. 3(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 §6.6.3, except that step (4) assumes the congruence

$$i'' = (i' - b + c) \pmod{2^{L+m}}. \tag{16}$$

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. (16) 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 §6.6.3, except that step (4) assumes the congruence

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

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

6.7 Incomparability of Composed and Cascaded Reinterpretation

As shown by Exs. 6.9 and 6.10 below, neither technique for basic-block reinterpretation is strictly better than the other.

Example 6.9. To see that cascaded reinterpretation can create a more precise $\text{KS}[V; V']$ element than composed reinterpretation, 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`):

$$\begin{aligned} \iota_3 &: \texttt{xor ax, ax} \\ \iota_4 &: \texttt{or \ ebx, eax} \end{aligned}$$

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. $\texttt{interpInstr}_{\text{KS}}(\iota_3, Id_{\text{KS}})$ creates the KS element $\left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right]$, which captures $(\texttt{ebx}' = \texttt{ebx}) \wedge (2^{16}\texttt{eax}' = 0)$. The two approaches to reinterpretation produce the following answers:

—*Composed reinterpretation*:

$$\begin{aligned} &\texttt{interpInstr}_{\text{KS}}(\iota_4, Id_{\text{KS}}) \circ_{\text{KS}} \texttt{interpInstr}_{\text{KS}}(\iota_3, Id_{\text{KS}}) \\ &\quad = \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 1 & 0 & -1 & 0 & 0 \end{array}\right] \circ_{\text{KS}} \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\ &\quad = \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\ &\quad = (2^{16}\texttt{eax}' = 0). \end{aligned}$$

—*Cascaded reinterpretation*:

$$\begin{aligned} &\texttt{interpInstr}_{\text{KS}}(\iota_4, \texttt{interpInstr}_{\text{KS}}(\iota_3, Id_{\text{KS}})) \\ &\quad = \texttt{interpInstr}_{\text{KS}}(\iota_4, \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right]) \\ &\quad = \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ 0 & 2^{16} & 0 & -2^{16} & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\ &\quad = (2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx}) \wedge (2^{16}\texttt{eax}' = 0). \end{aligned}$$

The reinterpretation of "ι_4: `or ebx, eax`" takes place in a context in which the two low-order bytes of `eax` are partially constant (see §6.6.4.1 and §6.6.4.2); in particular, $2^{16}\texttt{eax} = 0$ holds. Because of this additional piece of information, the reinterpretation technique described in §6.6.4.2 recovers the additional conjunct "$2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx}$."

Put more succinctly, what the above example shows is that because of the evaluation methods used to increase precision for partially constant values (§6.6.4.1 and §6.6.4.2), it is possible to have

$$\texttt{interpInstr}_{\mathrm{KS}}(\iota, Id_{\mathrm{KS}}) \circ K \sqsupset \texttt{interpInstr}_{\mathrm{KS}}(\iota, K).$$

□

Example 6.10. To see that composed reinterpretation can create a more precise KS$[V; V']$ element than cascaded reinterpretation, consider the following code fragment:

$$\begin{array}{l} \iota_5 : \texttt{test} \;\; \texttt{eax}, \texttt{eax} \\ \iota_6 : \texttt{setnz} \; \texttt{cl} \\ \iota_7 : \texttt{add} \;\;\; \texttt{esi}, \texttt{ecx} \end{array}$$

Cascaded KS-reinterpretation obtains

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

whereas composed KS-reinterpretation obtains the more precise answer

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

The instruction "`test eax, eax`" sets the x86 processor flags according to the result of the bitwise logical-and of the value of register `eax` with itself. In particular, it sets the `ZF` flag to *true* iff the result is 0 (which, for this instruction, would happen exactly when the value of `eax` is 0). The instruction "`setnz cl`" sets the low-order byte of register `ecx` to 1 if `ZF` is *true*, and 0 if `ZF` is *false*. Because the value of `ecx` is unknown on entry to the code fragment, the prefix "$[\![\iota_5; \iota_6]\!]_{\mathrm{KS}}$" of the cascaded reinterpretation computes a KS$[V, V']$ element $\mathrm{KS}_{5,6}$ in which nothing is known about the values of `ecx` and `ecx`′. When $\texttt{interpInstr}_{\mathrm{KS}}(\iota_7, \mathrm{KS}_{5,6})$ is performed, the KS$[V; \{i\}]$ value retrieved for the value of argument `ecx` of "`add esi, ecx`" is $\top_{\mathrm{KS}[V;\{i\}]}$ (see §6.5.1), and hence cascaded reinterpretation is unable to identify the desired relationship between `esi`, `ecx`′, and `esi`′.

In contrast, with composed reinterpretation, the KS$[V, V']$ element given to $\texttt{interpInstr}_{\mathrm{KS}}$ for "`add esi, ecx`" is Id_{KS}. When $\texttt{interpInstr}_{\mathrm{KS}}(\iota_7, Id_{\mathrm{KS}})$ is performed, the KS$[V; \{i\}]$ value retrieved for the value of `ecx` is $\left[\begin{array}{cccccc|c} \scriptstyle i & \scriptstyle \texttt{eax} & \scriptstyle \texttt{ebx} & \scriptstyle \texttt{ecx} & \scriptstyle \texttt{esi} & \scriptstyle 1 & \\ 1 & 0 & 0 & -1 & 0 & 0 \end{array}\right]$.

The addition of `ecx` to `esi` results in $\left[\begin{array}{cccccc|c} \scriptstyle i & \scriptstyle \texttt{eax} & \scriptstyle \texttt{ebx} & \scriptstyle \texttt{ecx} & \scriptstyle \texttt{esi} & \scriptstyle 1 & \\ 1 & 0 & 0 & -1 & -1 & 0 \end{array}\right]$. As discussed in §6.5.2, the function call "`updateState(S2, dstOp, res)`" on line (27) of Fig. 1(a) would return the KS$[V, V']$ element

$$\left[\begin{array}{cccccccc|c} \scriptstyle \texttt{eax} & \scriptstyle \texttt{ebx} & \scriptstyle \texttt{ecx} & \scriptstyle \texttt{esi} & \scriptstyle \texttt{eax}' & \scriptstyle \texttt{ebx}' & \scriptstyle \texttt{ecx}' & \scriptstyle \texttt{esi}' & \scriptstyle 1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 0 & 1 & 0 \end{array}\right],$$

which Howellizes to

$$\begin{array}{c} \begin{array}{cccccccc} \texttt{eax} & \texttt{ebx} & \texttt{ecx} & \texttt{esi} & \texttt{eax}' & \texttt{ebx}' & \texttt{ecx}' & \texttt{esi}' \end{array} \\ \left[\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 0 \end{array}\right] \end{array} \quad \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} \texttt{1} \\ \\ \\ \\ \end{array} \tag{17}$$

When matrix (17) is composed with

$$\texttt{interpInstr}_{\mathrm{KS}}(\iota_6, Id_{\mathrm{KS}}) \circ \texttt{interpInstr}_{\mathrm{KS}}(\iota_5, Id_{\mathrm{KS}}),$$

$\texttt{ecx}$ and $\texttt{ecx}'$ are havocked, but the last row is retained, i.e., $\texttt{esi}' = \texttt{esi} + \texttt{ecx}'$.

In this example, the shorter range over which an instruction's effect is considered during composed reinterpretation allows an affine relation to be identified with respect to a *post-state* variable, even though all information about the value of the corresponding pre-state variable is lost. Because of this phenomenon it is possible to have

$$\texttt{interpInstr}_{\mathrm{KS}}(\iota, K) \sqsupset \texttt{interpInstr}_{\mathrm{KS}}(\iota, Id_{\mathrm{KS}}) \circ K.$$

The example shows that identifying relationships with post-state variables can be important. Because reinterpretation performs forward (evaluation-order) evaluation, it is natural to use $\mathrm{KS}[V; \{i\}]$ values that keep relationships between a computed value and the values of pre-state variables. To create reinterpreted integers as relations of the form $\mathrm{KS}[V'; \{i\}]$, with respect to post-state variables, would seem to require a backward reinterpretation, performed counter to normal evaluation order. □

Note that appending the code fragment from Ex. 6.9 at the end of the one from Ex. 6.10 creates a code fragment in which the respective KS elements obtained from cascaded and composed reinterpretation are incomparable. One could always perform both reinterpretations and take the meet of the two results.

7. SYMBOLIC ABSTRACTION

Cousot and Cousot [1979] gave the following *specification* of the most-precise abstract interpretation of a concrete operator ("transformer") τ that is possible in a given abstract domain:

> Given a Galois connection $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between concrete domain $\mathcal{C}$ and abstract domain $\mathcal{A}$, the *best abstract transformer*, $\tau^{\#} : \mathcal{A} \to \mathcal{A}$, 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 $\mathcal{A}$.

Unfortunately, there are several problems that make it difficult to obtain best abstract transformers in practice.

(1) 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 values or concrete states—that is either infinite or too large to fit in computer memory.

(2) Best abstract operators are not closed under composition. That is, if $\tau_1^{\#}$ and $\tau_2^{\#}$ are the best abstract operators that over-approximate τ_1 and τ_2, $\tau_2^{\#} \circ \tau_1^{\#}$ is not necessarily the best abstraction of $\tau_2 \circ \tau_1$. Consequently, one cannot obtain the best abstract operator for an entity by combining best abstractions of the entity's constituents.

In particular, even if one has best KS operators for the set of machine-integer operations (cf. §6), an operator-by-operator reinterpretation method, like the one used in the TSL system, will not necessarily create the best abstract transformer for an individual assignment statement, basic block, or loop-free program fragment.

In practice, whether analyzing source code or machine code, the concrete semantics typically includes 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, issue (2) above can amplify the deficiencies of an abstract domain in tracking the result of a computation because of cascade effects when reinterpretation is applied to a large expression.

An operator-by-operator reinterpretation method abstracts operations in isolation, and is therefore rather "myopic." In contrast, a more "far-sighted" 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, the notion of *symbolic abstraction* [Reps et al. 2004] both (i) adopts a global outlook, and (ii) provides an algorithm for obtaining abstract transformers. Symbolic abstraction is parameterized on a logic $\mathcal{L}$ that is assumed to be rich enough to express the semantics of the constructs of the programming language in use.

—Abstract domain $\mathcal{A}$ is said to support a *symbolic implementation of the α function* of a Galois connection (or "*symbolic abstraction*," for short) if, for every logical formula $\psi \in \mathcal{L}$ 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 $\mathcal{L}$.

—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 $\varphi_\iota \in \mathcal{L}$ that captures the concrete semantics of each instruction ι (or basic block, or loop-free program fragment), and then performing $\widetilde{\alpha}(\varphi_\iota)$ or $\widehat{\alpha}(\varphi_\iota)$.

In our work, $\mathcal{L}$ is quantifier-free bit-vector logic (QFBV). 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 2013, §4.1.5]. That is, given ι, the desired formula φ_ι is $[\![\iota]\!]_{QFBV}$, which can be obtained by evaluating $\texttt{interpInstr}_{QFBV}(\iota, Id_{QFBV})$.

King and Søndergaard [2010, Fig. 2] gave an algorithm for symbolic abstraction for the KS domain: given a QFBV formula φ, their algorithm returns an element

in KS that over-approximates $[\![\varphi]\!]$. As originally stated, their algorithm did not return the *best* element in KS that over-approximates $[\![\varphi]\!]$ because, as discussed in §5.1.2, it used row-echelon form—rather than Howell form—for the projections that take place in the algorithm's join operations (cf. Ex. 5.1). However, with our more precise projection operation from §5.1.2, the symbolic-abstraction algorithm does return the best element in KS that over-approximates $[\![\varphi]\!]$.

$\widehat{\alpha}$ for the KS domain has the potential to create more precise abstract transformers than operator-by-operator reinterpretation, because $\widehat{\alpha}$ 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 $\mathcal{B}$. Consider the following examples:

Example 7.1. $\mathcal{B}$ contains a store to memory of register `eax`'s value, and a subsequent load from memory of that value into `ebx`.

$$\begin{array}{l} \iota_8 : \texttt{push eax} \\ \iota_9 : \texttt{pop ebx} \end{array}$$

Because the formula $[\![\iota_8; \iota_9]\!]_{QFBV}$ given to $\widehat{\alpha}$ captures the two memory operations, $\widehat{\alpha}$ finds the KS$[V; V']$ element that represents the transformation $(\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ebx}' = \texttt{eax})$. In contrast, KS-reinterpretation produces $\top_{\text{KS}[V;V']}$ because it interprets memory-access/update operations very conservatively.

Example 7.2.

$$\begin{array}{lll} \iota_{10} : \texttt{test} & \texttt{eax}, \texttt{eax} \\ \iota_{11} : \texttt{jz} & \texttt{label} \\ \iota_{12} : \langle \texttt{some instr} \rangle & \\ \iota_{13} : \textit{label} : \ldots & \end{array}$$

Suppose that we want to obtain the KS transformer for the path that consists of ι_{10}, ι_{11}, and the jump to *label*. The jump is performed only if `ZF` is true. `ZF` is set to true by ι_{10} exactly when `eax` is zero. Because the formula $[\![\iota_{10}; \iota_{11}; \textit{label}]\!]_{QFBV}$ given to $\widehat{\alpha}$ captures the flag-access/update operations performed in ι_{10} and ι_{11}, $\widehat{\alpha}$ finds the KS$[V; V']$ element that represents the transformation $(\texttt{eax}' = \texttt{0}) \wedge (\texttt{ebx}' = \texttt{ebx})$. In contrast, KS-reinterpretation obtains the weaker result $(\texttt{ebx}' = \texttt{ebx})$.

Pseudo-code for the improved King and Søndergaard algorithm, which we will denote by $\widehat{\alpha}^{\uparrow}_{\text{KS}}(\varphi)$, is shown in Fig. 4(a). The matrix *lower* is maintained in Howell form throughout. In line (6), $\widehat{\alpha}^{\uparrow}_{\text{KS}}(\varphi)$ 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:

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

If $\overset{x_1}{a_1} \overset{\cdots}{\cdots} \overset{x_k}{a_k} \overset{x'_1}{a'_1} \overset{\cdots}{\cdots} \overset{x'_k}{a'_k} \overset{1}{\big|\, b}$ 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. 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 $\widehat{\alpha}^{\uparrow}_{\text{KS}}(\varphi)$ 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)

$$lower \leftarrow lower \sqcup \beta(S). \tag{18}$$

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

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

or, equivalently,

$$S = [\vec{X} \mapsto \vec{v}, \vec{X}' \mapsto \vec{v}']. \tag{19}$$

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):

$$\beta(S) \stackrel{\text{def}}{=} \left[\begin{array}{cc|c} \multicolumn{2}{c|}{\begin{array}{cc}\vec{X} & \vec{X}'\end{array}} & 1 \\ \hline I & 0 & (-\vec{v})^t \\ 0 & I & (-\vec{v}')^t \end{array}\right] \tag{20}$$

7.1 Correctness of $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$

As originally stated, the King and Søndergaard algorithm only created best abstract transformers for the Boolean case (KS_{2^1}). For the general KS_{2^w} domain, their algorithm creates a sound, but not necessarily best, transformer (i.e., it is an algorithm for $\widetilde{\alpha}(\varphi)$). They claimed to prove its correctness, but the proof is flawed for the general KS_{2^w} domain. In this section and App. E, we prove that Fig. 4(a) implements $\widehat{\alpha}(\varphi)$. Our proof also provides insight on how the algorithm works.

The argument that Fig. 4(a) is correct is somewhat subtle. In particular, one of the tricky aspects of Fig. 4(a) is the indexing into matrix *lower* via variable i: i is used to index rows of *lower* relative to the *last* row of *lower*. Initially, *lower* is the one-row matrix $\left[\begin{array}{cc|c} \vec{X} & \vec{X}' & 1 \\ \hline 0 & 0 & 1 \end{array}\right]$, which represents $\bot_{\mathrm{KS}}$, and $i = 1$ indexes the last 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 "$\mathit{lower} \leftarrow \bot_{\mathrm{KS}} \sqcup \beta(S)$" (line (12)), after which *lower* holds the value $\beta(S)$. Thus, as can be seen from Eqn. (20), 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. (18)). 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. E.4, 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 *last* row of *lower*.

Algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ 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 $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})] = \top \sqsupseteq \widehat{\alpha}(\varphi)$.[5] In App. E, we prove a sequence of lemmas that establish that Algorithm

[5] When $i = 1$, the range $(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})$ is empty, and $\mathit{lower}[(\texttt{rows}(\mathit{lower}) -$

$\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ maintain these invariants properly. The lemmas are used to prove the following theorem:

THEOREM 7.3. *If algorithm $\widehat{\alpha}^{\uparrow}_{KS}$ from Fig. 4(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. See App. E. □

7.2 An Improvement to $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$

Algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ from Fig. 4(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[6]—i.e., line (6) of Fig. 4(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 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].

Fig. 4(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. 4(a) and (b) are highlighted in gray. 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 (*lower*) 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 (see line (7) of Fig. 4(b)).

In the proof of correctness of $\widehat{\alpha}_{\mathrm{KS}}$ in App. E, it is convenient to abbreviate $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})]$ as "*upper*," and restate invariant (2) as $\mathit{upper} \sqsupseteq \widehat{\alpha}(\varphi)$. In $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER}}$, the over-approximation of $\widehat{\alpha}(\varphi)$ is obtained by materializing the ghost variable *upper* from invariant (2) of §7.1 as an actual variable. $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ initializes *upper* to $\top$ on line (3) of of Fig. 4(b). At any stage in the algorithm, $\mathit{upper} \sqsupseteq \widehat{\alpha}(\varphi)$ holds. By exactly the same argument given in Lem. E.2, it is

$i + 2) \ldots \texttt{rows}(\mathit{lower})]$ denotes the empty set of constraints, which equals $\top$.

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

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, $p \not\sqsupseteq upper$, which guarantees progress because $p \sqcap upper \sqsubset upper$. Termination is guaranteed by the same argument used in Thm. 7.3.

In case of a decision-procedure timeout, $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$ 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 $lower[(\texttt{rows}(lower) - i + 2) \ldots \texttt{rows}(lower)] \sqsupseteq \widehat{\alpha}(\varphi)$ always holds. Consequently, the assignment $upper \leftarrow lower[(\texttt{rows}(lower) - i + 2) \ldots \texttt{rows}(lower)]$ need only be performed if line (7) is reached, and neither of the assignments on lines (3) and (10) need to be performed explicitly.

7.3 Symbolic Abstraction for the MOS Domain

It was not previously known how to perform symbolic abstraction for MOS. Using $\widehat{\alpha}_{\mathrm{KS}}$ in conjunction with the algorithms from §3 and §4.4, we can obtain an algorithm for $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ as follows:

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

8. 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: $\widehat{\alpha}_{\mathrm{KS}}$ (§7), 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 in the program's control-flow graph
   —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 (§5.1.5), 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

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.

application needs to obtain an affine relation that over-approximates $\gamma_{\text{MOS}}(\mathcal{B})$.[7] 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.

To address the questions posed above, we performed two kinds of experiments. Both 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.

§8.1 describes an experiment that involved performing reinterpretation on a large corpus of x86 instruction instances; this experiment answers questions (1) and (2). §8.2 describes an experiment in which flow-sensitive, context-sensitive, interprocedural affine-relation analysis was performed on nine Windows utilities, using different sets of abstract transformers; this experiment answers question (3). §8.3 discusses how the evaluation order chosen by a dataflow analyzer can affect the precision of the results obtained with KS abstract transformers, due to the fact that the KS domain is non-distributive.

8.1 Reinterpretation of Individual Instructions

Experimental Setup. 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 2008; 2013],[8] 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*.

[7] The process by which Müller-Olm and Seidl extract *one-vocabulary* affine relations from an MOS value is not stated as an algorithm in their paper [2007]; however, an algorithm is implicit in the discussion, lemmas, and theorems of §2 of their paper.

In our case, we use the algorithm from §4.2 to convert an MOS value M to an over-approximating *two-vocabulary* KS value K. The reason that this algorithm extracts affine relations from M is because the set of rows in K can be read off as the desired affine relations. Moreover, *one-vocabulary* affine relations can be obtained from K by projecting K onto the desired vocabulary.

[8] The algorithm for obtaining MOS transformers via reinterpretation is similar to the reinterpretation method for obtaining KS transformers via reinterpretation given in §6. Reinterpretation for MOS is sketched in [Lim and Reps 2013, §3.1.4.2 and §4.1.2].

Total instructions	MOS-reinterpretation time (seconds)	KS-reinterpretation time (seconds)
19,066	23.3	382.9

Fig. 6. Comparison of the performance of MOS-reinterpretation and KS-reinterpretation for x86 instructions.

Identical precision	MOS-reinterpretation more precise	KS-reinterpretation more precise	Incomparable	Total
18,190	0	876	0	19,066

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

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.

Experimental Results. Figs. 6 and 7 summarize the results of the per-instruction experiment. They answer questions (1) and (2) posed at the beginning of §8.

—KS-reinterpretation created an abstract transformer that was more precise than the one created by MOS-reinterpretation for about 4.59% of the instructions. MOS-reinterpretation never created an abstract transformer that was more precise than the one created by KS-reinterpretation, and there were no cases in which the two abstract transformers were incomparable.

—However, MOS-reinterpretation is much faster: to generate abstract transformers for the entire corpus of instructions, MOS-reinterpretation is about 16.4 times faster than KS-reinterpretation.

Example 8.1. 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:

$$\varphi_\iota \stackrel{\text{def}}{=} \left(\texttt{ebx}' = \left(\begin{array}{l}(\texttt{ebx}\ \&\ \texttt{0xFFFF00FF}) \\ |\ ((\texttt{ebx} + 256 * (\texttt{eax}\ \&\ \texttt{0xFF}))\ \&\ \texttt{0xFF00})\end{array}\right)\right) \\ \wedge (\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx}). \tag{21}$$

Eqn. (21) shows that the semantics of the instruction involves non-linear bit-masking operations.

The abstract transformer created via MOS-reinterpretation represents the function that havocs `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

$$\left[\begin{array}{cccccc|c} \texttt{eax} & \texttt{ebx} & \texttt{ecx} & \texttt{eax}' & \texttt{ebx}' & \texttt{ecx}' & 1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2^{24} & 0 & 0 & -2^{24} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right], \tag{22}$$

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

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

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`.

The result of KS-reinterpretation of the semantics of instruction ι is less precise than $\widehat{\alpha}_{\mathrm{KS}}(\varphi_\iota)$ because (i) a KS$[R; \{i\}]$ value can hold onto only a limited class of properties of a subexpression of the specification of the semantics of ι, and (ii) this lack of precision propagates to enclosing subexpressions. Roughly speaking, the semantics of ι involves the following TSL expression:

```
let a = regAccess(S, BH());
    b = regAccess(S, AL());
    c = a + b;
in ( updateState(S, BH(), c) )
```

The subexpression regAccess(S, BH()) accesses the second-least significant byte of register `ebx`. For the domain KS$[R; \{i\}]$, R is the set of 32-bit registers. Unfortunately, KS$[R; \{i\}]$ cannot express a constraint on the value of i in terms of input register `ebx`, and hence the KS-reinterpretation of regAccess(S, BH()) is $\top$ (in the domain KS$[R; \{i\}]$). In contrast, the subexpression regAccess(S, AL()) accesses the least-significant byte of `eax`. In this case, the equality relationship is captured precisely as the KS$[R; \{i\}]$ value

$$\left[\begin{array}{cccc|c} i & \texttt{eax} & \texttt{ebx} & \texttt{ecx} & 1 \\ \hline -2^{24} & 2^{24} & 0 & 0 & 0 \end{array}\right].$$

Program Name	Measures of Size						
					WPDS Rules		
	Instrs	Procs	BBs	Branches	Δ_0	Δ_1	Δ_2
write	322	11	184	26	11	151	48
finger	562	19	318	48	19	275	72
subst	1119	17	627	74	17	556	128
label	1193	17	591	103	17	579	98
chkdsk	1494	19	805	119	19	769	136
convert	1953	39	1031	161	39	1041	119
route	2125	41	1006	243	41	1019	181
comp	2403	36	1279	224	36	1297	170
logoff	2484	47	1157	306	47	1272	155
setup	4745	68	1876	596	68	2233	197

Fig. 8. Program information for the application code only. (Information about the library code called by the applications is given in Fig. 9.) 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).

However, the lack of precision in the value of a propagates to c: i.e., the evaluation of c = a + b yields $\top$. Finally, updateState(S, BH(), $\top$) can only capture the fact that the least-significant byte of ebx′ equals the least-significant byte of ebx. Consequently, the final KS$[R; R']$ value obtained for ι is the one shown in Eqn. (22). □

8.2 Interprocedural Analysis

Experimental Setup. We performed flow-sensitive, context-sensitive, interprocedural affine-relation analysis on the executables of nine Windows utilities, using three 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 $\widehat{\alpha}_{\mathrm{KS}}$. (See Remark 8.2 for more details about the symbolic-abstraction method.)

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.

Program Name	Measures of Size						
					WPDS Rules		
	Instrs	Procs	BBs	Branches	Δ_0	Δ_1	Δ_2
write	311967	5319	126847	35264	5319	139486	17811
finger	98416	1487	38088	11137	1487	42905	4951
subst	79871	1133	30955	9211	1133	35358	3891
label	84131	1174	32522	9703	1174	37173	4109
chkdsk	84211	1176	32645	9702	1176	37276	4126
convert	84951	1203	32999	9761	1203	37669	4126
route	319631	5336	131490	36492	5336	144361	18763
comp	84221	1228	32911	9666	1228	37451	4114
logoff	79590	1103	30373	9438	1103	35299	3635
setup	66652	851	24389	7759	851	28633	2902

Fig. 9. Program information for the library code called by the applications. 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).

Fig. 8 lists several size parameters of the executables (number of instructions, procedures, basic blocks, branches, and number of WPDS rules), when the code for libraries called by the application is not included. Fig. 9 lists size parameters for the library code called by the applications. 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.

As discussed in §6.3, for a basic block $B = \iota_1; \ldots; \iota_m$, there are two approaches to performing KS$[V; V']$ reinterpretation: composed reinterpretation and cascaded reinterpretation. Our experiments use cascaded reinterpretation for the following reasons:

—In the case of $\widehat{\alpha}_{\mathrm{KS}}$, 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 $w^B_{\mathrm{KS}} \leftarrow \widehat{\alpha}_{\mathrm{KS}}(\varphi_{\mathcal{B}})$. Letting $QFBV$ denote the type of quantifier-free bit-vector formulas, the $QFBV$ reinterpretation of `interpInstr` has the type

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

Symbolic execution is performed by cascaded reinterpretation:

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

—For our experiments, we wanted to control for any effects on precision that might be due solely to the use of cascaded reinterpretation or composed reinterpretation; thus, we used cascaded reinterpretation for *all* of the weight-generation methods.[9]

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

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

caller-save and callee-save registers across call sites. Running a WPDS-based analysis to find the least fixed-point 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 of the $\widehat{\alpha}_{\mathrm{KS}}$ method for weight generation (see §8.2), KS-reinterpretation is used for the library-code weights in the $\widehat{\alpha}_{\mathrm{KS}}$-based analysis. This approach made it feasible to create sound basic-block transformers for the library code for the $\widehat{\alpha}_{\mathrm{KS}}$-based analysis, so that we could analyze the application plus library code for all three weight-generation methods. Thus, in §8.2, when we compare analysis results obtained via the $\widehat{\alpha}_{\mathrm{KS}}$-based analysis against those obtained via KS-reinterpretation, we are measuring the impact of using more precise abstract transformers in the application code only.

The analysis also needed to model system calls. In our experiments, system calls were modeled approximately (albeit unsoundly, in general) by "`havoc(eax')`".

To implement $\widehat{\alpha}_{\mathrm{KS}}$, 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.

Remark 8.2. This remark 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, the method of §6, is performed first.

(2) The generalized-Stålmarck algorithm of Thakur and Reps [2012]—instantiated for the KS domain—is applied next, *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) The final step is to apply $\widehat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}$, from Fig. 4(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. □

	WPDS-construction time for the application code			
	MOS-reinterp	KS-reinterp	$\widehat{\alpha}_{\mathrm{KS}}$	
name	WPDS	WPDS	WPDS	t/o
write	0.234	0.905	84.942	4(1.90%)
finger	0.219	1.389	248.432	10(2.73%)
subst	0.374	2.371	554.628	6(0.86%)
label	0.421	2.512	551.82	9(1.30%)
chkdsk	0.499	2.917	526.863	9(0.97%)
convert	0.67	3.697	476.83	10(0.83%)
route	0.78	5.382	783.604	19(1.53%)
comp	0.812	5.07	1048.46	9(0.60%)
logoff	0.852	6.677	1131.91	19(1.29%)
setup	1.591	11.794	1924.98	62(2.48%)

Fig. 10. WPDS-construction time for the application code only, using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$ to generate weights for basic blocks. (WPDS-construction time for library code is not included.) The column labeled "t/o" reports the number of WPDS rules for which weight generation timed out during symbolic abstraction via $\widehat{\alpha}_{\mathrm{KS}}$.

	WPDS-construction time for the library code		
	MOS-reinterp	KS-reinterp	$\widehat{\alpha}_{\mathrm{KS}}$
name	WPDS	WPDS	WPDS
write	113.085	748.493	742.544
finger	34.102	238.728	239.789
subst	26.13	190.991	191.132
label	29.188	207.153	216.903
chkdsk	28.657	203.316	203.737
convert	28.782	202.987	202.614
route	109.527	749.222	750.781
comp	28.282	200.46	201.224
logoff	26.338	188.012	189.869
setup	20.904	151.945	151.414

Fig. 11. WPDS-construction time for the library code, using MOS-reinterpretation, KS-reinterpretation, and KS-reinterpretation to generate weights for basic blocks. (Due to the high cost of the $\widehat{\alpha}_{\mathrm{KS}}$ method for weight generation, KS-reinterpretation is used for the library-code weights in the $\widehat{\alpha}_{\mathrm{KS}}$-based analysis.)

Experimental Results. Fig. 10 shows the times for WPDS construction—in particular, the times for constructing the weights that serve as abstract transformers—for the application code only. Column 5 of Fig. 10 shows the number of $\widehat{\alpha}$ calls for which weight generation timed out during $\widehat{\alpha}_{\mathrm{KS}}$. During WPDS construction, if Yices times out during $\widehat{\alpha}_{\mathrm{KS}}$, 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.

The number of WPDS rules is given in Fig. 9; a timeout occurred for about 1.45% of the $\widehat{\alpha}_{\mathrm{KS}}$ calls (computed as a geometric mean[10]).

The experiment showed that the cost of constructing weights via $\widehat{\alpha}_{\mathrm{KS}}$ is high,

[10]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

Prog. name	Performance of Interprocedural Analysis					
	MOS-reinterp		KS-reinterp		$\widehat{\alpha}_{\mathrm{KS}}$	
	post* + path summary	queries at branch points	post* + path summary	queries at branch points	post* + path summary	queries at branch points
write	300.162	0.203	69.264	0.265	69.81	3.713
finger	78.749	0.328	18.814	0.39	18.751	3.479
subst	57.345	0.484	14.556	0.546	14.523	4.93
label	63.476	0.734	17.129	0.748	17.535	7.207
chkdsk	61.09	0.811	15.21	0.858	15.179	108.047
convert	62.182	0.936	15.241	1.139	15.288	20.733
route	312.531	1.732	71.698	2.184	71.557	63.913
comp	63.46	1.731	15.226	1.716	15.304	31.948
logoff	57.23	2.012	14.103	3.151	14.196	52.854
setup	52.01	2.106	12.573	6.147	12.604	277.867

Fig. 12. Performance of WPDS-based interprocedural analysis. The times, in seconds, for performing interprocedural dataflow analysis (i.e., running post* and performing path summary) and finding one-vocabulary affine relations at all branch instructions in the application code, using the MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$-based analyses.

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 167 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 6.1 times slower than creating MOS weights using MOS-reinterpretation. The latter number is different from the 16.4-fold slowdown reported in §8.1 for two reasons: (i) §8.1 reported the cost of creating KS and MOS abstract transformers for individual instructions, whereas in Figs. 10 and 11 the transformers are for basic blocks, and (ii) the WPDS construction times in Figs. 10 and 11 include the cost of creating merge functions for use at procedure-exit sites, which was about the same for KS-reinterpretation and MOS-reinterpretation.

Fig. 10 shows the WPDS-construction time for the application code only; Fig. 11 shows the WPDS-construction time for the library code used by the application. As mentioned earlier, for the $\widehat{\alpha}_{\mathrm{KS}}$-based analysis, we use $\widehat{\alpha}_{\mathrm{KS}}$ to generate weights for basic blocks in the application code, and KS-reinterpretation to generate weights for basic blocks in the library code. Thus, columns 3 and 4 of Fig. 11 show roughly the same times because both analyses use KS-reinterpretation to generate weights for library code. Fig. 12 shows the time for performing interprocedural dataflow analysis via post* and path summary, as well as for finding one-vocabulary affine relations at all branch instructions in the application code.

Figs. 13, 14, and 15 present three studies that compare the precision of the

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	WPDS Weights: MOS-reinterp $\subsetneq$ KS-reinterp	WPDS Weights: KS-reinterp $\subsetneq$ MOS-reinterp	WPDS Weights: $\widehat{\alpha}_{\mathrm{KS}} \subsetneq$ KS-reinterp
write	151	0(0.00%)	0(0.00%)	11(7.28%)
finger	275	0(0.00%)	0(0.00%)	29(10.55%)
subst	556	0(0.00%)	0(0.00%)	60(10.79%)
label	579	0(0.00%)	0(0.00%)	84(14.51%)
chkdsk	769	0(0.00%)	0(0.00%)	86(11.18%)
convert	1041	0(0.00%)	2(0.19%)	131(12.58%)
route	1019	0(0.00%)	5(0.49%)	156(15.31%)
comp	1297	0(0.00%)	6(0.46%)	166(12.80%)
logoff	1272	0(0.00%)	5(0.39%)	200(15.72%)
setup	2233	0(0.00%)	29(1.30%)	472(21.14%)

Fig. 13. Comparison of the precision of the WPDS weights computed using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (E.g., KS-reinterp $\subsetneq$ MOS-reinterp reports the number of rules for which the KS-reinterp weight was more precise than the MOS-reinterp weight.)

Prog. name	Branches	1-Vocabulary Affine Relations at Branch Points: MOS-reinterp $\subsetneq$ KS-reinterp	1-Vocabulary Affine Relations at Branch Points: KS-reinterp $\subsetneq$ MOS-reinterp	1-Vocabulary Affine Relations at Branch Points: $\widehat{\alpha}_{\mathrm{KS}} \subsetneq$ KS-reinterp
write	26	0(0.00%)	0(0.00%)	3(11.54%)
finger	48	0(0.00%)	0(0.00%)	11(22.92%)
subst	74	0(0.00%)	0(0.00%)	12(16.22%)
label	103	0(0.00%)	0(0.00%)	7(6.80%)
chkdsk	119	0(0.00%)	0(0.00%)	4(3.36%)
convert	161	0(0.00%)	0(0.00%)	49(30.43%)
route	243	0(0.00%)	3(1.23%)	47(19.34%)
comp	224	0(0.00%)	0(0.00%)	9(4.02%)
logoff	306	0(0.00%)	24(7.84%)	81(26.47%)
setup	596	0(0.00%)	8(1.34%)	29(4.87%)

Fig. 14. Comparison of the precision of the one-vocabulary affine relations identified by interprocedural analysis as holding at branch points in the application code, using weights created using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (E.g., KS-reinterp $\subsetneq$ MOS-reinterp reports the number of branch points at which the KS-reinterp results were more precise than the MOS-reinterp results.)

analysis results obtained in the application code via MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (In all three cases, the library code was also analyzed.)

Fig. 13 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 13.13% of the WPDS rules (computed as a geometric mean). The "$\widehat{\alpha}_{\mathrm{KS}} \subsetneq$ KS-reinterp" column of Fig. 13 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 §8.1. The numbers in Fig. 13 differ from that study in two ways: (i) Fig. 13 compares the precision of abstract transformers for basic blocks rather than for individual instructions; and (ii) Fig. 13

Prog. name	Procs	2-Vocabulary Procedure Summaries		
		MOS-reinterp $\subsetneq$ KS-reinterp	KS-reinterp $\subsetneq$ MOS-reinterp	$\widehat{\alpha}_{\mathrm{KS}} \subsetneq$ KS-reinterp
write	11	0(0.00%)	0(0.00%)	0(0.00%)
finger	19	0(0.00%)	0(0.00%)	0(0.00%)
subst	17	0(0.00%)	0(0.00%)	0(0.00%)
label	17	0(0.00%)	0(0.00%)	0(0.00%)
chkdsk	17	0(0.00%)	0(0.00%)	0(0.00%)
convert	39	0(0.00%)	0(0.00%)	0(0.00%)
route	40	0(0.00%)	1(2.50%)	0(0.00%)
comp	36	0(0.00%)	0(0.00%)	0(0.00%)
logoff	47	0(0.00%)	0(0.00%)	3(6.38%)
setup	64	0(0.00%)	0(0.00%)	6(9.38%)

Fig. 15. Comparison of the precision of the two-vocabulary affine relations identified by interprocedural analysis as holding at procedure-exit points in the application code, using weights created using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{\mathrm{KS}}$. (E.g., KS-reinterp $\subsetneq$ MOS-reinterp reports the number of procedure-exit points at which the KS-reinterp results were more precise than the MOS-reinterp results.)

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.

Figs. 14 and 15 answer question (3) posed at the beginning of §8:

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

Figs. 14 and 15 summarize the results obtained from comparing the precision of the affine relations identified via interprocedural analysis using the different weights.[11]

In the studies reported in Figs. 13, 14, and 15, we never observed cases in which MOS-reinterpretation produced results that were incomparable with the KS-reinterpretation or $\widehat{\alpha}_{\mathrm{KS}}$ results.

Compared to runs based on either KS-reinterpretation or MOS-reinterpretation, the analysis runs based on $\widehat{\alpha}_{\mathrm{KS}}$ weights identified more precise one-vocabulary affine relations at a substantial number of branch points in the application code (Fig. 14, col. 5). The $\widehat{\alpha}_{\mathrm{KS}}$ analysis results are strictly more precise than the KS-reinterpretation results at 14.21% of all application-code branch points (computed as a geometric mean).

Two-vocabulary affine relations that hold at procedure-exit points describe procedure summaries. The $\widehat{\alpha}_{\mathrm{KS}}$ analysis results are strictly more precise than the KS-reinterpretation results at 1.53% of all application-code procedure exits (computed as a geometric mean)—see Fig. 15, col. 5.

The lower percentage of precision improvement for procedure exits compared to that obtained for branch points can be explained as follows: most precision

[11] 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*".

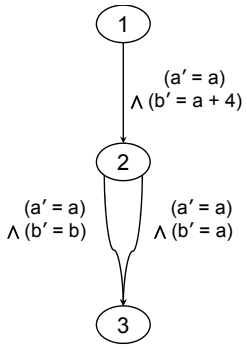


Fig. 16. Simplified version of an example that caused KS results to be less precise than MOS results, due to compose not distributing over join in the KS domain.

improvements obtained at an instruction ι are lost at join points that occur in the control flow of the procedure between ι and the procedure's exit point.

8.3 Imprecision Due to Non-Distributivity of KS

Although it did not show up in the analysis runs reported in §8.2, in some earlier experiments we observed cases in which the MOS-reinterpretation results were more precise than the KS-reinterpretation results (as well as the $\widehat{\alpha}_{\mathrm{KS}}$ results). When we examined such cases, we 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 §5.1.3).

Fig. 16 is a simplified version of the actual transformers that caused the KS-based analyses to return a less-precise element than the MOS-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 `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 "$a' = a \wedge 2^{30}\mathtt{b}' = 2^{30}\mathtt{a}$".

9. RELATED WORK

9.1 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 §8, we were able to apply our methods to interprocedural program analysis.

As shown in §5.1.2, 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, §5.1.2, and §5.1.3; see also §9.2 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 9.1. 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

$$\begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & 1 \end{array} \\ \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \end{array}\right], \end{array} \tag{23}$$

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. (23) we use the following KS element over $\mathbb{Z}^3_4$:

$$\begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & 1 \end{array} \\ \left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \end{array}\right], \end{array} \tag{24}$$

which corresponds to the affine relation $x_1 + x_2 + x_3 + 3 = 0$. The matrix in

Eqn. (24) 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; Bagnara et al. 2008; 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.

Müller-Olm and Seidl [2005b] have given improved algorithms for Granger's domain, along with an interprocedural extension. The abstract domain they use for interprocedural analysis is similar to what we have called the MOS domain in this paper: each domain element e is a finite set of "basis" matrices, and e denotes the set of matrices that are in the span of the basis matrices.

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.

9.2 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 2005c] 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].

9.3 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 $\widetilde{\alpha}_{\mathrm{MOS}}(\varphi)$ (§7.3).

10. 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., $\gamma_{\mathrm{KS}}(v_{\mathrm{KS}}) \subseteq \gamma_{\mathrm{MOS}}(v_{\mathrm{MOS}})$—and to convert an MOS element w_{MOS} to an over-approximating KS element w_{KS}—i.e., $\gamma_{\mathrm{MOS}}(w_{\mathrm{MOS}}) \subseteq \gamma_{\mathrm{KS}}(w_{\mathrm{KS}})$.

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. §7 discusses symbolic abstraction for the KS domain, which provides not only a more global approach to creating KS abstract transformers, but one that can attain the fundamental limits on precision that abstract-interpretation theory establishes. §8 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. 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.

The results presented in the paper provide insight on the range of options that one has for performing affine-relation analysis in a program analyzer, and should thereby serve as a guide for anyone interested in creating an analysis component for performing ARA.

Acknowledgments. We thank Evan Driscoll and Aditya Thakur for their comments on a draft of this paper. We thank the referees for their extensive comments on the submission, which have helped us to improve the presentation. We thank Referee 1 for suggesting the KS-to-MOS conversion method presented in §4.4.1 as an alternative to the one given in §4.4.2.

REFERENCES

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.* 219–235.

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *SCP 72,* 1–2, 3–21.

Bouajjani, A., Esparza, J., and Touili, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *POPL.* 62–73.

Burstall, R. 1969. Proving properties of programs by structural induction. *Comp. J. 12,* 1, 41–48.

Clauss, P. 1996. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Int. Conf. Supercomputing*. 278–285.

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*. 198–215.

Fahringer, T. 1998. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing 12,* 3, 227–252.

Fredrikson, M. and Jha, S. 2013. Personal communication.

Granger, P. 1989. Static analysis of arithmetical congruences. *Int. J. of Comp. Math. 30,* 3–4, 165–190.

Gulwani, S. and Necula, G. 2003. Discovering affine equalities using random interpretation. In *POPL*. 74–84.

Gulwani, S. and Necula, G. 2005. Precise interprocedural analysis using random interpretation. In *POPL*. 324–337.

Howell, J. 1986. Spans in the module $(\mathbb{Z}_m)^s$. *Linear and Multilinear Algebra 19,* 1, 67–77.

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*. 281–293.

King, A. and Søndergaard, H. 2010. Automatic abstraction for congruences. In *VMCAI*. 197–213.

Knoop, J. and Steffen, B. 1992. The interprocedural coincidence theorem. In *CC*. 125–140.

Lal, A. and Reps, T. 2006. Improving pushdown system model checking. In *CAV*. 343–357.

Lal, A., Reps, T., and Balakrishnan, G. 2005. Extended weighted pushdown systems. In *CAV*. 434–448.

Lim, J. and Reps, T. 2008. A system for generating static analyzers for machine instructions. In *CC*. 36–52.

Lim, J. and Reps, T. 2013. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *TOPLAS 35,* 1 (Apr.), 4.

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*. 330–341.

Müller-Olm, M. and Seidl, H. 2005a. Analysis of modular arithmetic. In *ESOP*. 46–60.

Müller-Olm, M. and Seidl, H. 2005b. A generic framework for interprocedural analysis of numerical properties. In *SAS*. 235–250.

Müller-Olm, M. and Seidl, H. 2005c. 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*. 156–171.

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

Pugh, W. 1994. Counting solutions to Presburger formulas: How and why. In *PLDI*. 121–134.

Reps, T., Sagiv, M., and Yorsh, G. 2004. Symbolic implementation of the best transformer. In *VMCAI*. 252–266.

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.), 206–263.

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, 189–233.

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

Tawbi, N. 1994. Estimation of nested loop execution time by integer arithmetic in convex polyhedra. In *Int. Parallel Processing Symp.* 217–221.

Thakur, A., Elder, M., and Reps, T. 2012. Bilateral algorithms for symbolic abstraction. In *SAS*. 111–128.

Thakur, A. and Reps, T. 2012. A method for symbolic computation of abstract operations. In *CAV*. 174–192.

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 $\left(M^{-1}\right)^t$.

THEOREM 3.4 *For any matrix M,* $\text{null}^t M = \text{row}\, M^\perp$ *and* $\text{row}\, M = \text{null}^t M^\perp$.

PROOF. Let L, D, and R be the diagonal decomposition of M (see Defn. 3.1, and construct T from D as in Lem. 3.3. Recall that L is invertible. To see that $\text{row}\, M = \text{null}^t M^\perp$,

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

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 set of coefficients $\{v_B \mid B \in \mathcal{B}\}$, we have

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

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 $v = (v_1, \ldots, v_{|\mathcal{B}|})$ be the vector created by listing the coefficients $\{v_B \mid B \in \mathcal{B}\}$ in the same order as the G_B were stacked to create $\mathbb{G}$. Now consider the following vector, $\vec{V} = (v_1 \left[\begin{array}{c|c} 1 & x \end{array}\right], \ldots, v_{|\mathcal{B}|}\left[\begin{array}{c|c} 1 & x \end{array}\right])$, as a $(1 \times (|\mathcal{B}|(k+1)))$-vector of coefficients

for the rows of $\mathbb{G}$:

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

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

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.

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

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

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

$\square$

Theorem 4.6 *For $G \in AG$, $\gamma_{AG}(G) \subseteq \gamma_{MOS}(\textsc{Shatter}(\textsc{MakeExplicit}(G)))$.*

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

$$\gamma_{\text{AG}}(G) \subseteq \gamma_{\text{MOS}}(\text{SHATTER}(\text{MAKEEXPLICIT}(G))).$$

□

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

□

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

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

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

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

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

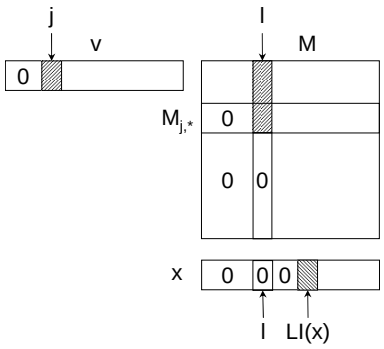
$$(R+S) \perp (\text{null}^t G_R \cap \text{null}^t G_S).$$

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, $(R+S) \perp (R' \cap S')$. $\square$

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 $LI\left(M_{LI(y),*}\right) \geq LI\,(x)$. Consequently, x can be stated as a linear combination of rows with leading indexes $LI\,(x)$ or greater; i.e., $x \in \operatorname{row}([M]_{LI(x)})$. $\square$

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

PROOF. We know that $\operatorname{row} M \perp \operatorname{null}^t M$, and that $\operatorname{row}([M]_i) \perp \operatorname{null}^t([M]_i)$. Let E_i be the module space generated by $\{e_j \mid j < i\}$, and let F_i be the module space generated by $\{e_j \mid j \geq i\}$. Clearly, $E_i \perp F_i$, and thus, by Lem. C.2,

$$(\operatorname{null}^t M + E_i) \perp (\operatorname{row} M \cap F_i).$$

By Thm. C.5, we have that $\text{row}([M]_i) = (\text{row}\, M \cap F_i)$. Consequently,

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

and thus, by Lem. C.4, we have

$$\text{null}^t([M]_i) = (\text{null}^t\, M + E_i), \tag{25}$$

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

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

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 $\left[\begin{smallmatrix} x \\ 1 \end{smallmatrix}\right]$ 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. CORRECTNESS OF $\widehat{\alpha}^{\uparrow}_{\text{KS}}$

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

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

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

Both invariants are established before the loop is entered on line (4) of Fig. 4(a): (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 $\mathit{lower}[(\texttt{rows}(\mathit{lower}) - i + 2) \ldots \texttt{rows}(\mathit{lower})] = \top \sqsupseteq \widehat{\alpha}(\varphi)$.[12]

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

Lemma E.1. The assignment "$i \leftarrow i + 1$" on line (9) of Fig. 4(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. □

[12] 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 $\top$.

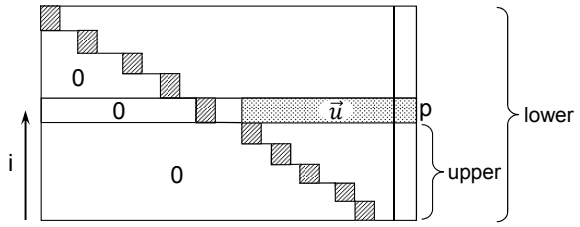


Fig. 17. Depiction of the structure of *lower* during $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$. The shaded blocks running diagonally represent the leading values of the different rows. The dotted block labeled $\vec{u}$ represents a portion of row p.

Lemma E.2. The assignment "$i \leftarrow i + 1$" on line (9) of Fig. 4(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,

$$\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!]) \sqsubseteq \alpha([\![p]\!]) = \widehat{\alpha}(\widehat{\gamma}(p)). \tag{26}$$

The structure of *lower* is depicted in Fig. 17. Row $p = lower[(\texttt{rows}(lower) - i + 1)]$ 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., $lower[(\texttt{rows}(lower) - i + 1) \ldots \texttt{rows}(lower)]$). 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. 17—are exactly the ones with which p is augmented when $\widehat{\alpha}(\widehat{\gamma}(p))$ is put in Howell form. Consequently, by Eqn. (26) and invariant (2),

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

Hence, after the assignment "$i \leftarrow i + 1$" on line (9) of Fig. 4(a), invariant (2) again holds: $lower[(\texttt{rows}(lower) - i + 2) \ldots \texttt{rows}(lower)] \sqsupseteq \widehat{\alpha}(\varphi)$. □

Lemma E.3. The assignment "$lower \leftarrow lower \sqcup \beta(S)$" on line (12) of Fig. 4(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 $lower \sqcup \beta(S) \sqsubseteq \widehat{\alpha}(\varphi)$, so invariant (1) holds after the assignment on line (12). □

Lemma E.4. The assignment "$lower \leftarrow lower \sqcup \beta(S)$" on line (12) of Fig. 4(a) does not change *upper*.

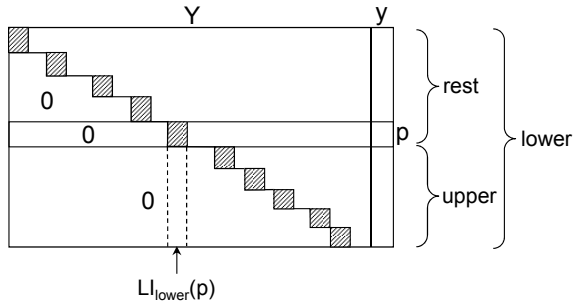
Proof. S is a state that satisfies $\varphi \wedge \neg\widehat{\gamma}(p)$. From Eqns. (19) and (20), we have

$S = [\vec{X} \mapsto \vec{v}, \vec{X}' \mapsto \vec{v}']$ and

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

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

$$\left[\begin{array}{cccc} I & -v & 0 & 0 \\ 0 & (-Yv - y) & Y & y \end{array}\right]. \tag{27}$$

Note that the column $\left[\begin{array}{c} -v \\ (-Yv - y) \end{array}\right]$ is the only column that causes matrix (27) 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), 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 $S \in \gamma(\mathit{upper})$.

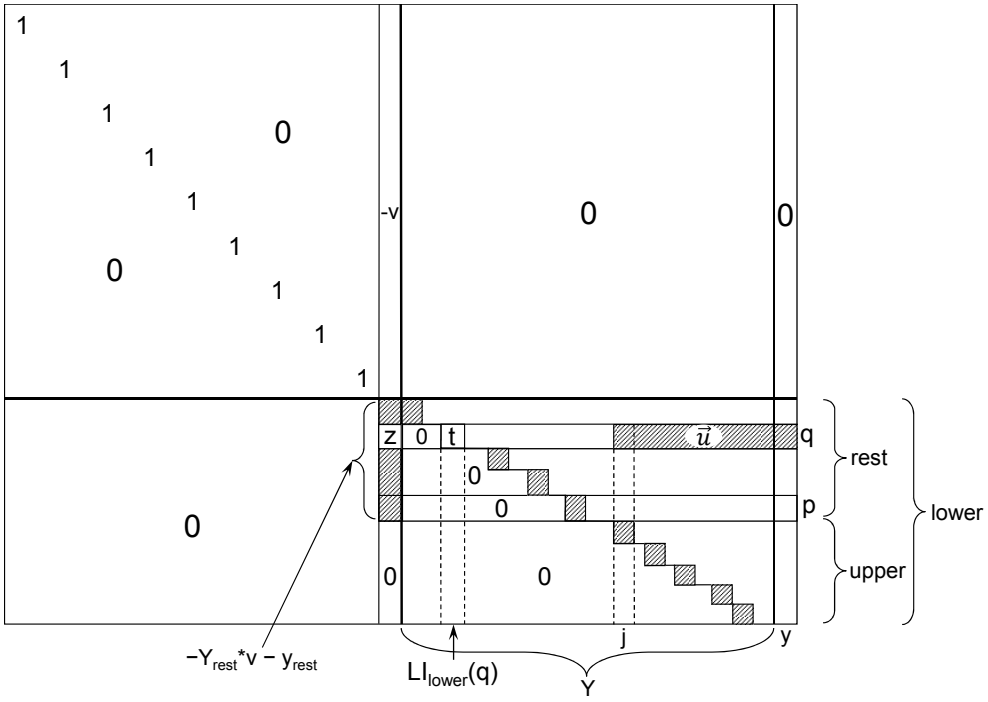


Fig. 18. Depiction of the structure of the partially Howellized matrix (28) that arises during the operation $lower \sqcup \beta(S)$.

The latter fact can be expressed as $S \models \widehat{\gamma}(upper)$, or in matrix terms, as $\left[\, Y_{upper} \,\middle|\, y_{upper} \,\right] \begin{bmatrix} v \\ 1 \end{bmatrix} = 0$. Therefore, matrix (27) has the following structure

$$\begin{bmatrix} I & -v & 0 & 0 \\ 0 & (-Y_{rest} * v - y_{rest}) & Y_{rest} & y_{rest} \\ 0 & 0 & Y_{upper} & y_{upper} \end{bmatrix}, \tag{28}$$

which is depicted in more detail in Fig. 18.

We now want to argue that none of the remaining steps carried out to finish putting matrix (28) 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 (28) 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 (28) to enforce Defn. 2.1(4) cannot change *upper* either. For instance, consider a row such as row q in Fig. 18, 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 (28), $\left[\begin{array}{ccccccccc} 0 & \ldots & 0 & z & 0 & \ldots & 0 & t & \ldots & \vec{u} \end{array}\right]$. 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 $\left[\begin{array}{cccc} 0 & \ldots & 0 & 2^s\vec{u} \end{array}\right]$. $\square$

From Lem. E.4, we conclude the following:

Corollary E.5. The assignment "$lower \leftarrow lower \sqcup \beta(S)$" on line (12) of Fig. 4(a) maintains invariant (2).

THEOREM 7.3 *If algorithm $\widehat{\alpha}^{\uparrow}_{KS}$ from Fig. 4(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 E.1–E.3 and Cor. E.5. Thus, if algorithm $\widehat{\alpha}^{\uparrow}_{\mathrm{KS}}$ from Fig. 4(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 $ans = lower = \widehat{\alpha}(\varphi)$. $\square$