Interprocedural Analysis of Modular Arithmetic

Markus Müller-Olm*
FernUniversität Hagen
LG Praktische Informatik 5
58084 Hagen, Germany

Helmut Seidl
TU München
Lehrstuhl für Informatik II
80333 München, Germany

Abstract

We consider integer arithmetic modulo a power of 2 as provided by mainstream programming languages like C or Java. The difficulty here is that the ring $\mathbb{Z}_m$ of integers modulo $m = 2^w$, $w > 1$, may have zero divisors and thus cannot be embedded into a field. Not withstanding that, we present precise intra- and inter-procedural algorithms for inferring for every program point u, all affine relations between program variables valid at u. Our algorithms run in time linear in the program size and polynomial in the number of program variables. Moreover, these algorithms can be implemented by using the same modular integer arithmetic as the target language to be analyzed. We also explain how the methods can be extended to arithmetic w.r.t. rings $\mathbb{Z}_m$ for arbitrary $m > 1$.

1 Introduction

Analyses for automatically finding linear invariants in programs have been studied for a long time [8, 7, 9, 14, 11, 10]. None of these analyses, however, can find out, that the linear invariant $\mathtt{n} - 1 = 0$ holds upon termination of the following simple Java program, i.e., that `n` is actually a constant of value 1 at program exit:

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

In order to convince yourself that this is indeed the case, run the program twice, with and without a parameter.

*On leave from Universität Dortmund, FB 4, LS 5, 44221 Dortmund, Germany.

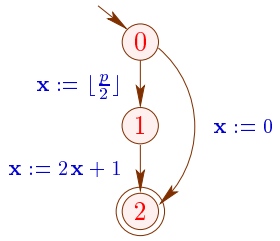

Figure 1: $\mathbb{Z}_p$ interpretation is unsound.

Why is this? In order to allow implementing arithmetic operations by the efficient instructions provided by processors, Java, like other programming languages, performs arithmetic operations for expressions of the 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` [6, p. 32]. Variable `n` is a constant of value 1 because the equation $21 * 1022611261 = 21 - 20$ is valid modulo 2^{32}. In order to work with mathematical structures with nice properties, however, analyses for finding linear invariants typically interpret variables by members from a field, often from $\mathbb{Q}$ the set of rational numbers [8, 11, 12], or $\mathbb{Z}_p = \mathbb{Z}/(p\mathbb{Z})$ where p is a prime number [7]. Note that even an analysis that interprets variables by integers, i.e., by members from $\mathbb{Z}$, cannot detect that x is a constant, because $21 * 1022611261 \neq 1$. Thus, such analyses are inherently incomplete with respect to the modulo interpretation used in practice. Even worse: analyses based on $\mathbb{Z}_p$ may yield unsound results. 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 $\mathbb{Z}_m$.

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 modulo arithmetic. Our analyses are thus more precise than analyses based on computing

over $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{Z}_p$ and, in contrast to analyses based on computing over $\mathbb{Z}_p$, they are sound w.r.t. the arithmetic used in mainstream programming languages.

We generalize our analyses based on techniques from linear algebra that we have studied previously [12, 11]. The major new difficulty is that unlike $\mathbb{Q}$ and $\mathbb{Z}_p$, $\mathbb{Z}_m$ is not a field. In particular, $\mathbb{Z}_m$ now may have zero divisors implying that no longer every non-zero element is invertible. Therefore, standard 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 $\mathbb{Z}_m$. We establish the properties of $\mathbb{Z}_m$ that still enable us to implement similar algorithms as in [12, 11] and show how this can be done.

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: they can straightforwardly be implemented using modulo arithmetic. Clearly, if we implement an algorithm based on computing in $\mathbb{Q}$, we must use some representation for rational numbers. If we use 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. In this case we must cope with overflows. Or we represent integers by arbitrarily long bit strings. In this case the size of our representation might explode. If we compute over $\mathbb{Z}_p$ also special care is needed to get the arithmetic right. The algorithms proposed in this paper, however, can simply be implemented using the modulo arithmetic provided by the host language itself. In particular, without any additional effort this totally bans the danger of 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 prime powers m 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 introduce our basic notion of programs together with their concrete semantics. In section 4, we introduce affine relations and show how the basic technology provided in [11] for fields can be adapted to work also for rings $\mathbb{Z}_m$ where m is a prime power such as 2^{32}. In section 5, we generalize these results to rings $\mathbb{Z}_m$ for arbitrary $m > 1$. In section 6, we explain how to compute all valid affine relations over $\mathbb{Z}_m$ in absence of procedures. Here, it suffices to show how to modify the algorithm from [12] to work with modular arithmetic. Finally in section 7 we summarize and explain further directions of research.

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

In this section we collect basic facts about the residue class ring $\mathbb{Z}_m$ for $m = 2^w$, $w \geq 1$. Similar properties hold for arbitrary prime powers instead of 2^w. Since we will use the more general results in section 5, we present our results for $m = p^w$ where p is any prime and $w \geq 1$.

Lemma 1 *Assume $a \in \mathbb{Z}_m$ is different from 0. Then we have:*

1. *If $p \mid a$, i.e., p divides a, 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 $p \nmid a$, then a is invertible, i.e., $a \cdot b = 1$ modulo m for some $b \in \mathbb{Z}_m$. Moreover, the inverse b can be computed using arithmetic modulo m in time $\mathcal{O}(\log(m))$.*

Proof: Assume $a = p \cdot a'$. Then $a \cdot p^{w-1} = p^w \cdot a' = 0$ modulo m. If, on the other hand, p does not divide a, 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. Note, however, that this algorithm cannot be executed modulo m. In the case where $w = 1$, we know that $\mathbb{Z}_m$ is in fact the field $\mathbb{Z}_p$. The invertible elements of $\mathbb{Z}_p$ form a cyclic group of order $p - 1$. Thus, the inverse of a is simply given by a^{p-2}. If on the other hand $w > 1$, we may use the Euclidean algorithm to determine integers x_1 and y_1 with $1 = a \cdot x_1 + p^{w-1} \cdot y_1$. By computing the p-th power of both sides of this equation, we obtain:

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

Every summand of the right-hand side except of the very first contains p^w as a factor and equals thus 0 modulo m. Therefore, b can be chosen as $a^{p-1} x_1^p$ (modulo m). Powers $p - 2$, $p - 1$ or p can be computed with $\mathcal{O}(\log(p))$ operations. Since the Euclidean algorithm uses at most $\mathcal{O}(\log(m))$ operations, the complexity statement follows.

Example 1 *Consider $p = 2$, $w = 32$ 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.* □

Since computing of inverses can be rather expensive, we will avoid these whenever possible. For $a \in \mathbb{Z}_m$, we define the *rank* of a as $r \in \{0, \ldots, w\}$ iff $a = p^r \cdot a'$ 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 for $p = 2$, 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 easily be computed with $\mathcal{O}(\log(w)) = \mathcal{O}(\log\log(m))$ arithmetic operations using a variant of binary search.

A subset $M \subseteq \mathbb{Z}_m^k$ of vectors[1] $(x_1, \ldots, x_k)^{\mathrm{t}}$ 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 *generator* of M iff $M = \{\sum_{i=1}^{r} r_i g_i \mid r \geq 0, r_i \in \mathbb{Z}_m, g_i \in G\}$. In this case, we also say that M is generated by G and write $M = \langle G \rangle$.

For a non-zero vector $x = (x_1, \ldots, x_k)^{\mathrm{t}}$, we call i the leading index iff $x_i \neq 0$ and $x_{i'} = 0$ for all $i' < i$. In this case, we also call x_i the leading entry of x. A set of non-zero vectors is in *triangular* iff for all distinct vectors $x, x' \in G$, the leading indices of x and x' are pairwise distinct. Note that every set G in triangular form contains at most k elements. Accordingly, 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 $(k - s) \cdot w$.

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 $p^r d$ (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 $\bar{G} = G \cup \{x\}$.
2. i is the leading index of some vector $y \in G$ where the leading entry equals $p^{r'} d'$.
   (a) If $r' \leq r$, then we compute $x' = d' \cdot x - p^{r-r'} d \cdot y$. 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 $y' = d \cdot y - p^{r'-r} d' \cdot x'$. Thus, the i-th entry of y' equals 0, and we proceed with G' and y'.

It should be clear that eventually, we arrive at a set $\bar{G}$ having the desired properties. Moreover, either $\bar{G}$ 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 $\mathcal{O}(k \cdot (k + \log\log(m)))$ 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 generators of cardinality at most k.*

2. Given any finite set G' of generators of cardinality n, we can compute in time $\mathcal{O}(n \cdot k \cdot (k + \log\log(m)))$ a set G of cardinality at most k 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 $s \leq k \cdot w = k \cdot \log(m)$.

Proof: The second statement follows from our construction of triangular sets of generators. Starting from the empty set, which is in triangular form for trivial reasons, we add the n vectors in G' one after the other with the procedure described above. The complexity estimation follows by summing up the operations of these n inclusions.

Since $\mathbb{Z}_m^k$ is finite, the first statement trivially follows from the second statement. Therefore, 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. Then we construct a sequence of triangular sets generating the same modules as follows. G'_0 is the triangular set constructed for G_0 whereas 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 triangular sets 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 *Assume again $p = 2$. In order to keep the numbers small, we choose here and in the following examples of this section $w = 4$, i.e., $m = 16$. Then consider the vectors $x = (2, 6, 9)^{\mathrm{t}}$ and $y = (0, 2, 4)^{\mathrm{t}}$ with leading indices 1 and 2 and both with leading entry 2. Thus, the set $G = \{x, y\}$ is triangular. Let $z = (1, 2, 1)^{\mathrm{t}}$. We construct a triangular set of generators equivalent to $\{x, y, z\}$ from G and z as follows. 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 $x' = x - 2 \cdot z = (0, 2, 7)^{\mathrm{t}}$. 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,*

[1] 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).

we can subtract a suitable multiple of y to bring the second component of x' to 0 as well. We compute $x'' = x' - 1 \cdot y = (0,0,3)^{\mathrm{t}}$. As triangular set we finally return $\bar{G} = \{z, y, x''\}$. □

Note that for a set of generators G being triangular, does not imply being a *minimal* set of generators. For $p = 2$ and $w = 3$ consider, e.g., the triangular set $G = \{x, y\}$ where $x = (4,1)^{\mathrm{t}}, y = (0,2)^{\mathrm{t}}$. Multiplying x with 2 results in: $2 \cdot x = (8,2)^{\mathrm{t}} = (0,2)^{\mathrm{t}} = y$. Thus as a multiple, the second vector y in G is redundant implying that G is not a minimal set of generators.

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

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

We turn to computing 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 may w.l.o.g. 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 $a_{ij} \in \mathbb{Z}_m$, $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathrm{t}}$ is a column vector of unknowns and $b = (b_1, \ldots, b_k)^{\mathrm{t}}$ 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 $\mathbf{0} = (0, \ldots, 0)^{\mathrm{t}}$. It is well-known that, if the system $A\,\mathbf{x} = b$ has at least one solution x, then the set of all solution 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 *Let A denote a diagonal $(k \times k)$-matrix over $\mathbb{Z}_m$ where the diagonal elements are given by $a_{ii} = p^{w_i} d_i$ for invertible d_i ($w_i = w$ means $a_{ii} = 0$). Then we have:*

1. *The system $A\,\mathbf{x} = b$ has a solution iff for all i, the rank w_i of a_{ii} does not exceed the rank of b_i.*
2. *If $A\,\mathbf{x} = b$ is solvable, then one solution is given by: $x = (x_1, \ldots, x_k)^{\mathrm{t}}$ with $x_i = p^{w_i' - w_i} \cdot d_i^{-1} b_i'$ where $b_i = p^{w_i'} b_i'$ 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: $e^{(j)} = (e_{1j}, \ldots, e_{kj})^{\mathrm{t}}, j = 1, \ldots, k$, where $e_{ij} = p^{w - w_i}$ if $i = j$ and $e_{ij} = 0$ otherwise.* □

In contrast to the situation for 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, however, that in contrast to inhomogeneous systems, a set of generators for the homogeneous system can be computed without ever computing inverses.

Example 3 *Let $p = 2$, $w = 4$, 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 from 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 $\mathcal{O}(\log\log(m) \cdot k^3)$. 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 $d = p^r d'$ 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 p^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). □

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 $\mathcal{O}(\log\log(m)\cdot k^3)$.*

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 $\mathcal{O}(\log(m)\cdot k + \log\log(m)\cdot k^3)$.

Example 4 *Consider, for $p = 2$ and $w = 4$, 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

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

□

3 Affine Programs

For a better comparison with the related work [11] on the inter-procedural analysis of programs whose variables take values in a field, we use the same conventions as there which we recall here for reasons of self-containedness. In particular, we model programs 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 the program operates on. We use $\mathbf{x}$ to denote the column vector of variables $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathrm{t}}$. 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 conveniently modeled by a k-dimensional (column) vector $x = (x_1, \ldots, x_k)^{\mathrm{t}} \in \mathbb{Z}_m^k$; x_i is the value assigned to variable $\mathbf{x}_i$. For convenience, we distinguish variables and

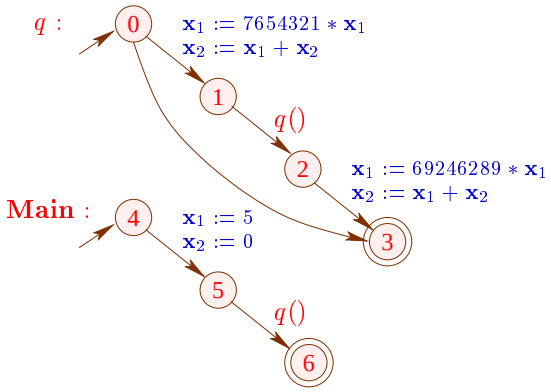

Figure 2: An inter-procedural program.

their values by their fonts. 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 $(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_k)^{\mathrm{t}}$ as usual.

In the programs we are going to analyze, we assume the basic statements either to be *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 2 with sequences of assignments. Since assignments $\mathbf{x}_j := \mathbf{x}_j$ have no effect onto the program state, they are also called skip-statements and omitted in the picture. Skip-statements can be used to abstract guards. Non-deterministic assignments $\mathbf{x}_j :=?$ can be used as a safe abstraction of statements in a source program which our analysis cannot handle precisely, for example of assignments $\mathbf{x}_j := t$ with non-affine expressions t or of read statements $\mathsf{read}(\mathbf{x}_j)$. By Stmt we denote the set of all basic statements.

In this setting, a *program* comprises a finite set Proc of *procedure names* together with one distinguished procedure **Main**. Execution starts with a call to **Main**. Each procedure name $q \in \mathsf{Proc}$ is associated with a *control flow graph* $G_p = (N_q, E_q, A_q, \mathsf{st}_q, \mathsf{ret}_q)$ consisting of:

- a set N_q of *program points*;
- a set of edges $E_q \subseteq N_q \times N_q$;
- a mapping $A_q : E_q \to \mathsf{Stmt} \cup \mathsf{Proc}$ that annotates each edge with a basic statement of the form described above or a procedure call;
- a special *entry (or start) point* $\mathsf{st}_q \in N_q$; and
- a special *return point* $\mathsf{ret}_q \in N_q$.

We assume that the program points of different procedures are disjoint: $N_q \cap N_{q'} = \emptyset$ for $q \neq q'$. This can always be enforced, e.g., by renaming program points.

We write N for $\bigcup_{q \in \mathsf{Proc}} N_q$, E for $\bigcup_{q \in \mathsf{Proc}} E_q$, and A for $\bigcup_{q \in \mathsf{Proc}} A_q$. We agree that $\mathsf{Base} = \{e \mid A(e) \in \mathsf{Stmt}\}$ is the set of base edges and $\mathsf{Call}_q = \{e \mid A(e) \equiv q\}$ is the set of edges that call procedure q.

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 $t_0, \ldots, t_k \in \mathbb{Z}_m$. We write Runs for the set of runs. The set of runs *reaching program point* $u \in N$ is characterized as the least solution of a system of subset constraints on run sets. First, the set of executions of base edges e are defined. If e is annotated by an affine assignment, i.e., $A(e) \equiv \mathbf{x}_j := t$, there is just one single execution: $\mathbf{S}(e) = \{\mathbf{x}_j := t\}$. Base edges e annotated by a non-deterministic assignment $\mathbf{x}_j :=?$ are meant to assign *any* value. Therefore, they give rise to the set of all constant assignments:

$$\mathbf{S}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{Z}_m\}.$$

In order to cope with procedure calls, we proceed in two steps. First, we characterize *same-level runs*. Same-level runs of procedures capture complete runs of procedures in isolation. In order to accumulate these sets, we also consider same-level runs of program nodes u, i.e., runs reaching u in a procedure q from a call to q on same-level, i.e., after all procedures called by q have terminated. The sets of same-level runs of procedures and program nodes are the smallest solution of the constraint system $\mathbf{S}$:

$$\begin{array}{lllll}
[\mathrm{S1}] & \mathbf{S}(q) & \supseteq \mathbf{S}(\mathsf{ret}_q) & & \\
[\mathrm{S2}] & \mathbf{S}(\mathsf{st}_q) & \supseteq \{\varepsilon\} & & \\
[\mathrm{S3}] & \mathbf{S}(v) & \supseteq \mathbf{S}(u)\,;\,\mathbf{S}(e) & \text{if } e = (u, v) \in \mathsf{Base} & \\
[\mathrm{S4}] & \mathbf{S}(v) & \supseteq \mathbf{S}(u)\,;\,\mathbf{S}(q) & \text{if } e = (u, v) \in \mathsf{Call}_q &
\end{array}$$

Here, "ε" denotes the empty run, and the operator ";" denotes concatenation of run sets. By [S1], the set of same-level runs of a procedure q comprises all same-level runs reaching the return point of q. By [S2], the set of same-level runs of the entry point of a procedure contains the empty run. By [S3] and [S4], the same-level runs for a program point v are obtained by considering all ingoing edges $e = (u, v)$. In both cases, we concatenate the same-level runs reaching u with all runs

corresponding to the edge. If e is a base edge, we concatenate with the edges from $\mathbf{S}(e)$. If e is a call to a procedure q, we take all same-level runs of q.

In the second step, the runs reaching program points are characterized. They are the smallest solution of the constraint system $\mathbf{R}$:

$$\begin{array}{lllll} [\mathrm{R1}] & \mathbf{R}(\mathbf{Main}) & \supseteq \{\epsilon\} & & \\ [\mathrm{R2}] & \mathbf{R}(q) & \supseteq \mathbf{R}(u) & \text{if } (u,_) \in \mathsf{Call}_q \\ [\mathrm{R3}] & \mathbf{R}(u) & \supseteq \mathbf{R}(q)\,;\,\mathbf{S}(u) & \text{if } u \in N_q \end{array}$$

By [R1], the procedure **Main** is reachable by the empty run. By [R2], every procedure q is reachable by a run reaching a call of q. By [R3], we obtain a run reaching a program point u in some procedure q, by composing a run reaching q with a same-level run reaching u.

Sets of program executions can be seen as a symbolic operational semantics of procedural flow graphs which describes all sequences of assignments possibly reaching program points. Obviously, each of these runs gives rise to a transformation of the underlying program state $x \in \mathbb{Z}_m^k$. 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 Id is the identical mapping and $[\![ra]\!] = [\![a]\!] \circ [\![r]\!]$.

A closer look reveals that the state transformation of an affine assignment $\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i \mathbf{x}_i$ in fact is an affine transformation. Hence, it can be written in the form $[\![\mathbf{x}_j := t]\!]x = Ax + b$ with a matrix $A \in \mathbb{Z}_m^{k\times k}$ and a (column) vector $b \in \mathbb{Z}_m^k$ where A and b are indicated below:

$$A = \left(\begin{array}{c|c} I_{j-1} & 0 \\ \hline t_1 \;\ldots\; t_k & \\ \hline 0 & I_{k-j} \end{array}\right) \qquad b = \left(\begin{array}{c} 0 \\ \hline t_0 \\ \hline 0 \end{array}\right) \tag{1}$$

Here, I_i is the unit matrix with i rows and columns and 0 denotes zero matrices and vectors of appropriate dimension. In b, t_0 appears as j-th component. Note that this observation is independent of whether the domain of values of variables is a field or just a ring. 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 $\mathbb{Z}_m$. For any run r, let $A_r \in \mathbb{Z}_m^{k\times k}$ and $b_r \in \mathbb{Z}_m^k$ be such that $[\![r]\!]x = A_r x + b_r$.

4 Affine Relations and Weakest Preconditions

We now turn to the definition of *affine relations* over $\mathbb{Z}_m$ which 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 $\mathbb{Z}_m$-modul 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 scalar product. We write $y \models a$ to denote this fact.

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. Literally the same argumentation works as well in the more general case of arbitrary rings. More specifically, this linear transformation is given by the following $(k+1)\times(k+1)$ matrix W_r:

$$W_r = \left(\begin{array}{c|c} 1 & b_r^{\mathrm{t}} \\ \hline 0 & A_r^{\mathrm{t}} \end{array}\right) \tag{2}$$

Note that the only operations we have to apply to A_r and b_r are transposition and combination into a $(k+1)\times(k+1)$-matrix. We find that for every $x \in \mathbb{Z}_m^k$:

$$[\![r]\!]x \models a \quad \text{iff} \quad x \models W_r\, a\,. \tag{3}$$

Consequently, the matrix W_r provides us also over $\mathbb{Z}_m$ with a finite description of the weakest precondition transformer for affine relations of a single program execution r.

Recall that also over $\mathbb{Z}_m$, the only affine relation which is true for *all program states* is the relation $\mathbf{0} = (0, \ldots, 0)^{\mathrm{t}}$. Thus, the affine relation a is valid after run r iff $W_r\, a = \mathbf{0}$, because the initial state is arbitrary. Moreover, the affine relation a is valid at a program point u, iff it is valid after all runs $r \in \mathbf{R}(u)$. Summarizing, we obtain here:

Lemma 4 *The affine relation $a \in \mathbb{Z}_m^{k+1}$ is valid at program point u iff $W_r a = \mathbf{0}$ for all $r \in \mathbf{R}(u)$.* □

This is good news, since this shows that as in the case of fields, we may use the set $\mathcal{W} = \{W_r \mid r \in \mathbf{R}(u)\}$ to solve the validity problem for affine relations. While in our case this set is finite because $\mathbb{Z}_m$ is finite, it can be very large. We are thus left with the problem to represent $\mathcal{W}$ 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 at least can be considered as a $\mathbb{Z}_m$-module isomorphic to $\mathbb{Z}_m^{(k+1)^2}$. We observe:

Lemma 5 *Let M denote a set of $n \times n$ matrices from $\mathbb{Z}_m^{n\times n}$.*

a) For every $W \in M$, the set $\{a \in \mathbb{Z}_m^n \mid W\,a = \mathbf{0}$ forms a submodule of $\mathbb{Z}_m^n$.

b) As an intersection of $\mathbb{Z}_m$-modules, the set $\{a \in \mathbb{Z}_m^n \mid \forall W \in M : W\,a = \mathbf{0}\}$ forms a submodule of $\mathbb{Z}_m^n$.

c) For every $a \in \mathbb{Z}_m^n$, the following three statements are equivalent:

- *$W\,a = \mathbf{0}$ for all $W \in M$;*
- *$W\,a = \mathbf{0}$ for all $W \in \langle M\rangle$.*
- *$W\,a = \mathbf{0}$ for all W in a generating system of $\langle M\rangle$.* □

We conclude that we can work with $\langle\mathcal{W}\rangle$, i.e., the submodule of $\mathbb{Z}_m^{(k+1)\times(k+1)}$ generated by $\mathcal{W}$ without losing interesting information. By Theorem 1, $\langle\mathcal{W}\rangle$ can be described by a generating system of at most $(k+1)^2$ matrices. Based on these observations, we can determine the set of all affine relations at program point u from a generating system of the $\mathbb{Z}_m$-module $\langle\{W_r \mid r \in \mathbf{R}(u)\}\rangle$.

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

a) Affine relation $a \in \mathbb{Z}_m^{k+1}$ is valid at program point 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 $\mathcal{O}(k^4 \cdot (k + \log\log(m)))$.

Proof: Statement a) follows directly from Lemma 4 and Lemma 5,c).

For seeing b), consider that by a) the affine relation a is valid at u iff a is a solution of all the equations

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

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 $\mathcal{O}(k^4 \cdot (k + \log\log(m)))$. 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 $\mathcal{O}(\log\log(m) \cdot k^3)$ operations. The latter amount, however, is dominated by the former, and the complexity statement follows. □

As in the case of fields, we are left with the task to compute, for every program point u, a generating system of $\langle\{W_r \mid r \in \mathbf{R}(u)\}\rangle$. Following the approach there, we try to compute this submodule of $\mathbb{Z}_m^{(k+1)\times(k+1)}$ as an abstract interpretation of the constraint system for $\mathbf{R}(u)$ from Section 3. From Section 2 we know that $\mathbf{Sub}(\mathbb{Z}_m^{(k+1)\times(k+1)})$ 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 described by the mapping $\alpha : 2^{\mathsf{Runs}} \to \mathbf{Sub}(\mathbb{Z}_m^{(k+1)\times(k+1)})$:

$$\alpha(R) = \langle\{W_r \mid r \in R\}\rangle\,.$$

where:

$$\begin{aligned}\alpha(\emptyset) &= \langle\emptyset\rangle = \{\mathbf{0}\}\\ \alpha(\{r\}) &= \langle\{W_r\}\rangle\end{aligned}$$

for a single run r. By Equation (2) we get for the empty run,

$$\alpha(\{\epsilon\}) = \langle\{I_{k+1}\}\rangle$$

because $A_\varepsilon = I_k$ and $b_\varepsilon = \mathbf{0}$.

We also verify that the mapping α is monotonic (w.r.t. subset ordering on sets of runs and submodules) and commutes with arbitrary unions. In order to solve the constraint system for the run sets $\mathbf{R}(u)$ over abstract domain $\mathbf{Sub}(\mathbb{Z}_m^{(k+1)\times(k+1)})$, we now must provide adequate abstract versions of the operators and constants in this constraint system. In the case of fields, we have provided matrix multiplication (lifted to vectorspaces of matrices) as abstraction of the concatenation of run sets. Since matrix multiplication does not involve computation of inverses, we may use the same idea for $\mathbb{Z}_m$ as well. Thus, for $M_1, M_2 \subseteq \mathbb{Z}_m^{(k+1)\times(k+1)}$, we define:

$$M_1 \circ M_2 = \langle\{A_1A_2 \mid A_i \in M_i\}\rangle\,.$$

The following lemma and its proof is literally identical to the corresponding lemmas for fields. We have included it for completeness in order to demonstrate that the techniques directly carry over.

Lemma 6 *For all sets of matrices M_1, M_2,*

$$\langle M_1\rangle \circ \langle M_2\rangle = M_1 \circ M_2\,.$$

Proof: Observe first that $\langle M_i\rangle \supseteq M_i$ and therefore,

$$\langle M_1\rangle \circ \langle M_2\rangle \supseteq M_1 \circ M_2$$

by monotonicity of "$\circ$".

For the reverse inclusion, consider arbitrary elements $B_i = \sum_j \lambda_j^{(i)} \cdot A_j^{(i)}$ in $\langle M_i\rangle$ for suitable $A_j^{(i)} \in M_i$. Then

$$B_1B_2 = \sum_m \sum_j \lambda_m^{(1)}\lambda_j^{(2)} \cdot A_m^{(1)}A_j^{(2)}$$

by linearity of matrix multiplication. Since each $A_m^{(1)} A_j^{(2)}$ is contained in $M_1 \circ M_2$, $B_1 B_2$ is contained in $M_1 \circ M_2$ as well. Therefore, also the inclusion "$\subseteq$" follows. □

Lemma 6 implies that a generating system for $M_1 \circ M_2$ can be computed from generating systems G_1, G_2 for M_1 and M_2 by multiplying each matrix in G_1 with each matrix in G_2.

The following lemma that ensures that "$\circ$" precisely abstracts the concatenation of run sets is proved as for fields:

Lemma 7 *Let $R_1, R_2 \subseteq \mathsf{Runs}$. Then*

$$\alpha(R_1) \circ \alpha(R_2) = \alpha(R_1 \,;\, R_2)\,.$$

□

The abstractions of base edges are also very similar to the field case. For a base edge $e \in \mathsf{Base}$ annotated by an affine assignment, i.e., $A(e) \equiv x_j := t$ where $t \equiv t_0 + \sum_{i=1}^{n} t_i \mathbf{x}_i$, we have $\mathbf{S}(e) = \{\mathbf{x}_j := t\}$. The weakest precondition for an affine relation $a \in \mathbb{Z}_m^{k+1}$ is computed by substituting t into $\mathbf{x}_j$ of the corresponding affine combination. Thus, the corresponding abstract transformer is given by

$$\alpha(\mathbf{S}(e)) \;=\; \left\langle \left\{ \left(\begin{array}{c|c|c} I_j & t_0 & 0 \\ \hline 0 & \vdots & \hline I_{k-j} \\ & t_k & \end{array} \right) \right\} \right\rangle$$

For a base edge $e \in \mathsf{Base}$ annotated by $\mathbf{x}_j :=?$ we have $\mathbf{S}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{Z}_m\}$ by definition. In order to abstract this set of $m = p^w$ runs, we observe that $\alpha(\mathbf{S}(e))$ is in fact generated by just two matrices:

Lemma 8

$$\begin{aligned} \alpha(\mathbf{S}(e)) &= \alpha(\{\mathbf{x}_j := c \mid c \in \mathbb{Z}_m\}) \\ &= \langle\{T_0, T_1\}\rangle\,, \end{aligned}$$

where $T_c = W_{\mathbf{x}_j := c}$ is the matrix obtained from I_{k+1} by replacing the $j+1$-th column with $(c, 0, \ldots, 0)^{\mathrm{t}}$.

Proof: Only the second equation requires a proof. From Equations (1) and (2) we get $\alpha(\{\mathbf{x}_j := c \mid c \in \mathbb{Z}_m\}) = \langle\{T_c \mid c \in \mathbb{Z}_m\}\rangle$. We verify: $T_c = (1-c) \cdot T_0 + c \cdot T_1$. Hence, $T_c \in \langle\{T_0, T_1\}\rangle$ and we conclude $\langle\{T_c \mid c \in \mathbb{Z}_m\}\rangle = \langle\{T_0, T_1\}\rangle$. □

Thus, as in the case of fields, we obtain the abstraction by taking the matrices for just two different values. Note, however, that other than in the case of fields, we cannot choose these values arbitarily. For example for $p = 2$, T_0 and T_2 are not sufficient to combine T_1.

Proceeding as in the case of fields, we take the constraint systems $\mathbf{S}$ and $\mathbf{R}$ for run sets and construct the constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$ by the application of α. The variables in the new constraint systems now take submodules of $\mathbb{Z}_m^{(k+1)\times(k+1)}$ as values. We apply α to the occurring constant sets $\{\epsilon\}$ and $\mathbf{S}(e)$ and replace the concatenation operator ";" with "$\circ$":

$$\begin{array}{lll} \mathbf{S}_\alpha(q) & \supseteq \mathbf{S}_\alpha(\mathsf{ret}_q) & \\ \mathbf{S}_\alpha(\mathsf{st}_q) & \supseteq \langle\{I_{k+1}\}\rangle & \\ \mathbf{S}_\alpha(v) & \supseteq \mathbf{S}_\alpha(u) \circ \alpha(\mathbf{S}(e)) & \text{if } e = (u,v) \in \mathsf{Base} \\ \mathbf{S}_\alpha(v) & \supseteq \mathbf{S}_\alpha(u) \circ \mathbf{S}_\alpha(q) & \text{if } e = (u,v) \in \mathsf{Call}_q \\ \mathbf{R}_\alpha(\mathbf{Main}) & \supseteq \langle\{I_{k+1}\}\rangle & \\ \mathbf{R}_\alpha(q) & \supseteq \mathbf{R}_\alpha(u) & \text{if } (u,_) \in \mathsf{Call}_q \\ \mathbf{R}_\alpha(u) & \supseteq \mathbf{R}_\alpha(q) \circ \mathbf{S}_\alpha(u) & \text{if } u \in N_q \end{array}$$

Again as in the case of fields, the resulting constraint system can be solved by computing on generating systems. In constrast to fields, however, we do no longer have the notion of a basis available. Instead, we rely on sets of generators in triangular form. In order to avoid full solving of a system of equations over $\mathbb{Z}_m$ whenever a new vector is going to be 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.

As usual, 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 $|N| + |E|$. If all program nodes have bounded out-degree (which is typically the case), this implies that $n = \mathcal{O}(|N|)$.

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

a) The values:
$\langle\{W_r \mid r \in \mathbf{S}(u)\}\rangle$, $u \in N$,
$\langle\{W_r \mid r \in \mathbf{S}(q)\}\rangle$, $q \in \mathsf{Proc}$,
$\langle\{W_r \mid r \in \mathbf{R}(q)\}\rangle$, $q \in \mathsf{Proc}$, *and*
$\langle\{W_r \mid r \in \mathbf{R}(u)\}\rangle$, $u \in N$,
are the least solutions of the constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$, respectively.

b) These values can be computed in time $\mathcal{O}(\log(m) \cdot n \cdot k^6 \cdot (k^2 + \log\log(m)))$.

c) The sets of all valid affine relations at program point u, $u \in N$, can be computed in time $\mathcal{O}(\log(m) \cdot n \cdot k^6 \cdot (k^2 + \log\log(m)))$.

In our main application, $m = 2^w$ where $w = \log(m)$ equals 32 or 64. The term $\log\log(m)$ in the complexity estimation accounts for the necessary rank computations. In our application, $\log\log(m)$ 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 $\log(m) = 32$ 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 lengthes of ascending chains in the lattice but stabilize much earlier.

Proof: Statement a) asserts that the least solution of constraint systems $\mathbf{S}_\alpha$ and $\mathbf{R}_\alpha$ is obtained from the least solution of $\mathbf{S}$ and $\mathbf{R}$ by applying the abstraction α. As in the case of fields, this follows from the Transfer Lemma known in fixpoint theory (see, e.g., [1, 3]), which can be applied since α commutes with arbitrary unions, the concatenation operator is precisely abstracted by the operator $\circ$ (Lemma 7), and the constant run sets $\{\varepsilon\}$ and $\mathbf{S}(e)$ are replaced by their abstractions $\alpha(\{\varepsilon\}) = \langle\{I_{k+1}\}\rangle$ and $\alpha(\mathbf{S}(e))$, respectively.

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 [13, 2, 5]. The corresponding semi-naive fixpoint algorithm for computing $\mathbf{S}_\alpha(r)$ for all program points or procedures r is shown in Figure 3. In order to deal uniformly with deterministic and non-deterministic assignments e, we introduce sets $\mathcal{M}(e)$ where $\mathcal{M}(e) = \{T_0, T_1\}$ if $e \in Base$ is annotated by $\mathbf{x}_j :=?$ and $\mathcal{M}(e) = \{T\}$ if e is nnotated by $\mathbf{x}_j := t$, $t \equiv t_0 + \sum_{i=1}^n t_i \mathbf{x}_i$, where T is obtained from I_{k+1} by replacing the $j+1$ column by $(t_0, \ldots, t_k)$.

Informally, the algorithm works as follows. It maintains a workset W holding pairs (r, M) where r is a node or procedure 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 $\emptyset$. Accordingly, the workset receives the elements $(\mathsf{st}_q, I_{k+1}), q \in \mathsf{Proc}$. In the main loop, the algorithm successively takes pairs (r, M) out of the workset and propagates the new value for r to all uses of the variable $\mathbf{S}_\alpha(r)$ until the workset is empty.

If r is a procedure, then the new matrix M must be propagated to all edges (u, v) at which r is called. At every such edge, the algorithm computes the set $new = \{M_1 \cdot M \mid M_1 \in \mathbf{S}_\alpha(u)\}$ 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 $\mathbf{S}_\alpha(v)$. Thus, we

```
W = ∅ ;
forall (u ∈ N)  S_α(u) = ∅;
forall (q ∈ Proc) {
    S_α(q) = ∅;
    S_α(st_q) = {I_{k+1}};
    W = W ∪ {(st_q, I_{k+1})};
}

while (W ≠ ∅) {
    (r, M) = Extract(W);
    if (r ∈ Proc) {
        forall ((u, v) ∈ Call_r) {
            new = {M_1 · M | M_1 ∈ S_α(u)};
            forall (M_2 ∈ new)
                if (M_2 ∉ ⟨S_α(v)⟩) {
                    S_α(v) = Add(S_α(v), M_2);
                    W = W ∪ {(v, M_2)};
                }
        }
    }  else { // i.e., if r ∈ N
        if (r ≡ ret_q)
            if (M ∉ ⟨S_α(q)⟩) {
                S_α(q) = Add(S_α(q), M);
                W = W ∪ {(q, M)};
            }
        forall ((r, v) ∈ E) {
            if (e = (r, v) ∈ Call_q)
                new = {M · M_1 | M_1 ∈ S_α(q)};
            else  // i.e., if e ∈ Base
                new = {M · M_1 | M_1 ∈ M(q)};
            forall (M_2 ∈ new)
                if (M_2 ∉ ⟨S_α(v)⟩) {
                    S_α(v) = Add(S_α(v), M_2);
                    W = W ∪ {(v, M_2)};
                }
        }
    }
}
```

Figure 3: The semi-naive fixpoint algorithm for $\mathbf{S}_\alpha$.

successively check for every $M_2 \in new$ whether or not it is contained in the module generated by $\mathbf{S}_\alpha(v)$. If not, then we *add* it to $\mathbf{S}_\alpha(v)$ and insert the pair (v, M_2) into the workset.

Now assume that r is not a procedure but a program point. First, we consider the case where r is the return point of some procedure q. Then the new matrix M for r must be added to the set $\mathbf{S}_\alpha(q)$, if M is not subsumed by the matrices in $\mathbf{S}_\alpha(q)$. In this case the pair (q, M) is inserted into the workset. Second, we consider all out-going edges (r, v) of r. If (r, v) is

a call to some procedure q, then we compute the set $new = \{M \cdot M_1 \mid M_1 \in \mathbf{S}_\alpha(q)\}$ consisting of all candidate matrices potentially to be added to the set $\mathbf{S}_\alpha(v)$. If on the other hand, $e = (r, v)$ is a basic edge, then the set new is determined as $new = \{M \cdot M_1 \mid M_1 \in \mathcal{M}(e)\}$. 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 $\mathbf{S}_\alpha(v)$. If not, then we *add* it to $\mathbf{S}_\alpha(v)$ 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 be inserted into W twice. Second, the $\mathbb{Z}_m$-modules $\mathbf{S}_\alpha(r)$ for every program point or procedure r are represented as a set of generators (of cardinality at most $(k+1)^2$). Finally, we do not perform an exact check whether or not a new matrix M is subsumed by a set of matrices. Instead, we keep the sets $\mathbf{S}_\alpha(r)$ in triangular form. Whenever a new matrix M is checked for containment in $\langle \mathbf{S}_\alpha(r) \rangle$, we compute a triangular set G of generators for $\mathbf{S}_\alpha(r) \cup \{M\}$ by means of the algorithm from Theorem 1. If the new set equals the old one, we know for sure that M is contained in $\mathbf{S}_\alpha(r)$. Otherwise, the new set at least comprises both all old vectors together with the new one (i.e., is safe) 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 multiplications and at most as many additions of a vector to a triangular set. The matrix multiplications can be performed in time $\mathcal{O}(\log(m) \cdot k^7)$. By our considerations for Theorem 1, the insertions need $\mathcal{O}(\log(m) \cdot k^6 \cdot (k^2 + \log\log(m)))$ arithmetic operations. Multiplying that with the number of fixpoint variables, we arrive at the complexity stated in statement b).

Having thus computed the values, $\mathbf{S}_\alpha(r)$, r a procedure name or program node, it remains to determine the values $\mathbf{R}_\alpha(r)$. Again we use semi-naive iteration to compute the least solution of $\mathbf{R}_\alpha$. The algorithm is shown in Figure 4. It proceeds in the same spirit as the algorithm for computing the values $\mathbf{S}_\alpha(r)$. Now, the set $\mathbf{R}_\alpha(\mathbf{Main})$ for the main procedure is initialized with $\{I_{k+1}\}$. The new matrices in the reachability set of a procedure q are then propagated to every program point u of q by multiplying with all matrices in the set $\mathbf{S}_\alpha(u)$. Moreover, whenever a new matrix arrives at the start point of an edge calling some procedure q', then this matrix must be added to the set $\mathbf{R}_\alpha(q')$. An analogous argument shows that this algorithm has essentially

```
W = ∅ ;
forall (u ∈ N)  R_α(u) = ∅;
forall (q ∈ Proc)  R_α(q) = ∅;
R_α(Main) = {I_{k+1}};
W = W ∪ {(Main, I_{k+1})};

while (W ≠ ∅) {
    (r, M) = Extract(W);
    if (r ∈ Proc) {
        forall (u ∈ N_r) {
            new = {M · M_1 | M_1 ∈ S_α(u)};
            forall (M_2 ∈ new)
                if (M_2 ∉ ⟨R_α(u)⟩) {
                    R_α(u) = Add(R_α(u), M_2);
                    W = W ∪ {(u, M_2)};
                }
        }
    } else { // i.e., if r ∈ N
        forall ((r, v) ∈ E)
            if (e = (r, _) ∈ Call_q)
                if (M ∉ ⟨R_α(q)⟩) {
                    R_α(q) = Add(R_α(q), M);
                    W = W ∪ {(q, M)};
                }

    }
}
```

Figure 4: The semi-naive fixpoint algorithm for $\mathbf{R}_\alpha$.

the same complexity as our algorithm for computing the least solution of $\mathbf{S}_\alpha$. This proves b).

Finally, for c) we recall that we know from Theorem 3 that, from generators for $\langle \{W_r \mid r \in \mathbf{R}(u)\} \rangle$ for all program points u, we can compute the sets of all valid affine relations within the stated complexity bounds. □

Example 5 *Consider again 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 given by:*

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

since $c \cdot 69246289 = 1$ mod 2^{32}. For $\mathbf{R}_\alpha(q)$, the algorithm successively finds 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:*

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

This gives us the following equation system for the affine relations valid at the exit of main:

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

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, e.g., by using the corresponding analysis over the field* $\mathbb{Q}$*.* □

5 Extension to Arbitrary Rings $\mathbb{Z}_m$

In this section we generalize our results to the analysis of linear arithmetic modulo an arbitrary $m > 1$. The crucial point here is to explain how systems of equations over the residue class ring $\mathbb{Z}_m$ can be solved. So assume that $m = m_1 \cdot \ldots \cdot m_r$, $r > 1$, for pairwise relatively prime m_i. The key observation is that every element $x \in \mathbb{Z}_m$ is uniquely determined by the values $x_i = x \bmod m_i$, $i = 1, \ldots, r$. This result is known as Chinese Remainder Theorem. More precisely by Euclid's algorithm, we can find numbers y_i, z_i such that $m_i \cdot y_i + \frac{m}{m_i} \cdot z_i = 1$. For $i = 1, \ldots, r$, we define $s_i = \frac{m}{m_i} \cdot z_i \bmod m$. Thus for every i, j,

$$s_i \bmod m_j = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases} \tag{4}$$

Since every element in $\mathbb{Z}_m$ is uniquely determined by its residues modulo the m_i, we obtain $x = x_1 s_1 + \ldots + x_r s_r \bmod m$. Note that the coefficients s_i can be constructed once and for all from the factorization $m = m_1 \cdot \ldots \cdot m_m$. Given these coefficients, the reconstruction of x needs just $\mathcal{O}(r)$ arithmetic operations. Let us call this operation *Chinese remainder reconstruction* and denote it by $x_1 * \ldots * x_r$. This operation has interesting properties. We have:

Lemma 9 *The Chinese remainder reconstruction commutes with componentwise addition and multiplication:*

1. $(x_1 * \ldots * x_r) + (y_1 * \ldots * y_r) = (x_1 + y_1) * \ldots * (x_r + y_r)$
2. $(x_1 * \ldots * x_r) \cdot (y_1 * \ldots * y_r) = (x_1 \cdot y_1) * \ldots * (x_r \cdot y_r)$

for all $x_i, y_i \in \mathbb{Z}_{m_i}$*.*

Proof: The assertion for addition immediately follows from the definition. For second assertion, it suffices to prove that the left-hand side modulo m_i equals $x_i \cdot y_i$ for all i. We have modulo m_i:

$$\begin{aligned} (x_1 * \ldots * x_r) \cdot (y_1 * \ldots * y_r) &= (\textstyle\sum_i s_i x_i) \cdot \left(\sum_j s_j y_j\right) \\ &= \textstyle\sum_{i,j} s_i s_j x_i y_j \\ &= x_i y_i \end{aligned}$$

where the last equality follows from equation (4). □

For the following, we extend the modulo operation to vectors. Thus for a vector $x = (x_1, \ldots, x_k)^{\mathrm{t}}$, we denote by $x \bmod q$ the vector $(x_1 \bmod q, \ldots, x_k \bmod q)^{\mathrm{t}}$. Moreover, for vectors $x_i \in \mathbb{Z}_{m_i}^k$, we denote by $x_1 * \ldots * x_k$ the vector in $\mathbb{Z}_m$ whose j-th component is obtained from the j-th components of the vectors x_i by applying the Chinese remainder reconstruction. We show that this reconstruction can be generalized to modules. Assume $M \subseteq \mathbb{Z}_m^k$ is a $\mathbb{Z}_m$-module and for $i = 1, \ldots, r$, $M_i = \{x \bmod m_i \mid x \in M\} \subseteq \mathbb{Z}_{m_i}^k$. Then we have:

Theorem 5 *1. If* G *is a set of generators of* M*, then* $G_i = \{x \bmod m_i \mid x \in G\}$ *is a set of generators of* M_i*.*

2. *For $i = 1, \ldots, r$, let $G_i = \{g_1^{(i)}, \ldots, g_n^{(i)}\}$ denote a set of generators for M_i. Then $G = \{g_1, \ldots, g_n\}$ is a set of generators of M for $g_j = g_j^{(1)} * \ldots * g_j^{(n)}$.*

Proof: We only prove the second statement. First, we prove that $G \subseteq M$ and therefore also $\langle G \rangle \subseteq M$. Let $\pi_i = 0 * \ldots * 0 * 1 * 0 * \ldots * 0$ be the Chinese reconstruction of $r-1$ zeros together with a single 1 at position i. Now consider any Chinese reconstruction $g = g_1 * \ldots * g_r$ where $g_i \in M_i$. Then there are $x_i \in M$ such that $g_i = x_i \bmod m_i$. Hence by lemma 9,

$$\begin{aligned} \pi_i \cdot x_i &= \pi_i \cdot ((x_i \bmod m_1) * \ldots * (x_i \bmod m_r)) \\ &= \mathbf{0} * \ldots * \mathbf{0} * g_i * \mathbf{0} * \ldots * \mathbf{0} \end{aligned}$$

(i.e., the Chinese reconstruction of $r-1$ zero vectors together with g_i at position i). Again by lemma 9, we have (modulo m),

$$g = \sum_{i=1}^{r} \mathbf{0} * \ldots * \mathbf{0} * g_i * \mathbf{0} * \ldots * \mathbf{0} = \sum_{i=1}^{r} \pi_i x_i$$

As a linear combination of $x_1, \ldots, x_r$, g therefore must also be contained in M. In particular, this proves $G \subseteq M$.

It remains to prove the reverse inclusion. Assume $x \in M$. Then for all i, $x_i = x \mod m_i \in M_i$. Therefore, $x_i = \sum_{j=1}^{n} \lambda_{ij} g_j^{(i)}$ for suitable $\lambda_{ij} \in \mathbb{Z}_{m_i}$. Consequently, we can define values $\lambda_j = \lambda_{1j} * \ldots * \lambda_{rj}$, $j = 1, \ldots, n$. Let $x' = \sum_{j=1}^{n} \lambda_j \cdot g_j$. Then by construction, $x' \mod m_i = x_i$. Therefore by the Chinese Remainder Theorem, $x' = x$ — implying that $x \in \langle G \rangle$. □

Theorem 5 implies that the reconstruction always results in sets of generators for M — no matter how the sets G_i of generators for the M_i are chosen or their elements are ordered. In particular, if the m_i are prime powers, the sets G_i may be chosen triangular and of cardinality at most k. Therefore we conclude:

Corollary 1 *1. Every $\mathbb{Z}_m$-module $M \subseteq \mathbb{Z}_m^k$ is generated by at most k vectors.*

2. *Assume that $m = p_1^{w_1} \cdot \ldots \cdot p_r^{w_r}$ for pairwise different prime numbers p_i. Then every strictly increasing chain of $\mathbb{Z}_m$-modules $M_0 \subset \ldots \subset M_r \subseteq \mathbb{Z}_m^k$ has length at most $k \cdot (w_1 + \ldots + w_r)$.* □

We turn to computing the solutions of systems of equations. Let $A\ \mathbf{x} = b$ denote a system of equations over $\mathbb{Z}_m$ where A is a $(k \times k)$-matrix with entries in $\mathbb{Z}_m$, $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)^{\mathrm{t}}$ is a column vector of unknowns, and $b \in \mathbb{Z}_m^k$ is a column vector of values in $\mathbb{Z}_m$. Assume again that $m = m_1 \cdot \ldots \cdot m_r$ for pairwise relatively prime numbers m_i. For every $i = 1, \ldots, r$, let A_i, b_i denote the matrix A and vector b with entries modulo m_i. Then we have:

Lemma 10 *1. Assume $x \in \mathbb{Z}_m^k$ is a solution of $A\ \mathbf{x} = b$, then for every i, $x_i = x \bmod m_i$ is a solution of the system $A_i\ \mathbf{x} = b_i$.*

2. *Assume that for $i = 1, \ldots, r$, $x_i \in \mathbb{Z}_{m_i}^k$ is a solution of the system $A\ \mathbf{x} = b_i$. Then the vector $x = x_1 * \ldots * x_r$ is a solution of $A\ \mathbf{x} = b$.* □

Lemma 10 allows us to reduce solving of systems of equations over $\mathbb{Z}_m$ to solving equations over the $\mathbb{Z}_{m_i}$. We obtain:

Theorem 6 *Assume $m = m_1 \cdot \ldots \cdot m_r$ and $m_i = p_i^{w_i}$ for pairwise different prime numbers p_i.*

1. *$A\ \mathbf{x} = b$ has a solution over $\mathbb{Z}_m$ iff $A_i\ \mathbf{x} = b_i$ has a solution over $\mathbb{Z}_{m_i}$ for all $i = 1, \ldots, r$.*

2. *If for $i = 1, \ldots, r$, G_i is a set of generators of the $\mathbb{Z}_{m_i}$-module of solutions of the homogeneous system $A_i\ \mathbf{x} = \mathbf{0}$, then every combination of the G_i to a set of $\mathbb{Z}_m$-vectors is a set of generators of the $\mathbb{Z}_m$-module of solutions of the homogeneous system $A\ \mathbf{x} = \mathbf{0}$.*

3. *The set of solutions of a linear system $A\ \mathbf{x} = b$ over $\mathbb{Z}_m$ can be computed in time $\mathcal{O}(\log(m) \cdot k^3)$.* □

Now we have the necessary tools at hand to generalize the computation of valid affine relations over rings $\mathbb{Z}_{p^w}$ (p prime) to general rings $\mathbb{Z}_m$. In essence, we perform the same analysis for each prime power $p_i^{w_i}$ of m separately. Then we combine the resulting modules by means of Chinese remainder reconstruction. By theorem 5, this gives the desired results. Taking into account that we have to add up the work to be performed for each prime power, we therefore obtain:

Theorem 7 *For every program of size n with k variables, the sets of all valid affine relations at program point u, $u \in N$, can be computed in time $\mathcal{O}(\log(m) \cdot n \cdot k^6 \cdot (k^2 + \log(m)))$.* □

6 The Intra-Procedural Case

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 [12], we have presented an efficient algorithm which computes

all intra-procedurally valid affine relations in time $\mathcal{O}(n \cdot k^3)$ where every arithmetic operation is counted for 1. This algorithm improves on the original algorithm by Karr [8] by a factor of k. It assumes that the program variables to be analyzed take values in a field, namely $\mathbb{Q}$ — but any other field $\mathbb{Z}_p$ (p prime) would do as well.

We will not rephraze this algorithm in all details but just remark that the key idea is to compute, for every program point u, the affine hull of all program states reaching u where the affine hull (if non-empty) is represented by one vector v together with (a basis of) a linear vector space V. In a second phase then (a basis of) the vector space of affine relations valid at u is computed as the solution of a linear system of equations. Note that the much better complexity of the intra-procedural analysis is possible since in absence of procedures, it suffices to compute with vector spaces of states, i.e., vectors with k components — instead of vector spaces of weakest precondition transformers, i.e., matrices with $(k+1)^2$ entries.

From this toplevel description it should be clear how the intra-procedural algorithm from [12] must be modified in order to work for rings $\mathbb{Z}_m$. First, for every program point u, the non-empty affine hull of reachable states now is described by one vector together with a reduced triangular set of generators of a $\mathbb{Z}_m$-module where the containment test of a newly occurring state at u is replaced with a subsumption test based on Theorem 1. Finally in the second phase, we replace solving of systems of equations over a field with our method from Theorem 2 for rings.

Example 6 *As an example, assume again $m = 2^w$ for some $w > 0$ and consider the control-flow graph from Figure 5. The fixpoint iteration starts by setting the*

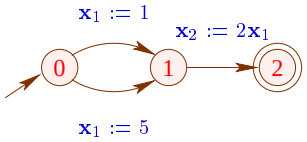

Figure 5: A small program without procedures.

computed approximation of the set of reachable states for the start point 0 to $\mathbb{Z}_m^2$. This affine space is represented by one vector, e.g., the zero vector together with the $\mathbb{Z}_m$-module generated by the vectors $(1,0)^{\mathrm{t}}$ and $(0,1)^{\mathrm{t}}$. Propagating these two states along the edges from 0 to 1, we obtain the four states:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 1 \end{pmatrix}$$

whose affine hull can be represented by the first state together with the $\mathbb{Z}_m$-module generated by the vectors:

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 5 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$$

Since the state $(5,1)^{\mathrm{t}}$ is subsumed by the other three, only the first three are propagated to program point 2. This gives us the states $(1,2)^{\mathrm{t}}$ and $(5,10)^{\mathrm{t}}$ for program point 2 whose affine hull is represented by the state $(1,2)^{\mathrm{t}}$ together with the $\mathbb{Z}_m$-module generated by the vector $(4,8)^{\mathrm{t}}$.

In order to determine the set of all affine relations $a_0 + a_1\mathbf{x}_1 + a_2\mathbf{x}_2 = 0$ valid, e.g., at program point 2, we use the (representation of the) states for program point 2 to put up the following linear equation system:

$$\begin{array}{rcl} a_0 + a_1 \quad + a_2 \cdot 2 & = & 0 \\ a_1 \cdot 4 + a_2 \cdot 8 & = & 0 \end{array}$$

Using our technique from Theorem 2, we can compute the reduced diagonal matrix D corresponding to the matrix of coefficients of the equation system together with the inverse R^{-1} of the right transformation matrix:

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad R^{-1} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

The $\mathbb{Z}_m$-module of solutions of the transformed equation system $D\,\mathbf{a} = \mathbf{0}$ is generated by the vectors $(0, 2^{w-2}, 0)^{\mathrm{t}}$ and $(0,0,1)^{\mathrm{t}}$. Be applying the transformation R^{-1}, we arrive at the following two generators of all valid affine relations (modulo m) at program point 2:

$$\begin{array}{rcl} a & = & (0,-2,1)^{\mathrm{t}} \\ a' & = & (-2^{w-2}, 2^{w-2}, 0)^{\mathrm{t}} \end{array}$$

The affine relation a means $2 \cdot \mathbf{x}_1 - \mathbf{x}_2 = 0$ and could have been found also by performing program analysis over $\mathbb{Q}$. The affine relation a', however, is due to the refined computation over $\mathbb{Z}_m$. It amounts to the statement: $-2^{w-2} + 2^{w-2} \cdot \mathbf{x}_1 = 0$ or $2^{w-2} \cdot (\mathbf{x}_1 - 1) = 0$, which means that $\mathbf{x}_1$ equals 1 modulo 4. □

Summarizing, we obtain the following Theorem:

Theorem 8 *Consider an affine program of size n with k variables but without procedures. Then for every $m > 1$, the set of all affine relations at all program points which are valid over $\mathbb{Z}_m$ can be computed in time linear in n and polynomial in k and $\log(m)$. More precisely:*

1. *If m is a prime power, then the complexity of the analysis is bounded by $\mathcal{O}(\log(m) \cdot n \cdot k^2 \cdot (k + \log\log(m)))$.*
2. *If m is a product of several prime powers, then the complexity of the analysis can be bounded by $\mathcal{O}(\log(m) \cdot n \cdot k^2 \cdot (k + \log(m)))$.* □

7 Conclusion

We have presented intra- and inter-procedural algorithms for computing all valid affine relations in affine programs over rings $\mathbb{Z}_m$. In particular, these techniques allow to analyze integer arithmetic in programming languages like C or Java precisely. All these algorithms were obtained from the corresponding algorithms in [11] and [12] by replacing techniques for vector spaces with corresponding techniques for modules over rings. The key difficulty here is that the ring $\mathbb{Z}_m$ may have zero divisors — implying that not every element in the ring is invertible. Also the notion of a vector space had to be replaced with the weaker concept of a $\mathbb{Z}_m$-module. Since we succeeded in maintaining the toplevel structure of the analysis algorithms, we could essentially achieve the same complexity bounds as in the case of fields — upto one extra factor $\log(m)$ which was due to the increased height of the complete lattice of submodules of $\mathbb{Z}_m^k$. Beyond that, 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 or numerical instability are thus resolved.

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.

More work remains to be done. For example, one method to extend the linear algebra approach to deal with inequalities in conditions is to use *polyhedra* for abstracting sets of vectors [4]. 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 problem of analysis for modular arithmetic.

References

[1] K. R. Apt and G. D. Plotkin. Countable Nondeterminism and Random Assignment. *Journal of the ACM*, 33(4):724–767, 1986.

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

[3] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: `www.elsevier.nl/locate/entcs/volume6.html`.

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

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

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

[7] S. Gulwani and G. Necula. Discovering Affine Equalities Using Random Interpretation. In *Proceedings 30th POPL*, pages 74–84, 2003.

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

[9] J. Leroux. *Algorithmique de la Vérification des Systèmes à Compteurs: Approximation et Accélération*. PhD thesis, Ecole Normale Supérieure de Cachan, 2003.

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

[11] M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *Proceedings 31st POPL*, pages 330–341, 2004.

[12] M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *ICALP 2004*, to appear. Available from `http://ls5-www.cs.uni-dortmund.de/~mmo`.

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

[14] 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.