# Analysis of Modular Arithmetic

Markus Müller-Olm[1] and Helmut Seidl[2]

[1] Universität Dortmund, Fachbereich Informatik, LS 5
Baroper Str. 301, 44221 Dortmund, Germany
markus.mueller-olm@cs.uni-dortmund.de
[2] TU München, Institut für Informatik, I2
80333 München, Germany
seidl@in.tum.de

**Abstract.** 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 inter-procedural algorithms for inferring for every program point $u$, affine relations between program variables valid at $u$. Our algorithms are not only sound but also *complete* in that they detect *all* valid affine relations. 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.

## 1 Introduction

Analyses for automatically finding linear invariants in programs have been studied for a long time [6, 3, 4, 7, 12, 11, 9]. With the notable exception of Granger [3], none of these analyses can find out, that the linear invariant $21 \cdot \mathbf{x} - \mathbf{y} = 1$ holds upon termination of the following Java program:

```
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));
    }
}
```

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 $w = 32$, if the result expression is of type `int`, and $w = 64$, if the result expression is of type `long` [2, 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

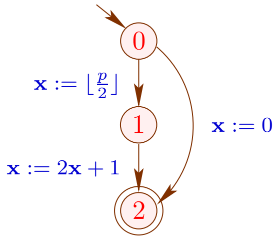


**Fig. 1.** $\mathbb{Z}_p$ interpretation is unsound.

with nice properties analyses for finding linear invariants typically interpret variables by members from a field, e.g., the set $\mathbb{Q}$ of rational numbers [6, 11, 8], or $\mathbb{Z}_p = \mathbb{Z}/(p\mathbb{Z})$ for prime numbers $p$ [4]. Even worse: analyses based on $\mathbb{Z}_p$ for a fixed prime $p$ alone may yield unsound results.[3] In the small flow graph in Fig. 1, 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 $\mathbb{Z}_m$. 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 intra-procedural, and no upper complexity bound to his algorithm is known.

In this paper we present intra- and inter-procedural analyses that are sound and, up to the common abstraction of guarded to non-deterministic branching, complete 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 *sound* w.r.t. the arithmetic used in mainstream programming languages. Technically, our new analyses are based on the methods from linear algebra that we have studied previously [8, 11]. 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, 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 [5, 13]. Here, we simplify the general techniques to establish the properties of $\mathbb{Z}_m$ which suffice to implement similar algorithms as in [8, 11]. Interestingly, 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.

Besides the soundness and completeness issues discussed above, there is another advantage of our analyses that is perhaps more important from a practical point of view than precision. For any algorithm based on computing in $\mathbb{Q}$, we

[3] If the primes of the analysis are chosen randomly, the resulting analysis is at least "probabilistically sound" [4].

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 rely on integers of bounded size as provided by the host language. In this case we must cope with overflows. Or we represent integers by arbitrarily long bit strings. In this case 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.

Our 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 technology 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 system of linear equations over $\mathbb{Z}_m$. In Section 3, we show how these insights can be used to construct sound and complete program analyses. We introduce our basic notion of programs together with their concrete semantics. We introduce affine relations and adapt the basic technology provided in [11] for fields to work for rings $\mathbb{Z}_m$. Finally, we sketch how to obtain highly efficient *intraprocedural* analyses of affine relations over $\mathbb{Z}_m$. In Section 4, we summarize and explain further directions of research.

## 2 The Ring $\mathbb{Z}_m$ for Powers $m = 2^w$

In [5, 13], 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 $\mathbb{Z}_m$ for prime powers $m$ which is a special case of a PIR. Accordingly, the general methods from [5, 13] are applicable. It turns out, however, that for prime powers $m$, the ring $\mathbb{Z}_m$ has a very 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 is (mostly) abandoned. Since the abstract values of our program analyses will be *submodules* of $\mathbb{Z}_m^N$ for suitable $N$, we also compute 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. In particular, we 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 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 $b \in \mathbb{Z}_m$. Using arithmetic modulo $m$, the inverse $b$ can be computed in time $\mathcal{O}(w)$.*

*Proof.* Assume $a = 2 \cdot a'$. Then $a \cdot 2^{w-1} = 2^w \cdot a' = 0$ (modulo $m$). If, on the other hand, $a$ is odd, then $a$ and $m$ are relative prime. Therefore, we can use the

Euclidean algorithm to determine integers $x$ and $y$ such that $1 = a \cdot x + m \cdot y$. Accordingly, $b = x$ (modulo $m$) is the inverse of $a$. This algorithm, however, cannot be executed modulo $m$. In the case where $w = 1$, we know that $\mathbb{Z}_m$ is in fact the field $\mathbb{Z}_2$. Thus, the inverse of $a \neq 0$ (modulo 2) is given by $a$. If on the other hand $w > 1$, we can use the Euclidean algorithm to determine odd integers $x_1$ and $y_1$ with $1 = a \cdot x_1 + 2^{w-1} \cdot y_1$. By computing the square of both sides of this equation, we obtain:

$$1 \;=\; a^2 x_1^2 + 2 \cdot a x_1 2^{w-1} y_1 + 2^{(w-1)2} y_1^2$$

Every summand of the right-hand side except the first contains $2^w$ as a factor and thus equals 0 (modulo $m$). Hence, $b = a x_1^2$ (modulo $m$). Since the Euclidean algorithm uses $\mathcal{O}(\log(m))$ operations, the complexity statement follows. $\square$

*Example 1.* Consider $w = 32$ and $a = 21$. We use the familiar notation of Java int values as elements in the range $[-2^{31}, 2^{31} - 1]$. The Euclidean algorithm applied to $a$ and $m' = 2^{31}$ (or: $-2^{31}$ in signed notation) gives us $x_1 = -1124872387$ and $y_1 = 11$. Then $b = 21 \cdot x_1^2 = 1022611261$ modulo $2^{32}$ is the inverse of $a$. $\square$

Since computing inverses can be rather expensive, we will avoid these whenever possible. For $a \in \mathbb{Z}_m$, we define the *rank* of $a$ as $r \in \{0, \dots, w\}$ 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 $M \subseteq \mathbb{Z}_m^k$ of vectors[4] $(x_1, \dots, x_k)^{\mathrm{t}}$ 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 $x = (x_1, \dots, x_k)^{\mathrm{t}}$, 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 *triangular form* iff for all distinct vectors $x, x' \in G$, the leading indices of $x$ and $x'$ are distinct. Every set $G$ in triangular form contains at most $k$ elements. We define the *rank* of a triangular set $G$ of cardinality $s$ as the sum of the ranks of the leading entries of the vectors of $G$ plus $(k - s) \cdot w$ (to account for $k - s$ zero vectors). Note that this deviates from the common notion of the rank of a matrix.

Assume that we are given a set $G$ in triangular form together with a new vector $x$. Our goal is to construct a set $\bar{G}$ in triangular 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 $2^r d$ ($d$ invertible) denote the leading index and leading entry of $x$, respectively. We distinguish several cases:

1. The $i$-th entry of all vectors $x' \in G$ are 0. Then we choose $\bar{G} = G \cup \{x\}$.

[4] 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).

2. $i$ is the leading index of some $y \in G$ where the leading entry equals $2^{r'}d'$ ($d'$ invertible).
   (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, and we proceed with $G$ and $x'$.
   (b) If $r' > r$, then we construct a new set $G'$ by replacing $y$ with the vector $x$. Furthermore, 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 $y'$.

Eventually, we arrive at a set $\bar{G}$ having the desired properties. Moreover, either the resulting $\bar{G}$ equals $G$ or the rank of $\bar{G}$ is strictly less than the rank of $G$.

Overall, computing a triangular set for a given triangular set and a new vector amounts to at most $\mathcal{O}(k)$ computations of ranks together with $\mathcal{O}(k^2)$ arithmetic operations. On the whole, it therefore can be done with $\mathcal{O}(k \cdot (k + \log(w)))$ operations. Accordingly, we obtain the following theorem:

**Theorem 1.** *1. Every $\mathbb{Z}_m$-module $M \subseteq \mathbb{Z}_m^k$ is generated by some set $G$ of generators of cardinality at most $k$.*
*2. Given a set $G'$ of generators of cardinality $n$, a set $G$ of cardinality at most $k$ can be computed in time $\mathcal{O}(n \cdot k \cdot (k + \log(w)))$ such that $\langle G \rangle = \langle G' \rangle$.*
*3. Every strictly increasing chain of $\mathbb{Z}_m$-modules $M_0 \subset M_1 \subset \ldots \subset M_s \subseteq \mathbb{Z}_m^k$, has length $s \leq k \cdot w$.*

*Proof.* The second statement follows from our construction of triangular sets of generators. Starting from the empty set, which is triangular by definition, we successively add the vectors in $G'$ with the procedure described above. The complexity is then estimated by summing up the operations of these $n$ inclusions.

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 triangular sets generating the same modules as follows. $G'_0$ is the triangular set constructed for $G_0$. 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 triangular 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 $k \cdot w$ and ranks are non-negative, the third statement follows. □

*Example 2.* 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 triangular. Let $z = (1, 2, 1)^{\mathsf{t}}$. We want to construct a triangular set of generators 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 $x'' = x' - 1 \cdot y = (0, 0, 3)^{\mathsf{t}}$. As triangular set we finally return $\bar{G} = \{z, y, x''\}$. □

For a set of generators $G$ being triangular, does not imply being a *minimal* set of generators. For $w = 3$ consider, e.g., the triangular set $G = \{x, y\}$ where $x = (4,1)^{\mathsf{t}}, y = (0,2)^{\mathsf{t}}$. Multiplying $x$ with 2 results in: $2 \cdot x = (8,2)^{\mathsf{t}} = (0,2)^{\mathsf{t}} = y$. Thus $\{x\}$ generates the same module as $G$ implying that $G$ is not minimal.

It is well-known that the submodules of $\mathbb{Z}_m^k$ are closed under intersection. Ordered by set inclusion they thus form a complete lattice $\mathbf{Sub}(\mathbb{Z}_m^k)$, like the linear subspaces of $\mathbb{F}^k$ for a field $\mathbb{F}$. However, while the height of the lattice of linear subspaces of $\mathbb{F}^k$ is $k$ for dimension reasons, the height of the lattice of submodules of $\mathbb{Z}_m^k$ is precisely $k \cdot w$. By Theorem 1, $k \cdot w$ is an upper bound for the height and it is not hard to actually construct a chain of this length. The least element of $\mathbf{Sub}(\mathbb{Z}_m^k)$ is $\{\mathbf{0}\}$, the greatest element is $\mathbb{Z}_m^k$ 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 systems of linear equations in $k$ 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 $k$ equations. Such a system can be denoted as $A\,\mathbf{x} = b$ where $A$ is a square $(k \times k)$-matrix $A = (a_{ij})_{1 \le i,j \le k}$ with entries $a_{ij} \in \mathbb{Z}_m$, $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathsf{t}}$ is a column vector of unknowns and $b = (b_1, \ldots, b_k)^{\mathsf{t}}$ is a column vector of elements $b_i \in \mathbb{Z}_m$. Let $\mathbb{L}$ denote the set of all solutions of $A\,\mathbf{x} = b$. Let $\mathbb{L}_0$ denote the set of all solutions of the corresponding *homogeneous system* $A\,\mathbf{x} = \mathbf{0}$ where $\mathbf{0} = (0, \ldots, 0)^{\mathsf{t}}$. It is well-known that, if the system $A\,\mathbf{x} = b$ has at least one solution $x$, then the set of all its solutions can be obtained from $x$ by adding solutions of the corresponding homogeneous system, i.e.,

$$\mathbb{L} = \{x + y \mid y \in \mathbb{L}_0\}$$

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

1. *$A\,\mathbf{x} = b$ has a solution iff for all $i$, $w_i$ does not exceed the rank of $b_i$.*
2. *If $A\,\mathbf{x} = b$ is solvable, then one solution is given by: $x = (x_1, \ldots, x_k)^{\mathsf{t}}$ with $x_i = 2^{w'_i - w_i} \cdot d_i^{-1} b'_i$ where $b_i = 2^{w'_i} b'_i$ for invertible elements $b'_i$.*
3. *The set of solutions of the homogeneous system $A\,\mathbf{x} = \mathbf{0}$ is the $\mathbb{Z}_m$-module generated from the vectors: $e^{(j)} = (e_{1j}, \ldots, e_{kj})^{\mathsf{t}}, j = 1, \ldots, k$, where $e_{ij} = 2^{w - w_i}$ if $i = j$ and $e_{ij} = 0$ otherwise.* □

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 also, that in contrast to inhomogeneous systems, sets of generators for homogeneous systems can be computed *without* computing inverses.

*Example 3.* Let $w = 4$, i.e., $m = 16$, and

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

Then the $\mathbb{Z}_m$-module of solutions of $A\,\mathbf{x} = 0$ is generated by the two vectors $e^{(1)} = (8, 0)^{\mathrm{t}}$ and $e^{(2)} = (0, 2)^{\mathrm{t}}$. □

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 3.** *Let $A$ denote an arbitrary $(k \times k)$-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 invertible $(k \times k)$-matrices over $\mathbb{Z}_m$.*
2. *W.r.t. this decomposition, $x$ is a solution of $A\,\mathbf{x} = b$ iff $x = R^{-1}x'$ for a solution $x'$ of the system $D\,\mathbf{x} = b'$ for $b' = L^{-1}b$.*
3. *The matrix $D$ together with the matrix $R^{-1}$ and the vector $b' = L^{-1}\,b$ can be computed in time $\mathcal{O}(\log(w) \cdot k^3)$. In particular, computation of inverses is not needed for the decomposition.*

*Proof.* In order to prove 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. Recall that the idea for fields consisted in successively selecting 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 $(i, j)$ 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 Pivot elements by always selecting as a Pivot element the $(i, j)$ where the entry $d = 2^r d'$ of the current matrix has *minimal rank* $r$, and $d'$ is invertible over $\mathbb{Z}_m$. Since $r$ has been chosen minimal, still 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. Proceeding with 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 $L$. Since the elementary transformations which we apply only express exchange of columns or rows, multiplication with an invertible element or adding of a multiple of one column / row to the other, these transformations are also invertible over $\mathbb{Z}_m$.

Now it should be clear how the matrix $D$ together with the matrix $R^{-1}$ and the vector $b' = L^{-1}b$ 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 $A$. Also, the vector $b'$ is obtained by performing on $b$ the same sequence of row transformations as on $A$. In particular, this provides us with the complexity bound as stated in item (3). $\square$

Putting lemmas 2 and 3 together we obtain:

**Theorem 2.** *1. A representation of the set $\mathbb{L}_0$ of a homogeneous equation system $A\,\mathbf{x} = 0$ over $\mathbb{Z}_m$ can be computed without resorting to the computation of inverses in time $\mathcal{O}(\log(w) \cdot k^3)$.*

*2. A representation of the set $\mathbb{L}$ of all solutions of an equation system $A\,\mathbf{x} = b$ over $\mathbb{Z}_m$ can be computed in time $\mathcal{O}(w \cdot k + \log(w) \cdot k^3)$.*

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

$$\begin{aligned} 12\mathbf{x}_1 + 6\mathbf{x}_2 &= 10 \\ 14\mathbf{x}_1 + 4\mathbf{x}_2 &= \ \ 8 \end{aligned}$$

We start with

$$A_0 = \begin{pmatrix} 12 & 6 \\ 14 & 4 \end{pmatrix}, \ b_0 = \begin{pmatrix} 10 \\ 8 \end{pmatrix}, \ R_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

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 $A_0$ and in $b_0$:

$$A_1 = \begin{pmatrix} 12 & 6 \\ 2 & 0 \end{pmatrix}, \ b_1 = \begin{pmatrix} 10 \\ 4 \end{pmatrix}, \ R_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

By subtracting twice the second column from the first in $A_1$ and $R_1$, we obtain:

$$A_2 = \begin{pmatrix} 0 & 6 \\ 2 & 0 \end{pmatrix}, \ b_2 = \begin{pmatrix} 10 \\ 4 \end{pmatrix}, \ R_2 = \begin{pmatrix} 1 & 0 \\ 14 & 1 \end{pmatrix}$$

Now, we exchange the columns 1 and 2 in $A_3$ and $R_3$:

$$A_3 = \begin{pmatrix} 6 & 0 \\ 0 & 2 \end{pmatrix}, \ b_3 = \begin{pmatrix} 10 \\ 4 \end{pmatrix}, \ R_3 = \begin{pmatrix} 0 & 1 \\ 1 & 14 \end{pmatrix}$$

Since $3 \cdot 11 = 1 \bmod 16$, we can easily read off $x'_0 = \binom{11 \cdot 5}{2} = \binom{7}{2}$ as a solution of $A_3\,\mathbf{x} = b_3$. We also see that the two vectors $x'_1 = \binom{8}{0}$ and $x'_1 = \binom{0}{8}$ generate the module of solutions of the homogeneous system $A_3\mathbf{x} = \mathbf{0}$. Consequently, $x_0 = R_3\,x'_0 = \binom{2}{3}$ is a solution of $A_0\,\mathbf{x} = b_0$ and the two vectors $x_1 = R_3\,x'_1 = \binom{0}{8}$ and $x_1 = R_3\,x'_2 = \binom{8}{0}$ generate the module of solutions of the homogeneous system $A_0\,\mathbf{x} = \mathbf{0}$. We conclude that the set of solutions of $A_0\,\mathbf{x} = b_0$ (over $\mathbb{Z}_{16}$) is

$$\mathbb{L} = \left\{ \binom{2+8a}{3+8b} \mid a, b \in \mathbb{Z}_{16} \right\} = \left\{ \binom{2}{3}, \binom{2}{11}, \binom{10}{3}, \binom{10}{11} \right\}$$

$\square$

## 3 Affine Program Analysis

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 $\mathbb{Z}_m$. In this section, we show how these algorithms can be plugged into the algorithmic skeletons of the sound and complete analyses of affine relations over fields as, e.g., presented in [11] to obtain sound and complete analyses of affine relations over $\mathbb{Z}_m$. For the sake of an easier comparison, we use the same conventions as in [11] which we recall here briefly in order to be self-contained. Thus, programs are modeled by systems of non-deterministic flow graphs that can recursively call each other as in Figure 2. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of (global) variables

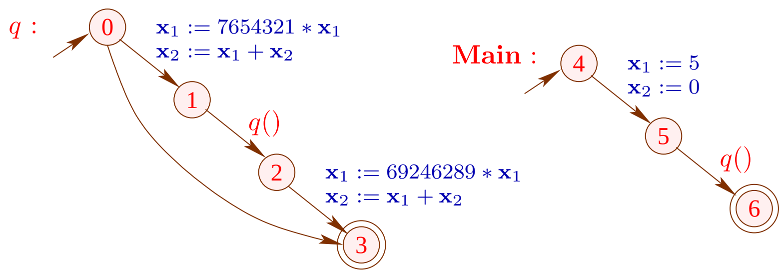


**Fig. 2.** An inter-procedural program.

the program operates on. We use $\mathbf{x}$ to denote the column vector of variables $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^t$. In this paper, we assume that the variables take values in the ring $\mathbb{Z}_m$. Thus, a *state* assigning values to the variables is modeled by a $k$-dimensional (column) vector $x = (x_1, \ldots, x_k)^t \in \mathbb{Z}_m^k$; $x_i$ being the value assigned to variable $\mathbf{x}_i$. For a state $x$, a variable $\mathbf{x}_i$ and a value $c \in \mathbb{Z}_m$, we write $x[\mathbf{x}_i \mapsto c]$ for the state $(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_k)^t$ as usual.

We assume that the basic statements in our programs either are *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 $\mathbf{x}_j \in \mathbf{X}$). We annotated the edges in Fig. 2 with sequences of assignments just in order to reduce the number of program points. Since assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect onto the program state, they are skip-statements and omitted in pictures. Skip-statements can be used to abstract guards. This amounts to replacing conditional branching in the original program with non-deterministic branching. It is relative to this common abstraction when we say an analysis is complete.

Non-deterministic assignments $\mathbf{x}_j :=?$ can be used as a safe abstraction of statements our analysis cannot handle precisely, for example of assignments $\mathbf{x}_j := t$ with non-affine expressions $t$ or of read statements $\mathsf{read}(\mathbf{x}_j)$.

In this setting, a *program* comprises a finite set $\mathsf{Proc}$ of *procedure names* together 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 $\mathsf{st}_q$ and a single return point $\mathsf{ret}_q$ where each edge is either labeled with an assignment or a call to some procedure.

The key idea of [11] which we take up here for the analysis of modular arithmetic, is to construct a precise abstract interpretation of a constraint system characterizing the program executions that reach program points. For that, program executions or *runs* are represented by sequences $r$ of affine assignments:

$$r \equiv s_1; \ldots ; s_m$$

where $s_i$ are assignments of the form $\mathbf{x}_j := t$, $\mathbf{x}_j \in \mathbf{X}$ and $t \equiv t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ for some $t_0, \ldots, t_k \in \mathbb{Z}_m$. (Non-deterministic assignments give rise to multiple runs.) We write $\mathsf{Runs}$ for the set of runs. Every assignment statement $\mathbf{x}_i := t$ induces a state transformation $[\![\mathbf{x}_j := t]\!] : \mathbb{Z}_m^k \to \mathbb{Z}_m^k$ given by

$$[\![\mathbf{x}_j := t]\!]x \;=\; x[\mathbf{x}_j \mapsto t(x)] \, ,$$

where $t(x)$ is the value of term $t$ in state $x$. This definition is inductively extended to runs: $[\![\varepsilon]\!] = \mathsf{Id}$, where $\mathsf{Id}$ is the identical mapping and $[\![ra]\!] = [\![a]\!] \circ [\![r]\!]$.

A closer look reveals that the state transformation of an affine assignment $\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ is in fact an affine transformation. As a composition of affine transformations, the state transformer of a run is therefore also an affine transformation — no matter whether we compute over fields or some $\mathbb{Z}_m$. Hence, for any run $r$, we can choose $A_r \in \mathbb{Z}_m^{k^2}$ and $b_r \in \mathbb{Z}_m^k$ such that $[\![r]\!]x = A_r x + b_r$.

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 $a = (a_0, \ldots, a_k)^{\mathrm{t}}$. 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 vector $y \in \mathbb{Z}_m^k$ *satisfies* the affine relation $a$ iff $a_0 + a' \cdot y = 0$ where $a' = (a_1, \ldots, a_k)^{\mathrm{t}}$ and "$\cdot$" denotes the scalar product. This fact is denoted by $y \models a$.

We say that the affine relation $a$ is *valid* after a single run $r$ iff $[\![r]\!]x \models a$ for all $x \in \mathbb{Z}_m^k$, i.e., iff $a_0 + a' \cdot [\![r]\!]x = 0$ for all $x \in \mathbb{Z}_m^k$; $x$ represents the unknown initial state. Thus, $a_0 + a' \cdot [\![r]\!]\mathbf{x} = 0$ is the *weakest precondition* for validity of the affine relation $a$ after run $r$. In [11], we have shown in the case of fields, that the weakest precondition can be computed by a *linear transformation* applied to the vector $a$. The very same argumentation works as well in the more general case of arbitrary rings. More specifically, this linear transformation is given by the following $(k+1)^2$ matrix $W_r$:

$$W_r = \left(\begin{array}{c|c} 1 & b_r^{\mathrm{t}} \\ \hline 0 & A_r^{\mathrm{t}} \end{array}\right) \tag{1}$$

Also over $\mathbb{Z}_m$, the only affine relation which is true for *all program states* is the relation $\mathbf{0} = (0, \ldots, 0)^{\mathrm{t}}$. Since the initial state is arbitrary, an affine relation $a$ is thus valid at a program point $u$ iff $W_r \, a = \mathbf{0}$ for all runs $r$ that reach $u$.

This is good news, since it shows that as in the case of fields, we may use the set $\mathcal{W} = \{W_r \mid r \text{ reaches } u\}$ to solve the validity problem for affine relations by setting up and solving the linear equation system $W a = \mathbf{0}$, $W \in \mathcal{W}$. While in our case this set is finite because $\mathbb{Z}_m$ is finite, it can be large. We are thus left with the problem of representing $\mathcal{W}$ compactly. In the case of fields, we could observe that the set of $(k+1) \times (k+1)$ matrices forms a vector space. Over $\mathbb{Z}_m$ this is no longer the case. However, this set is still a $\mathbb{Z}_m$-*module* isomorphic to $\mathbb{Z}_m^{(k+1)^2}$. We observe that as in the case of fields we can use a generating system of the submodule $\langle \mathcal{W} \rangle$ instead of $\mathcal{W}$ to set up this linear equation system without changing the set of solutions. By Theorem 1, $\langle \mathcal{W} \rangle$ can be described by a generating system of at most $(k+1)^2$ matrices. Therefore, in order to determine the set of all affine relations at program point $u$, it suffices to compute a set of generators for the module $\langle\{W_r \mid r \text{ reaches } u\}\rangle$. This is the contents of the next theorem:

**Theorem 3.** *Assume we are given a generating system $G$ of cardinality at most $(k+1)^2$ for the set $\langle\{W_r \mid r \text{ reaches } u\}\rangle$. Then we have:*

*a) The affine relation $a \in \mathbb{Z}_m^{k+1}$ is valid at $u$ iff $W\,a = \mathbf{0}$ for all $W \in G$.*
*b) A generating system for the $\mathbb{Z}_m$-submodule of all affine relations valid at program point $u$ can be computed in time $\mathcal{O}(k^4 \cdot (k + \log(w)))$.*

*Proof.* We only consider the complexity estimation. By statement a), the affine relation $a$ is valid at $u$ iff $a$ is a solution of all the equations

$$\textstyle\sum_{j=0}^{k} w_{ij}\mathbf{a}_j \;=\; 0$$

for each matrix $W = (w_{ij}) \in G$ and $i = 0, \ldots, k$. The generating system $G$ contains at most $(k+1)^2$ matrices each of which contributes $k+1$ equations. First, we can bring this set into triangular form. By Theorem 1, this can be done in time $\mathcal{O}(k^4 \cdot (k + \log(w)))$. The resulting system has at most $k+1$ equations. By Theorem 2, a generating system for the $\mathbb{Z}_m$-module of solutions of this system can be computed with $\mathcal{O}(\log(w) \cdot k^3)$ operations. The latter amount, however, is dominated by the former, and the complexity statement follows. $\square$

As in the case of fields, we are left with the task of computing, for every program point $u$, a generating system for $\langle\{W_r \mid r \text{ reaches } u\}\rangle$. Following the approach in [11], we compute this submodule of $\mathbb{Z}_m^{(k+1)^2}$ as an abstract interpretation of a constraint system for set of runs reaching $u$. From Section 2 we know that $\mathbf{Sub}(\mathbb{Z}_m^{(k+1)^2})$ is a complete lattice of height $\mathcal{O}(k^2 \cdot w)$ such that we can compute fixpoints effectively. The desired abstraction of run sets is given by $\alpha : 2^{\mathsf{Runs}} \to \mathbf{Sub}(\mathbb{Z}_m^{(k+1)^2})$, $\alpha(R) = \langle\{W_r \mid r \in R\}\rangle$. Indeed, the mapping $\alpha$ is monotonic (w.r.t. subset ordering on sets of runs and submodules) and commutes with arbitrary unions. Similar to the case of fields, we set up a constraint system. The variables in the new constraint system take submodules of $\mathbb{Z}_m^{(k+1)^2}$ as values:

| | | | |
|---|---|---|---|
| $[\mathrm{S}_\alpha 1]$ | $\mathbf{S}_\alpha(q)$ | $\supseteq \mathbf{S}_\alpha(\mathsf{ret}_q)$ | |
| $[\mathrm{S}_\alpha 2]$ | $\mathbf{S}_\alpha(\mathsf{st}_q)$ | $\supseteq \langle\{I_{k+1}\}\rangle$ | |
| $[\mathrm{S}_\alpha 3]$ | $\mathbf{S}_\alpha(v)$ | $\supseteq \mathbf{S}_\alpha(u) \circ \langle\{W_{\mathbf{x}_j:=t}\}\rangle$ | for an edge $(u,v)$ labeled $\mathbf{x}_j := t$ |
| $[\mathrm{S}_\alpha 4]$ | $\mathbf{S}_\alpha(v)$ | $\supseteq \mathbf{S}_\alpha(u) \circ \langle\{W_{\mathbf{x}_j:=0}, W_{\mathbf{x}_j:=1}\}\rangle$ | for an edge $(u,v)$ labeled $\mathbf{x}_j :=?$ |
| $[\mathrm{S}_\alpha 5]$ | $\mathbf{S}_\alpha(v)$ | $\supseteq \mathbf{S}_\alpha(u) \circ \mathbf{S}_\alpha(q)$ | for an edge $(u,v)$ calling $q$ |
| $[\mathrm{R}_\alpha 1]$ | $\mathbf{R}_\alpha(\mathbf{Main})$ | $\supseteq \langle\{I_{k+1}\}\rangle$ | |
| $[\mathrm{R}_\alpha 2]$ | $\mathbf{R}_\alpha(q)$ | $\supseteq \mathbf{R}_\alpha(u)$ | for an edge $(u,\_)$ calling $q$ |
| $[\mathrm{R}_\alpha 3]$ | $\mathbf{R}_\alpha(u)$ | $\supseteq \mathbf{R}_\alpha(q) \circ \mathbf{S}_\alpha(u)$ | if $u$ is a program point of $q$ |

The variable $\mathbf{S}_\alpha(q)$ is meant to capture the abstract effect of the procedure $q$. By the constraints $\mathbf{S}_\alpha 1$, this value is obtained as the module of transformations $\mathbf{S}_\alpha(\mathsf{ret}_q)$ accumulated for the return point $\mathsf{ret}_q$ of $q$. According to $\mathbf{S}_\alpha 2$, this accumulation starts at the start point $\mathsf{st}_q$ with (the module generated by) the identity transformation. The constraints $\mathbf{S}_\alpha 3$ and $\mathbf{S}_\alpha 4$ deal with affine and non-deterministic assignments, respectively, while the constraints $\mathbf{S}_\alpha 5$ correspond to calls. The abstract effects of procedures are then used in the second part of the constraint system to determine for every procedure and program point the module of transformations induced by reaching runs. The constraint $\mathbf{R}_\alpha 1$ indicates that we start before the call to **Main** with the identity transformation. The constraints $\mathbf{R}_\alpha 2$ indicate that transformations reaching a procedure should comprise all transformation reaching its calls. Finally, the constraints $\mathbf{R}_\alpha 3$ state that the transformations for a program point $u$ of some procedure $q$ should contain the composition of transformations reaching $q$ with the transformation accumulated for $u$ from the start point $\mathsf{st}_q$ of $q$.

In this constraint system, the operator "$\circ$" abstracts concatenation of run sets. As in the case of fields, it is defined in terms of matrix multiplication by

$$M_1 \circ M_2 = \langle \{A_1 A_2 \mid A_i \in M_i\} \rangle$$

for sets of matrices $M_1, M_2 \subseteq \mathbb{Z}_m^{(k+1)^2}$. An edge annotated by $\mathbf{x}_j := ?$ induces the set of all runs $\mathbf{x}_j := c$, $c \in \mathbb{Z}_m$. As in the case of fields, however, the module spanned by the matrices $W_{\mathbf{x}_j := c}, c \in \mathbb{Z}_m$, is generated by the two matrices $W_{\mathbf{x}_j := 0}$ and $W_{\mathbf{x}_j := 1}$. Therefore, these two suffice to abstract the effect of $\mathbf{x}_j := ?$.

Again as in the case of fields, the constraint system from above can be solved by computing on generating systems. In contrast to fields, however, we no longer have the notion of a basis available. Instead, we rely on sets of generators in triangular form. In order to avoid the need to solve a system of equations over $\mathbb{Z}_m$ fully whenever a new vector is added to a set of generators, we use our algorithm from Theorem 1 to bring the enlarged set again into triangular form. A set of generators, thus, may have to be changed — even if the newly added vector is implied by the original one. The update, however, then *decreases* the rank of the set of generators implying that ultimately stabilization is detected.

We assume that the basic statements in the given program have size $\mathcal{O}(1)$. Thus, we measure the size $n$ of the given program by $|N| + |E|$. We obtain:

**Theorem 4.** *For every program of size $n$ with $k$ variables the following holds:*

*a) The values: $\langle \{W_r \mid r \text{ reaches } X\} \rangle$, $X$ a procedure or program point, equal the components of the least solution of the constraint system for the $\mathbf{R}_\alpha(X)$.*
*b) These values can be computed in time $\mathcal{O}(w \cdot n \cdot k^6 \cdot (k^2 + \log(w)))$.*
*c) The sets of all valid affine relations at program point $u$, $u \in N$, can be computed in time $\mathcal{O}(w \cdot n \cdot k^6 \cdot (k^2 + \log(w)))$.* □

A full proof of Theorem 4 can be found in [10]. In our main application, $w$ equals 32 or 64. The term $\log(w)$ in the complexity estimation accounts for the necessary rank computations. In our application, $\log(w)$ equals 5 or 6 and thus is easily dominated by $k^2$. We conclude that the extra overhead over the corresponding complexity from [11] for the analysis over fields (w.r.t. unit cost for basic arithmetic operations) essentially consists in one extra factor $w = 32$ or 64 which is due to the increased height of the lattice used. We expect, however, that fixpoint computations in practice will not exploit full maximal lengths of ascending chains in the lattice but stabilize much earlier.

*Example 5.* Consider the inter-procedural program from Figure 2 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 weakest precondition transformers for $s_1 \equiv \mathbf{x}_1 := 7654321 \cdot \mathbf{x}_1; \mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$ and $s_2 \equiv \mathbf{x}_1 := 69246289 \cdot \mathbf{x}_1; \mathbf{x}_2 := \mathbf{x}_1 + \mathbf{x}_2$ are:

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

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

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

None of these is subsumed by the other where the corresponding triangular set of generators is given by $G_1 = \{I_{k+1}, P\}$ where

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

The next iteration then results in the matrix

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

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

In order to obtain the weakest precondition transformers, e.g., for the endpoint 6 of **Main**, we additionally need the transformation $B_0$ for $\mathbf{x}_1 := 5; \mathbf{x}_2 := 0$:

$$B_0 = \begin{pmatrix} 1 & 5 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Using the set $\{I_{k+1}, P\}$ of generators for $\mathbf{S}_\alpha(q)$, we thus obtain for $\mathbf{R}_\alpha(6)$ the generators:

$$W_1 = B_0 \cdot I_{k+1} = \begin{pmatrix} 1 & 5 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad W_2 = B_0 \cdot P = \begin{pmatrix} 0 & 0 & 5c+5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

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 shows that the set of all solutions is generated by:

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

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 $\mathbb{Q}$. $\square$

The runtime of our inter-procedural analysis is linear in the program size $n$ but polynomial in the number of program variables $k$ of a rather high degree. In [8], we have presented an efficient algorithm which in absence of procedures, computes all valid affine relations in time $\mathcal{O}(n \cdot k^3)$ — given that all arithmetic operations count for 1. This algorithm improves on the original algorithm by Karr [6] for the same problem by one factor of $k$. Due to lack of space we will neither rephraze this nor Karr's original algorithm but remark that both algorithms assume that the program variables take values in a field, namely $\mathbb{Q}$ — but any other field $\mathbb{Z}_p$ ($p$ prime) would do as well. Similar to the algorithm from [11], they compute with finite-dimensional vector spaces represented through sets of generators. Just as in our exposition for the interprocedural analysis, we obtain a sound and complete *intraprocedural* analysis if we replace the vector spaces of the algorithm in [8] with the corresponding $\mathbb{Z}_m$-modules and use the algorithms from Section 2 for reducing the cardinalities of sets of generators and solving sets of homogeneous equations. Summarizing, we obtain:

**Theorem 5.** *Consider an affine program of size $n$ with $k$ variables but without procedures. Then for $m = 2^w$, the set of* all *affine relations at all program points which are valid over $\mathbb{Z}_m$ can be computed in time $\mathcal{O}(w \cdot n \cdot k^2 \cdot (k + \log(w)))$.* □

## 4 Conclusion

We have presented sound and complete inter- and intraprocedural algorithms for computing valid affine relations in affine programs over rings $\mathbb{Z}_m$ where $m = 2^w$. These techniques allow us to analyze integer arithmetic in programming languages like Java precisely (upto abstraction of guards). Our new algorithms were obtained from the corresponding algorithms in [11] and [8] by replacing techniques for vector spaces with techniques for $\mathbb{Z}_m$-modules. The difficulty here is that for $w > 1$, the ring $\mathbb{Z}_m$ has zero divisors — implying that not every element in the ring is invertible. Since we maintained the top-level structure of the analysis algorithms, we achieved the same complexity bounds as in the case of fields — upto one extra factor $w$ due to the increased height of the used complete lattices. We carefully avoid explicit computation of inverses in our algorithms for reducing sets of generators and for solving homogeneous linear equation systems. Otherwise the complexity estimates for the algorithms would be worse because computation of inverses cannot reasonably be assumed constant time.

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 [10] we also show how to extend the analyzes to $\mathbb{Z}_m$ for an arbitrary $m > 2$.

We remark that in [11], we also have shown how the linear algebra methods over fields can be extended to deal with 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.

One method to deal with *inequalities* instead of equalities is to use *polyhedra* for abstracting sets of vectors [1]. It is a challenging question what kind of impact modular arithmetic has on this abstraction.

*Acknowledgments* We thank Martin Hofmann for pointing us to the topic of analyzing modular arithmetic and the anonymous referees for valuable comments.

## References

1. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *5th ACM Symp. on Principles of Programming Languages (POPL)*, pages 84–97, 1978.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
3. P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 169–192. LNCS 493, Springer-Verlag, 1991.
4. S. Gulwani and G. Necula. Discovering Affine Equalities Using Random Interpretation. In *30th ACM Symp. on Principles of Programming Languages (POPL)*, pages 74–84, 2003.
5. J. Hafner and K. McCurley. Asymptotically Fast Triangularization of Matrices over Rings. *SIAM J. of Computing*, 20(6):1068–1083, 1991.
6. M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
7. J. Leroux. *Algorithmique de la Vérification des Systèmes à Compteurs: Approximation et Accélération*. PhD thesis, Ecole Normale Supérieure de Cachan, 2003.
8. M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *31st Int. Coll. on Automata, Languages and Programming (ICALP)*, pages 1016–1028. Springer Verlag, LNCS 3142, 2004.
9. M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. *Information Processing Letters (IPL)*, 91(5):233–244, 2004.
10. M. Müller-Olm and H. Seidl. Interprocedural Analysis of Modular Arithmetic. Technical Report 789, Fachbereich Informatik, Universität Dortmund, 2004.
11. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
12. T. Reps, S. Schwoon, and S. Jha. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. In *Int. Static Analysis Symposium (SAS)*, pages 189–213. LNCS 2694, Springer-Verlag, 2003.
13. A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, ETH Zürich, Diss. ETH No. 13922, 2000.