A FUNCTIONAL PROGRAM FOR GAUSSIAN ELIMINATION

Fer-Jan de Vries

Department of Philosophy

University of Utrecht

Logic Group
Preprint Series
No. 23

Department of Philosophy

University of Utrecht

A Functional Program For Gaussian Elimination

Fer-Jan de Vries

September 1987

Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

* the author is partially sponsored by the Dutch Ministry of Science and Education through the project "Parallel Reduction Machine".

Contents

1 Introduction

This paper has been written as a contribution to the Dutch Parallel Reduction Machine Project. The paper is second in a sequence of papers that gives functional presentations for fundamental algorithms. The aim is to verify experimentally the following claim by functional programmers [BvL]:

functional programming is good for writing structured software;
better than the so-called imperative von Neumann-languages.

In this paper the traditional algorithm of Gaussian Elimination of a system of n linear equations in n indeterminates is presented as a functional program in Miranda. First we will discuss appropriate candidates for the type of matrices and the type of arbitrary n-dimensional arrays, then we give the functional algorithm for Gaussian Elimination.

With pleasure I acknowledge discussions with Henk Barendregt, Rinus Plasmeijer and Gerard Renardel.

2 Miranda

As in [V87] we take the functional programming language *Miranda* (meaning: to be admired) as our representative of a modern functional programming language because:

(i) Miranda has a pleasantly working implementation (we used version 0.378)

(ii) it has a type treatment more or less similar to the language TALE (cf. [BvL] proposed by some workers on the Parallel Reduction Machine Project.

Information on Miranda can be found in the on-line Miranda system-manual, and in articles by the creator of Miranda, D.A. Turner [T84, T85]. A very short introduction to the main notations can be found in [V87]. Roughly speaking one can say that -like other modern functional programming languages- Miranda is based on the idea of using recursion equations to define data types and functions between them [T84].

3 The types of vectors, matrices and arrays

In compilers of the functional language Miranda, like in most other functional programming languages, no implementation for matrices or arrays is built in. The reason is that up until now no satisfying efficient implementation has been developed allowing for random access to the elements of the matrices and arrays.

It is easy to define a type containing all rectangular matrices of a fixed size, let us say the type of all 2x3-matrices:

```
mat_2x3 * ::= ((*,*,*),(*,*,*))
```

However, we need more. We are interested in a type containing rectangular matrices of all sizes, that can serve for instance as the domain for the Gaussian Elimination algorithm we describe in the next chapter. Otherwise we would need to write for each **n** a special Gaussian Elimination program that can solve the systems of **n** linear equations in **n** indeterminates.

We will give such a definition and we will generalize it to a definition of a type that contains arrays of all dimensions and sizes. We do not strive with this array type definition for the most efficient implementation of arrays in Miranda, we merely present it as a straightforward way of defining a type that contains vectors, matrices, blocks, etc.. The main point is that a language like Miranda is powerful enough to define such general types in.

In the functional language TALE (cf. [BvL]) notations for arrays are introduced. In a subsequent paper we will develop a suitable notion of arrays such that inside Miranda we can define the operations on arrays that are allowed in TALE.

3.1 The type of vectors

When we are given a type *, then it is easy in Miranda to define a type **vectors** * containing all vectors $[a_1,...,a_n]$, where the a_i are elements of *. One simply defines

```
vectors * == [*].
```

Note, that already with lists one has to peel n layers before one can gets at the n-th element of a list **x**. The notation **x!n** only suggests a random access to the n-th element of **x**.

3.2 The type of matrices

3.2.1 Matrices as lists of lists

Lists are the tools Miranda provides us to create matrices with. The natural idea then is that a rectangular matrix is nothing but a list of vectors of the same size. Obviously, [[*]] is the type of lists of lists of elements in a type *. But it contains also lists of vectors of unequal sizes.

Therefore, we define a predicate **reject::[[*]]->bool** on [[*]] with the property that

reject x is true if and only if **x** contains two consecutive elements of different length:

```
reject :: [[*]]->bool
reject [] = False
reject [a] = False
reject (a:(b:x)) = (~#a=#b \/ reject (b:x))
```

Now we are ready to define a type of all matrices:

```
matrix * ::= P [vector *]
P x => error "no matrix", reject x
```

We prefer such a restricted datatype **matrices** * containing only matrices above a combination of [[*]] and a predicate that singles out the matrices in [[*]], because, to our opinion, one looses a certain rigor if one defines operations on an intended domain of matrices as partial functions on the type [[*]], meanwhile taking care by means of the predicate that the functions are only applied to well-formed matrices. We believe that if one want to compose different programs or functions, one avoids many errors by using total functions with well described domains and codomains.

Note, that we did not compromise ourselves on the question whether matrices are built from rows or from columns. Depending on the context we can make a choice. The type **matrices** is neutral enough to admit both interpretations. Details like height and width we did not build into the definition: these have to be read off from the matrices by counting.

3.2.2 Matrices as functions

The above approach to matrices is not the only one. In other contexts one can be interested in matrices that are functions from a some rectangular domain of pairs of integers into the codomain *.

A way of programming this approach is the following:

```
ord_pairs ::= O num num
O a b => error "no ordered pair of int", a>b \/ ~(integer a) \/ ~(integer b)
descriptors == (ord_pairs,ord_pairs)
matrices2 * ::= (descriptors,((num,num)->*))
```

For us aiming at a program for Gaussian elimination, this approach is not useful. To write out the actual definition of such a matrix function seems a bit tiresome (one could circumvent this maybe by writing some matrix building operations, though). Note, in contrast with the costs of the function **transpose** in 4.1, how easy and cheap (only $O(1)$) one can define the transpose of a matrix with this approach:

```
permute :: (*,*)->(*,*)
permute (a,b) = (b,a)
transpose :: (matrices2 *)->(matrices2 *)
transpose (d,f) = (permute d,h)
```

```
        where
        h (x,y) = f(permute(x,y))
```

(Observe that Miranda lacks a simple notation for composition of functions.)

3.3 The type of arbitrary arrays

Vectors, matrices and blocks are all examples of more dimensional arrays. With a similar trick as above we will define a type **arrays** containing all arrays of arbitrary dimensions. The reason we have chosen to give a general definition above a more down-to-earth approach of defining a type containing the **n**-dimensional arrays for each **n** is that now we are able to give definitions for all kind of array operations, like update and exchange operations, descriptor transformations and the like.

What is a good picture for arrays? One can think of arrays as functions from a block of natural or integer indices given by a descriptor of the form $((l_1,r_1), ..., (l_n,r_n))$. Miranda, however, does not allow such a vague array concept as a type **A** of all functions from a described block to some type **C** of elements unless we give a precise description of the type **D** of all n-tuples that according to the descriptor belongs to the described block, i.e. **A == D=>C**. The type of all arrays would then be a sort of union of such types. How natural such a picture may be for us, it does not result in a constructive definition for a type containing all arbitrary sized arrays with elements of a particular type.

For a *constructive* definition it seems better to think of arrays as being built from smaller arrays of identical structure, like in the case of matrices. For instance, we think of a 3x3x3-block as the result of the glueing of three 3x3-matrices together. So, we have to consider more aspects of the structure than the mere length of **v** and components of **m**, as we did in the definition of **matrices ***. The predicate **reject2** is a bit more complicated than before, we have to reject lists of arrays that do not have the same structure:

```
arrays * ::= V (vectors *) | Q [arrays *]
Q x => error "no array", reject2 x
        where
        reject2 [] = False
        reject2 [a] = False
        reject2 (a:(b:x)) = (~structure a=structure b \/ reject2 (b:x))
                            where
                            structure (V x) = [#x]
                            structure (Q []) = [0]
                            structure (Q (a:x)) = (1+#x):(structure a)
```

structure counts in each dimension the number of elements.

If one worries about too many different empty arrays one can map them with help of the constructors on one unique constructor representing all empty arrays: **Nil_array**.

Note, however, that this type of arrays is a simple one. It is just a way of storing n-dimensional blocks of data, without much bookkeeping tools. The notion of array defined here can be made more useful by putting extra pieces of information, like descriptors, in the definition (cf. [V87a]).

4 Gaussian Elimination

In this chapter we will give a description of the algorithm that performs the Gaussian Elimination on a system of **n** linear equations in **n** variables, intertwined with a script or program in Miranda ending with a function that, given a matrix representing the equations, produces either the solution of the system or the message "*no unique solution*". We take as type of matrices:

```
matrix * ::= P [[*]]
P x => error "no matrix", reject x
       where
       reject [] = False
       reject [a] = False
       reject (a:(b:x)) = (~#a=#b \/ reject (b:x))
```

Note, that we still are in a position to interpret this definition in two different ways: either we can think a matrix to be built from vertical vectors, or from horizontal vectors. Rather arbitrarily we will picture matrices as lists of *horizontal* vectors, simply because this will be useful for the algorithm of Gaussian elimination.

With each function we will give an estimate of the costs it takes to evaluate this function. We measure the costs in the number of elementary operations that have to be calculated:

(i) operations on numbers: +,-,*,/,abs,=,<,>,<=,>=.

(ii) operations on booleans: ~,\/

(iii) operations on lists: :,<-

The following operations on lists, ! and ++, are proportional to the length of the input lists.

4.1 Elementary functions on matrices

We want to be able to extract the n-th row of a matrix m.

```
get_row :: num->(matrices *)->(vector *)
get_row n (P x) = x!n
```

To define a function that extracts the n-th column from a matrix m is just as easy:

```
get_column :: num->(matrices *)->(vector *)
get_column n (P x) = [(v!n)| v<-x]    (i.e., v∈ x)
```

Note, however, that the costs to compute these two functions differ in the order costs, because we defined the matrices as a list of rows instead of columns. **Get_row n mat** takes O(**n**) operations, but **get_column n mat** costs as many as O(**n.width(mat)**).

As an example of a typical elementary function on matrices we take the transposition of a matrix.We will use the function **get_column** to calculate the *transposition* (α_{ji}) of a matrix (α_{ij}), with help of the dimensions of a matrix, the height and the width:

```
height :: (matrix *)->num
height (P x) = #x
```

Note that the costs of height are of the order of the height of the matrix.

```
width :: (matrix *)->num
width (P []) = error "meaningless notion"
width (P (a:x)) = #a
```

Note, that the costs to compute the the width of a matrix are of the order of its width.

```
transpose :: (matrix *)->(matrix *)
transpose (P x) =   P [get_column (w-i+1) (P x) | 1<=i>=w]
                    where
                    w = width (P x)
```

The cost of this function **transpose mat** is $O(n^3)$, where **n** is the width of **mat**.

4.3 Manipulations with linear equations

Sticking to convention we will read linear equations with the polynomial part on the left. We will treat a system of **n** equations in one big **nx(n+1)** matrix:

$$\begin{array}{l} a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n=b_1 \\ a_{12}x_1+a_{22}x_2+\ldots+a_{2n}x_n=b_2 \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ a_{n1}x_1+a_{n2}x_2+\ldots+a_{nn}x_n=b_n \end{array} \quad \text{becomes} \quad \begin{array}{l} a_{11}\ a_{12}\ \cdots\ a_{1n}\ b_1 \\ a_{12}\ a_{22}\ \cdots\ a_{2n}\ b_2 \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ a_{n1}\ a_{n2}\ \cdots\ a_{nn}\ b_n \end{array}$$

As everybody will know, Gaussian elimination consists of a forward phase and a backward phase. In the forward phase one tries to rewrite the system of equations: working from left to right one produces a triangle of zero's.

In the backward phase the solution of the system is found by substitution.

4.4 The forward phase

The idea of the forward phase of Gaussian elimination is to clean or to triangulate the matrix:

$$\begin{array}{cccccc} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{12} & a_{22} & \cdots & a_{2n} & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}$$

in the following form:

$$\begin{array}{cccccc} 1 & c_{12} & \cdots & \cdots & c_{1n} & b'_1 \\ 0 & 1 & \cdots & \cdots & c_{2n} & b'_2 \\ 0 & 0 & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \cdots & 1 & b'_n \end{array}$$

We will explain this in detail for the left-most column. This process has to be repeated on the other columns - each time with an decreased number of rows.

First, we searches in the column for the entry with the highest absolute value. In the case that all entries are equal to zero, there is *no unique solution*, i.e., either there is no solution or there are too many, and the algorithm stops. If there is a non-zero element in the column, then we place the equation (row) to which this element belongs above the other equations (i.e., on top of the matrix). Next we subtract the new top row from the other rows taking care that we first multiply it by well-choosen constants, such that we produce zeros in the first column below the top equation. This is called elimination.

Having finished elimination of this column, we skip the top row and the column on the left and proceed similar with the remaining matrix. The final result of this triangulation process is a matrix with a triangle of zero's in the left-lower corner.

We will now describe in Miranda the algorithms to perform all this.

The first thing we need is a function **index_of_largest (x,l,n,p)** that given a vector or list **v** returns to us the index of the in absolute value largest element in that list or, if such an entry can not be found, produces the error message *no unique solution*. We define this function with help of a function **largest_in_list**, that will do the actual search, by searching from left to right through the list, bookkeeping with each step position **l** of the last found possible largest element, the next position **n** to be checked and the value **p** of the up to then largest element (note that in the function **index_of_largest** these bookkeeping details are hidden):

```
largest_in_list([ ],l,n,p)  = error "no unique solution", a=0
                            = p
largest_in_list((b:x),l,n,p)  = largest_in_list (x,b,n+1,n+1), abs b>=abs l
                              = largest_in_list (x,l,n+1,p)
```

and now:

```
index_of_largest  v  =  largest_in_list  (v,0,0,0)
```

Note that **index_of_largest x** takes $O(\#x)$ operations.

Having searched in a column for the index **n** of the in absolute value largest entry we want to put the corresponding row on top of the matrix **m**. Extracting columns is simple with the already defined function **get_column**. Putting them together in the intended order in a list and reformatting of this list in a matrix is just as straightforward.

```
put_row_on_top  m n = P ((get_row n m):[(get_row i m) |
                                        i<-[1..height  m]; i~=n])
```

An informal description of **put_row_on_top** is the following

$$\text{put\_row\_on\_top}\begin{pmatrix} \text{row}_1 \\ \text{row}_2 \\ \dots \\ \dots \\ \dots \\ \dots \\ \text{row}_k \end{pmatrix} n = \begin{pmatrix} \text{row}_n \\ \text{row}_1 \\ \dots \\ \text{row}_{n-1} \\ \text{row}_{n+1} \\ \dots \\ \text{row}_k \end{pmatrix}$$

Observe that the costs **put_row_on_top** are $O(n^2)$, where **n** is the number of equations, i.e., the height of the matrix, as one repeats **n** times a job costing $O(n)$.

We combine these reconstructing activities in one function:

```
reconstruct x  =  put_row_on_top  x  n
                  where
                  n=index_of_largest  (get_column  1  x)
```

Clearly, the costs of **reconstruct** are $O(n^2)$, where **n** is the width of the matrix it gets as input.

The elimination step in the forward phase we program as follows

```
eliminate  (P[])  =  P[]
eliminate  (P  (v:m))  =  triangulate  v  m
```

where the function **triangulate** will perform the described subtractions of the top row **v** from all other rows in **m** working from top to bottom:

```
triangulate  v  (P[])  =  P[]
triangulate  v  (P  (w:m)  =  P  ((clear_vec  v  w):(triangulate  v  m))
clear_vec  (a:v)  (b:w)  =  clear  (a:v)  (b:w)  (b/a)
                            where
                            clear  (a:v)  (b:w)  c  =  (b-a*c):(clear  v  w  c)
                            clear  []  []  c  =  []
```

Explanation: **clear_vec** $(a_1,\dots,a_n)$ $(b_1,\dots,b_n) = (0,b_2-a_2.(b_1/a_1),\dots,b_n-a_n.(b_1/a_1))$
Observe that the costs of **clear_vec v w** are order of the costs of **clear v w c**, that is $O(\#v)$. Hence the costs of **triangulate v m** are $O(n^2)$, where **n=#v=1+height m**. It follows that

the costs of **eliminate v m** are also $O(n^2)$, using the same notation.

To facilitate the substitution in the backward phase, we divide all elements of the top row we are working with by the first element of the top row

```
divide_top_row (P(v:m)) = P (div_list v (v!1)) m
                          where
                          div_list (a:v) c = (a/c):(div_list v c)
                          div_list [] c = []
```

that is: $v=(a_1,...,a_n)$ in **P v m** is replaced by $(1,a_2/a_1,...,a_n/a_1)$. Clearly the costs are $O(n)$, where $n=\#v$.

It is straightforward to define a function that removes the left most column of a given matrix **m**, if we recall that we can use a function **tl** to remove the head of a list and to pertain its tail.

```
delete_left_most_column (P[]) = []
delete_left_most_column (P(v:m)) = (tl v):(delete_left_most_column m)
```

Note that the costs of **delete_left_most_column m** are $O(\text{width}(m).\text{height}(m))$, i.e., in our case of an **(n+1)xn** matrix just $O(n^2)$.

With all equipment developed up until now we can define the function **forward** that performs the forward phase as described in the beginning of this section:

```
forward (P(v:m)) = (forward(P(delete_left_most_column(eliminate new))))
                   ++[reverse (get_row 1 new)]
                   where
                   new = divide_row (reconstruct (P(v:m)))
forward (P[]) = []
```

Note that **forward** produces in backward order a list of reversed vectors becoming shorter from left to right, so that the result will be of the form:

$$
\begin{array}{lllll}
(d_n & 1 & & & \\
(d_{n-1} & 1 & c_{n-1,n} & & \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
(d_1 & 1 & c_{12} & \ldots & c_{1n})
\end{array}
\qquad \text{meaning} \qquad
\begin{array}{l}
d_n = x_n \\
d_{n-1} = x_n + c_{n-1,n}x_{n-1} \\
\ldots = \ldots \\
d_1 = x_n + c_{1,n-1}x_{n-1} + \ldots + c_{11}x_1
\end{array}
$$

The reversing is useful for the following backward phase, which input will be this list of vectors.

The costs of **forward** applied to a system of **n** equations are $O(n^3)$.

4.5 The backward phase

If we return to the meaning of the above mentioned list of vectors in terms of equations, then we now have to solve the following system of linear equations:

$$d_n = x_n$$
$$d_{n-1} = x_n + c_{n-1,n} x_{n-1}$$
$$\ldots = \ldots$$
$$d_1 = x_n + c_{1,n-1} x_{n-1} + \ldots + c_{11} x_1$$

This clearly can be done, working from top to bottom and substituting the values found for $x_n, x_{n-1}, \ldots$ etc.

```
substitute x [ ] = [ ]
substitute x (v:m) = (row_subst x v):(substitute x m)
                     where
                     row_subst x (d:(c:v)) = (d-c*x):v
```

Explanation: **row_subst x** $(d_k = c_{kn}x_n + \ldots + c_{k2}x_{(n-k)+1} + x_{n-k}) =$
$(d_k - c_{kn}x = c_{k+1\,n}x_{n-1} + \ldots + c_{k2}x_{(n-k+1} + x_{n-k})$,
and **substitute** performs this substitution in all rows of the matrix.
Costs of **row_subst x v** are $O(\#v)$, and the costs of **substitute x m** are $O(n)$, when **m** is an **nx(n+1)**-matrix.

Finally we define the function **backward**

```
backward (v:(w:m)) = (backward(substitute (v!1) (w:m)))++[v!1]
backward (v:[]) = [v!1]
```

Note that **backwards** lists the found values in backward order so that the final result is a list representing the solution of the system in the usual order. The costs of **backward m** are $O(n^2)$.

4.6 Forward + Backward = Gaussian Elimination

The remaining script is pleasantly simple. Apply the forward and the backward functions to a **nx(n+1)**-matrix representing **n** linear equations with **n** indeterminates. The output will be either a list containing the solution, or an error message "*no unique solution*" produced by **forward**.

```
gaussian_elimination m = backward(forward m)
```

The costs are simply $O(n^3)$.

5 Concluding remarks

5.1 Possibilities for parallel computing

It is clear from the description of the algorithm that Gaussian Elimination is a very sequential process, allowing not much parallel computation.

We will collect some small, local examples that allow parallel processing. Recall the definition of the function **put_row_on_top**. It resembles the following general form:

```
f(n) = [g(i) | i<-[1..n]]
```

The computation of **f** allows parallel processing of all the **g(i)**. It will be time-saving only if the complexity of **g** exceeds a certain implementation dependent bound.

The other obvious places where one could compute in parallel occur in the definitions of **triangulate** and **substitute**. All of them involve recursion. Evaluating the possibilities of parallel processing seems to ask for a research of different cases of recursion. Recursion in Miranda can be introduced at two places: a recursive *type* definition or a recursive *function* definition. As an example: the following situation can arise:

```
f(n) = g(h(n)),f(n-1)))
```

where g is a function or a constructor. It feels that hardly any gain can be made by computing **h(n)** and **f(n-1)** in parallel, unless the calculation of **h(n)** is a particularly time-consuming business.

One can think of two approaches: (i) the user somehow gives hints at some places in the program from which the compiler deduces whether or not to process that particular part in parallel, (ii) one provides the compiler with a set of heuristics to make the decisions where to process in parallel by itself.

5.2 Suppose there was a random access possible on matrices...

We began this paper with the observation that up until now there are no implementations of functional languages that allow for random access on matrices. The idea is of course that random access improves speed. To what extent? Let us reconsider the above algorithm. It is $O(n^3)$, where **n** is the number of equations. Essentially because the forward phase is of this order. The forward phase is an itself repeating process on a diminishing input. The components of the process are all $O(n^2)$. One can expect to improve the functions **delete_left_most_column, reconstruct** and maybe the form of the definition of **forward** itself. But the order of the function **eliminate** is $O(n^2)$ and will not improve by the introduction of random access possibilities for matrices. Hence the order of **gaussian elimination** can not be improved by random access for arrays...

The conclusion seems to be that only for simple, i.e., low order algorithms on matrices and arrays one can expect much gain from random access for arrays, and that for algorithms of high orders the gain can be negligible.

5.3 Conclusion

Returning to the claim on functional programming mentioned in the introduction, we can say that:

(i) this exercise of writing a Miranda program for Gaussian elimination supports the feeling that a functional programming language like Miranda is a good medium to write structured software in. The resulting script, or program, seems to be short and

conspicuous compared to similar programs in von Neumann-languages like Pascal.

(ii) the formulations of mathematical notions in a language like Miranda can be expressed in a concise and precise way, not too far from the mathematical way of describing them. The troubles one encounters seem to stem from an acceptable vagueness with respect to computability of concepts in every day life mathematics in contrast to the precise and constructive notions a computer asks for.

6 Literature

[BvL] Barendregt, Henk, & Marc van Leeuwen, *Functional programming and the language TALE,* preprint 412, Dept. of Math., University of Utrecht, 1986, pp.87.

[T84] Turner, D.A., *Functional programs as executable specifications,* Phil. Trans. R. Soc. Lond. A 312, 363-388 (1984).

[T85] Turner, D.A., *Miranda: A non-strict functional language with polymorphic types,* in: proceedings IFIP international conference on functional programming languages and computer architecture, Nancy , September 1985, (Springer Lecture Notes in Computer Science, 201)

[S83] Sedgewick, Robert, *Algorithms,* Addison-Wesley Publishing Company, Reading Massachusetts etc, 1983.

[V87] Vries, F.J. de, *A functional program for the Fast Fourier Transform,* to appear in ACM Sigplan Notices, 1987.

[V87a] Vries, F.J. de, *Arrays in a functional language*, to appear in the Logic Group Preprint Series, University of Utrecht, 1987.

7 Appendix

```
|| The following program performs Gaussian elimination according to the
|| traditional algorithm. Maybe one better starts reading at the end.
||
|| As an example one can try
|| gaussian_elimination m2
||
|| Utrecht September 15 1987
|| Fer-Jan de Vries (RUU) (mcvax!asterix!ferjan)

m2 = P [[3,2,1,10],[2,3,1,14],[1,2,1,6]]
v2 = [10,14,6]

vector * == [*]
matrix * ::= P [[*]]
P x => error "no matrix", reject x
       where
       reject [] = False
       reject [a] = False
       reject (a:(b:x)) = (~#a=#b \/ reject (b:x))
|| a matrix is rejected if and only if it is a list of vectors of different
|| length

get_column :: num->matrix *->vector *
get_column n (P x) = [(v!n)|v<-x]

get_row :: num->matrix *->vector *
get_row n (P x) = x!n

width :: matrix *->num
width (P []) = error "meaningless notion"
width (P (a:x)) = #a

height :: matrix *->num
height (P x) = #x

transpose :: matrix *->matrix *
transpose (P x) = P[get_column i (P x) | i<-[1..(width (P x))]]

largest_in_list ([],l,n,p) = error "no unique solution", l=0
                           = p

largest_in_list ((b:x),l,n,p) = largest_in_list (x,b,n+1,n+1), abs b>= abs
l
                              = largest_in_list (x,l,n+1,p)
index_of_largest x = largest_in_list (x,0,0,0)

put_row_on_top m n = P ((get_row n m):[(get_row i m)| i<-[1..height m];
i~=n])

reconstruct x = put_row_on_top x n
                where
                n = index_of_largest (get_column 1 x)

eliminate (P []) = P []
```

```
eliminate (P (v:m)) = P(triangulate v m)

triangulate v [] = []
triangulate v (w:m) = ((clear_vec v w):(triangulate v m))

clear_vec (a:v) (b:w) = clear (a:v) (b:w) (b/a)
                        where
                        clear (a:v) (b:w) c = (b-a*c):(clear v w c)
                        clear [] [] c = []

divide_top_row (P(v:m)) = P ((div_list v (v!1)):m)
                          where
                          div_list (a:v) c = (a/c):(div_list v c)
                          div_list [] c = []

delete_left_most_column (P[]) = []
delete_left_most_column (P(v:m)) = (tl v):(delete_left_most_column (P m))

forward (P(v:m)) = (forward (P(dlmc (eliminate new))))++[reverse (get_row 1
new)
]
                   where
                   dlmc = delete_left_most_column
                   new = divide_top_row (reconstruct (P(v:m)))

forward (P[])=[]

substitute x [] = []
substitute x (v:m) = (row_subst x v):(substitute x m)
                     where
                     row_subst x (d:(c:v)) = (d-c*x):v

backward (v:(w:m)) = (backward (substitute (v!1) (w:m))) ++ [v!1]
backward (v:[]) = [v!1]

gaussian_elimination m = backward(forward m)
```