

Gauss-Jordan Imperative

By Jose Divasón and Jesús Aransay

January 20, 2015

Contents

1 Gauss-Jordan with mutable arrays	2
1.1 Elementary operations	2

```
theory Gauss-Jordan-Imp
imports
  $ISABELLE-HOME/src/HOL/Main
  $ISABELLE-HOME/src/HOL/Imperative-HOL/Imperative-HOL
  $ISABELLE-HOME/src/HOL/Imperative-HOL/ex/Subarray
  $ISABELLE-HOME/src/HOL/Multivariate-Analysis/Multivariate-Analysis
  $ISABELLE-HOME/src/HOL/Library/Code-Target-Numeral
  Code-Bit
begin

instantiation bit::heap
begin

instance
proof
show  $\exists to\_nat::bit \Rightarrow nat. inj\ to\_nat$ 
by (rule exI[of -  $\lambda n::bit. if\ n=0\ then\ 0\ else\ 1$ ], simp add: inj-on-def)
qed

end

fun swap :: bit array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit Heap where
  swap a i j = do {
    x  $\leftarrow$  Array.nth a i;
    y  $\leftarrow$  Array.nth a j;
    Array.upd i y a;
    Array.upd j x a;
    return ()
  }

definition example = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5, 0, 1705, 8, 3, 15];
```

```

    swap a 0 1;
    return a
  }

```

ML-val $\ll @\{\textit{code example}\} () \gg$

1 Gauss-Jordan with mutable arrays

1.1 Elementary operations

Arrays must be proven to be instances of heap type class, over any type being a heap itself.

```

instantiation array :: (heap) heap
begin
instance
proof
qed
end

```

Interchange rows is equivalent to swap:

```

fun interchange-rows :: (bit array) array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit Heap where
  interchange-rows M i j = do {
    x  $\leftarrow$  Array.nth M i;
    y  $\leftarrow$  Array.nth M j;
    Array.upd i y M;
    Array.upd j x M;
    return ()
  }

```

```

definition example2 = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5/7, 0, 1705, 8];
  b  $\leftarrow$  Array.of-list [24, 1, 4, 6, 9, 517, 98];
  M  $\leftarrow$  Array.of-list [a, b, a, b];
  interchange-rows M 0 3;
  return M
}

```

ML-val $\ll @\{\textit{code example2}\} () \gg$

The following proof fails with "fun"; no argument decreases, but the difference between "i" and "j":

Another interesting point is that we use a second "aux" bit array; this is to avoid later unwanted lateral effects:

It may be more elegant to return a bit array, but then it should be built inside of the recursive function, I guess...

```

function mult-array-rec :: (bit array)  $\Rightarrow$  (bit array)  $\Rightarrow$  bit  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit
Heap
  where
    mult-array-rec v w q i j = (if (i < j) then
      do {
        x  $\leftarrow$  Array.nth v i;
        w  $\leftarrow$  Array.upd i (x * q) w;
        mult-array-rec v w q (i + 1) j
      }
    else return ())
by pat-completeness auto
termination by (relation measure ( $\lambda(v,w,q,i,j). j - i$ )) simp+

```

Using the previous function, imposing bounds 0 and "Array.len v", and a newly allocated array, we recursively multiply the input array by a constant:

```

fun mult-array :: (bit array)  $\Rightarrow$  bit  $\Rightarrow$  (bit array) Heap where
  mult-array v q =
    do {
      l  $\leftarrow$  Array.len v;
      w  $\leftarrow$  Array.make l (%x. 0);
      mult-array-rec v w q 0 l;
      return w
    }

```

```

definition example3 = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5, 0, 1705, 8::bit];
  w  $\leftarrow$  mult-array a 3;
  return w
}

```

```

ML-val  $\ll$  @{code example3} () $\gg$ 

```

```

fun mult-row :: (bit array) array  $\Rightarrow$  nat  $\Rightarrow$  bit  $\Rightarrow$  unit Heap where
  mult-row M i q = do {
    x  $\leftarrow$  Array.nth M i;
    y  $\leftarrow$  mult-array x q;
    Array.upd i y M;
    return ()
  }

```

```

definition example4 = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5, 0, 1705, 8];
  b  $\leftarrow$  Array.of-list [24, 1, 4, 6, 9, 517, 98];
  M  $\leftarrow$  Array.of-list [a, b, a, b];
  mult-row M 2 0.5;
  return M
}

```

```

ML-val  $\ll$  @{code example4} () $\gg$ 

```

```

function add-array-rec :: (bit array)  $\Rightarrow$  (bit array)  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit Heap
where
  add-array-rec v w i j = (if (i < j) then
    do {
      a  $\leftarrow$  Array.nth v i;
      b  $\leftarrow$  Array.nth w i;
      v'  $\leftarrow$  Array.upd i (a + b) v;
      add-array-rec v' w (i + 1) j
    }
    else return ())
by pat-completeness auto
termination by (relation measure ( $\lambda(v,q,i,j). j - i$ )) simp+

```

Using the previous function, imposing bounds 0 and "Array.len v", we recursively add two arrays (they are supposed to be of equal length:)

```

fun add-array :: (bit array)  $\Rightarrow$  (bit array)  $\Rightarrow$  unit Heap where
  add-array v w =
    do {
      l  $\leftarrow$  Array.len v;
      add-array-rec v w 0 l;
      return()
    }

```

```

definition example5 = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5, 0, 1705, 8];
  b  $\leftarrow$  Array.of-list [24, 1, 4, 6, 9, 517, 98];
  add-array a b;
  return a
}

```

```

ML-val  $\ll$  @{code example5} () $\gg$ 

```

Row i is added row j times q:

```

fun row-add :: (bit array) array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bit  $\Rightarrow$  unit Heap where
  row-add M i j q = do {
    a  $\leftarrow$  Array.nth M i;
    b  $\leftarrow$  Array.nth M j;
    c  $\leftarrow$  mult-array b q;
    add-array a c;
    Array.upd i a M;
    return ()
  }

```

```

definition example6 = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5, 0, 1705, 8];
  b  $\leftarrow$  Array.of-list [24, 1, 4, 6, 9, 517, 98];
  c  $\leftarrow$  Array.of-list [24, 1, 4, 6, 9, 517, 98];
  d  $\leftarrow$  Array.of-list [42::bit, 2, 3, 5, 0, 1705, 8];

```

```

     $M \leftarrow \text{Array.of-list } [a, b, c, d];$ 
    row-add  $M$  1 2 0.5;
    return  $M$ 
  }

```

ML-val $\ll @\{\text{code example6}\} () \gg$

In the following function the first condition might be removed by using a tail recursive function, but I encountered some problems to prove it terminating.

```

function lnzero-position-f-ind :: bit array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat Heap where
  lnzero-position-f-ind  $v$   $i$   $j$  = (if ( $i < j$ ) then do {
     $x \leftarrow \text{Array.nth } v$   $i$ ;
    (if ( $x \neq 0$ ) then (return  $i$ ) else (lnzero-position-f-ind  $v$  ( $i + 1$ )  $j$ ))
  }
  else (raise "array lookup: index out of range"))
by pat-completeness auto
termination by (relation measure ( $\lambda(v,i,j). j - i$ )) simp+

```

```

fun lnzero-position-array-upt :: (bit array)  $\Rightarrow$  nat  $\Rightarrow$  nat Heap where
  lnzero-position-array-upt  $v$   $i$  =
  do {
     $l \leftarrow \text{Array.len } v$ ;
     $j \leftarrow \text{lnzero-position-f-ind } v$   $i$   $l$ ;
    return  $j$ 
  }

```

```

fun lnzero-position-array :: (bit array)  $\Rightarrow$  nat Heap where
  lnzero-position-array  $v$  =
  do {
     $l \leftarrow \text{Array.len } v$ ;
     $i \leftarrow \text{lnzero-position-f-ind } v$  0  $l$ ;
    return  $i$ 
  }

```

```

definition example7 = do {
   $v \leftarrow \text{Array.of-list } [0, 0, 0, 5, 0, 0, 9];$ 
   $b \leftarrow \text{lnzero-position-array-upt } v$  4;
  return  $b$ 
}

```

ML-val $\ll @\{\text{code example7}\} () \gg$

A predicate to detect columns array where there is no element to be pivoted.

```

function pivot-rec :: (bit array)  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bool Heap
where
  pivot-rec  $v$   $i$   $j$  = (if ( $i < j$ ) then do {
     $x \leftarrow \text{Array.nth } v$   $i$ ;
    (if ( $x = 0$ ) then pivot-rec  $v$  ( $i + 1$ )  $j$  else return True)
  }
  else return False)

```

```

    else return False)
by pat-completeness auto
termination by (relation measure ( $\lambda(v,i,j). j - i$ )) simp+

```

```

fun pivot :: (bit array)  $\Rightarrow$  nat  $\Rightarrow$  bool Heap
  where
    pivot v i = do {
      l  $\leftarrow$  Array.len v;
      b  $\leftarrow$  pivot-rec v i l;
      return b
    }

```

```

definition example8 = do {
  v  $\leftarrow$  Array.of-list [0, 0, 0, 5, 0, 0, 7];
  b  $\leftarrow$  pivot v 4;
  return b
}

```

```

ML-val  $\ll$  @{code example8} () $\gg$ 

```

The next definition requires that the element in position $M\ i\ j$ is equal to one; therefore, we must first pivot an element into position $M\ i\ j$, and then invoke it.

Using row i , every row in the interval k to l is reduced in column j .

```

function row-reduce-rec :: (bit array) array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit
Heap
  where
    row-reduce-rec M i j k l = (if (k < l) then do
      {
        (if (k = i) then row-reduce-rec M i j (k + 1) l else do {
          fk  $\leftarrow$  Array.nth M k;
          mkj  $\leftarrow$  Array.nth fk j;
          row-add M k i (- mkj);
          row-reduce-rec M i j (k + 1) l
        })
      }
    )
  else return ()
by (pat-completeness) auto
termination by (relation measure ( $\lambda(M,i,j,k,l). l - k$ )) simp+

```

Applying row reduction with row i in column j

```

fun row-reduce :: (bit array) array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit Heap
  where
    row-reduce M i j = do
      {
        nr  $\leftarrow$  Array.len M;
        row-reduce-rec M i j 0 nr
      }

```

}

```
definition example9 = do {
  a ← Array.of-list [42::bit, 2, 3, 5];
  b ← Array.of-list [24::bit, 1, 4, 6];
  c ← Array.of-list [24::bit, 1, 4, 6];
  d ← Array.of-list [42::bit, 2, 3, 5];
  M ← Array.of-list [a, b, c, d];
  row-reduce M 1 1;
  return M
}
```

ML-val $\ll \text{@}\{\text{code example9}\} \text{()}\gg$

We need an operation to get a column of a matrix; we use an auxiliary array in the second parameter.

function *column-rec* :: (*bit array*) *array* ⇒ *bit array* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *unit*
Heap

where

```
column-rec M w k i j = (if (i < j) then do {
  fi ← Array.nth M i;
  mij ← Array.nth fi k;
  Array.upd i mij w;
  column-rec M w k (i + 1) j
}
else return ())
```

by (*pat-completeness*) *auto*

termination by (*relation measure* ($\lambda(M,w,k,i,j). j - i$)) *simp+*

fun *column* :: (*bit array*) *array* ⇒ *nat* ⇒ (*bit array*) *Heap*

where

```
column M k = do {
  l ← Array.len M;
  w ← Array.make l (%x. 0);
  column-rec M w k 0 l;
  return w
}
```

```
definition example10 = do {
  a ← Array.of-list [42::bit, 2, 3, 5];
  b ← Array.of-list [24::bit, 1, 4, 6];
  c ← Array.of-list [24::bit, 1, 4, 6];
  d ← Array.of-list [42::bit, 2, 3, 5];
  M ← Array.of-list [a, b, c, d];
  w ← column M 3;
  return w
}
```

ML-val $\ll \text{@}\{\text{code example10}\} \text{()}\gg$

Still need to make zeros above and below the pivot;

```
fun Gauss-Jordan-in-ij :: (bit array) array  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  unit Heap
where
  Gauss-Jordan-in-ij M i j = do {
    cj  $\leftarrow$  column M j;
    n  $\leftarrow$  lnzero-position-array-upt cj i;
    pivot  $\leftarrow$  Array.nth cj n;
    interchange-rows M i n;
    mult-row M i (1 / pivot);
    row-reduce M i j;
    return ()
  }
```

```
definition example11 = do {
  a  $\leftarrow$  Array.of-list [42, 2, 3, 5];
  b  $\leftarrow$  Array.of-list [24, 1, 4, 6];
  c  $\leftarrow$  Array.of-list [24, 1, 0, 6];
  d  $\leftarrow$  Array.of-list [42, 2, 3, 5];
  M  $\leftarrow$  Array.of-list [a, b, c, d];
  Gauss-Jordan-in-ij M 2 2;
  return M
}
```

ML-val $\ll @\{code\ example11\} () \gg$

```
fun Gauss-Jordan-column-k :: ((bit array) array  $\times$  nat)  $\Rightarrow$  nat  $\Rightarrow$  ((bit array)
array  $\times$  nat) Heap
where
  Gauss-Jordan-column-k Mp k = do {
    ck  $\leftarrow$  column (fst Mp) k;
    b  $\leftarrow$  pivot ck (snd Mp);
    nr  $\leftarrow$  Array.len (fst Mp);
    (if (( $\neg$  b)  $\vee$  (nr = (snd Mp))) then return (((fst Mp), (snd Mp)))
    else do {
      Gauss-Jordan-in-ij (fst Mp) (snd Mp) k;
      return (((fst Mp), (snd Mp) + 1))
    })
  }
```

```
definition example12 = do {
  a  $\leftarrow$  Array.of-list [42::bit, 2, 3, 10];
  b  $\leftarrow$  Array.of-list [24::bit, 1, 4, 6];
  c  $\leftarrow$  Array.of-list [24::bit, 1, 0, 6];
  d  $\leftarrow$  Array.of-list [42::bit, 2, 3, 5];
  M  $\leftarrow$  Array.of-list [a, b,c, d];
  Gauss-Jordan-column-k (M, 0) 0;
  return M
}
```


ML-val $\ll \text{@}\{\text{code example12}\} \text{ ()}\gg$

In the following function k denotes the last column to be processed, i denotes the column being processed in each recursive call, and j the row in which the pivot in row i is being looked for.

function *Gauss-Jordan-upt-k-rec* :: (bit array) array $\Rightarrow$ nat $\Rightarrow$ nat $\Rightarrow$ nat $\Rightarrow$ unit Heap

where

Gauss-Jordan-upt-k-rec $M\ k\ i\ j =$ (if ($i < k$) then do {
 $Mp \leftarrow$ *Gauss-Jordan-column-k* (M, j) i ;
Gauss-Jordan-upt-k-rec (*fst* Mp) $k\ (i + 1)\ (\text{snd } Mp)$
} else return ())

by (pat-completeness) auto

termination by (relation measure ($\lambda(M,k,i,j). k - i$)) simp+

Using the previous function, starting from column 0 and row 0, we define a new function to produce Gauss-Jordan up to a column k .

fun *Gauss-Jordan-upt-k* :: (bit array) array $\Rightarrow$ nat $\Rightarrow$ unit Heap

where

Gauss-Jordan-upt-k $M\ k =$ *Gauss-Jordan-upt-k-rec* $M\ k\ 0\ 0$

definition *example13* = do {
 $a \leftarrow$ *Array.of-list* [42, 2, 3, 10];
 $b \leftarrow$ *Array.of-list* [24, 1, 4, 6];
 $c \leftarrow$ *Array.of-list* [24, 1, 0, 6];
 $d \leftarrow$ *Array.of-list* [42, 2, 3, 5];
 $M \leftarrow$ *Array.of-list* [a, b, c, d];
Gauss-Jordan-upt-k $M\ 3$;
return M
}

ML-val $\ll \text{@}\{\text{code example13}\} \text{ ()}\gg$

Note that we only count the elements in the first row to compute the number of columns; input matrices have to be well formed.

fun *cols* :: (bit array) array $\Rightarrow$ nat Heap

where *cols* $M =$ do {

$v \leftarrow$ *Array.nth* $M\ 0$;

$n \leftarrow$ *Array.len* v ;

return n

}

I am not pretty sure if the column to stop is *ncols* or "*ncols* - 1"

fun *Gauss-Jordan* :: (bit array) array $\Rightarrow$ unit Heap

where

Gauss-Jordan $M =$ do {

$k \leftarrow$ *cols* M ;

Gauss-Jordan-upt-k $M\ k$;

```

    return ()
  }

```

```

definition example14 = do {
  a ← Array.of-list [42, 2, 3, 10];
  b ← Array.of-list [24, 1, 4, 6];
  c ← Array.of-list [24, 1, 0, 6];
  d ← Array.of-list [42, 2, 3, 5];
  M ← Array.of-list [a, b, c, d];
  Gauss-Jordan M;
  return M
}

```

```

ML-val ⟨⟨ @{code example14} ()⟩⟩

```

```

definition example15 = do {
  a ← Array.of-list [1, 0, 0, 0];
  b ← Array.of-list [0, 1, 0, 0];
  c ← Array.of-list [0, 0, 1, 0];
  d ← Array.of-list [1, 0, 1, 0];
  M ← Array.of-list [a, b, c, d, a, c, d];
  Gauss-Jordan M;
  return M
}

```

```

ML-val ⟨⟨ @{code example15} ()⟩⟩

```

```

export-code Gauss-Jordan example14 in SML
  file Gauss-Jordan-imp.sml

```

```

definition matrix-z2 = do {
  a ← Array.of-list [0, 1, 0, 1::bit];
  b ← Array.of-list [0, 1, 1, 1];
  c ← Array.of-list [1, 1, 1, 1];
  d ← Array.of-list [1, 0, 0, 1];
  M ← Array.of-list [a, b, c, d];
  return M
}

```

```

definition print-result-Gauss M = do {
  Gauss-Jordan M;
  return M
}

```

```

term Gauss-Jordan
term print-result-Gauss

```

```
export-code matrix-z2 print-result-Gauss Gauss-Jordan in SML  
  module-name Gauss-SML  
  file Gauss-Jordan-Imp.sml  
  
end
```