Analysis of Modular Arithmetic

MARKUS MÜLLER-OLM
Westfälische Wilhelms-Universität Münster
and
HELMUT SEIDL
TU München

We consider integer arithmetic modulo a power of 2 as provided by mainstream programming languages like Java or standard implementations of C. The difficulty here is that the ring $\mathbb{Z}_m$ of integers modulo $m = 2^w$, $w > 1$, has zero divisors and thus cannot be embedded into a field. Not withstanding that, we present intra- and interprocedural algorithms for inferring for every program point u, affine relations between program variables valid at u. If conditional branching is replaced with nondeterministic branching, our algorithms are not only sound but also *complete* in that they detect *all* valid affine relations in a natural class of programs. Moreover, they run in time linear in the program size and polynomial in the number of program variables and can be implemented by using the same modular integer arithmetic as the target language to be analyzed. We also indicate how our analysis can be extended to deal with equality guards even in an interprocedural setting.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; D.3.4 [**Programming Languages**]: Processors—*optimization*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: program analysis, modular arithmetic, abstract interpretation, affine relation, interprocedural analysis

1. INTRODUCTION

Analyses for automatically finding linear invariants in programs have been studied for a long time [Karr 1976; Granger 1991; Gulwani and Necula 2003; Leroux 2003; Reps et al. 2003; Müller-Olm and Seidl 2004d,2004b]. With the notable exception of an analysis by Granger [1991], however, none of these analyses can find out that the linear invariant $21 \cdot \mathbf{x} - \mathbf{y} = 1$ holds upon termination of the Java program in Figure 1. Why is this? In order to allow implementing arithmetic operations by the efficient instructions provided by processors, Java, like other common programming languages, performs arithmetic operations for integer types modulo $m = 2^w$ where

Author's address: Markus Müller-Olm, Westfälische Wilhelms-Universität Münster, Institut für Informatik, Fachbereich Mathematik und Informatik, Einsteinstraße 62, 48149 Münster, email: `mmo@math.uni-muenster.de`; Helmut Seidl, TU München, Institut für Informatik, I2, 80333 München, Germany, email: `seidl@in.tum.de`

```
class Eins {
    public static void main(String [] argv) {
        int x = 1022611261;
        int y = 0;
        if (argv.length > 0) {
            x = 1;
            y = 20;
        }
        System.out.println("" + (21*x-y));
    }
}
```

Fig. 1. Analyses of linear invariants are incomplete.

$w = 32$, if the result expression is of type `int`, and $w = 64$, if the result expression is of type `long` [Gosling et al. 1996, p. 32]. The invariant $21 \cdot \mathbf{x} - \mathbf{y} = 1$ is valid because $21 * 1022611261 = 1$ modulo 2^{32}. In order to work with mathematical structures with good properties, analyses for finding linear invariants typically interpret variables by members from a field, e.g., the set $\mathbb{Q}$ of rational numbers [Karr 1976; Müller-Olm and Seidl 2004d,2004a], or $\mathbb{Z}_p = \mathbb{Z}/(p\mathbb{Z})$ for prime numbers p [Gulwani and Necula 2003]. As illustrated by the above example, such analyses are inherently incomplete with respect to the modulo interpretation used in practice. Worse, analyses based on $\mathbb{Z}_p$ for a fixed prime p alone may yield unsound results.[1] In the small flow graph in Figure 2, for instance, x is a constant at program point 2 if variables take values in $\mathbb{Z}_p$ for a prime number $p > 2$, but it is not a constant if variables take values in $\mathbb{Z}_m$ as p equals 0 modulo p but not modulo m. Interestingly, the given problem is resolved by Granger's analysis, which not only detects all affine relations between integer variables but also all affine *congruence* relations. On the other hand, Granger's analysis is just intraprocedural and a polynomial-time version of his analysis has been presented only recently [Müller-Olm and Seidl 2005a].

Practically important examples of affine relations that are valid modulo 2^w but missed by analyses over fields are properties about divisibility modulo powers of 2. Such properties could, e.g., be useful for checking correct alignment of computed addresses. Note that a number x is divisible by $2^{w'}$ for $w' < w$ if and only if $2^{w-w'} \cdot x = 0$ modulo 2^w. A number x is even (odd), for instance, iff $2^{w-1} \cdot x = 0$ (or $2^{w-1} \cdot x = 1$, respectively). Tom Reps reports that they observed and exploited such properties in their implementation of our analysis in the Codesurfer/x86 tool [Reps et al. 2006; Reps 2006].

In this paper we present intra- and interprocedural analyses that are sound and that interpret completely all assignments with affine right hand sides with respect to arithmetic modulo powers of 2. Our analyses are thus tightly tailored for the arithmetic used in mainstream programming languages. For this arithmetic, our analyses are more precise than analyses based on computing over $\mathbb{Q}$ or $\mathbb{Z}_p$, and, in contrast to analyses based on computing over $\mathbb{Z}_p$ with a fixed prime p, they are

[1] If the primes of the analysis are chosen randomly, the resulting analysis is "probabilistically sound" [Gulwani and Necula 2003].

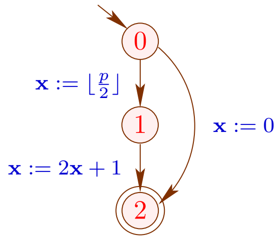

0
1
2
$\mathbf{x} := \lfloor \frac{p}{2} \rfloor$
$\mathbf{x} := 2\mathbf{x} + 1$
$\mathbf{x} := 0$

Fig. 2. $\mathbb{Z}_p$ interpretation is unsound.

sound w.r.t. this arithmetic. Technically, our new analyses are based on the methods from linear algebra that we have studied previously [2004a; 2004d]. The major new difficulty is that unlike $\mathbb{Q}$ and $\mathbb{Z}_p$, $\mathbb{Z}_m$ is no longer a field. In particular, $\mathbb{Z}_m$ has zero divisors implying that not every non-zero element is invertible. Therefore, $\mathbb{Z}_m$ cannot be embedded into a field. Consequently results from linear algebra over fields do not apply to sets of vectors and matrices over $\mathbb{Z}_m$. However, these sets are still *modules* over $\mathbb{Z}_m$. An extensive account of linear algebra techniques for modules over abstract rings can, e.g., be found in [Hafner and McCurley 1991; Storjohann 2000]. Here, we simplify the general techniques to establish the properties of $\mathbb{Z}_m$ that suffice to implement similar algorithms as in [2004a; 2004d].

Besides the soundness and completeness issues discussed above, there is another advantage of our analyses that is perhaps even more important from a practical point of view than precision. For any algorithm based on computing in $\mathbb{Q}$, we must use some representation for rational numbers. When using floating point numbers, we must cope with rounding errors and numerical instability. Alternatively, we may represent rational numbers as pairs of integers. Then we can either use integers of bounded size as provided by the host language or represent integers by arbitrarily long bit strings. In the first case we must cope with overflows in the second one the sizes of our representations may explode. On the other hand, when computing over $\mathbb{Z}_p$, p a prime, special care is needed to get the analysis right. The algorithms proposed in this paper, however, can be implemented using the modulo arithmetic provided by the host language itself. In particular, without any additional effort this totally prevents explosion of number representations, rounding errors, and numerical instability.

This paper is based on [Müller-Olm and Seidl 2005b]. Beyond that conference version we have the following contributions: firstly, we unify the intra- and interprocedural algorithm. Now, also the interprocedural algorithm works by forward iteration such that in absence of procedures it specializes directly to the intraprocedural algorithm. Secondly, we explicate the semi-naive analysis algorithms. Thirdly, we discuss an intra- and interprocedural extension to equality guards. Here, we show that precise analysis of affine relations becomes DEXPTIME-complete in general and still remains PSPACE-complete in absence of procedures. As a way out, we present an *approximate* treatment of affine equality guards both in the intra- and interprocedural setting. We also indicate that the new analyses provide extra useful information beyond analyses over $\mathbb{Q}$ alone, for instance, whether or not a variable is always a multiple of 2.

The paper is organized as follows. In Section 2, we investigate the properties of the ring $\mathbb{Z}_m$ for powers $m = 2^w$ and provide basic techniques for dealing with generating systems of $\mathbb{Z}_m$-modules. In particular we show how to compute (a finite description of) all solutions of a homogeneous system of linear equations over $\mathbb{Z}_m$. These descriptions enable an implementation of our algorithms within $\mathbb{Z}_m$ arithmetic without consulting external references. Then we show how these insights can be used to construct sound and complete program analyses. In Section 3, we introduce basic notions about affine programs and define the collecting semantics of such programs. In Section 4, we argue that, for determining sets of valid affine relations, it suffices to consider the linear hull of the (extended) program states reaching individual program points. In Section 5, we therefore provide efficient algorithms for computing these linear hulls. In Section 6, we provide matching upper and lower complexity bounds for the precise verification of equality invariants and then indicate how the interprocedural analysis of affine relations can be enhanced to approximately take equality guards into account. Finally, in Section 7, we summarize and explain further directions of research.

2. THE MODULAR RING FOR POWERS OF 2

In [Hafner and McCurley 1991; Storjohann 2000], efficient methods are developed for computing various normal forms of matrices over *principal ideal rings* (PIR's). Here, we are interested in the residue class ring $\mathbb{Z}_m$ where m is a power of two which is a special case of a PIR. Accordingly, the general methods from [Hafner and McCurley 1991; Storjohann 2000] are applicable. For our choice of m, however, the ring $\mathbb{Z}_m$ has a special structure. In this section, we show how this structure can be exploited to obtain specialized algorithms in which the computation of (generalized) gcd's (greatest common divisors) is abandoned. Since the abstract values of our program analyses will be *submodules* of $\mathbb{Z}_m^N$ for suitable N, we also determine the exact maximal length of a strictly ascending chain of such submodules. Since we need *effective representations* of modules, we provide algorithms for dealing with sets of generators. We also show how to solve homogeneous systems of linear equations over $\mathbb{Z}_m$ without gcd computations. In the sequel, let $m = 2^w$, $w \geq 1$. We begin with the following observation.

Lemma 2.1. *Assume $a \in \mathbb{Z}_m$ is different from 0. Then we have:*

(*1*) *If a is even, then a is a zero divisor, i.e., $a \cdot b = 0$ (modulo m) for some $b \in \mathbb{Z}_m$ different from 0.*

(*2*) *If a is odd, then a is invertible, i.e., $a \cdot b = 1$ modulo m for some $b \in \mathbb{Z}_m$. The inverse of a can be computed in time $\mathcal{O}(\log(w))$.*

Proof. Let $a = 2 \cdot a'$. Then $a \cdot 2^{w-1} = 2^w \cdot a' = 0$ (modulo m) which proves (1).

For (2), assume a is odd. For $x, y \in \{0, \ldots, m-1\}$, $a \cdot x = a \cdot y$ (modulo m) implies $a \cdot (x - y) = 0$ (modulo m) and, as the absolute value of $x - y$ is strictly smaller than 2^m and a does not contain a factor 2, this implies $x - y = 0$, i.e., $x = y$. Hence the mapping that maps x to $a \cdot x$ (modulo m) is injective on $\{0, \ldots, m-1\}$ and hence also bijective. Therefore, there is a number b with $a \cdot b = 1$ (modulo m), the multiplicative inverse of a in $\mathbb{Z}_m$.

The multiplicative inverse of an odd number a in $\mathbb{Z}_m$ can be computed by Newton's method by iteratively calculating the sequence

$$x_{n+1} = x_n \cdot (2 - a \cdot x_n) \text{ (modulo } m)$$

starting from any odd number x_0, e.g., $x_0 = 1$. This sequence stabilizes after $\mathcal{O}(\log(w))$ iterations with the multiplicative inverse of a [Warren 2003, pp. 195-196]. □

Example 2.2. Consider $w = 32$ and $a = 21$. We use the familiar notation of Java int values as elements in the range $[-2^{31}, 2^{31} - 1]$. Starting from $x_0 = 1$ we obtain the sequence $x_1 = -19$, $x_2 = -7619$, $x_3 = -1219047619$, $x_4 = x_5 = 1022611261$. Hence $b = 1022611261$ is the multiplicative inverse of a in $\mathbb{Z}_m$ for $m = 2^{32}$. □

All algorithms developed in this section avoid computing inverses. Thus, there is no need to compute inverses in the program analysis algorithms developed in Section 5. Nevertheless, the observations of Lemma 2.1 are needed for our reasoning,

For $a \in \mathbb{Z}_m$, we define the *rank* of a as $r \in \{0, \ldots, w\}$ iff $a = 2^r \cdot a'$ for some invertible element a'. In particular, the rank is 0 iff a itself is invertible, and the rank is w iff $a = 0$ (modulo m). Note that the rank of a can be computed by determining the length of suffix of zeros in the bit representation of a. If there is no hardware support for this operation, it can be computed with $\mathcal{O}(\log(w))$ arithmetic operations using a variant of binary search.

A subset $M \subseteq \mathbb{Z}_m^N$ of vectors[2] $[x_1, \ldots, x_N]^t$ with entries x_i in $\mathbb{Z}_m$ is a $\mathbb{Z}_m$*-module* iff it is closed under vector addition and scalar multiplication with elements from $\mathbb{Z}_m$. A subset $G \subseteq M$ is a *set of generators* of M iff $M = \{\sum_{i=1}^{l} r_i g_i \mid l \geq 0, r_i \in \mathbb{Z}_m, g_i \in G\}$. Then M is *generated* by G and we write $M = \langle G \rangle$.

For a non-zero vector $x = [x_1, \ldots, x_N]^t$, we call i the *leading index* iff $x_i \neq 0$ and $x_{i'} = 0$ for all $i' < i$. In this case, x_i is the *leading entry* of x. A set of non-zero vectors is in *echelon form* iff for all distinct vectors $x, x' \in G$, the leading indices of x and x' are distinct. Every set G in echelon form contains at most N elements. We define the *rank* of a set G in echelon form of cardinality s as the sum of the ranks of the leading entries of the vectors of G plus $(N - s) \cdot w$ (to account for $N - s$ zero vectors). Note that this deviates from the common notion of the rank of a matrix.

Assume that we are given a set $G \subseteq \mathbb{Z}_m^N$ in echelon form together with a new vector x. Our goal is to construct a set $\bar{G}$ in echelon form generating the same $\mathbb{Z}_m$-module as $G \cup \{x\}$. If x is the zero vector, then we simply can choose $\bar{G} = G$. Otherwise, let i and $d \cdot 2^r$ (d invertible) denote the leading index and leading entry of x, respectively. We distinguish several cases:

(1) The leading indices of all vectors $x' \in G$ are different from i. Then we choose $\bar{G} = G \cup \{x\}$. Note that the rank of $\bar{G}$ is strictly smaller than the rank of G due to the additional summand in the definition of the rank.

(2) i is the leading index of some $y \in G$ where the leading entry equals $d' \cdot 2^{r'}$ (d' invertible).

[2] The superscript "t" denotes the *transpose* operation which mirrors a matrix at the main diagonal and changes a row vector into a column vector (and vice versa).

(a) If $r' \leq r$, then we compute $x' = d' \cdot x - 2^{r-r'} d \cdot y$. Thus, the i^{th} entry of x' equals 0. This operation is called a *reduction step* applied to x. We proceed with G and x'.

(b) If $r' > r$, then no reduction step can be applied to x. Instead, we construct a new set G' by replacing y with the vector x. Then we apply a reduction step to y w.r.t. G', i.e., we compute $y' = d \cdot y - 2^{r'-r} d' \cdot x$. Thus, the i^{th} entry of y' equals 0, and we proceed with G' and y'. Note that the rank of G' is strictly smaller than the rank of G.

Both vectors x' and y' to be reduced further in (a) or (b), respectively, have a strictly greater leading index than x. This implies termination of the algorithm after at most N reduction steps.

Eventually, we arrive at a set $\bar{G}$ in echelon form generating the same $\mathbb{Z}_m$-module as $G \cup \{x\}$.

Example 2.3. In order to keep the numbers small, we choose here and in the following examples of this section $w = 4$, i.e., $m = 16$. Consider the vectors $x = [2, 6, 9]^{\mathsf{t}}$ and $y = [0, 2, 4]^{\mathsf{t}}$ with leading indices 1 and 2 and both with leading entry 2. Thus, the set $G = \{x, y\}$ is in echelon form. Let $z = [1, 2, 1]^{\mathsf{t}}$. We want to construct a set of generators in echelon form equivalent to $G \cup \{z\}$. Since the leading index of z equals 1, we compare the leading entries of x and z. The ranks of the leading entries of x and z are 1 and 0, respectively. Therefore, we exchange x in the generating set with z while continuing with $x' = x - 2 \cdot z = [0, 2, 7]^{\mathsf{t}}$. The leading index of x' has now increased to 2 . Comparing x' with the vector y, we find that the leading entries have identical ranks. Thus, we can subtract a suitable multiple of y to bring the second component of x' to 0 as well. We compute $x'' = x' - 1 \cdot y = [0, 0, 3]^{\mathsf{t}}$. We finally return $\bar{G} = \{z, y, x''\}$ as the set in echelon form generating the same module as $G \cup \{z\}$. □

If the set $\bar{G}$ computed by the algorithm for G and x equals G, then the algorithm performs a sequence of reduction steps which are solely applied to x which ultimately results in the 0 vector. If this is the case, we call x *reducible.* If x is reducible, then x is contained in the $\mathbb{Z}_m$-module generated by G. Note, however, that the converse might not be true in general. Thus, x being not reducible w.r.t. a set G of generators in echelon form does not necessarily imply that $x \notin \langle G \rangle$.

Example 2.4. Consider a set of generators consisting of the single vector $z = [8, 1, 3]^{\mathsf{t}}$, and let $x = [0, 2, 6]^{\mathsf{t}}$. Obviously, x is not reducible w.r.t. $\{z\}$. On the other hand, $x = 2 \cdot z$. Thus x is contained in the module generated by $\{z\}$. □

In order to allow us to use reduction for deciding module membership, we work with special sets of generators called *saturated* sets. We call a set of generators G in echelon form *saturated* iff $2^{w-r} \cdot x$ is reducible w.r.t. $G \backslash \{x\}$ for every $x \in G$ where r is the rank of the leading entry of x. Then we have:

Lemma 2.5. *Assume $G \subseteq \mathbb{Z}_m^N$ is a saturated set of generators in echelon form of rank r. Then the following holds for every vector $x \in \mathbb{Z}_m^N$:*

(1) $x \in \langle G \rangle$ iff x is reducible.

(2) For G and x, a saturated set $\bar{G}$ of generators can be constructed such that $\langle G \cup \{x\} \rangle = \langle \bar{G} \rangle$.

Let r' denote the rank of the resulting saturated set $\bar{G}$. Then $r' \leq r$ where $r' = r$ iff $G = \bar{G}$ iff x is reducible w.r.t. G. Moreover, the algorithm runs in time $\mathcal{O}((r - r' + 1) \cdot N \cdot (N + \log(w)))$.

Proof. Obviously, if x is reducible then x is also contained in $\langle G \rangle$. For the reverse implication, assume that $x \in \langle G \rangle$. Let $G = \{z_1, \ldots, z_k\}$. Since $x \in \langle G \rangle$, $x = \sum_{i=1}^{k} a_i z_i$ for suitable $a_i \in \mathbb{Z}_m$. We perform induction on the cardinality k of the set G. If $k = 0$, x equals the zero vector and the assertion is trivially true. Therefore, assume $k > 0$ and z_k is the vector with minimal leading index. If $a_k = 0$, the assertion follows by the induction hypothesis. Otherwise, $a_k = a' \cdot 2^r \neq 0$ for some odd number a'. Let i be the leading index and $b = b' \cdot 2^s$ the leading entry of z_k for some odd number b'. Note that the i^{th} entry of x is $a' \cdot b' \cdot 2^{s+r}$. We distinguish two cases.

If $s + r < w$, the vector $x' = b' \cdot x - 2^r \cdot a' \cdot b' \cdot z_k = b' \cdot (x - a_k z_k)$ is obtained from x in a single reduction step. This vector x' is contained in the module generated by $G' = \{z_1, \ldots, z_{k-1}\}$. By induction hypothesis, x' is reducible w.r.t. G'. Therefore, x is also reducible w.r.t. G.

If on the other hand, $s + r \geq w$, the vector $a_k z_k$ is a multiple of the vector $z' = 2^{w-s} \cdot z_k$. As G is saturated, z' is reducible w.r.t. $G' = \{z_1, \ldots, z_{k-1}\}$. Thus, z' and hence also $a_k z_k$ is contained in the module generated by G'. Therefore, $a_k z_k$ can be represented as $a_k z_k = \sum_{i=1}^{k-1} c_i z_i$. This means that $x = \sum_{i=1}^{k-1} (a_i + c_i) z_i$ and therefore is contained in $\langle G' \rangle$. Hence, the assertion again follows by induction hypothesis. This proves the first assertion.

For the second assertion, we simply apply our reduction algorithm to x and G. If x is found reducible, then nothing must be done. Otherwise, after a few reduction steps, x has been reduced to x' with leading index i and leading entry $b \cdot 2^r$ for some odd b. We consider two cases.

If no generator in G has leading index i, then we add x' to G. The leading entry of x' is annihilated by multiplication with 2^{w-r}. Therefore, we recursively apply the algorithm to $G \cup \{x'\}$ and the vector $x'' = 2^{w-r} x'$. Note that either x'' is already the empty vector or has a leading index greater than i and thus in effect is only reduced further w.r.t. G.

Now assume on the other hand, that there is a vector $z \in G$ with the same leading index i and leading entry $c \cdot 2^s$, c odd, where $s > r$ (otherwise x is simply reduced further). Then we add x' to G and remove z, and now continue with first calling the algorithm for $G \setminus \{z\} \cup \{x'\}$ and $x'' = 2^{w-r} x'$, followed by a call for the resulting saturated set of generators together with z.

This algorithm initiates reduction of additional vectors x'' in order to saturate the computed set of generators whenever a vector in the set of generators is replaced by another one. As each replacement reduces the rank of the set of generators, the number of these additional vectors is bounded by the decrease in rank of the resulting saturated set of generators. Moreover, a reduction sequence applied to a single vector has at most length N where each reduction step requires $\mathcal{O}(1)$ computation of ranks together with $\mathcal{O}(N)$ arithmetic operations. This gives the complexity estimation in the second assertion. □

Accordingly, we obtain the following theorem:

Theorem 2.6. *(1) Every $\mathbb{Z}_m$-module $M \subseteq \mathbb{Z}_m^N$ is generated by some saturated set G of generators of cardinality at most N.*

(2) Given a set G' of generators of cardinality s, a saturated set G of cardinality at most N can be computed in time $\mathcal{O}((s + wN) \cdot N \cdot (N + \log(w)))$ such that $\langle G \rangle = \langle G' \rangle$.

(3) Every strictly increasing chain of $\mathbb{Z}_m$-modules $M_0 \subset M_1 \subset \ldots \subset M_s \subseteq \mathbb{Z}_m^N$, has length $s \leq wN$.

Proof. The second statement follows from our construction of saturated sets of generators in echelon form from Lemma 2.5. Starting from the empty set, which is in echelon form by definition, we successively add the vectors in G'. The complexity is then estimated by summing up the operations of these s inclusions. The given estimate exploits that the sum of the rank differences is bounded by the maximal rank wN as ranks are non-negative.

The first statement trivially follows from the second because M is a finite generator of itself. It remains to consider the third statement. Assume that $M_i \subset M_{i+1}$ for $i = 0, \ldots, s-1$. Consider finite sets G_i of generators for M_i. We construct a sequence of (saturated) sets in echelon form generating the same modules as follows. G'_0 is the saturated set in echelon form constructed from G_0. For $i > 0$, G'_i is obtained from G'_{i-1} by successively adding the vectors in G_i to the set G'_{i-1}. Since $M_{i-1} \neq M_i$, the set G'_{i-1} is necessarily different from the set G'_i for all $i = 1, \ldots, s$. Therefore, the ranks of the G'_i are strictly decreasing. Since the maximal possible rank is wN and ranks are non-negative, the third statement follows. □

It is well-known that the submodules of $\mathbb{Z}_m^N$ are closed under intersection. Ordered by set inclusion they thus form a complete lattice $\mathbf{Sub}(\mathbb{Z}_m^N)$, like the linear subspaces of $\mathbb{F}^N$ for a field $\mathbb{F}$. However, while the height of the lattice of linear subspaces of $\mathbb{F}^N$ is N for dimension reasons, the height of the lattice of submodules of $\mathbb{Z}_m^N$ is precisely wN. By Theorem 2.6, wN is an upper bound for the height and it is not hard to actually construct a chain of this length. An example is this, where e_j denotes the j^{th} unit vector: $\langle 0 \rangle \subset \langle 2^{w-1}e_1 \rangle \subset \langle 2^{w-2}e_1 \rangle \subset \cdots \subset \langle 2^0 e_1 \rangle \subset \langle e_1, 2^{w-1}e_2 \rangle \subset \langle e_1, 2^{w-2}e_2 \rangle \subset \cdots \subset \langle e_1, 2^0 e_2 \rangle \subset \langle e_1, e_2, 2^{w-1}e_3 \rangle \subset \cdots \subset \langle e_1, \ldots, e_N \rangle$. The least element of $\mathbf{Sub}(\mathbb{Z}_m^N)$ is $\{\mathbf{0}\}$, the greatest element is $\mathbb{Z}_m^N$ itself. The least upper bound of two submodules M_1, M_2 is given by

$$M_1 \sqcup M_2 = \langle M_1 \cup M_2 \rangle = \{m_1 + m_2 \mid m_i \in M_i\}.$$

We turn to the computation of the solutions of *homogeneous* systems of linear equations in N variables over $\mathbb{Z}_m$. Here, we consider only the case where the number of equations is at most as large as the number of variables. By adding extra equations with all coefficients equal to zero, we can assume that every such system has precisely N equations. Such a system can be denoted as $A\,\mathbf{x} = \mathbf{0}$ where A is a square $(N \times N)$-matrix $A = [a_{ij}]_{1 \leq i,j \leq N}$ with entries $a_{ij} \in \mathbb{Z}_m$, $\mathbf{x} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^{\mathrm{t}}$ is a column vector of unknowns and and $\mathbf{0} = [0, \ldots, 0]^{\mathrm{t}}$ is the zero vector. Let $\mathbb{L}_A$ denote the set of all solutions of $A\,\mathbf{x} = \mathbf{0}$.

Let us first consider the case where the matrix A is *diagonal*, i.e., $a_{ij} = 0$ for all $i \neq j$. The following lemma deals completely with this case.

Lemma 2.7. *Assume A is a diagonal $(N \times N)$-matrix over $\mathbb{Z}_m$ where the diagonal elements are given by $a_{ii} = d_i \cdot 2^{w_i}$ for invertible d_i ($w_i = w$ means $a_{ii} = 0$).*

The set of solutions of the homogeneous system $A\,\mathbf{x} = \mathbf{0}$ is the $\mathbb{Z}_m$-module generated from the vectors: $l_j = 2^{w-w_i}e_j$ for $j = 1, \ldots, N$ where e_j is the j^{th} unit vector.

Proof. It is easily checked that $A\,l_j = \mathbf{0}$ for $j = 1, \ldots, N$. For proving that these solutions indeed generate $\mathbb{L}_A$ assume that $y = [y_1, \ldots, y_N]^{\mathsf{t}}$ is an arbitrary solution of the homogeneous system. Then in particular, $a_{ii} \cdot y_i = d_i \cdot 2^{w_i} \cdot y_i = 0$ for every i. Thus, y_i must be an annihilator of 2^{w_i} which means that $y_i = y'_i \cdot 2^{w-w_i}$ for some y'_i. Consequently, we can write the vector y also as the linear combination $y = y'_1 \cdot l_1 + \ldots + y'_N \cdot l_N$. We conclude that y is in the $\mathbb{Z}_m$-module generated by the vectors $l_1, \ldots, l_N$. □

In contrast to equation systems over fields, a homogeneous system $A\,\mathbf{x} = \mathbf{0}$ thus may have non-trivial solutions, even if all entries a_{ii} are different from 0. Note that the set of generators in item (2) for the $\mathbb{Z}_m$-module $\mathbb{L}_A$ is trivially saturated. Note also, that sets of generators for homogeneous systems can be computed *without* computing inverses.

Example 2.8. Let $w = 4$, i.e., $m = 16$, and

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}$$

Then the $\mathbb{Z}_m$-module of solutions of $A\,\mathbf{x} = 0$ is generated by the two vectors $l_1 = [8, 0]^{\mathsf{t}}$ and $l_2 = [0, 2]^{\mathsf{t}}$. □

For the case where the matrix A is not diagonal, we adapt the concept of invertible column and row transformations known from linear algebra to bring A into diagonal form. More precisely, we have:

Lemma 2.9. *Let A denote an arbitrary $(N \times N)$-matrix over $\mathbb{Z}_m$. Then we have:*

(1) A can be decomposed into matrices: $A = L \cdot D \cdot R$ where D is diagonal and L, R are $(N \times N)$-matrices that are invertible over $\mathbb{Z}_m$.

(2) W.r.t. this decomposition, x is a solution of $A\ \mathbf{x} = \mathbf{0}$ iff $x = R^{-1}x'$ for a solution x' of the system $D\,\mathbf{x} = \mathbf{0}$.

(3) The matrix D together with the matrix R^{-1} can be computed in time $\mathcal{O}(\log(w) \cdot N^3)$. Computation of inverses is not needed for determining D and R^{-1}.

Proof. In order to argue that every matrix A can indeed be decomposed into a product $A = L \cdot D \cdot R$ for a diagonal matrix D and invertible matrices L, R over $\mathbb{Z}_m$, we recall the corresponding technique over fields from linear algebra first. Recall that the idea for fields is to successively select a non-zero pivot element (i, j) in the current matrix. Since every non-zero element in a field is invertible, the entry d at (i, j) has an inverse d^{-1}. By multiplying the row with d^{-1}, one can bring the entry at (i, j) to 1. Then one can apply column and row transformations to bring all other elements in the same column or row to zero. Finally, by exchanging suitable columns or rows, one can bring the former pivot entry into the diagonal. In contrast, when computing in the ring $\mathbb{Z}_m$, we do not have inverses for all non-zero elements, and even if there are inverses, we would like to avoid their construction. Therefore, we refine the selection rule for the pivot element (i, j) by always selecting an entry $d = 2^r d'$ (where d' is invertible) with *minimal rank* r. Since r has been chosen minimal, all other elements in row i and column j are multiples of 2^r. Therefore, all these entries can be brought to 0 by multiplying the corresponding row or column with d' and then subtracting a suitable multiple of the i^{th} row or j^{th} column, respectively. These elementary transformations are invertible since d' is invertible. Finally, by suitable exchanges of columns or rows, the entry (i, j) can be moved into the diagonal. As in the classical construction for fields, the inverses of the chosen elementary column transformations are collected in the matrix R. while the inverses of the chosen elementary row transformations are collected in the matrix L. All the elementary transformations applied in this algorithm (exchange of columns or rows, multiplication with an invertible element, or adding a multiple of one column/row to another one) are also invertible over $\mathbb{Z}_m$.

Now it should be clear how the matrix D together with the matrix R^{-1} can be computed. The matrix R^{-1} is obtained by starting from the unit matrix and then performing the same sequence of column operations on it as on A. In particular, this provides us with the complexity bound as stated in item (3).

From the decomposition, item (2) follows exactly as in the field case: $A\,x = \mathbf{0}$ iff $L \cdot D \cdot R\,x = \mathbf{0}$ iff $D \cdot R\,x = L^{-1}\,\mathbf{0} = \mathbf{0}$ iff $x = R^{-1}\,x'$ for a solution x' of $D\,\mathbf{x} = \mathbf{0}$. □

Putting Theorem 2.6, Lemma 2.7 and Lemma 2.9 together we obtain:

THEOREM 2.10. *A saturated set G of generators in echelon form for the set $\mathbb{L}_A$ of solutions of a homogeneous equation system $A\,\mathbf{x} = \mathbf{0}$ over $\mathbb{Z}_m$ can be computed without resorting to computation of inverses in time $\mathcal{O}(w \cdot N^2 \cdot (N + \log(w))$.*

PROOF. In order to justify the complexity estimate, we summarize the algorithm for homogeneous equation systems. First, we compute the matrices D and R^{-1} of the decomposition $A = L \cdot D \cdot R$ in time $\mathcal{O}(\log(w) \cdot N^3)$ according to Lemma 2.9. Secondly, we compute the vectors $l_1, \ldots, l_N$ that generate the set of solutions of $D\,\mathbf{x} = 0$ according to Lemma 2.7. Thirdly, we compute $x_i = R^{-1}\,l_i$ for $i = 1, \ldots, N$ such that, by Lemma 2.9 and linearity, $\{x_1, \ldots, x_N\}$ generates the set of solutions of $A\,\mathbf{x} = 0$. Clearly, the effort of Steps 2 and 3 is majorized by the first step. Finally, we compute a saturated set of generators in echelon form for the set of solutions of $A\,\mathbf{x} = 0$ from $\{x_1, \ldots, x_N\}$ according to Theorem 2.6 in time $\mathcal{O}(w \cdot N^2 \cdot (N + \log(w))$. Indeed, the effort for this last step majorizes the effort for the other steps. □

Note that Theorem 2.10 is concerned with computing a set of generators in very specific format, a saturated set in echelon form. The reason is that we do not just want to determine any solution by forming linear combinations. Instead, we insist on a representation G of the $\mathbb{Z}_m$-module of all solutions of the homogeneous system which additionally allows us to decide quickly whether any given vector z is a solution or not. In fact, the latter check simply amounts to reducing z w.r.t. G.

Example 2.11. Consider, for $w = 4$, i.e., $m = 16$, the equation system with the two equations

$$\begin{aligned} 12\mathbf{x}_1 + 6\mathbf{x}_2 &= 0 \\ 14\mathbf{x}_1 + 4\mathbf{x}_2 &= 0 \end{aligned}$$

We start with the matrix of coefficients A_0, and the identity matrix T_0 that is to become the transformation matrix R^{-1}:

$$A_0 = \begin{bmatrix} 12 & 6 \\ 14 & 4 \end{bmatrix}, \; T_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We cannot use $(1,1)$ with entry 12 as a pivot, since the rank of 12 exceeds the ranks of 14 and 6. Therefore we choose $(1,2)$ with entry 6. We bring the entry at $(2,2)$ to 0 by multiplying the second row with 3 and subtracting the first row twice in A_0:

$$A_1 = \begin{bmatrix} 12 & 6 \\ 2 & 0 \end{bmatrix}, \; T_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

By subtracting the second column twice from the first one in A_1 and T_1, we obtain:

$$A_2 = \begin{bmatrix} 0 & 6 \\ 2 & 0 \end{bmatrix}, \; T_2 = \begin{bmatrix} 1 & 0 \\ 14 & 1 \end{bmatrix}$$

Now, we exchange the columns 1 and 2 in A_2 and T_2:

$$A_3 = \begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix}, \; T_3 = \begin{bmatrix} 0 & 1 \\ 1 & 14 \end{bmatrix}$$

Both diagonal elements of A_3 have rank 1. Therefore, according to Lemma 2.7, the two vectors $l_1 = \left[\begin{smallmatrix} 8 \\ 0 \end{smallmatrix}\right]$ and $l_2 = \left[\begin{smallmatrix} 0 \\ 8 \end{smallmatrix}\right]$ generate the module of solutions of the homogeneous system $A_3\,\mathbf{x} = \mathbf{0}$. Consequently, the two vectors $x_1 = T_3\,l_1 = \left[\begin{smallmatrix} 0 \\ 8 \end{smallmatrix}\right]$ and $x_2 = T_3\,l_2 = \left[\begin{smallmatrix} 8 \\ 0 \end{smallmatrix}\right]$ generate the module of solutions of the homogeneous system $A_0\,\mathbf{x} = \mathbf{0}$ (Lemma 2.9). In this example, the set of generators $G = \{x_1, x_2\}$ is already saturated and in echelon form. We conclude that the set of solutions of $A_0\,\mathbf{x} = \mathbf{0}$ (over $\mathbb{Z}_{16}$) is

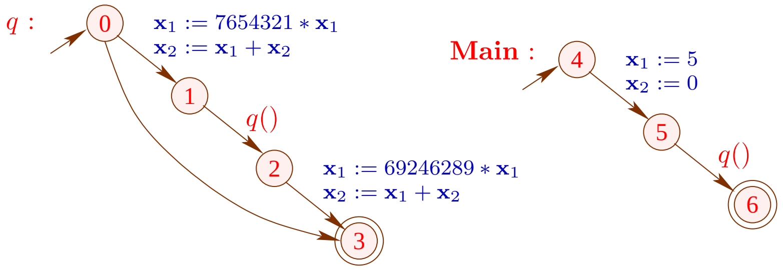
$$\mathbb{L} = \left\{ \begin{bmatrix} 8a \\ 8b \end{bmatrix} \mid a, b \in \mathbb{Z}_{16} \right\} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 8 \end{bmatrix}, \begin{bmatrix} 8 \\ 0 \end{bmatrix}, \begin{bmatrix} 8 \\ 8 \end{bmatrix} \right\}$$

□

3. THE GENERAL SET-UP

In the last section, we have proposed algorithms for reducing sets of generators of $\mathbb{Z}_m$-modules and for solving systems of (homogeneous) linear equations over $\mathbb{Z}_m$. Now, we use these algorithms in order to construct sound and complete analyses of affine relations over $\mathbb{Z}_m$.

Programs are modeled by systems of non-deterministic flow graphs that can recursively call each other as in Figure 3. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of (global) variables the program operates on. Here, we assume that the variables take values in $\mathbb{Z}_m$ for $m = 2^w$. In the programs we analyze, we assume the basic statements to be either *affine assignments* of the form $\mathbf{x}_j := t_0 + \sum_{i=1}^{k} t_i \mathbf{x}_i$ (with $t_i \in \mathbb{Z}_m$ for $i = 0, \ldots, k$ and $\mathbf{x}_j \in \mathbf{X}$) or *non-deterministic assignments* of the form $\mathbf{x}_j :=?$ (with $\mathbf{x}_j \in \mathbf{X}$). It is to reduce the number of program points in the example, that we annotated the edges in Figure 3 with sequences of assignments. Also, we use assignments $\mathbf{x}_j := \mathbf{x}_j$ which have no effect onto the program state as skip-statements and omit these in pictures. For the moment, skip-statements are used to



q :
0
$\mathbf{x}_1 := 7654321 * \mathbf{x}_1$
$\mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$
1
q()
2
$\mathbf{x}_1 := 69246289 * \mathbf{x}_1$
$\mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$
3
Main :
4
$\mathbf{x}_1 := 5$
$\mathbf{x}_2 := 0$
5
q()
6

Fig. 3. An interprocedural program.

abstract guards. Later, we will present methods which treat affine equality guards more precisely. Non-deterministic assignments $\mathbf{x}_j$:=? can be used for handling read statements and as a safe abstraction of statements which our analysis cannot handle precisely, for example of assignments $\mathbf{x}_j := t$ with non-affine expressions t.

In this setting, an *affine program* comprises a finite set Proc of *procedure names* with one distinguished procedure **Main**. Execution starts with a call to **Main**. Each procedure $q \in \mathsf{Proc}$ is specified by a distinct edge-labeled *control flow graph* with a single start point st_q and a single return point ret_q where each edge (u, s, v) has a label s which is either a procedure name q or a deterministic or nondeterministic assignment. Later, we will also consider edges labeled with equality guards.

The basic approach of [Müller-Olm and Seidl 2004d; 2004a; 2005b] which we take up here for the analysis of modular arithmetic, is to construct a precise abstract interpretation of a constraint system characterizing the concrete program semantics. Similar to [Granger 1991; Müller-Olm and Seidl 2004a; 2005a; Müller-Olm et al. 2005], we find it convenient to start from the *collecting* semantics. For that, we model a *state* attained by program execution when reaching a program point or procedure by a k-dimensional (column) vector $x = [x_1, \ldots, x_k]^{\mathrm{t}} \in \mathbb{Z}_m^k$ of ring elements where x_i is the value assigned to variable $\mathbf{x}_i$. For convenience, we consider *extended* states $[1, x_1, \ldots, x_k]^{\mathrm{t}}$ containing an extra component 1. Then every assignment $\mathbf{x}_j := t$, $\mathbf{x}_j \in \mathbf{X}$, $t \equiv t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$, induces a *linear* transformation $[\![\mathbf{x}_j := t]\!] : \mathbb{Z}_m^{k+1} \to \mathbb{Z}_m^{k+1}$ of the extended state which is described by the matrix:

$$[\![\mathbf{x}_j := t]\!] \;=\; \left[\begin{array}{c|c} I_j & 0 \\ \hline t_0 \ldots t_{j-1} & t_j \ldots t_k \\ \hline 0 & I_{k-j} \end{array}\right]$$

where I_j is the identity matrix in $\mathbb{Z}_m^{j^2}$. This definition is readily extended to sets of extended states. Composition of transformations is captured by matrix multiplication. Since linear mappings are closed under composition, the effect of a single run can be represented by one matrix in $\mathbb{Z}_m^{(k+1)^2}$. Since in general, procedures have multiple runs, we model their semantics by *sets* of linear transformations. These are characterized by the constraint system $\mathbf{S}$:

$$\begin{array}{lll} [\mathbf{S}1] & \mathbf{S}(\mathsf{st}_q) \supseteq \{I_{k+1}\} & \\ [\mathbf{S}2] & \mathbf{S}(v) \supseteq \{[\![\mathbf{x}_j := t]\!]\} \cdot \mathbf{S}(u) & \text{for edge } (u, \mathbf{x}_j := t, v) \\ [\mathbf{S}3] & \mathbf{S}(v) \supseteq \{[\![\mathbf{x}_j := c]\!] \mid c \in \mathbb{Z}_m\} \cdot \mathbf{S}(u) & \text{for edge } (u, \mathbf{x}_j :=?, v) \\ [\mathbf{S}4] & \mathbf{S}(v) \supseteq \mathbf{S}(\mathsf{ret}_q) \cdot \mathbf{S}(u) & \text{for edge } (u, q, v) \end{array}$$

In these constraints we write $\cdot$ for the pointwise extension of matrix multiplication to sets of matrices. The set of effects of complete runs through the procedure q is captured by the set of transformations accumulated for the return point ret_q of q. According to $\mathbf{S}1$, this accumulation starts at the start point st_q with the identity transformation. The constraints $\mathbf{S}2$ and $\mathbf{S}3$ deal with affine and nondeterministic assignments, respectively, while the constraints $\mathbf{S}4$ correspond to calls.

Given the effects of procedures, we characterize the sets of extended states reaching program points and procedures by the constraint system $\mathbf{R}$:

$$\begin{array}{lll} [\mathbf{R}1] & \mathbf{R}(\mathsf{st}_{\mathbf{Main}}) \supseteq \{1\} \times \mathbb{Z}_m^k & \\ [\mathbf{R}2] & \mathbf{R}(\mathsf{st}_q) \supseteq \mathbf{R}(u) & \text{for edge } (u, q, \_) \\ [\mathbf{R}3] & \mathbf{R}(v) \supseteq [\![\mathbf{x}_j := t]\!](\mathbf{R}(u)) & \text{for edge } (u, \mathbf{x}_j := t, v) \\ [\mathbf{R}4] & \mathbf{R}(v) \supseteq \bigcup\{[\![\mathbf{x}_j := c]\!](\mathbf{R}(u)) \mid c \in \mathbb{Z}_m\} & \text{for edge } (u, \mathbf{x}_j :=?, v) \\ [\mathbf{R}5] & \mathbf{R}(v) \supseteq \mathbf{S}(\mathsf{ret}_q)(\mathbf{R}(u)) & \text{for edge } (u, q, v) \end{array}$$

Here, application of matrices to vectors is extended to sets in the obvious way. The constraint $\mathbf{R}1$ indicates that we start before the call of **Main** with the full (extended) state space. The constraints $\mathbf{R}2$ indicate that the extended states reaching the start point of a procedure includes all extended states reaching its calls. The constraints $\mathbf{R}3$ through $\mathbf{R}5$ then are completely analogous to a usual forward propagating definition of the *intraprocedural* collecting semantics only that at a call edge the set of transformations obtained for the called procedure is applied (constraints $\mathbf{R}5$).

By the fixpoint theorem of Knaster-Tarski, the constraint systems $\mathbf{S}$ and $\mathbf{R}$ have least solutions. For convenience, we denote the components of these least solutions by $\mathbf{S}(u)$, and $\mathbf{R}(u)$, respectively (u a program point).

4. THE LINEAR ABSTRACTION

The definition of *affine relations* over $\mathbb{Z}_m$ is completely analogous to affine relations over fields. So, an *affine relation* over $\mathbb{Z}_m^k$ is an equation $a_0 + a_1\mathbf{x}_1 + \ldots + a_k\mathbf{x}_k = 0$ for some $a_i \in \mathbb{Z}_m$. As for fields, we represent such a relation by the column vector $a = [a_0, \ldots, a_k]^t$. Instead of a vector space, the set of all affine relations now forms a $\mathbb{Z}_m$-module isomorphic to $\mathbb{Z}_m^{k+1}$. We say that the extended vector $y \in \mathbb{Z}_m^{k+1}$ *satisfies* the affine relation a iff $a \cdot y = 0$ where "$\cdot$" denotes the scalar product. We say that the affine relation a is *valid* for a set $X \subseteq \mathbb{Z}_m^{k+1}$ of extended states iff $a \cdot x = 0$ for all $x \in X$. We observe that a is valid for the set X iff a is valid for the $\mathbb{Z}_m$-module $\langle X \rangle$ generated by X. This follows since the set of states x which satisfy a itself is a $\mathbb{Z}_m$-module.

We conclude that it does not make any difference whether we determine the affine relations valid for the states in $\mathbf{R}(u)$, i.e., the extended states reaching the program point u, or for the states in the linear closure of $\mathbf{R}(u)$. Therefore, we consider the abstraction $\alpha_{\mathbb{Z}_m}$ which abstracts a set $V \subseteq \mathbb{Z}_m^{k+1}$ of extended states by the $\mathbb{Z}_m$-*linear closure* of V:

$$\alpha_{\mathbb{Z}_m}(V) = \langle V \rangle = \{\lambda_1 v_1 + \ldots + \lambda_s v_s \mid s \geq 0, \lambda_i \in \mathbb{Z}_m, v_i \in V\}$$

Due to the extension of states by an extra 0-th component, the abstraction adds all *linear* combinations of vectors in V with the understanding that only those vectors in the closure are meaningful whose 0-th component equals 1.

The linear abstraction has been extensively studied for different rings. In [Granger 1991], it is used with the ring $\mathbb{Z}$ to analyze linear congruence relations. In [Müller-Olm and Seidl 2004a], this abstraction is applied for fields to speed up Karr's analysis [Karr 1976] of affine relations. Note, however, that our interprocedural analyses of affine relations [Müller-Olm and Seidl 2004d; 2005b] over fields and modular rings $\mathbb{Z}_m$, $m = 2^w$, do not directly rely on abstractions of the collecting semantics but on linear abstractions of sets of weakest precondition transformers.

Assume that the $\mathbb{Z}_m$-module $\langle \mathbf{R}(u) \rangle$ is generated by the finite set $G \subseteq \mathbb{Z}_m^{k+1}$. Then we can determine the set of all valid linear relations at X as the set of all solutions of the homogeneous system of equations:

$$\mathbf{a} \cdot x = 0\,, \quad x \in G$$

where $\mathbf{a} = [\mathbf{a}_0, \ldots, \mathbf{a}_k]$ is a row vector of variables. Together with Theorem 2.10 we obtain:

Theorem 4.1. *Assume we are given a generating system G of cardinality at most $k+1$ for the $\mathbb{Z}_m$-module $\langle \mathbf{R}(u) \rangle$. Then we have:*

(*1*) *The affine relation $a \in \mathbb{Z}_m^{k+1}$ is valid at u iff $a \cdot x = 0$ (modulo $\mathbb{Z}_m$) for all $x \in G$.*

(*2*) *A saturated set of generators in echelon form for the $\mathbb{Z}_m$-submodule of all affine relations valid at program point u can be computed in time $\mathcal{O}(w \cdot k^2 \cdot (k + \log(w)))$.* □

Once we are given a set G of generators for the $\mathbb{Z}_m$-module $\mathbf{R}^\sharp(v)$, we can also easily determine all valid affine relations modulo any other $m' = 2^{w'}$ for $1 \leq w' < w$. For this we determine the $\mathbb{Z}_{m'}$-module of all vectors $a \in \mathbb{Z}_{m'}^{k+1}$ that satisfy $a \cdot x' = 0$ (modulo $\mathbb{Z}_{m'}$) for all the vectors x' obtained from the vectors $x \in G$ by reading the components modulo m'. This module can again be computed by solving the corresponding homogeneous equation system, this time over $\mathbb{Z}_{m'}$.

We are left with the task of computing, for every program point u, a generating system for $\langle \mathbf{R}(u) \rangle$. Following the approach in [Müller-Olm and Seidl 2004a; 2005a], we compute this submodule of $\mathbb{Z}_m^{k+1}$ as an abstract interpretation of the constraint system $\mathbf{R}$ for the collecting semantics, i.e., the set of extended states reaching program point u. This is elaborated in the next section.

5. CONSTRUCTING INTERPROCEDURAL ANALYSES

We have seen that for affine programs, the effects of procedures are given by sets of linear transformations, or matrices. As matrices can be viewed as vectors with quadratically many components, we can use the same abstraction α for effects as for sets of extended state vectors. By applying $\alpha_{\mathbb{Z}_m}$ to the constraint systems $\mathbf{S}$ and $\mathbf{R}$, we obtain constraint systems $\mathbf{S}^\sharp$ and $\mathbf{R}^\sharp$:

$$\begin{array}{lll}
[\mathbf{S}^\sharp 1] & \mathbf{S}^\sharp(\mathsf{st}_q) \sqsupseteq \langle\{I_{k+1}\}\rangle & \\
[\mathbf{S}^\sharp 2] & \mathbf{S}^\sharp(v) \;\; \sqsupseteq \langle\{[\![\mathbf{x}_j := t]\!]\}\rangle \cdot \mathbf{S}^\sharp(u) & \text{for edge } (u, \mathbf{x}_j := t, v) \\
[\mathbf{S}^\sharp 3] & \mathbf{S}^\sharp(v) \;\; \sqsupseteq \langle\{[\![\mathbf{x}_j := 0]\!], [\![\mathbf{x}_j := 1]\!]\}\rangle \cdot \mathbf{S}^\sharp(u) & \text{for edge } (u, \mathbf{x}_j :=?, v) \\
[\mathbf{S}^\sharp 4] & \mathbf{S}^\sharp(v) \;\; \sqsupseteq \mathbf{S}^\sharp(\mathsf{ret}_q) \cdot \mathbf{S}^\sharp(u) & \text{for edge } (u, q, v)
\end{array}$$

As in [Müller-Olm and Seidl 2004d; 2004a], the abstract effect of a non-deterministic assignment $\mathbf{x}_j :=?$ can be modeled by the span of the two transformations $[\![\mathbf{x}_j := 0]\!]$ and $[\![\mathbf{x}_j := 1]\!]$.

The constraint system $\mathbf{S}^\sharp$ closely resembles the corresponding constraint systems as presented in [Müller-Olm and Seidl 2004d] and [Müller-Olm and Seidl 2005b]. There, however, the accumulated transformations are interpreted as *weakest precondition transformers* and therefore accumulated from the rear. The constraint system now accumulates values in a forward fashion. Accordingly, the second constraint system $\mathbf{R}^\sharp$ is in the spirit of the forward *intraprocedural* accumulation as used, e.g., in [Müller-Olm and Seidl 2004a]. Thus, in contrast to [Müller-Olm and Seidl 2004d; 2005b], the second constraint system *directly* speaks about abstract sets of values and not about abstract sets of transformations:

$$\begin{array}{llll}
[\mathbf{R}^\sharp 1] & \mathbf{R}^\sharp(\mathsf{st}_{\mathbf{Main}}) & \sqsupseteq \mathbb{Z}_m^{k+1} & \\
[\mathbf{R}^\sharp 2] & \mathbf{R}^\sharp(\mathsf{st}_q) & \sqsupseteq \mathbf{R}^\sharp(u) & \text{for edge } (u, q, \_) \\
[\mathbf{R}^\sharp 3] & \mathbf{R}^\sharp(v) & \sqsupseteq [\![\mathbf{x}_j := t]\!]\,(\mathbf{R}^\sharp(u)) & \text{for edge } (u, \mathbf{x}_j := t, v) \\
[\mathbf{R}^\sharp 4] & \mathbf{R}^\sharp(v) & \sqsupseteq [\![\mathbf{x}_j := 0]\!]\,(\mathbf{R}^\sharp(u)) \sqcup & \\
 & & \quad [\![\mathbf{x}_j := 1]\!]\,(\mathbf{R}^\sharp(u)) & \text{for edge } (u, \mathbf{x}_j :=?, v) \\
[\mathbf{R}^\sharp 5] & \mathbf{R}^\sharp(v) & \sqsupseteq \mathbf{S}^\sharp(\mathsf{ret}_q)(\mathbf{R}^\sharp(u)) & \text{for edge } (u, q, v)
\end{array}$$

By the fixpoint theorem of Knaster-Tarski, the constraint systems $\mathbf{S}^\sharp$ and $\mathbf{R}^\sharp$ have least solutions. Again, we denote the components of these least solutions by $\mathbf{S}^\sharp(u)$ and $\mathbf{R}^\sharp(u)$, respectively (u a program point). Abstracting the collecting semantics according to constraint system $\mathbf{R}^\sharp$ has the advantage that it relies on matrices only for procedure calls. This means that we can take advantage from any improvements on the abstractions, e.g., for guards $g = 0$ (g an affine combination) or non-affine assignments which have been proposed for intraprocedural analysis [Karr 1976; Granger 1991].

We verify that the abstraction commutes with application and with composition of transformations. By linearity we have:

Lemma 5.1. *Let $V \subseteq \mathbb{Z}_m^{k+1}$ be a set of vectors and $M, M_1, M_2 \subseteq \mathbb{Z}_m^{(k+1)^2}$ sets of matrices. Then we have*

(1) $\langle\{A\,x \mid x \in V, A \in M\}\rangle = \langle\{A\,x \mid x \in \langle V\rangle, A \in \langle M\rangle\}\rangle$ and

(2) $\langle\{A_1\,A_2 \mid A_i \in M_i\}\rangle = \langle\{A_1\,A_2 \mid A_i \in \langle M_i\rangle\}\rangle$. □

By the fixpoint transfer lemma, we obtain from Lemma 5.1, for the constraint systems $\mathbf{S}^\sharp$ and $\mathbf{R}^\sharp$:

Theorem 5.2. *For every program point u,* *(1)* $\mathbf{S}^\sharp(u) = \langle \mathbf{S}(u) \rangle$ *as well as*

(2) $\mathbf{R}^\sharp(u) = \langle \mathbf{R}(u) \rangle$. □

Theorem 5.2 gives a precise characterization of the linear closure of the collecting semantics through a constraint system. Note that a similar characterization also holds when the program is assumed to take values not in $\mathbb{Z}_m$ but in a field or in the ring $\mathbb{Z}$ or, more generally, in an arbitrary *principal ideal ring* R [Müller-Olm and Seidl 2005a]. In case of the ring $\mathbb{Z}_m$ we have seen that the maximal length of a strictly ascending chain of sub-modules of $\mathbb{Z}_m^r$ is bounded by $r \cdot w$. Furthermore, we recall that every $\mathbb{Z}_m$-submodule M of $\mathbb{Z}_m^r$ can be represented as $M = \langle G \rangle$ for a saturated set G of at most r generators. In order to decide whether a submodule $\langle G \rangle$ is included in another submodule $\langle G' \rangle$ it suffices to test whether each vector $v \in G$ is included in $\langle G' \rangle$. For this test we can simply use our reduction algorithm if the set G' is in saturated echelon form. Therefore, Theorem 5.2 gives rise to an effective analysis over $\mathbb{Z}_m$.

Summarizing, we have:

Theorem 5.3. *Assume p is an affine program with n edges. Then the least solutions of the constraint systems $\mathbf{S}^\sharp$ and $\mathbf{R}^\sharp$ are computable.*

In absence of procedures, the algorithm uses $\mathcal{O}(n \cdot w \cdot k^2 \cdot (k + \log(w)))$ operations. In presence of procedures, the algorithm uses $\mathcal{O}(n \cdot w \cdot k^7 \cdot (k + \log(w)))$. □

Proof. In order to argue about the complexity of the analysis, we will not analyze an ordinary worklist-based fixpoint algorithm, since this would result in much too conservative estimations. Instead, we prefer to consider a *semi-naive* iteration strategy here [Paige and Koenig 1982; Balbin and Ramamohanarao 1987; Fecht and Seidl 1998]. The corresponding semi-naive fixpoint algorithm for computing $\mathbf{S}^\sharp(u)$ for all program points u is shown in Figure 4.

In order to deal uniformly with calls of procedures q as well as deterministic or non-deterministic assignments $\mathbf{x}_j := t$ and $\mathbf{x}_j :=?$, we introduce sets $\mathcal{M}_{\mathbf{S}^\sharp}(s)$ where

$$\begin{array}{lcl} \mathcal{M}_{\mathbf{S}^\sharp}(q) & = & \mathbf{S}^\sharp(\mathsf{ret}_q) \\ \mathcal{M}_{\mathbf{S}^\sharp}(\mathbf{x}_j := t) & = & \{[\![\mathbf{x}_j := t]\!]^\sharp\} \\ \mathcal{M}_{\mathbf{S}^\sharp}(\mathbf{x}_j :=?) & = & \{[\![\mathbf{x}_j := 0]\!]^\sharp, [\![\mathbf{x}_j := 1]\!]^\sharp\} \end{array}$$

Informally, the algorithm works as follows. It maintains a workset W holding pairs (u, M) where u is a node whose set of generators has changed through addition of the matrix M. The fixpoint variables are initialized by setting all entry nodes of procedures to the sets $\{I_{k+1}\}$ and all other variables to the empty set. Accordingly, the workset initially receives the elements $(\mathsf{st}_q, I_{k+1}), q \in \mathsf{Proc}$. In the main loop, the algorithm successively takes pairs (u, M) out of the workset and propagates the new value for u to all uses of the variable $\mathbf{S}^\sharp(u)$ until the workset is empty.

If u is the exit point of the procedure q, then the new matrix M must be propagated to all edges (u', q, v) at which q is called by constraint $\mathbf{S}^\sharp 4$. At every such edge, the algorithm computes the set $\mathit{new} = \{M \cdot M_1 \mid M_1 \in \mathbf{S}^\sharp(u')\}$ consisting of all products of M with matrices in the current set for the entry point of the edge u'. This set consists of all candidate matrices potentially to be added to the set $\mathbf{S}^\sharp(v)$. Thus, we successively check for every $M_2 \in \mathit{new}$ whether or not it is contained in

```
$W = \emptyset$ ;
forall $(u \in N)$  $\mathbf{S}^\sharp(u) = \emptyset$;
forall $(q \in \mathsf{Proc})$ {
    $\mathbf{S}^\sharp(\mathsf{st}_q) = \{I_{k+1}\}$;
    $W = W \cup \{(\mathsf{st}_q, I_{k+1})\}$;
}
while $(W \neq \emptyset)$ {
    $(u, M) = \mathsf{Extract}(W)$;
    if $(u \equiv \mathsf{ret}_q$ with $q \in \mathsf{Proc})$ {
        forall $(u', v$ with $(u', q, v) \in E)$ {
            $new = \{M \cdot M_1 \mid M_1 \in \mathbf{S}^\sharp(u')\}$;
            forall $(M_2 \in new)$
                if $(M_2 \notin \langle \mathbf{S}^\sharp(v) \rangle)$ {
                    $\mathbf{S}^\sharp(v) = \mathsf{Add}(\mathbf{S}^\sharp(v), M_2)$;
                    $W = W \cup \{(v, M_2)\}$;
                }
        }
    }
    forall $(s, v$ with $(u, s, v) \in E)$ {
        $new = \{M_1 \cdot M \mid M_1 \in \mathcal{M}_{\mathbf{S}^\sharp}(s)\}$;
        forall $(M_2 \in new)$
            if $(M_2 \notin \langle \mathbf{S}^\sharp(v) \rangle)$ {
                $\mathbf{S}^\sharp(v) = \mathsf{Add}(\mathbf{S}^\sharp(v), M_2)$;
                $W = W \cup \{(v, M_2)\}$;
                }
    }
}
```

Fig. 4. The semi-naive fixpoint algorithm for $\mathbf{S}^\sharp$.

the module generated by $\mathbf{S}^\sharp(v)$. If not, then we *add* it to $\mathbf{S}^\sharp(v)$ and insert the pair (v, M_2) into the workset.

Then we consider all out-going edges (u, s, v) of u. First, we compute the set $new = \{M_1 \cdot M \mid M_1 \in \mathcal{M}_{\mathbf{S}^\sharp}(s)\}$. With this set *new*, we then proceed as above, i.e., we iteratively check for every M_2 in *new* whether or not it is contained in the module generated by $\mathbf{S}^\sharp(v)$. If not, then we *add* it to $\mathbf{S}^\sharp(v)$ and insert the pair (v, M_2) into the workset.

We make three technical remarks. First, in the real implementation we do not need to resort to some kind of set implementation for the workset W. Instead, an ordinary list will do for W, since the same pair (r, M) is never inserted into W twice. Second, the $\mathbb{Z}_m$-modules $\mathbf{S}^\sharp(u)$ are represented as a saturated set of generators (of cardinality at most $(k+1)^2$) in echelon form. Whenever a new matrix M is checked for containment in $\langle \mathbf{S}^\sharp(u) \rangle$, we compute a new saturated set G of generators in echelon form for $\mathbf{S}^\sharp(u) \cup \{M\}$ by means of the algorithm from Theorem 2.6. If the new set equals the old one, we know for sure that M is contained in $\langle \mathbf{S}^\sharp(u) \rangle$. Otherwise, the new set is different from G and and has lower rank. Since the maximal rank of a triangular set of vectors (with $(k+1)^2$ components) is $(k+1)^2 \cdot w$, we conclude that the number of insertions of pairs into W is bounded by $n \cdot (k+1)^2 \cdot w$. In particular, this implies that the algorithm always terminates. Moreover, for every edge it performs at most $2 \cdot w \cdot (k+1)^4$ matrix

```
forall $(u \in N)$ $\mathbf{R}^\sharp(u) = \emptyset$;
$\mathbf{R}^\sharp(\mathsf{st}_{\mathbf{Main}}) = \{e_0, \ldots, e_k\}$;
$W = \{(\mathsf{st}_{\mathbf{Main}}, e_0), \ldots, (\mathsf{st}_{\mathbf{Main}}, e_k)\}$ ;
while $(W \neq \emptyset)$ {
    $(u, x) = \mathsf{Extract}(W)$;
    forall $(s, v$ with $(u, s, v) \in E)$ {
        if $(s \in \mathsf{Proc})$
            if $(x \notin \langle \mathbf{R}^\sharp(\mathsf{st}_s) \rangle)$ {
                $\mathbf{R}^\sharp(\mathsf{st}_s) = \mathsf{Add}(\mathbf{R}^\sharp(\mathsf{st}_s), x)$;
                $W = W \cup \{(\mathsf{st}_s, x)\}$;
            }
        $new = \{M\,x \mid M \in \mathcal{M}_{\mathbf{S}^\sharp}(s)\}$;
        forall $(x' \in new)$ {
            if $(x' \notin \langle \mathbf{R}^\sharp(v) \rangle)$ {
                $\mathbf{R}^\sharp(v) = \mathsf{Add}(\mathbf{R}^\sharp(v), x')$;
                $W = W \cup \{(v, x')\}$;
                }
            }
        }
    }
```

Fig. 5. The semi-naive fixpoint algorithm for $\mathbf{R}^\sharp$.

multiplications and at most as many additions of a vector to a triangular set. The matrix multiplications can be performed in time $\mathcal{O}(w \cdot k^7)$. By our considerations for Theorem 2.6, the insertions need $\mathcal{O}(w \cdot k^6 \cdot (k^2 + \log(w)))$ arithmetic operations. Multiplying that with the number of fixpoint variables, we arrive at the complexity statement.

Having thus computed the values, $\mathbf{S}^\sharp(u)$, u a program node, it remains to determine the values $\mathbf{R}^\sharp(u)$. Again we use semi-naive iteration to compute the least solution of $\mathbf{R}^\sharp$. The algorithm is shown in Figure 5. It proceeds in the same spirit as the algorithm for computing the values $\mathbf{S}^\sharp(u)$. This time, however, we compute with submodules of $\mathbb{Z}_m^{k+1}$. In fact, this algorithm is the straight-forward extension of the improved version of Karr's intraprocedural algorithm as presented in [Müller-Olm and Seidl 2004a]. According to this algorithm, the set $\mathbf{R}^\sharp(\mathsf{st}_{\mathbf{Main}})$ for the start point of the main procedure is initialized with the unit vectors $\{e_0, \ldots, e_k\}$. The new vectors in the reachability set of a node u are then propagated along every outgoing edge of u. The extension consists in additionally taking procedure calls into account. This means that the new vectors arriving at the start edge u of a call to procedure q must also be propagated to the start point st_q of the called procedure. Also, the new vectors arriving at u must be transformed by the set of generators for $\mathbf{S}^\sharp(\mathsf{ret}_q)$ to determine the new contributions for the end point of the calling edge.

In absence of procedure calls, a similar complexity estimation can be applied as in [Müller-Olm and Seidl 2004a]. the only difference is that the maximal length of a strictly ascending chain now can be a factor w larger.

In presence of procedure calls, a new vector for a program point u amounts for every outgoing edge to at most $(k+1)^2$ matrix vector multiplications and likewise $\mathcal{O}((k+1)^2)$ updates of saturated sets in echelon form. Since there are at most

$n\cdot w\cdot(k+1)$ possible increments, we obtain the complexity $\mathcal{O}(n\cdot w\cdot k^3\cdot(k+\log(w)))$ for the second phase. Since this complexity is dominated by the complexity of computing the least solution of $\mathbf{S}^\sharp$, the assertion of the theorem follows. □

Example 5.4. Consider the interprocedural program from Figure 3 and assume that we want to infer all valid affine relations modulo $\mathbb{Z}_m$ for $m = 2^{32}$, and let c abbreviate 7654321. The linear transformation induced by $s_1 \equiv \mathbf{x}_1 := 7654321 \cdot \mathbf{x}_1; \mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$ and $s_2 \equiv \mathbf{x}_1 := 69246289 \cdot \mathbf{x}_1; \mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$ are:

$$B_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & 0 \\ 0 & c & 1 \end{bmatrix} \qquad B_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c^{-1} & 0 \\ 0 & c^{-1} & 1 \end{bmatrix}$$

since $c \cdot 69246289 = 1 \bmod 2^{32}$. For $\mathbf{S}^\sharp(3)$, we find the matrices I_{k+1} and

$$P_1 = B_2 \cdot B_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & c+1 & 1 \end{bmatrix}$$

None of these is subsumed by the other. The corresponding saturated set of generators in echelon form is given by $G_1 = \{I_{k+1}, P\}$ where

$$P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & c+1 & 0 \end{bmatrix}$$

The next iteration then results in the matrix

$$P_2 = B_2 \cdot P_1 \cdot B_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & (c+1)^2 & 1 \end{bmatrix}$$

Since $P_2 = I_{k+1} + (c+1)\cdot P$, computing a saturated set G_2 of generators in echelon from for G_1 together with P_2 will result in $G_2 = G_1$, and the fixpoint iteration terminates.

In order to determine the affine closure of the collecting semantics, e.g., for the endpoint 6 of **Main**, we find that $\mathbf{R}^\sharp(4)$ is generated by the vectors e_0, e_1, e_2. Successively applying the transformers for $\mathbf{x}_1 := 5$; and $\mathbf{x}_2 := 0$ to e_0 results in the vector:

$$b_0 = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}$$

whereas applying this sequence of transformers to the vectors e_1, e_2 contributes the zero vector only. Therefore, $\{b_0\}$ constitutes a set of generators for $\mathbf{R}^\sharp(5)$ which is obviously in saturated echelon form. Using the set $\{I_{k+1}, P\}$ of generators for $\mathbf{S}^\sharp(3)$, we thus obtain for $\mathbf{R}^\sharp(6)$ the vector b_0 together with:

$$b_1 = P\, b_0 = \begin{bmatrix} 0 \\ 0 \\ 5c+5 \end{bmatrix}$$

This gives us the following equations for the affine relations at the exit of **Main**:

$$\begin{aligned}\mathbf{a}_0 + 5\mathbf{a}_1 \qquad\qquad &= 0\\ (5c+5)\mathbf{a}_2 &= 0\end{aligned}$$

Solving this equation system over $\mathbb{Z}_m$ according to Theorem 2.10 shows that the set of all solutions is generated by:

$$a = \begin{bmatrix} -5 \\ 1 \\ 0 \end{bmatrix} \qquad a' = \begin{bmatrix} 0 \\ 0 \\ 2^{31} \end{bmatrix}$$

The vector a means $-5 + \mathbf{x}_1 = 0$ or, equivalently, $\mathbf{x}_1 = 5$. The vector a' means that $2^{31} \cdot \mathbf{x}_2 = 0$ or, equivalently, $\mathbf{x}_2$ is *even*. Both relations are non-trivial and could not have been derived by using the corresponding analysis over $\mathbb{Q}$. □

6. GUARDS

A draw-back of the interprocedural analyses of Section 5 is that conditional branching is abstracted by non-deterministic choice. A natural class of guards to be taken into account in the context of the linear abstraction are *affine equality guards* of the form $g = 0$ for $g \equiv g_0 + g_1\mathbf{x}_1 + \ldots + g_k\mathbf{x}_k$. Over infinite fields, already the problem of deciding whether a variable $\mathbf{x}$ always equals 0 or not at a given program point u (i.e. validity of the affine relation $\mathbf{x} = 0$ at u) is undecidable in presence of equality guards [Müller-Olm and Seidl 2004a]. Of course, if variables take values in a finite structure like $\mathbb{Z}_m$, $m = 2^w$, the problem becomes trivially decidable, as in principle we can compute the collecting semantics completely. However, as we show next, the problem still stays computationally hard, even in absence of procedures.

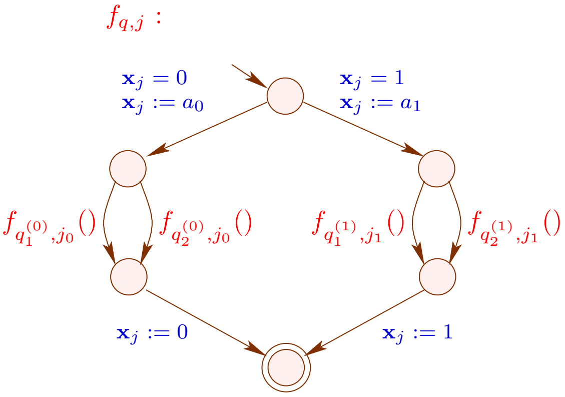


Fig. 6. Procedure constructed for an existential state q with successors $q_1^{(b)}, q_2^{(b)}$ $(b \in \{0,1\})$.

Specifically, we show interprocedural constant detection is DEXPTIME-complete while intraprocedural constant detection is still PSPACE-complete in presence of equality guards. We have:

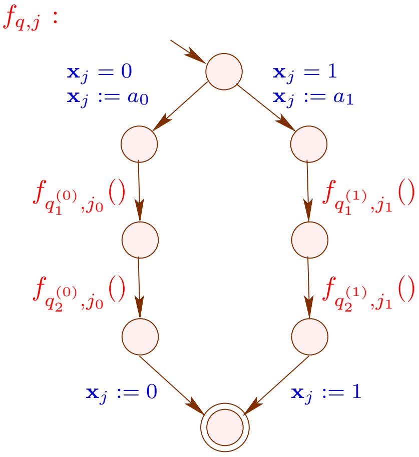


Fig. 7. Procedure constructed for a universal state q with successors $q_1^{(b)}, q_2^{(b)}$ ($b \in \{0, 1\}$).

Theorem 6.1. *Determining for affine programs over $\mathbb{Z}_m$ with equality guards whether at program end, the equality $\mathbf{x}_1 = 0$ holds or not, is DEXPTIME-complete in general. For programs without procedures, this problem is still PSPACE-complete.*

Proof. Let us first consider the upper complexity bounds. In absence of procedures, we can simulate an execution of the program in polynomial space. By guessing such a run we can falsify the equality $\mathbf{x}_1 = 0$ whenever possible. Since PSPACE is closed under complement, the upper bound for intraprocedural analysis follows.

For the exponential upper bound, we rely on a straight-forward representation of effects of procedures through value-tables. Each entry contains the set of possible variable assignments after a call. It is readily checked that all these tables together can be computed in deterministic exponential time and the assertion in presence of procedures follows as well.

For a proof of the exponential lower bound, we consider a linear-space bounded alternating Turing machine. Whether or not such a machine M accepts an input w is DEXPTIME-hard [Chandra et al. 1981]. For simplicity we may assume that the tape of M always has length n and that the tape alphabet is just $\{0, 1\}$. We construct an affine program with guards that has an execution iff M has an accepting run from its initial configuration. The n cells of the tape are represented by the variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$. For every pair of a state q and a head position $j \in \{1, \ldots, n\}$, we introduce a procedure $f_{q,j}$. W.l.o.g. we assume that M has no transitions in an accepting state, and that M in every non-accepting state first reads the cell at the current head position and then depending on the outcome, moves its head and either selects one of two successor states or continues independently with two successor states. The first kind of behavior is chosen for *existential* states, while the latter applies to *universal* states. Assume that q is an existential state which when reading b, sets the j-th cell to a_b, moves the head position to j_b and proceeds with one of the states $q_1^{(b)}, q_2^{(b)}$. Then the corresponding simulating procedure is shown in Figure 6. Similarly, the procedure for a universal state q is shown in Figure 7.

When reading b, the machine M in state q then sets the j-th cell to a_b, moves the head position to j_b and proceeds with the states $q_1^{(b)}, q_2^{(b)}$ in parallel. The program in Figure 7 calls the corresponding procedures one after the other in order to ensure that both execution of M succeed. Finally, the procedure $f_{q,j}$ for an accepting state q immediately returns, i.e., consists of a single program point only. Note that, according to Figures 6 and 7, after the recursive calls, we reverse the modification of the variable in question. By this, we guarantee that, whenever a call of a procedure $f_{q,j}$ terminates, it will leave the values of the variables unchanged. Assume now that we want to check whether the machine M halts when started on the initial configuration which consists of the tape $w = x_1 \dots x_n$, head position 1 and the initial state q_0. Then we construct a main program which initializes the variables $\mathbf{x}_j$ to x_j, then calls the procedure $f_{q_0,1}$ and finally sets $\mathbf{x}_1 :=?$. The resulting program can reach the end of the main procedure iff M has an accepting run when started in the initial configuration. Since the equality $\mathbf{x}_1 = 0$ only holds at program exit if the program exit is not reachable, we conclude that $\mathbf{x}_1 = 0$ hold iff M does not accept w. This proves the DEXPTIME-hardness for the general case.

Now consider a linear-space bounded Turing machine M having existential states only. Deciding whether or not there is an accepting computation for such a machine is just a PSPACE-hard problem. Please note that in the corresponding simulation of such a machine by affine programs, the construction of Figure 7 is not needed. Therefore the program contains no consecutive calls such that we can abandon the reconstruction of the original values of the variables $\mathbf{x}_j$ after calls. This makes all procedure calls within the $f_{q,j}$ *tail-recursive*. Thus, the resulting program can also be realized by a single control-flow graph. This observation implies the lower bound for affine programs without procedures. □

We remark that the upper bound also holds for more general programs where, e.g., also multiplication in expressions is allowed, and also for arbitrary affine relations. On the other hand, the lower bound construction only builds on the existence of two distinct values 0 and 1 and thus also holds for any other nontrivial finite data domain. In summary, we conclude that any efficient analysis of programs with guards must be approximate.

Intraprocedurally, an approximative treatment of equality guards has been considered both by Karr for fields [Karr 1976] and by Granger for $\mathbb{Z}$ [Granger 1991]. In both cases, the effect of such a guard amounts to intersection of linear spaces of extended states. This idea also works for $\mathbb{Z}_m$-modules of extended states:

$$[\![g = 0]\!]^\sharp\, M \;=\; M \sqcap \langle \{[x_0, \dots, x_k]^{\mathrm{t}} \mid \sum_{j=0}^{k} g_j x_j = 0\} \rangle$$

Computing this intersection can be reduced to solving a homogeneous linear equation system: Assume $M = \langle G \rangle$ where G is a finite set of generators. Let V denote a matrix containing the vectors of G as column vectors. Then we obtain a system of generators for $[\![g = 0]\!]^\sharp\, M$ by solving the homogeneous system of equations:

$$(g'\, V) \cdot \mathbf{y} = \mathbf{0}$$

for the row vector $g' = [g_0, \dots, g_k]$ and a column vector $\mathbf{y} = [\mathbf{y}_1, \dots, \mathbf{y}_q]^{\mathrm{t}}$ of variables. It expresses that $V\mathbf{y}$ is a linear combination of the vectors in G that satisfies

$g = 0$. Consequently, if $Y = \{y_1, \ldots, y_m\}$ is a set of generators for the $\mathbb{Z}_m$-module of solutions, the intersection is generated by the set $\{V \cdot y_1, \ldots, V \cdot y_m\}$.

An unsatisfiable guard such as $2 = 0$ could still result in a nonempty intersection. In this case, however, it does not contain any extended state. More generally, we can improve the transformer $[\![g = 0]\!]^\sharp$ by checking whether or not the resulting $\mathbb{Z}_m$-module $M' = [\![g = 0]\!]^\sharp\, M$ contains a vector with 0^{th} component 1. If this is not the case, the $\mathbb{Z}_m$-module M' does not represent any concrete extended state and therefore safely can be replaced by the trivial $\mathbb{Z}_m$-module $\langle\,\rangle$.

Assume that the $\mathbb{Z}_m$-module M' is generated by the finite set G' of generators. In order to decide whether M' contains at least one extended state, we observe that the following three statements are equivalent:

(1) M' contains a vector with 0^{th} component 1;
(2) M' contains a vector whose 0^{th} component is odd;
(3) G' contains a vector whose 0^{th} component is odd

where the third property is obviously the easiest to verify.

Example 6.2. Consider the guard $2 = 0$. The $\mathbb{Z}_m$-module of all solutions of this equation is generated by the vectors $2^{w-1} \cdot e_0$ and the unit vectors e_i for $i = 1, \ldots, k$, if k is the number of variables. Each of these vectors has an even 0^{th} component. Accordingly, this $\mathbb{Z}_m$-module does not contain extended states as elements. □

Summarizing, we have a method for dealing with guards intraprocedurally. It is not obvious, though, how intersections can be lifted to the transformer level to obtain an interprocedural generalization of this method. Therefore, we suggest to *postpone* the decision taken at the guard. Instead of performing the intersection, we introduce an extra *indicator variable* for every guard expression in the program. More precisely, assume that the edges with guards are numbered $k+1, \ldots, m$. Then we *instrument* the original program by introducing fresh variables $\mathbf{x}_{k+1}, \ldots, \mathbf{x}_m$, one for each guard. Initially, all these variables are assumed to have values 0. At the j^{th} guard $g = 0$, we place the assignment $\mathbf{x}_j := g$. This corresponds to the matrix:

$$\left[\begin{array}{c|c|c|c} I_{k+1} & \multicolumn{3}{c}{0} \\ \hline 0 & I_{j-k-1} & 0 & 0 \\ \hline g_0 \cdots g_k & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & I_{m-j} \end{array}\right]$$

The extra values stored in the indicator variables are then used for an improved treatment of calls in the constraint system $\mathbf{R}^\sharp$. As an invariant, we insist in $\mathbf{R}^\sharp_{\mathsf{R}}$ that all indicator variables have values 0, since this is the case for all program runs permitted by the guards. Thus the first constraint now reads:

$$[\mathbf{R}\sharp 1] \qquad \mathbf{R}^\sharp(\mathsf{st}_{\mathbf{Main}}) \sqsupseteq \mathbb{Z}_m^{k+1} \times \{0^{m-k}\}$$

Accordingly, we modify the constraints for calls to:

$$[\mathbf{R}^\sharp 5] \quad \mathbf{R}^\sharp(v) \sqsupseteq [\![\mathbf{x}_m = 0]\!]^\sharp\, (\ldots [\![\mathbf{x}_{k+1} = 0]\!]^\sharp\, (\mathbf{S}^\sharp(\mathsf{ret}_q)(\mathbf{R}^\sharp(u)))\ldots) \quad \text{for edge } (u, q, v)$$

Thus, having applied the transformations from $\mathbf{S}^\sharp(\mathsf{ret}_q)$, we select just those vectors from the result whose indicator variables all equal 0. These can be determined by solving an appropriate homogeneous system of linear equations. Altogether, we obtain an enhanced interprocedural analysis which deals with equality guards and conservatively extends the corresponding intraprocedural analysis. Note that an analogous technique can also be applied to other value domains such as fields or general PIRs [Müller-Olm and Seidl 2005a].

7. CONCLUSION

We have presented a sound interprocedural algorithm for computing valid affine relations in affine programs over rings $\mathbb{Z}_m$ where $m = 2^w$. This algorithm specializes to a more efficient intraprocedural algorithm for programs without procedures. In absence of guards, our techniques could be shown to be complete. Beyond that, we also considered affine programs enhanced with equality guards and showed that there the verification of non-trivial equalities immediately becomes intractable . Therefore, we provided methods for approximately dealing with equality guards both intra- and interprocedurally. These techniques allow us to analyze integer arithmetic in programming languages like Java precisely (up to abstraction of guards and non-linear statements). Our new algorithms are obtained from the corresponding algorithms in [Müller-Olm and Seidl 2004d; 2004a; 2005a] by replacing techniques for vector spaces with techniques for $\mathbb{Z}_m$-modules. The difficulty here is that for $m = 2^w$ with $w > 1$, the ring $\mathbb{Z}_m$ has zero divisors which implies that not every element in the ring is invertible. Interestingly, the analysis using modular arithmetic allows us not only to determine all affine relations modulo the given power 2^w but with little extra effort also all relations which are valid for any smaller power of 2. We achieve the same complexity bounds as in the case of fields — up to one extra factor w due to the increased height of the used complete lattices. Interestingly, there is no need to compute multiplicative inverses in our program analysis algorithms.

Our algorithms have the clear advantage that their arithmetic operations can completely be performed within the ring $\mathbb{Z}_m$ of the target language to be analyzed. All problems with excessively long numbers are thus resolved. In [Müller-Olm and Seidl 2004c] we also show how to extend the analyses to $\mathbb{Z}_m$ for an arbitrary $m > 2$.

We remark that in [Müller-Olm and Seidl 2004d], we have shown how the linear algebra methods over fields can be extended to handle local variables and return values of procedures besides just global variables. These techniques immediately carry over to arithmetic in $\mathbb{Z}_m$. The same is true for the generalization to the inference of all valid *polynomial* relations up to a fixed degree bound described for the intraprocedural case over fields in [Müller-Olm and Seidl 2004a]. The key idea is to extend state vectors by further derived components and to apply the linear abstraction on these extended vectors. In an interprocedural algorithm, the matrices collected as summaries for procedures would have to transform extended vectors.

One method to deal with *inequalities* instead of equalities is to use *polyhedra* for abstracting sets of vectors [Cousot and Halbwachs 1978]. It is a challenging question how to combine this abstraction with modular arithmetic.

ACKNOWLEDGMENTS

We thank Martin Hofmann for suggesting the topic of analyzing modular arithmetic and the anonymous referees for valuable comments on previous versions. Thomas Reps gave very useful and encouraging feedback from an implementation of our interprocedural algorithm in the context of the CodeSurfer/x86 tool [Balakrishnan and Reps 2004] and pointed us to the $\mathcal{O}(\log(w))$-algorithm for computing multiplicative inverses in $\mathbb{Z}_m$, $m = 2^w$, described in [Warren 2003].

REFERENCES

Balakrishnan, G. and Reps, T. W. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction, 13th Int. Conf. (CC)*. LNCS 2985. Springer-Verlag, 5–23.

Balbin, I. and Ramamohanarao, K. 1987. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming (JLP) 4,* 3, 259–262.

Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. 1981. Alternation. *Journal of the ACM 28,* 1, 114–133.

Cousot, P. and Halbwachs, N. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th ACM Symp. on Principles of Programming Languages (POPL)*. 84–97.

Fecht, C. and Seidl, H. 1998. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. *Nordic Journal of Computing (NJC) 5,* 4, 304–329.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley.

Granger, P. 1991. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*. LNCS 493, Springer-Verlag, 169–192.

Gulwani, S. and Necula, G. 2003. Discovering Affine Equalities Using Random Interpretation. In *30th ACM Symp. on Principles of Programming Languages (POPL)*. 74–84.

Hafner, J. and McCurley, K. 1991. Asymptotically Fast Triangularization of Matrices over Rings. *SIAM J. of Computing 20,* 6, 1068–1083.

Karr, M. 1976. Affine Relationships Among Variables of a Program. *Acta Informatica 6*, 133–151.

Leroux, J. 2003. Algorithmique de la vérification des systèmes à compteurs: Approximation et accélération. Ph.D. thesis, Ecole Normale Supérieure de Cachan.

Müller-Olm, M. and Seidl, H. 2004a. A Note on Karr's Algorithm. In *31st Int. Coll. on Automata, Languages and Programming (ICALP)*. Springer Verlag, LNCS 3142, 1016–1028.

Müller-Olm, M. and Seidl, H. 2004b. Computing Polynomial Program Invariants. *Information Processing Letters (IPL) 91,* 5, 233–244.

Müller-Olm, M. and Seidl, H. 2004c. Interprocedural Analysis of Modular Arithmetic. Tech. Rep. 789, Fachbereich Informatik, Universität Dortmund.

Müller-Olm, M. and Seidl, H. 2004d. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*. 330–341.

Müller-Olm, M. and Seidl, H. 2005a. A Generic Framework for Interprocedural Analysis of Numerical Properties. In *12th Static Analysis Symposium (SAS)*. Springer, LNCS 3672, 235–250.

Müller-Olm, M. and Seidl, H. 2005b. Analysis of Modular Arithmetic. In *European Symposium on Programming (ESOP)*. Springer Verlag, LNCS 3444, 46–60.

Müller-Olm, M., Seidl, H., and Steffen, B. 2005. Interprocedural Analysis of Herbrand Equalities. In *European Symposium on Programming (ESOP)*. Springer Verlag, LNCS 3444, 31–45.

Paige, B. and Koenig, S. 1982. Finite Differencing of Computable Expressions. *ACM Trans. Prog. Lang. and Syst. 4,* 3, 402–454.

Reps, T. 2006. Personal communication.

Reps, T., Schwoon, S., and Jha, S. 2003. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. In *Int. Static Analysis Symposium (SAS)*. LNCS 2694, Springer-Verlag, 189–213.

Reps, T. W., Balakrishnan, G., and Lim, J. 2006. Intermediate-representation recovery from low-level code. In *PEPM*, J. Hatcliff and F. Tip, Eds. ACM, 100–111.

Storjohann, A. 2000. Algorithms for Matrix Canonical Forms. Ph.D. thesis, ETH Zürich, Diss. ETH No. 13922.

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

Received September 2005