International Journal of Computer Mathematics
Vol. 82, No. 10, October 2005, 1177–1189

Taylor & Francis
Taylor & Francis Group

Computing with locally effective matrices

J. RUBIO*† and F. SERGERAERT‡

†Departamento de Matemáticas y Computación, Universidad de La Rioja, Edificio Vives, Calle Luis de Ulloa s/n, 26004 Logroño, La Rioja, Spain
‡Institut Fourier, Université Grenoble, BP 74, 38402 Saint-Martin-d'Hères Cedex, France

(Revised 22 July 2004; in final form 25 September 2004)

In this work, we start from the naive notion of integer *infinite matrix* (*i.e.*, the functions of the set $\mathbb{Z}^{\mathbb{N}\times\mathbb{N}} = \{f\colon \mathbb{N}\times\mathbb{N}\to\mathbb{Z}\}$). Then, several undecidability results are established, leading to a convenient data structure for effective machine computations. We call this data structure a *locally effective matrix*. We study when (and how) the standard matrix calculus (Ker and CoKer computations) can be extended to the infinite case. We find again several *undecidability barriers*. When these limitations are overcome, we describe effective procedures for computing in the locally effective case. Finally, the role played by these data structures in the development of real symbolic computation systems for algebraic topology (based on the *effective homology* notion) is illustrated.

Keywords: Data structure; Locally effective matrix; Algebraic topology

C.R. Category: F.2.2

In 1986, Sergeraert introduced the notion of *metamatrix* [1] in the field of symbolic computation in homological algebra and algebraic topology. A metamatrix [1] is a *finite* matrix (but of huge size) whose entries are calculated from an algorithm rather than explicitly stored in computer memory. The paradigmatic examples were the matrices associated to the chain complex of a space $K(\pi, n)$ (see, for instance, [2]), where π is a finite group. Nevertheless, it is now quite clear that the finiteness constraint can be relaxed (both on the group π in the example and on the matrices in general). We are going to smoothly explore these infinite data structures.

In order to get a convenient notation, we choose the Common Lisp programming language [3] to describe our algorithmic texts. Let us call *infinite* (*computable*) *matrix* to any (Common Lisp) algorithm

```
#'(lambda (i j) ...)
```

which from each pair of natural numbers i, j returns (and therefore ends its calculation) an integer number.

*Corresponding author. Email: julio.rubio@dmc.unirioja.es

International Journal of Computer Mathematics
ISSN 0020-7160 print/ISSN 1029-0265 online © 2005 Taylor & Francis
http://www.tandf.co.uk/journals
DOI: 10.1080/00207160512331323326

These objects can be handled by programs, at least if functional programming is available in our computer environment. For instance, the following Common Lisp function computes the sum of two infinite matrices (im).

```
(defun add (iml im2)
  #'(lambda (i j) (+ (funcall iml i j)(funcall im2 i j))))
```

Some relevant matrices can be explicitly constructed as follows:

```
(setf zero #'(lambda (i j) 0))
(setf one #'(lambda (i j) (if (= i j) 1 0)))
```

To interpret these constructions conveniently, let us introduce a bit of terminology. Let us denote by $i\mathcal{M}$ the set of (*mathematical*) *infinite matrices*: that is, the set of functions $\mathbb{Z}^{\mathbb{N}\times\mathbb{N}}$ $(= \{f : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}\})$. In addition, let us denote `iM` the set of Common Lisp objects previously evoked. Then, there exists a (mathematical!) function α: `iM` $\to i\mathcal{M}$ which expresses the mathematical object represented by a machine object. Explicitly, in our case: $\alpha(\texttt{iM})(i, j) :=$ (`funcall iM i j`). (Here, as usual, the representation for integer and natural numbers is assumed to be the *literal* one: integer numbers in the machine are identified with elements of $\mathbb{Z}$; this is even 'physically' visible in our presentation: i and `i` are indistinguished; for the sake of conciseness, we will also identify a Common Lisp expression with the result of its evaluation, when no confusion can arise.) This map α is the *abstraction function* in the representation of a datatype, following Hoare's terminology [4] (it has also been called *decoding function* in the functional coding context [5]).

Let us remark that this abstraction map is not injective: the Common Lisp object

```
#'(lambda (i j) (- (* i j) (* j i)))
```

is different (as a *machine* object) from the object previously binded to the symbol `zero`, but they represent (trough α) the same infinite matrix.

On the other hand, α is neither surjective, as a simple cardinality argument shows: the set of infinite matrices is uncountable, whereas the universe of machine objects is explicitly enumerable. The image of α exactly defines the *computable* infinite matrices (*i.e.*, the infinite matrices whose entries are computable). Later on, we will come back on these cardinality issues.

In this context, the previous Common Lisp function `add` proves that the mathematical operation $+: i\mathcal{M} \times i\mathcal{M} \to i\mathcal{M}$ is *computable* (with respect to the representation defined by α). In fact, as it is easy to see, the external product by an integer is computable and this can be expressed by saying that we have achieved a (correct) *implementation* of the abstract data type (ADT) abelian group (for details on this kind of functional ADT implementations, see [6]).

However, computability is a quite infrequent property on infinite data structures. We are going to prove that equality is undecidable. In other words, the mathematical function $=: i\mathcal{M} \times i\mathcal{M} \to \{\text{true, false}\}$ is not computable (with respect to α). The argument to prove this fact, and the rest of non-computability results in this article, can be traced back to Gödel (*i.e.*, to the very beginning of computability in modern terms), who very explicitly proved that every computability fact can be encoded on the natural numbers. This implies that any infinite data structure, if it is general enough, will be always on the frontier between the computable and the undecidable. Being matrices, our infinite data structures have a supplementary 'dimension' which will allow us to place the pathological counter-examples in different positions, depending on our interests.

As usual, instead of working with recursive functions to state our results, we will work with the more comfortable notion of Turing machine. First, choose a Common Lisp representation for Turing machines. (For simplicity, we will always work with *blank-tape Turing machines*; *i.e.*, with Turing machines which model input-free programs; it is well-known that each undecidability result has a version in this context (see, for instance, [6])). Then, it is very easy to program a function which, given a natural number i, determines whether a Turing machine stops in i or less steps (this program implements the *universal* Turing machine capable of simulating the running of any Turing machine). Let us introduce the corresponding Common Lisp function:

```
(defun halts-in-i-steps? (tm i) ...)
```

Then, we construct, associated to any Turing machine `tm`, the following infinite matrix:

```
(defun tm->im (tm)
  #'(lambda (i j)
      (if (not (= i j))
          0
          (if (halts-in-i-steps? tm i)
              1
              0))))
```

That is, the matrix associated to `tm` is diagonal and it has one 1 at entry (i, i) if `tm` stops in i or less steps; otherwise, it has 0 at the entry (i, i).

Now, let us assume that equality is decidable (under α), and let us call

```
(defun im-eq? (iml im2) ...)
```

the corresponding Common Lisp function.

Consider the function:

```
(defun halts? (tm)
  (not (im-eq? (tm->im tm) zero)))
```

This function would give a solution to the halting problem, thus leading to a contradiction. The contradiction comes from the existence of `im-eq?`. This proves that equality of (computable) infinite matrices is undecidable.

In order to interpret this first non-computability result, let us come back to the notion *representation* of an ADT. Following Hoare [4], an abstraction function α should be seen as a *partial* mapping from a (concrete, machine) type `T` to the abstract model $i\mathcal{M}$. The definition domain of α should be the subset of `T` where the abstraction α has a meaning (the set `iM` in our example). The characteristic function of this definition domain in `T` is called, by Hoare, *invariant* of the representation. This function inv: `T` $\rightarrow$ {true, false} is very important for the programmer, as it defines the property to be kept trough the program for the data in `T` which have a meaning in the program (the compiler is only capable of checking that each data is in `T`, but not if it actually has a sense). The other relevant notion in Hoare's approach to the representation of ADTs is that of *equality of the representation.* This is an equivalence relation '$\sim$' on the definition domain of α, which expresses when two elements of the concrete type are equal from the programmer point of view. In addition, this equality $\sim$ must be coherent

with the abstraction function; in symbols: $a \sim b$ implies $\alpha(a) = \alpha(b)$. As a consequence, the greatest equality of a representation is the relation defined by α; this particular equivalence relation is called *abstraction equality* of the representation. (For a more detailed presentation of these concepts and terminology, see [6].)

Thus, the previous undecidability result can be re-worded by saying that the *abstraction equality of our representation for infinite matrices is undecidable.* Moreover, the invariant of this representation is undecidable. To prove this property, let us re-take the cardinality argument previously evoked.

First, let us choose a type `T` for our representation. We take as concrete type, the universe `U` of all Common Lisp objects. (Alternatively, we could choose `T` as the subset of lexical closures with two natural parameters; as this subtype of `U` is decidable, to take one or another changes nothing in our arguments.) Then α: `U` $\rightarrow i\mathcal{M}$ is the partial function with definition domain `iM` defined earlier. We claim that the invariant inv: `U` $\rightarrow$ {true, false} is not computable.

To prove this result, first let us note that a (theoretical) Common Lisp function can be programmed in such a way that any object in `U` is explicitly enumerated. This is the concrete, language dependent, version of the Gödel numbering previously mentioned. For the sake of precision, let us call

```
(defun enumerate-U (n) ...)
```

such a function. (Note that a Common Lisp object is finally nothing but a bit-string; these bit-strings can be sorted by lexicographical order, for instance.)

Now, let us assume that the invariant of our representation is computable. This means that there exists a Common Lisp function

```
(defun inv? (x) ...)
```

such that it takes a Common Lisp object `x` and returns true if `x` is in `iM` (*i.e.*, if it is a lexical closure taking two natural numbers as input and *finishing* its calculation, returning an integer number) and false otherwise.

By using both `enumerate-U` and `inv?`, it is very simple to program a new Common Lisp function:

```
(defun enumerate-iM (n) ...)
```

which effectively enumerates any element of `iM`.

Then, we define the following function:

```
(setf paradoxical-im
  #'(lambda (i j)
      (if (not (= i j))
        0
        (+ 1 (funcall (enumerate-iM i) i i)))))
```

This is a diagonal matrix that at entry (i, i) has a value that corresponds to the entry (i, i) for the matrix number i, $+1$.

Obviously, the value of `paradoxical-im` belongs to `iM` and so it is very easy to compute a natural number k such that `(enumerate-iM k)` is exactly equal (as a Common Lisp object) to the object associated to `paradoxical-im`. Then the integer `(funcall paradoxical-im k k)` should be equal to `(+1 (funcall paradoxical-im k k))`, which is clearly a contradiction. Thus, a function such as `inv?` cannot exist, and we have proved that the invariant for our representation is undecidable. (Let us remark that the

halting problem has not been used in this proof; in fact, both problems, the halting problem and the undecidability of our invariant, are equivalent.)

Let us note that even if the representation is undecidable (both from the invariant and the equality viewpoints), this is the basis for a perfectly correct implementation of the abelian group ADT. In the sequel, we will try to show how, in addition, (some variants of) this implementation is also very useful for practical and challenging machine calculations.

To this aim, we need to enrich with additional operations the ADT of infinite matrices. The standard product of two matrices do not generalise to the infinite matrices in $i\mathcal{M}$. Therefore, let us introduce the notion of *locally finite matrix*: this is an element of $i\mathcal{M}$ which has a finite number of non-null entries at each column. Let us denote by $lf\mathcal{M}$ the corresponding subset of $i\mathcal{M}$. It is quite clear that on $lf\mathcal{M}$ the standard product of (finite) matrices is directly generalised. Symmetrically, we could impose the finiteness requirement on rows, but we have preferred to keep the standard conventions in matricial calculus. Indeed, each locally finite matrix can be seen as the coordinate matrix of an endomorphism of the free abelian group generated by the natural numbers $\mathbb{N}$. (We will denote this infinitely generated abelian group by $\mathbb{Z}^{\omega}$.) Thus, the product of locally finite matrices corresponds to the composition of endomorphisms, as usual in the finite dimensional case.

From the computational point of view, let us remark that the property of being locally finite is undecidable. More precisely, to determine if a datum `im` in `iM` is such that $\alpha(\texttt{im}) \in lf\mathcal{M}$ is undecidable. The proof is based on a *rotation* of our previous method to associate an infinite matrix to each Turing machine. As it is clear that if a Turing machine stops, the corresponding matrix has an infinite number of 1s on the diagonal. Therefore, if we locate this diagonal on the first column (for example), to test that it has a finite number of non-null entries at the first column is equivalent to the halting problem. We explicitly show this construction:

```
(defun tm->im2 (tm)
  #'(lambda (i j)
      (if (not (= j 0))
        0
          (if (halts-in-i-steps? tm i)
            1
            0))))
```

However, even if the subset of relevant matrices in `iM` is undecidable, we could tackle the problem of implementing the product on such matrices (after all, the subset `iM` is undecidable in `U`, but this does not forbid to calculate the addition of two elements of `iM` and so on).

Unfortunately, the product of two infinite matrices, even if we know that both are locally finite, is also non-computable. To see this, let us consider a variant of `halts-in-i-steps?` which checks if a Turing machine stops in exactly i steps. This can be programmed simply as follows:

```
(defun halts-in-exactly-i-steps? (tm i)
  (or (and (= i 0) (halts-in-i-steps? tm 0))
      (and (halts-in-i-steps? tm i)
           (not (halts-in-i-steps? tm (- i 1))))))
```

Now, we again play the game of associating an infinite matrix to each Turing machine:

```
(defun tm->im3 (tm)
 #'(lambda (i j)
   (if (not (= j 0))
    0
     (if (halts-in-exactly-i-steps? tm i)
      1
      0))))
```

Each matrix in the image of this mapping is locally finite (in fact, it is null or it has only 1 at its first column).

Now, we define the following locally finite matrix:

```
(setf test-lfm
 #'(lambda (i j)
   (if (not (= i 0))
    0
    1)))
```

which has 1s at its first row and 0s elsewhere.

Let us assume there exists a Common Lisp function:

```
(defun lfm-product (lfm1 lfm2) ...)
```

implementing the product of two locally finite matrices. Then, the following function will be a solution for the halting problem:

```
(defun halts2? (tm)
  (= 1 (funcall (lfm-product test-lfm (tm->im3 tm))
                0 0)))
```

leading to contradiction. Thus, the product of locally finite matrices is non-computable.

Nevertheless, there exists an infinite data structure which is a close relative of locally finite matrices and where products can be computed. The solution is to store explicitly a bound for each column, indicating a row index from which the column is null. (It is not necessary for the bound to be minimal, as the last non-null entry can be easily computed from the original bound.) Therefore, we introduce the type of *locally effective matrices,* whose elements are pairs (lists of length 2):

```
(list #'(lambda (j) ...)
      #'(lambda (i j) ...))
```

the second being an element of `iM` and the first being a lexical closure which from each natural number finishes its computation, returning another natural number. The constraint linking both

components of the pair is easier to express if we introduce the following notation: a locally effective matrix is (*b,m*) (*b* for bound, *m* for matrix), where:

- for each $j \in \mathbb{N}$, b_j, is also in $\mathbb{N}$;
- for each $i \in \mathbb{N}$ and $j \in \mathbb{N}$, $m_{ij} \in \mathbb{Z}$;
- and, in addition, $m_{ij} = 0, \forall i > b_j$.

Let us again insist that we do not demand $m_{b_j j} \neq 0$, in other words: bounds are not minimal. We will denote by `leM` the set of (Common Lisp) pairs (*b, m*) as mentioned earlier.

The type `leM` is not a subtype of `iM`, but there exists an algorithm from `leM` to `iM` which consists in extracting the second component. Obviously, there is no converse algorithm from `iM` to `leM`, even if we assume that the element `im` in `iM` is such that $\alpha(\texttt{im}) \in lf\mathcal{M}$. Indeed, the family of matrices generated by `tm->im3` shows that if bound *b* was calculable, then the halting problem would have an algorithmic solution.

Let us introduce a new representation for infinite matrices, based on locally effective matrices. The abstraction function will be called β: $U \rightarrow i\mathcal{M}$, with definition domain `leM`, and defined by: $\beta(\texttt{lem}) = \alpha((\texttt{second lem}))$. The new abstraction β is not surjective, as it maps on locally finite matrices, and besides it is even not surjective on $lf\mathcal{M}$ (the same cardinality argument as the earlier works).

The 'field' *b* is another source of non-injectivity for β. For instance, any *b* is a correct field for the zero matrix. On the other way around, for any matrix *known to be diagonal* (or upper triangular, in fact), the identity function `(lambda (j) j)` is a coherent bound function as locally effective matrix. This remark is important because many of our previous paradoxical examples are given with diagonal matrices, and then the proofs can be directly exported to the locally effective case. Indeed, note that the abstraction equality and the invariant of the new representation are both undecidable (as shown by the locally effective versions of `tm->im` and `paradoxical-im`, respectively).

Let us explicitly prove that the new representation can be used in an implementation of the ADT group.

```
(defun leadd (lem1 lem2)
  (list #' (lambda (j) (max (funcall (first lem1) j)
                            (funcall (first lem2) j)))
    (add (second lem1) (second lem2))))
```

(This is the reason why we do not demand a minimal bound: programming is easier.)

```
(setf lezero (list #'(lambda (j) 0)
                   zero))
```

The product by an integer can also be easily written, noting that the same bounding functions can be kept (even if the product is by 0!).

Moreover, `leM` is obviously the basis for an implementation on the ADT ring. To this aim, observe that

```
(setf leone (list #'(lambda (j) j)
                  one))
```

is a unity for the product of matrices. The product is easily programmed by applying the classical formula:

$$m_{ij} = \sum_{k=0}^{b_j^2} m_{ik}^1 m_{kj}^2,$$

where m^1 is in `iM` and (b^2, m^2) is in `leM`, and m is a representation of the corresponding matrix product $i\mathcal{M} \times lf\mathcal{M} \to i\mathcal{M}$ (at the mathematical level, the same formula works by replacing b_j by ∞; being locally finite, the series is actually a finite sum). As usual, even if we know that $\alpha(m^1) \in lf\mathcal{M}$, a bound function for the product matrix is not calculable (as `leone` is a unity for the product). Therefore, this product lacks of *stability* and, in particular, it does not allow to iterate the process. However, the product of two locally effective matrices (b^1, m^1) and (b^2, m^2) is again a locally effective matrix (b, m), where $b_j = \max(b_0^1, \ldots, b_{b_j^2}^1)$, so achieving the necessary stability.

In summary, we have implemented the ring of (computable) endomorphisms on $\mathbb{Z}^\omega$, by means of a representation on `leM`, the Common Lisp subset of locally effective matrices.

Now, it is necessary to fulfil our promise of doing interesting calculations on such infinite data structures. Let us imagine that we are interested in computing the *kernel* and *cokernel* of such matrices. First, let us explore these concepts in the well-known setting of finite (standard) matrices.

Let us start with the following example. It is the coordinate matrix of a homomorphism d from $\mathbb{Z}^5$ to $\mathbb{Z}^4$.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

In such a diagonal matrix, the Ker(nel) and CoKer(nel) can be directly read. Indeed, the kernel is a free finitely generated abelian group and then simply a natural number determines it. In our example, this (Betti or rank) number is: $2(=5-3)$ (being 3 the number of non-null diagonal entries in the matrix). The CoKer (*i.e.*, the quotient of the target $\mathbb{Z}^4$ by the image of the homomorphism) is a finitely generated abelian group and then is characterised by a natural number (the Betti number) and a finite sequence of natural numbers (each one is greater than 1). In our example, the rank of the CoKer is $1(=4-3)$ and the torsion coefficients are 2 and 3. Thus, $\mathrm{Ker}(d) \cong \mathbb{Z}^2$ and $\mathrm{CoKer}(d) \cong \mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z} \oplus \mathbb{Z}/3\mathbb{Z}$.

Now, it is very well-known (see, for instance, as one among many possible references, [8]) that any (finite) integer matrix can be diagonalised (to reach a Smith normal form, for instance; remark that the matrix in the earlier example is not in normal form), by means of elementary row and column operations, in such a way that the Ker and CoKer are maintained invariant (up to isomorphism). Therefore, there exists an algorithm computing the Ker and CoKer of any (finite) matrix.

What about infinite matrices? Evidently, it is necessary to impose new conditions: the Ker and CoKer should be of finite type (*i.e.*, finitely generated). Therefore, let us introduce a new set of infinite matrices, $lf\mathcal{M}_{ft}$, whose elements are locally finite matrices of finite type (*i.e.*, with Ker and CoKer of finite type). Therefore, it is possible to consider the problem of computing the Ker and CoKer of the elements in $lf\mathcal{M}_{ft}$.

Note that the null matrix does not belong to $lf\mathcal{M}_{ft}$, as Ker and CoKer are $\mathbb{Z}^\omega$. On the contrary, the unity matrix belongs to $lf\mathcal{M}_{ft}$, being both its Ker and CoKer the trivial group. In fact, this property characterise isomorphisms: a matrix d is (the coordinate matrix of) an isomorphism if and only if $\mathrm{Ker}(d) = \mathrm{CoKer}(d) = 0$. Let us observe that, among the diagonal infinite matrices with positive entries, the unique isomorphism is the identity matrix. These

elementary facts allow us, with variants of the construction `tm->im`, to prove some undecidability results. Let us first prove that it is undecidable to determine if a locally finite matrix is of finite type. For, let us define:

```
(defun tm->lem (tm)
  (list #'(lambda (j) j)
    #'(lambda (i j)
      (if (not (= i j))
        0
         (if (halts-in-i-steps? tm i)
          0
          1)))))
```

The matrices constructed in that way are diagonal. (They are the locally effective counterpart of the subtraction of the (`tm->im`)-matrix from the unity matrix.) It can be the unity matrix (if the corresponding Turing machine `tm` does not halt) or a matrix with only a finite number of 1s (otherwise). This implies that one such matrix is of finite type if and only if the corresponding Turing machine does not halt. Therefore, an algorithmic solution for deciding if a locally effective matrix is of finite type would give a solution for the halting problem.

Even worse, in the class of locally effective matrices *of finite type* (in other words, such as its image by β belongs to $lf\mathcal{M}_{ft}$), the Ker and CoKer are not calculable. This is proved by the following construction:

```
(defun tm->lem2 (tm)
  (list #'(lambda (j) j)
    #'(lambda (i j)
      (if (not (= i j))
        0
          (if (halts-in-exactly-i-steps? tm i)
            0
            1)))))
```

Now, each matrix is the identity or it has only one 0 in the diagonal. This means that Ker = CoKer = 0 or Ker = CoKer = $\mathbb{Z}$. Thus the matrices are always of finite type. However, if the Ker and the CoKer were computable, an algorithm could decide if any Turing machine halts or not.

Nevertheless, there exist cases in which the Ker and CoKer of a locally effective matrix are computable. A first, almost trivial, case is that in which it is known that the matrix d is an isomorphism, then the Ker and CoKer are null. This fact could be explicitly stored by defining a new data type whose elements are pairs of locally effective matrices, which inverses one of the another. In this case, the algorithm returns 0 and 0 as Ker and CoKer. Let us denote these pairs as (d, h) (which must satisfy $dh = id$ and $hd = id$).

Another trivial case is that in which the matrix, even being infinite, is in fact finite: two numbers n and m are known, such that the only non-null entries are concentrated on the first n rows and m columns. In this case, the kernel is infinite dimensional, but, being free, it is

(isomorphic to) $\mathbb{Z}^\omega$. Then, it can be considered an algorithm that computes a natural number if the kernel is of finite type, or responds the symbol ∞ otherwise. For the cokernel, it is similar: the rank is ∞ and the torsion coefficients are finite in number and, obviously, easily computable from the $n \times m$-truncated matrix. Let us elaborate a little bit the terminology in order to be able to generalise this case. A first approach is to enrich the `leM` type and to consider triples (*lem, n, m*) denoting a infinite matrix which can be safely *truncated* at row n and column m.

However, it will be better for our interests to enrich `leM` by means of a tuple $[d, p_m, i_m, P_n, i_n, \bar{d}]$, where

- d is a locally effective matrix;
- $p_k\colon \mathbb{Z}^\omega \to \mathbb{Z}^k$ is the canonical projection in the first k components, with $k = n, m$;
- $i_k\colon \mathbb{Z}^k \to \mathbb{Z}^\omega$ is the canonical injection in the first k components, with $k = n, m$; and
- $\bar{d}$ is a finite matrix obtained from d by truncating at row n and column m.

This gives a diagram such as:

$$\begin{array}{ccc} \mathbb{Z}^\omega & \xrightarrow{\ d\ } & \mathbb{Z}^\omega \\ {\scriptstyle p_m}\downarrow \uparrow{\scriptstyle i_m} & & {\scriptstyle p_n}\downarrow \uparrow{\scriptstyle i_n} \\ \mathbb{Z}^m & \xrightarrow[\ \bar{d}\ ]{} & \mathbb{Z}^n \end{array}$$

Note that this diagram always commutes, but in addition, the torsion information on the cokernel of d can be recovered from $\bar{d}$ when d is null outside $\bar{d}$. In fact, this is also the case if d is truly infinite, but the rest of the matrix defines an isomorphism. The figure in the diagonal case is as follows:

$$\begin{pmatrix} \begin{pmatrix} 0 & & & & \\ & \ddots & & & \\ & & 0 & & \\ & & & d_1 & \\ & & & & \ddots \\ & & & & & d_k \end{pmatrix} & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 & \\ & & & & \ddots \end{pmatrix},$$

with $d_i > 1, i = 1, \ldots, k$.

Thus, this is a mixture of two trivial cases (the isomorphism case and the 'finite' case), giving a data structure such as:

$$[d, h, p_m, i_m, p_n, i_n, \bar{d}],$$

where h is the locally effective matrix:

$$\begin{pmatrix} \begin{pmatrix} 0 & & \\ & \ddots & \\ & & 0 \end{pmatrix} & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ & & & & \ddots \end{pmatrix},$$

acting as a 'partial' inverse of d.

In this, case, the Ker and CoKer of d are exactly the same as those of $\bar{d}$, and so they can be easily computed (and, in fact, they are necessarily of finite type; thus, if such a data structure is provided for d, then $\beta(d) \in lf\mathcal{M}_{ft}$).

Let us make explicit the equations satisfied by these seven matrices:

1. $p_n d = \bar{d} p_m$;
2. $d i_m = i_n \bar{d}$;
3. $p_k i_k = id, k = n, m$(in fact, $\forall k \in \mathbb{N}$);
4. $p_m h = 0$;
5. $h i_n = 0$;
6. $dh + i_n p_n = id$;
7. $hd + i_m p_m = id$.

Now, let us call *reduction* to a tuple as mentioned earlier but where the ingredient matrices can be general. More precisely, a *reduction* is a tuple

$$[d, h, f_1, g_1, f_2, g_2, \bar{d}],$$

where

- d, h are locally effective matrices;
- $\bar{d}$ is a finite matrix, encoded in the usual way (as a two-dimensional array, for instance); let us call it, for symmetry, an *effective* matrix;
- the seven equalities mentioned above hold (replacing the ps by fs and the is by the gs).

Then, it is easy to check that the Ker and CoKer of d are isomorphic to those of $\bar{d}$, and hence we have determined a class of (enriched) locally effective matrices whose Ker and CoKer are easily computable. However, at this point, two natural questions arise:

1. where these structures appear? (because, obviously, a reduction cannot be computed from d, even if we know d is of finite type);
2. why *not* drop any information different from $\bar{d}$, as $\bar{d}$ contains all the relevant information (for Ker and CoKer computations)?

Starting from the second question, the reason is that a reduction is a very *stable* structure with respect to *perturbations* in the initial matrix d.

Let us imagine we have another locally effective matrix ρ and we are interested in the Ker and CoKer of $d + \rho$.

If the information contained in a reduction

$$[d, h, f_1, g_1, f_2, g_2, \bar{d}]$$

can be transformed in another reduction

$$[d + \rho, h', f_1', g_1', f_2', g_2', \bar{d}'],$$

then the problem of computing $\mathrm{Ker}(d + \rho)$ and $\mathrm{CoKer}(d + \rho)$ is solved (trough $\bar{d}'$) and, in addition, the process can be iterated.

Obviously, the perturbations must be restricted in some way, but once the right conditions are established, this also answers the first question, since starting from something essentially finite as:

$$\begin{pmatrix} \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix} & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ & & & & \ddots \end{pmatrix},$$

we could perform a sequence of perturbations until obtaining some complex infinite matrix.

Interestingly enough, the only condition to be imposed to ρ is related uniquely to h. If there exists a natural number k such that $(\rho h)^{k+1} = 0$ and then we define:

$$\Phi = 1 - \rho h + (\rho h)^2 - \cdots + (-1)^k (\rho h)^k,$$

then the expressions for the new reduction are:

$$h' = h\Phi, \quad f_1' = f_1\Phi, \quad g_1' = (1 - h\Phi\rho)g_1, \quad f_2' = f_2, \quad g_2' = g_2, \quad \bar{d}' = \bar{d} + f_2\Phi\rho g_1.$$

Let us see two examples in the finite case.

Let *d, h* be as follows, and f and g are the canonical injections and projections in the first component, respectively.

$$d = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}, \quad h = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}.$$

Then, $\bar{d}$ is nothing but the product by 2 over integers. If we define

$$\rho = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix},$$

it is easy to check that $(\rho h)^2 = 0$. The new torsion coefficient in $\mathrm{CoKer}(d + \rho)$ is 3 and this can be directly calculated from the formula mentioned earlier (no new diagonalisation process is necessary in this case).

On the contrary, if

$$\rho = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},$$

then

$$\rho h = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

and the 'series' Φ does not converge.

Nevertheless, the first question mentioned earlier admits a more general answer: all the structures and algorithms presented in this article appear in a natural way in the field of symbolic computation in algebraic topology. More concretely, the definition of locally effective object and reduction was introduced by Sergeraert when constructing the Common Lisp systems EAT (effective algebraic topology) and Kenzo (see [9–11]). In particular, the locally effective matrices appear linked to the chain complexes of locally effective simplicial sets, and our discussion on matrix perturbations is nothing but a particular case of the well-known basic perturbation lemma occurring in homological algebra (see, for instance, [12]). Therefore, the Ker and CoKer computations are only the homology group computations in the case of a chain complex with only two non-null consecutive chain groups. The elementary results in this article describes, in particular, and for the first time up to authors' knowledge, a chain complex whose homology is of finite type, but which is not computable.

Acknowledgement

The authors were partially supported by MCyT and FEDER, project TIC2002-01626.

References

[1] Sergeraert, F., 1986, *Calcul Symbolique et Formel*, **5**, 97–101.
[2] Eilenberg, S. and Mac Lane, S., 1953, *Annals of Mathematics*, **58**, 55–106.
[3] Steele, G., 1990, *Common Lisp. The language. Second Edition* (United States of America: Digital Press).
[4] Hoare, C.A.R., 1972, *Acta Informatica*, **1**, 271–281.
[5] Sergeraert, F., 1990, *Astérisque*, **192**, 57–67.
[6] Lambán, L. Pascual, V. and Rubio, J., 2003, *Applicable Algebra in Engineering, Communication and Computing*, **14**, 187–215.
[7] Phillips, I.C.C., 1992, *Background: Mathematical Structures*, Vol. 2 (Oxford: Oxford University Press).
[8] Yap, Ch.K., 2000, *Fundamental Problems in Algorithmic Algebra* (Oxford: Oxford University Press).
[9] Dousson, X. Sergeraert, F. and Siret, Y., 1999, *The Kenzo Program* (Grenoble: Institut Fourier). Available online at: http://www-fourier.ujf-grenoble.fr/∼sergerar/Kenzo/
[10] Rubio, J. and Sergeraert, F., 1997, *Lecture Notes Summer School in Fundamental Algebraic Topology* (Grenoble: Institut Fourier). Available online at: http://www-fourier.ujf-grenoble.fr/∼sergerar/Summer-School/
[11] Rubio, J. and Sergeraert, F., 2002, *Bulletin des Sciences Mathématiques*, **126**, 389–412.
[12] Brown, R., 1967, *Celebrazioni Arch. Secolo XX, Simp. Top*. pp. 34–37.