A verified algorithm for computing the Smith normal form of a matrix

Jose Divasón

March 17, 2025

Abstract

This work presents a formal proof in Isabelle/HOL of an algorithm to transform a matrix into its Smith normal form, a canonical matrix form, in a general setting: the algorithm is parameterized by operations to prove its existence over elementary divisor rings, while execution is guaranteed over Euclidean domains. We also provide a formal proof on some results about the generality of this algorithm as well as the uniqueness of the Smith normal form.

Since Isabelle/HOL does not feature dependent types, the development is carried out switching conveniently between two different existing libraries: the Hermite normal form (based on HOL Analysis) and the Jordan normal form AFP entries. This permits to reuse results from both developments and it is done by means of the lifting and transfer package together with the use of local type definitions.

Contents

1 Definition of Smith normal form in HOL Analysis

theory *Smith-Normal-Form*
imports
Hermite.Hermite
begin

1.1 Definitions

Definition of diagonal matrix

definition *isDiagonal-upt-k A k* = (∀ *a b*. (*to-nat a* ≠ *to-nat b* ∧ (*to-nat a* < *k* ∨ (*to-nat b* < *k*))) ⟶ *A* $ *a* $ *b* = *0*)
definition *isDiagonal A* = (∀ *a b*. *to-nat a* ≠ *to-nat b* ⟶ *A* $ *a* $ *b* = *0*)

lemma *isDiagonal-intro*:
fixes *A*::*'a*::{*zero*}^*'cols*::*mod-type*^*'rows*::*mod-type*
assumes ⋀*a*::*'rows*. ⋀*b*::*'cols*. *to-nat a* = *to-nat b*
shows *isDiagonal A*
⟨*proof*⟩

Definition of Smith normal form up to position k. The element $A_{k-1,k-1}$ does not need to divide $A_{k,k}$ and $A_{k,k}$ could have non-zero entries above and below.

definition *Smith-normal-form-upt-k A k* =
(
(∀ *a b*. *to-nat a* = *to-nat b* ∧ *to-nat a* + *1* < *k* ∧ *to-nat b* + *1*< *k* ⟶ *A* $ *a* $ *b dvd A* $ (*a*+*1*) $ (*b*+*1*))
∧ *isDiagonal-upt-k A k*
)

definition *Smith-normal-form A* =
(
(∀ *a b*. *to-nat a* = *to-nat b* ∧ *to-nat a* + *1* < *nrows A* ∧ *to-nat b* + *1* < *ncols A* ⟶ *A* $ *a* $ *b dvd A* $ (*a*+*1*) $ (*b*+*1*))
∧ *isDiagonal A*
)

1.2 Basic properties

lemma *Smith-normal-form-min*:
Smith-normal-form A = *Smith-normal-form-upt-k A* (*min* (*nrows A*) (*ncols A*))
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-0*[*simp*]: *Smith-normal-form-upt-k A 0*
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-intro*:
assumes (⋀*a b*. *to-nat a* = *to-nat b* ∧ *to-nat a* + *1* < *k* ∧ *to-nat b* + *1*< *k* ⟹ *A* $ *a* $ *b dvd A* $ (*a*+*1*) $ (*b*+*1*))
and (⋀*a b*. (*to-nat a* ≠ *to-nat b* ∧ (*to-nat a* < *k* ∨ (*to-nat b* < *k*))) ⟹ *A* $ *a* $ *b* = *0*)
shows *Smith-normal-form-upt-k A k*
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-intro-alt*:
assumes (⋀*a b*. *to-nat a* = *to-nat b* ∧ *to-nat a* + *1* < *k* ∧ *to-nat b* + *1* < *k* ⟹ *A* $ *a* $ *b dvd A* $ (*a*+*1*) $ (*b*+*1*))
and *isDiagonal-upt-k A k*
shows *Smith-normal-form-upt-k A k*
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-condition1*:
fixes *A*::*'a*::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
assumes *Smith-normal-form-upt-k A k*
and *to-nat a* = *to-nat b* **and** *to-nat a* + *1* < *k* **and** *to-nat b* + *1* < *k*
shows *A* $ *a* $ *b dvd A* $ (*a*+*1*) $ (*b*+*1*)
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-condition2*:
fixes *A*::*'a*::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
assumes *Smith-normal-form-upt-k A k*
and *to-nat a* ≠ *to-nat b* **and** (*to-nat a* < *k* ∨ *to-nat b* < *k*)
shows ((*A* $ *a*) $ *b*) = *0*
⟨*proof*⟩

lemma *Smith-normal-form-upt-k1-intro*:

fixes $A::'a::\{bezout\text{-}ring\}\hat{}'cols::mod\text{-}type\hat{}'rows::mod\text{-}type$
assumes s: $Smith\text{-}normal\text{-}form\text{-}upt\text{-}k\ A\ k$
and $cond1$: $A\ \$\ from\text{-}nat\ (k-1)\ \$\ from\text{-}nat\ (k-1)\ dvd\ A\ \$\ (from\text{-}nat\ k)\ \$\ (from\text{-}nat\ k)$
and $cond2a$: $\forall a.\ to\text{-}nat\ a > k \longrightarrow A\ \$\ a\ \$\ from\text{-}nat\ k = 0$
and $cond2b$: $\forall b.\ to\text{-}nat\ b > k \longrightarrow A\ \$\ from\text{-}nat\ k\ \$\ b = 0$
shows $Smith\text{-}normal\text{-}form\text{-}upt\text{-}k\ A\ (Suc\ k)$
⟨*proof*⟩

lemma *Smith-normal-form-upt-k1-intro-diagonal*:
fixes $A::'a::\{bezout\text{-}ring\}\hat{}'cols::mod\text{-}type\hat{}'rows::mod\text{-}type$
assumes s: $Smith\text{-}normal\text{-}form\text{-}upt\text{-}k\ A\ k$
and d: $isDiagonal\ A$
and $cond1$: $A\ \$\ from\text{-}nat\ (k-1)\ \$\ from\text{-}nat\ (k-1)\ dvd\ A\ \$\ (from\text{-}nat\ k)\ \$\ (from\text{-}nat\ k)$
shows $Smith\text{-}normal\text{-}form\text{-}upt\text{-}k\ A\ (Suc\ k)$
⟨*proof*⟩

end

2 Algorithm to transform a diagonal matrix into its Smith normal form

theory *Diagonal-To-Smith*
imports *Hermite.Hermite*
HOL−Types-To-Sets.Types-To-Sets
Smith-Normal-Form
begin

lemma *invertible-mat-1*: $invertible\ (mat\ (1::'a::comm\text{-}ring\text{-}1))$
⟨*proof*⟩

2.1 Implementation of the algorithm

type-synonym $'a\ bezout = 'a \Rightarrow 'a \Rightarrow 'a \times 'a \times 'a \times 'a \times 'a$

hide-const *Countable.from-nat*
hide-const *Countable.to-nat*

The algorithm is based on the one presented by Bradley in his article entitled "Algorithms for Hermite and Smith normal matrices and linear diophantine equations". Some improvements have been introduced to get a general version for any matrix (including non-square and singular ones).

I also introduced another improvement: the element in the position j does

not need to be checked each time, since the element A_{ii} will already divide A_{jj} (where $j \leq k$). The gcd will be placed in A_{ii}.

This function transforms the element A_{jj} in order to be divisible by A_{ii} (and it changes A_{ii} as well).

The use of *from-nat* and *from-nat* is mandatory since the same index i cannot be used for both rows and columns at the same time, since they could have different type, concretely, when the matrix is rectangular.

The following definition is valid, but since execution requires the trick of converting all operations in terms of rows, then we would be recalculating the Bézout coefficients each time.

Thus, the definition is parameterized by the necessary elements instead of the operation, to avoid recalculations.

```
definition diagonal-step A i j d v =
            (χ a b. if a = from-nat i ∧ b = from-nat i then d else
             if a = from-nat j ∧ b = from-nat j
             then  v * (A $ (from-nat j) $ (from-nat j)) else A $ a $ b)


fun diagonal-to-Smith-i ::
nat list ⇒ 'a::{bezout-ring}^'cols::mod-type^'rows::mod-type ⇒ nat ⇒ ('a bezout)
  ⇒ 'a^'cols::mod-type^'rows::mod-type
 where
diagonal-to-Smith-i [] A i bezout = A |
diagonal-to-Smith-i (j#xs) A i bezout = (
  if A $ (from-nat i) $ (from-nat i) dvd A $ (from-nat j) $ (from-nat j)
      then diagonal-to-Smith-i xs A i bezout
  else let (p, q, u, v, d) = bezout (A $ from-nat i $ from-nat i) (A $ from-nat j $
from-nat j);
          A' = diagonal-step A i j d v
      in diagonal-to-Smith-i xs A' i bezout
  )


definition Diagonal-to-Smith-row-i A i bezout
  = diagonal-to-Smith-i [i+1..<min (nrows A) (ncols A)] A i bezout

fun diagonal-to-Smith-aux :: 'a::{bezout-ring}^'cols::mod-type^'rows::mod-type
  ⇒ nat list ⇒ ('a bezout) ⇒ 'a^'cols::mod-type^'rows::mod-type
  where
  diagonal-to-Smith-aux A [] bezout = A |
  diagonal-to-Smith-aux A (i#xs) bezout
        = diagonal-to-Smith-aux (Diagonal-to-Smith-row-i A i bezout) xs bezout
```

The minimum arises to include the case of non-square matrices (we do not demand the input diagonal matrix to be square, just have zeros in non-diagonal entries).

This iteration does not need to be performed until the last element of the diagonal, because in the second-to-last step the matrix will be already in Smith normal form.

definition *diagonal-to-Smith A bezout*
= diagonal-to-Smith-aux A [0..<min (nrows A) (ncols A) − 1] bezout

2.2 Code equations to get an executable version

definition *diagonal-step-row*
where *diagonal-step-row A i j c v a = vec-lambda (%b. if a = from-nat i ∧ b = from-nat i then c else*
if a = from-nat j ∧ b = from-nat j
then v ∗ (A $ (from-nat j) $ (from-nat j)) else A $ a $ b)

lemma *diagonal-step-code [code abstract]:*
vec-nth (diagonal-step-row A i j c v a) = (%b. if a = from-nat i ∧ b = from-nat i then c else
if a = from-nat j ∧ b = from-nat j
then v ∗ (A $ (from-nat j) $ (from-nat j)) else A $ a $ b)
⟨*proof*⟩

lemma *diagonal-step-code-nth [code abstract]: vec-nth (diagonal-step A i j c v)*
= diagonal-step-row A i j c v
⟨*proof*⟩

Code equation to avoid recalculations when computing the Bezout coefficients.

lemma *euclid-ext2-code[code]:*
euclid-ext2 a b = (let ((p,q),d) = euclid-ext a b in (p,q, − b div d, a div d, d))
⟨*proof*⟩

2.3 Examples of execution

value *let A= list-of-list-to-matrix [[12,0,0::int],[0,6,0::int],[0,0,2::int]]::int^3^3*
in matrix-to-list-of-list (diagonal-to-Smith A euclid-ext2)

Example obtained from: https://math.stackexchange.com/questions/77063/how-do-i-get-this-matrix-in-smith-normal-form-and-is-smith-normal-form-unique

value *let A= list-of-list-to-matrix*
[
[[:−3,1:],0,0,0],
[0,[:1,1:],0,0],
[0,0,[:1,1:],0],
[0,0,0,[:1,1:]]]::rat poly^4^4
in matrix-to-list-of-list (diagonal-to-Smith A euclid-ext2)

Polynomial matrix

value *let A = list-of-list-to-matrix*

```
  [
    [[:−3,1:],0,0,0],
    [0,[:1,1:],0,0],
    [0,0,[:1,1:],0],
    [0,0,0,[:1,1:]],
    [0,0,0,0]]::rat poly^4^5
  in matrix-to-list-of-list (diagonal-to-Smith A euclid-ext2)
```

2.4 Soundness of the algorithm

lemma *nrows-diagonal-step*[*simp*]: *nrows* (*diagonal-step A i j c v*) = *nrows A*
⟨*proof*⟩

lemma *ncols-diagonal-step*[*simp*]: *ncols* (*diagonal-step A i j c v*) = *ncols A*
⟨*proof*⟩

context
fixes *bezout*::′*a*::{*bezout-ring*} ⇒ ′*a* ⇒ ′*a* × ′*a* × ′*a* × ′*a* × ′*a*
assumes *ib*: *is-bezout-ext bezout*
begin

lemma *split-beta-bezout*: *bezout a b* =
(*fst*(*bezout a b*),
fst (*snd* (*bezout a b*)),
fst (*snd*(*snd* (*bezout a b*))),
fst (*snd*(*snd*(*snd* (*bezout a b*)))),
snd (*snd*(*snd*(*snd* (*bezout a b*))))) ⟨*proof*⟩

The following lemma shows that *diagonal-to-Smith-i* preserves the previous element. We use the assumption *to-nat a* = *to-nat b* in order to ensure that we are treating with a diagonal entry. Since the matrix could be rectangular, the types of a and b can be different, and thus we cannot write either $a = b$ or $A \$ a \$ b$.

lemma *diagonal-to-Smith-i-preserves-previous-diagonal*:
fixes *A*::′*a*:: {*bezout-ring*}^′*b*::*mod-type*^′*c*::*mod-type*
assumes *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *to-nat a* ∉ *set xs* **and** *to-nat a* = *to-nat b*
and *to-nat a* ≠ *i*
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)
shows (*diagonal-to-Smith-i xs A i bezout*) $ *a* $ *b* = *A* $ *a* $ *b*
⟨*proof*⟩

lemma *diagonal-step-dvd1*[*simp*]:
fixes *A*::′*a*::{*bezout-ring*}^′*b*::*mod-type*^′*c*::*mod-type* **and** *j i*
defines *v*==*case bezout* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat j* $ *from-nat j*) *of* (*p*,*q*,*u*,*v*,*d*) ⇒ *v*
and *d*==*case bezout* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat j* $ *from-nat j*) *of* (*p*,*q*,*u*,*v*,*d*) ⇒ *d*

shows *diagonal-step A i j d v $ from-nat i $ from-nat i dvd A $ from-nat i $ from-nat i*
⟨*proof*⟩

lemma *diagonal-step-dvd2*[*simp*]:
fixes *A*::*'a*::{*bezout-ring*}^*b*::*mod-type*^*c*::*mod-type* **and** *j i*
defines *v*==*case bezout* (*A $ from-nat i $ from-nat i*) (*A $ from-nat j $ from-nat j*) *of* (*p*,*q*,*u*,*v*,*d*) ⇒ *v*
and *d*==*case bezout* (*A $ from-nat i $ from-nat i*) (*A $ from-nat j $ from-nat j*) *of* (*p*,*q*,*u*,*v*,*d*) ⇒ *d*
shows *diagonal-step A i j d v $ from-nat i $ from-nat i dvd A $ from-nat j $ from-nat j*
⟨*proof*⟩

end

Once the step is carried out, the new element A'_{ii} will divide the element A_{ii}

lemma *diagonal-to-Smith-i-dvd-ii*:
fixes *A*::*'a*::{*bezout-ring*}^*b*::*mod-type*^*c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
shows *diagonal-to-Smith-i xs A i bezout $ from-nat i $ from-nat i dvd A $ from-nat i $ from-nat i*
⟨*proof*⟩

Once the step is carried out, the new element A'_{ii} divides the rest of elements of the diagonal. This proof requires commutativity (already included in the type restriction *bezout-ring*).

lemma *diagonal-to-Smith-i-dvd-jj*:
fixes *A*::*'a*::{*bezout-ring*}^*b*::*mod-type*^*c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)
and *to-nat a* ∈ *set xs*
and *to-nat a* = *to-nat b*
and *to-nat a* ≠ *i*
and *distinct xs*
shows (*diagonal-to-Smith-i xs A i bezout*) $ (*from-nat i*) $ (*from-nat i*)
dvd (*diagonal-to-Smith-i xs A i bezout*) $ *a* $ *b*
⟨*proof*⟩

The step preserves everything that is not in the diagonal

lemma *diagonal-to-Smith-i-preserves-previous*:
fixes *A*::*'a*:: {*bezout-ring*}^*b*::*mod-type*^*c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *a-not-b*: *to-nat a* ≠ *to-nat b*
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)

shows (*diagonal-to-Smith-i xs A i bezout*) $ *a* $ *b* = *A* $ *a* $ *b*
⟨*proof*⟩

lemma *diagonal-step-preserves*:
fixes *A*::'*a*::{*times*}^'*b*::*mod-type*^'*c*::*mod-type*
assumes *ai*: *a* ≠ *i* **and** *aj*: *a* ≠ *j* **and** *a-min*: *a* < *min* (*nrows A*) (*ncols A*)
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *j-min*: *j* < *min* (*nrows A*) (*ncols A*)
shows *diagonal-step A i j d v* $ *from-nat a* $ *from-nat b* = *A* $ *from-nat a* $ *from-nat b*
⟨*proof*⟩

context *GCD-ring*
begin

lemma *gcd-greatest*:
assumes *is-gcd gcd'* **and** *c dvd a* **and** *c dvd b*
shows *c dvd gcd' a b*
⟨*proof*⟩

end

This is a key lemma for the soundness of the algorithm.

lemma *diagonal-to-Smith-i-dvd*:
fixes *A*::'*a*:: {*bezout-ring*}^'*b*::*mod-type*^'*c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)
and ∀ *a b*. *to-nat a*∈*insert i* (*set xs*) ∧ *to-nat a* = *to-nat b* ⟶
A $ (*from-nat c*) $ (*from-nat c*) *dvd A* $ *a* $ *b*
and *c* ∉ (*set xs*) **and** *c*: *c*<*min* (*nrows A*) (*ncols A*)
and *distinct xs*
shows *A* $ (*from-nat c*) $ (*from-nat c*) *dvd*
(*diagonal-to-Smith-i xs A i bezout*) $ (*from-nat i*) $ (*from-nat i*)
⟨*proof*⟩

lemma *diagonal-to-Smith-i-dvd2*:
fixes *A*::'*a*:: {*bezout-ring*}^'*b*::*mod-type*^'*c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)
and *dvd-condition*: ∀ *a b*. *to-nat a* ∈ *insert i* (*set xs*) ∧ *to-nat a* = *to-nat b* ⟶
A $ (*from-nat c*) $ (*from-nat c*) *dvd A* $ *a* $ *b*
and *c-notin*: *c* ∉ (*set xs*)
and *c*: *c* < *min* (*nrows A*) (*ncols A*)
and *distinct*: *distinct xs*
and *ab*: *to-nat a* = *to-nat b*

and *a-in*: *to-nat a* ∈ *insert i* (*set xs*)
shows *A* $ (*from-nat c*) $ (*from-nat c*) *dvd* (*diagonal-to-Smith-i xs A i bezout*) $ *a* $ *b*
⟨*proof*⟩

lemma *diagonal-to-Smith-i-dvd2-k*:
fixes *A*::*'a*::{*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*k* **and** *k*≤*min* (*nrows A*) (*ncols A*)
and *dvd-condition*: ∀ *a b*. *to-nat a* ∈ *insert i* (*set xs*) ∧ *to-nat a* = *to-nat b* ⟶
A $ (*from-nat c*) $ (*from-nat c*) *dvd A* $ *a* $ *b*
and *c-notin*: *c* ∉ (*set xs*)
and *c*: *c* < *min* (*nrows A*) (*ncols A*)
and *distinct*: *distinct xs*
and *ab*: *to-nat a* = *to-nat b*
and *a-in*: *to-nat a* ∈ *insert i* (*set xs*)
shows *A* $ (*from-nat c*) $ (*from-nat c*) *dvd* (*diagonal-to-Smith-i xs A i bezout*) $ *a* $ *b*
⟨*proof*⟩

lemma *diagonal-to-Smith-row-i-preserves-previous*:
fixes *A*::*'a*:: {*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *a-not-b*: *to-nat a* ≠ *to-nat b*
shows *Diagonal-to-Smith-row-i A i bezout* $ *a* $ *b* = *A* $ *a* $ *b*
⟨*proof*⟩

lemma *diagonal-to-Smith-row-i-preserves-previous-diagonal*:
fixes *A*::*'a*:: {*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *i-min*: *i* < *min* (*nrows A*) (*ncols A*)
and *a-notin*: *to-nat a* ∉ *set* [*i* + *1*..<*min* (*nrows A*) (*ncols A*)]
and *ab*: *to-nat a* = *to-nat b*
and *ai*: *to-nat a* ≠ *i*
shows *Diagonal-to-Smith-row-i A i bezout* $ *a* $ *b* = *A* $ *a* $ *b*
⟨*proof*⟩

context
fixes *bezout*::*'a*::{*bezout-ring*} ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*
assumes *ib*: *is-bezout-ext bezout*
begin

lemma *diagonal-to-Smith-row-i-dvd-jj*:

fixes $A::'a::\{bezout\text{-}ring\}^{\sim}b::mod\text{-}type^{\sim}c::mod\text{-}type$
assumes $to\text{-}nat\ a \in \{i..<min\ (nrows\ A)\ (ncols\ A)\}$
and $to\text{-}nat\ a = to\text{-}nat\ b$
shows $(Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ A\ i\ bezout)\ \$\ (from\text{-}nat\ i)\ \$\ (from\text{-}nat\ i)$
$dvd\ (Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ A\ i\ bezout)\ \$\ a\ \$\ b$
⟨*proof*⟩

lemma *diagonal-to-Smith-row-i-dvd-ii*:
fixes $A::'a::\{bezout\text{-}ring\}^{\sim}b::mod\text{-}type^{\sim}c::mod\text{-}type$
shows $Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ A\ i\ bezout\ \$\ from\text{-}nat\ i\ \$\ from\text{-}nat\ i\ dvd\ A\ \$\ from\text{-}nat\ i\ \$\ from\text{-}nat\ i$
⟨*proof*⟩

lemma *diagonal-to-Smith-row-i-dvd-jj'*:
fixes $A::'a::\{bezout\text{-}ring\}^{\sim}b::mod\text{-}type^{\sim}c::mod\text{-}type$
assumes *a-in*: $to\text{-}nat\ a \in \{i..<min\ (nrows\ A)\ (ncols\ A)\}$
and *ab*: $to\text{-}nat\ a = to\text{-}nat\ b$
and *ci*: $c \leq i$
and *dvd-condition*: $\forall\ a\ b.\ to\text{-}nat\ a \in (set\ [i..<min\ (nrows\ A)\ (ncols\ A)]) \wedge to\text{-}nat\ a = to\text{-}nat\ b$
$\longrightarrow A\ \$\ from\text{-}nat\ c\ \$\ from\text{-}nat\ c\ dvd\ A\ \$\ a\ \$\ b$
shows $(Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ A\ i\ bezout)\ \$\ (from\text{-}nat\ c)\ \$\ (from\text{-}nat\ c)$
$dvd\ (Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ A\ i\ bezout)\ \$\ a\ \$\ b$
⟨*proof*⟩
end

lemma *diagonal-to-Smith-aux-append*:
$diagonal\text{-}to\text{-}Smith\text{-}aux\ A\ (xs\ @\ ys)\ bezout$
$= diagonal\text{-}to\text{-}Smith\text{-}aux\ (diagonal\text{-}to\text{-}Smith\text{-}aux\ A\ xs\ bezout)\ ys\ bezout$
⟨*proof*⟩

lemma *diagonal-to-Smith-aux-append2*[*simp*]:
$diagonal\text{-}to\text{-}Smith\text{-}aux\ A\ (xs\ @\ [ys])\ bezout$
$= Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ (diagonal\text{-}to\text{-}Smith\text{-}aux\ A\ xs\ bezout)\ ys\ bezout$
⟨*proof*⟩

lemma *isDiagonal-eq-upt-k-min*:
$isDiagonal\ A = isDiagonal\text{-}upt\text{-}k\ A\ (min\ (nrows\ A)\ (ncols\ A))$
⟨*proof*⟩

lemma *isDiagonal-eq-upt-k-max*:
$isDiagonal\ A = isDiagonal\text{-}upt\text{-}k\ A\ (max\ (nrows\ A)\ (ncols\ A))$
⟨*proof*⟩

lemma *isDiagonal*:
assumes *isDiagonal A*
and *to-nat a* ≠ *to-nat b* **shows** *A $ a $ b* = *0*
⟨*proof*⟩

lemma *nrows-diagonal-to-Smith-aux*[*simp*]:
shows *nrows* (*diagonal-to-Smith-aux A xs bezout*) = *nrows A* ⟨*proof*⟩

lemma *ncols-diagonal-to-Smith-aux*[*simp*]:
shows *ncols* (*diagonal-to-Smith-aux A xs bezout*) = *ncols A* ⟨*proof*⟩

context
fixes *bezout*::*'a*::{*bezout-ring*} ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*
assumes *ib*: *is-bezout-ext bezout*
begin

lemma *isDiagonal-diagonal-to-Smith-aux*:
assumes *diag-A*: *isDiagonal A* **and** *k*: *k* < *min* (*nrows A*) (*ncols A*)
shows *isDiagonal* (*diagonal-to-Smith-aux A* [*0*..<*k*] *bezout*)
⟨*proof*⟩
end

lemma *to-nat-less-nrows*[*simp*]:
fixes *A*::*'a*^*'b*::*mod-type*^*'c*::*mod-type*
and *a*::*'c*
shows *to-nat a* < *nrows A*
⟨*proof*⟩

lemma *to-nat-less-ncols*[*simp*]:
fixes *A*::*'a*^*'b*::*mod-type*^*'c*::*mod-type*
and *a*::*'b*
shows *to-nat a* < *ncols A*
⟨*proof*⟩

context
fixes *bezout*::*'a*::{*bezout-ring*} ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*
assumes *ib*: *is-bezout-ext bezout*
begin

The variables a and b must be arbitrary in the induction

lemma *diagonal-to-Smith-aux-dvd*:
fixes *A*::*'a*::{*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *ab*: *to-nat a* = *to-nat b*
and *c*: *c* < *k* **and** *ca*: *c* ≤ *to-nat a* **and** *k*: *k*<*min* (*nrows A*) (*ncols A*)
shows *diagonal-to-Smith-aux A* [*0*..<*k*] *bezout $ from-nat c $ from-nat c*
dvd diagonal-to-Smith-aux A [*0*..<*k*] *bezout $ a $ b*
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-Suc-imp-k*:
fixes *A*::*'a*::{*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *s*: *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*Suc k*] *bezout*) *k*
and *k*: *k*<*min* (*nrows A*) (*ncols A*)
shows *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*k*] *bezout*) *k*
⟨*proof*⟩

lemma *Smith-normal-form-upt-k-le*:
assumes *a*≤*k* **and** *Smith-normal-form-upt-k A k*
shows *Smith-normal-form-upt-k A a* ⟨*proof*⟩

lemma *Smith-normal-form-upt-k-imp-Suc-k*:
assumes *s*: *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*k*] *bezout*) *k*
and *k*: *k*<*min* (*nrows A*) (*ncols A*)
shows *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*Suc k*] *bezout*) *k*
⟨*proof*⟩

corollary *Smith-normal-form-upt-k-Suc-eq*:
assumes *k*: *k*<*min* (*nrows A*) (*ncols A*)
shows *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*Suc k*] *bezout*) *k*

= *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*k*] *bezout*) *k*
⟨*proof*⟩

end

lemma *nrows-diagonal-to-Smith-i*[*simp*]: *nrows* (*diagonal-to-Smith-i xs A i bezout*) = *nrows A*
⟨*proof*⟩

lemma *ncols-diagonal-to-Smith-i*[*simp*]: *ncols* (*diagonal-to-Smith-i xs A i bezout*) = *ncols A*
⟨*proof*⟩

lemma *nrows-Diagonal-to-Smith-row-i*[*simp*]: *nrows* (*Diagonal-to-Smith-row-i A i bezout*) = *nrows A*
⟨*proof*⟩

lemma *ncols-Diagonal-to-Smith-row-i*[*simp*]: *ncols* (*Diagonal-to-Smith-row-i A i bezout*) = *ncols A*
⟨*proof*⟩

lemma *isDiagonal-diagonal-step*:
assumes *diag-A*: *isDiagonal A* **and** *i*: *i*<*min* (*nrows A*) (*ncols A*)
and *j*: *j*<*min* (*nrows A*) (*ncols A*)

shows *isDiagonal* (*diagonal-step A i j d v*)
⟨*proof*⟩

lemma *isDiagonal-diagonal-to-Smith-i*:
assumes *isDiagonal A*
and *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)
and *i*<*min* (*nrows A*) (*ncols A*)
shows *isDiagonal* (*diagonal-to-Smith-i xs A i bezout*)
⟨*proof*⟩

lemma *isDiagonal-Diagonal-to-Smith-row-i*:
assumes *isDiagonal A* **and** *i* < *min* (*nrows A*) (*ncols A*)
shows *isDiagonal* (*Diagonal-to-Smith-row-i A i bezout*)
⟨*proof*⟩

lemma *isDiagonal-diagonal-to-Smith-aux-general*:
assumes *elements-xs-range*: ∀ *x*. *x* ∈ *set xs* ⟶ *x*<*min* (*nrows A*) (*ncols A*)
and *isDiagonal A*
shows *isDiagonal* (*diagonal-to-Smith-aux A xs bezout*)
⟨*proof*⟩

context
fixes *bezout*::*'a*::{*bezout-ring*} ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*
assumes *ib*: *is-bezout-ext bezout*
begin

The algorithm is iterated up to position k (not included). Thus, the matrix is in Smith normal form up to position k (not included).

lemma *Smith-normal-form-upt-k-diagonal-to-Smith-aux*:
fixes *A*::*'a*::{*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *k*<*min* (*nrows A*) (*ncols A*) **and** *d*: *isDiagonal A*
shows *Smith-normal-form-upt-k* (*diagonal-to-Smith-aux A* [*0*..<*k*] *bezout*) *k*
⟨*proof*⟩

end

lemma *nrows-diagonal-to-Smith*[*simp*]: *nrows* (*diagonal-to-Smith A bezout*) = *nrows A*
⟨*proof*⟩

lemma *ncols-diagonal-to-Smith*[*simp*]: *ncols* (*diagonal-to-Smith A bezout*) = *ncols A*
⟨*proof*⟩

lemma *isDiagonal-diagonal-to-Smith*:
assumes *d*: *isDiagonal A*
shows *isDiagonal* (*diagonal-to-Smith A bezout*)

⟨*proof*⟩

This is the soundess lemma.

lemma *Smith-normal-form-diagonal-to-Smith*:
fixes *A*::*'a*::{*bezout-ring*}^*'b*::*mod-type*^*'c*::*mod-type*
assumes *ib*: *is-bezout-ext bezout*
and *d*: *isDiagonal A*
shows *Smith-normal-form* (*diagonal-to-Smith A bezout*)
⟨*proof*⟩

2.5 Implementation and formal proof of the matrices P and Q which transform the input matrix by means of elementary operations.

fun *diagonal-step-PQ* :: *'a*::{*bezout-ring*}^*'cols*::*mod-type*^*'rows*::*mod-type* ⇒ *nat* ⇒ *nat* ⇒ *'a bezout* ⇒
(
(*'a*::{*bezout-ring*}^*'rows*::*mod-type*^*'rows*::*mod-type*) ×
(*'a*::{*bezout-ring*}^*'cols*::*mod-type*^*'cols*::*mod-type*)
)
where *diagonal-step-PQ A i k bezout* =
(*let i-row* = *from-nat i*; *k-row* = *from-nat k*; *i-col* = *from-nat i*; *k-col* = *from-nat k*;
(*p*, *q*, *u*, *v*, *d*) = *bezout* (*A* $ *i-row* $ *from-nat i*) (*A* $ *k-row* $ *from-nat k*);
P = *row-add* (*interchange-rows* (*row-add* (*mat 1*) *k-row i-row p*) *i-row k-row*) *k-row i-row* (−*v*);
Q = *mult-column* (*column-add* (*column-add* (*mat 1*) *i-col k-col q*) *k-col i-col u*) *k-col* (−*1*)
in (*P*,*Q*)
)

Examples

value *let A* = *list-of-list-to-matrix* [[*12*,*0*,*0*::*int*],[*0*,*6*,*0*::*int*],[*0*,*0*,*2*::*int*]]::*int*^*3*^*3*;
i=0; *k=1*;
(*p*, *q*, *u*, *v*, *d*) = *euclid-ext2* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat k* $ *from-nat k*);
(*P*,*Q*) = *diagonal-step-PQ A i k euclid-ext2*
in matrix-to-list-of-list (*diagonal-step A i k d v*)

value *let A* = *list-of-list-to-matrix* [[*12*,*0*,*0*::*int*],[*0*,*6*,*0*::*int*],[*0*,*0*,*2*::*int*]]::*int*^*3*^*3*;
i=0; *k=1*;
(*p*, *q*, *u*, *v*, *d*) = *euclid-ext2* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat k* $ *from-nat k*);
(*P*,*Q*) = *diagonal-step-PQ A i k euclid-ext2*
in matrix-to-list-of-list (*P***(*A*)***Q*)

value *let A* = *list-of-list-to-matrix* [[*12*,*0*,*0*::*int*],[*0*,*6*,*0*::*int*],[*0*,*0*,*2*::*int*]]::*int*^*3*^*3*;
i=0; *k=1*;

$(p,\ q,\ u,\ v,\ d) = \mathit{euclid\text{-}ext2}\ (A\ \$\ \mathit{from\text{-}nat}\ i\ \$\ \mathit{from\text{-}nat}\ i)\ (A\ \$\ \mathit{from\text{-}nat}\ k\ \$\ \mathit{from\text{-}nat}\ k)$;
$(P,Q) = \mathit{diagonal\text{-}step\text{-}PQ}\ A\ i\ k\ \mathit{euclid\text{-}ext2}$
in $\mathit{matrix\text{-}to\text{-}list\text{-}of\text{-}list}\ (P{**}(A){**}Q)$

lemmas $\mathit{diagonal\text{-}step\text{-}PQ\text{-}def} = \mathit{diagonal\text{-}step\text{-}PQ.simps}$

lemma *from-nat-neq-rows*:
fixes $A{::}'a\hat{}'cols{::}mod\text{-}type\hat{}'rows{::}mod\text{-}type$
assumes $i{:}\ i{<}(nrows\ A)$ **and** $k{:}\ k{<}(nrows\ A)$ **and** $ik{:}\ i \neq k$
shows $\mathit{from\text{-}nat}\ i \neq (\mathit{from\text{-}nat}\ k{::}'rows)$
⟨*proof*⟩

lemma *from-nat-neq-cols*:
fixes $A{::}'a\hat{}'cols{::}mod\text{-}type\hat{}'rows{::}mod\text{-}type$
assumes $i{:}\ i{<}(ncols\ A)$ **and** $k{:}\ k{<}(ncols\ A)$ **and** $ik{:}\ i \neq k$
shows $\mathit{from\text{-}nat}\ i \neq (\mathit{from\text{-}nat}\ k{::}'cols)$
⟨*proof*⟩

lemma *diagonal-step-PQ-invertible-P*:
fixes $A{::}'a{::}\{bezout\text{-}ring\}\hat{}'cols{::}mod\text{-}type\hat{}'rows{::}mod\text{-}type$
assumes $PQ{:}\ (P,Q) = \mathit{diagonal\text{-}step\text{-}PQ}\ A\ i\ k\ bezout$
and $pquvd{:}\ (p,q,u,v,d) = bezout\ (A\ \$\ \mathit{from\text{-}nat}\ i\ \$\ \mathit{from\text{-}nat}\ i)\ (A\ \$\ \mathit{from\text{-}nat}\ k\ \$\ \mathit{from\text{-}nat}\ k)$
and $i\text{-}not\text{-}k{:}\ i \neq k$
and $i{:}\ i{<}min\ (nrows\ A)\ (ncols\ A)$ **and** $k{:}\ k{<}min\ (nrows\ A)\ (ncols\ A)$
shows $invertible\ P$
⟨*proof*⟩

lemma *diagonal-step-PQ-invertible-Q*:
fixes $A{::}'a{::}\{bezout\text{-}ring\}\hat{}'cols{::}mod\text{-}type\hat{}'rows{::}mod\text{-}type$
assumes $PQ{:}\ (P,Q) = \mathit{diagonal\text{-}step\text{-}PQ}\ A\ i\ k\ bezout$
and $pquvd{:}\ (p,q,u,v,d) = bezout\ (A\ \$\ \mathit{from\text{-}nat}\ i\ \$\ \mathit{from\text{-}nat}\ i)\ (A\ \$\ \mathit{from\text{-}nat}\ k\ \$\ \mathit{from\text{-}nat}\ k)$
and $i\text{-}not\text{-}k{:}\ i \neq k$
and $i{:}\ i{<}min\ (nrows\ A)\ (ncols\ A)$ **and** $k{:}\ k{<}min\ (nrows\ A)\ (ncols\ A)$
shows $invertible\ Q$
⟨*proof*⟩

lemma *mat-q-1*[*simp*]: $mat\ q\ \$\ a\ \$\ a = q$ ⟨*proof*⟩

lemma *mat-q-0*[*simp*]:
assumes $ab{:}\ a \neq b$

shows *mat q $ a $ b = 0* ⟨*proof*⟩

This is an alternative definition for the matrix P in each step, where entries are given explicitly instead of being computed as a composition of elementary operations.

lemma *diagonal-step-PQ-P-alt*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::*mod-type*^*'rows*::*mod-type*
assumes *PQ*: (*P*,*Q*) = *diagonal-step-PQ A i k bezout*
and *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *bezout* (*A $ from-nat i $ from-nat i*) (*A $ from-nat k $ from-nat k*)
and *i*: *i*<*min* (*nrows A*) (*ncols A*) **and** *k*: *k*<*min* (*nrows A*) (*ncols A*) **and** *ik*: *i* ≠ *k*
shows
P = (χ *a b*.
if a = *from-nat i* ∧ *b* = *from-nat i then p else*
if a = *from-nat i* ∧ *b* = *from-nat k then 1 else*
if a = *from-nat k* ∧ *b* = *from-nat i then* −*v* ∗ *p* + *1 else*
if a = *from-nat k* ∧ *b* = *from-nat k then* −*v else*
if a = *b then 1 else 0*)
⟨*proof*⟩

This is an alternative definition for the matrix Q in each step, where entries are given explicitly instead of being computed as a composition of elementary operations.

lemma *diagonal-step-PQ-Q-alt*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::*mod-type*^*'rows*::*mod-type*
assumes *PQ*: (*P*,*Q*) = *diagonal-step-PQ A i k bezout*
and *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *bezout* (*A $ from-nat i $ from-nat i*) (*A $ from-nat k $ from-nat k*)
and *i*: *i*<*min* (*nrows A*) (*ncols A*) **and** *k*: *k*<*min* (*nrows A*) (*ncols A*) **and** *ik*: *i* ≠ *k*
shows
Q = (χ *a b*.
if a = *from-nat i* ∧ *b* = *from-nat i then 1 else*
if a = *from-nat i* ∧ *b* = *from-nat k then* −*u else*
if a = *from-nat k* ∧ *b* = *from-nat i then q else*
if a = *from-nat k* ∧ *b* = *from-nat k then* −*q*∗*u*−*1 else*
if a = *b then 1 else 0*)
⟨*proof*⟩

P**A can be rewriten as elementary operations over A.

lemma *diagonal-step-PQ-PA*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::*mod-type*^*'rows*::*mod-type*
assumes *PQ*: (*P*,*Q*) = *diagonal-step-PQ A i k bezout*
and *b*: (*p*,*q*,*u*,*v*,*d*) = *bezout* (*A $ from-nat i $ from-nat i*) (*A $ from-nat k $ from-nat k*)
shows *P***A* = *row-add* (*interchange-rows*
(*row-add A* (*from-nat k*) (*from-nat i*) *p*) (*from-nat i*) (*from-nat k*)) (*from-nat k*) (*from-nat i*) (− *v*)

⟨*proof*⟩

lemma *diagonal-step-PQ-PAQ'*:
fixes *A*::*'a*::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
assumes *PQ*: (*P*,*Q*) = *diagonal-step-PQ A i k bezout*
and *b*: (*p*,*q*,*u*,*v*,*d*) = *bezout* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat k* $ *from-nat k*)
shows *P****A****Q* = (*mult-column* (*column-add* (*column-add* (*P****A*) (*from-nat i*) (*from-nat k*) *q*)
(*from-nat k*) (*from-nat i*) *u*) (*from-nat k*) (− *1*))
⟨*proof*⟩

corollary *diagonal-step-PQ-PAQ*:
fixes *A*::*'a*::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
assumes *PQ*: (*P*,*Q*) = *diagonal-step-PQ A i k bezout*
and *b*: (*p*,*q*,*u*,*v*,*d*) = *bezout* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat k* $ *from-nat k*)
shows *P****A****Q* = (*mult-column* (*column-add* (*column-add* (*row-add* (*interchange-rows*

(*row-add A* (*from-nat k*) (*from-nat i*) *p*) (*from-nat i*)
(*from-nat k*)) (*from-nat k*) (*from-nat i*) (− *v*)) (*from-nat i*) (*from-nat k*) *q*)
(*from-nat k*) (*from-nat i*) *u*) (*from-nat k*) (− *1*))
⟨*proof*⟩

lemma *isDiagonal-imp-0*:
assumes *isDiagonal A*
and *from-nat a* ≠ *from-nat b*
and *a* < *min* (*nrows A*) (*ncols A*)
and *b* < *min* (*nrows A*) (*ncols A*)
shows *A* $ *from-nat a* $ *from-nat b* = *0*
⟨*proof*⟩

lemma *diagonal-step-PQ*:
fixes *A*::*'a*::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
assumes *PQ*: (*P*,*Q*) = *diagonal-step-PQ A i k bezout*
and *b*: (*p*,*q*,*u*,*v*,*d*) = *bezout* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat k* $ *from-nat k*)
and *i*: *i*<*min* (*nrows A*) (*ncols A*) **and** *k*: *k*<*min* (*nrows A*) (*ncols A*) **and** *ik*: *i* ≠ *k*
and *ib*: *is-bezout-ext bezout* **and** *diag*: *isDiagonal A*
shows *diagonal-step A i k d v* = *P****A****Q*
⟨*proof*⟩

fun *diagonal-to-Smith-i-PQ* ::
nat list ⇒ *nat* ⇒ (*'a*::{*bezout-ring*} *bezout*)
⇒ ((*'a*^*'rows*::*mod-type*^*'rows*::*mod-type*)×(*'a*^*'cols*::*mod-type*^*'rows*::*mod-type*)×
(*'a*^*'cols*::*mod-type*^*'cols*::*mod-type*))
⇒ ((*'a*^*'rows*::*mod-type*^*'rows*::*mod-type*)× (*'a*^*'cols*::*mod-type*^*'rows*::*mod-type*)
× (*'a*^*'cols*::*mod-type*^*'cols*::*mod-type*))
where
diagonal-to-Smith-i-PQ [] *i bezout* (*P*,*A*,*Q*) = (*P*,*A*,*Q*) |
diagonal-to-Smith-i-PQ (*j*#*xs*) *i bezout* (*P*,*A*,*Q*) = (
if A $ (*from-nat i*) $ (*from-nat i*) *dvd A* $ (*from-nat j*) $ (*from-nat j*)
then diagonal-to-Smith-i-PQ xs i bezout (*P*,*A*,*Q*)
else let (*p*, *q*, *u*, *v*, *d*) = *bezout* (*A* $ *from-nat i* $ *from-nat i*) (*A* $ *from-nat j* $
from-nat j);
A' = *diagonal-step A i j d v*;
(*P'*,*Q'*) = *diagonal-step-PQ A i j bezout*
in diagonal-to-Smith-i-PQ xs i bezout (*P'*∗∗*P*,*A'*,*Q*∗∗*Q'*) — Apply the step
)

This is implemented by fun. This way, I can do pattern-matching for (P, A, Q).

fun *Diagonal-to-Smith-row-i-PQ*
where *Diagonal-to-Smith-row-i-PQ i bezout* (*P*,*A*,*Q*)
= *diagonal-to-Smith-i-PQ* [*i* + *1*..<*min* (*nrows A*) (*ncols A*)] *i bezout* (*P*,*A*,*Q*)

Deleted from the simplified and renamed as it would be a definition.

declare *Diagonal-to-Smith-row-i-PQ.simps*[*simp del*]
lemmas *Diagonal-to-Smith-row-i-PQ-def* = *Diagonal-to-Smith-row-i-PQ.simps*

fun *diagonal-to-Smith-aux-PQ*
where
diagonal-to-Smith-aux-PQ [] *bezout* (*P*,*A*,*Q*) = (*P*,*A*,*Q*) |
diagonal-to-Smith-aux-PQ (*i*#*xs*) *bezout* (*P*,*A*,*Q*)
= *diagonal-to-Smith-aux-PQ xs bezout* (*Diagonal-to-Smith-row-i-PQ i bezout*
(*P*,*A*,*Q*))

lemma *diagonal-to-Smith-aux-PQ-append*:
diagonal-to-Smith-aux-PQ (*xs* @ *ys*) *bezout* (*P*,*A*,*Q*)
= *diagonal-to-Smith-aux-PQ ys bezout* (*diagonal-to-Smith-aux-PQ xs bezout*
(*P*,*A*,*Q*))
⟨*proof*⟩

lemma *diagonal-to-Smith-aux-PQ-append2*[*simp*]:
diagonal-to-Smith-aux-PQ (*xs* @ [*ys*]) *bezout* (*P*,*A*,*Q*)
= *Diagonal-to-Smith-row-i-PQ ys bezout* (*diagonal-to-Smith-aux-PQ xs bezout*
(*P*,*A*,*Q*))
⟨*proof*⟩

context
fixes $A::'a::\{bezout\text{-}ring\}^{\wedge}cols::mod\text{-}type^{\wedge}rows::mod\text{-}type$
and $B::'a::\{bezout\text{-}ring\}^{\wedge}cols::mod\text{-}type^{\wedge}rows::mod\text{-}type$
and P **and** Q
and $bezout::'a\ bezout$
assumes PAQ: $P{**}A{**}Q = B$
and P: $invertible\ P$ **and** Q: $invertible\ Q$
and ib: $is\text{-}bezout\text{-}ext\ bezout$
begin

The output is the same as the one in the version where P and Q are not computed.

lemma *diagonal-to-Smith-i-PQ-eq*:
assumes $P'B'Q'$: $(P',B',Q') = diagonal\text{-}to\text{-}Smith\text{-}i\text{-}PQ\ xs\ i\ bezout\ (P,B,Q)$
and xs: $\forall x.\ x \in set\ xs \longrightarrow x < min\ (nrows\ A)\ (ncols\ A)$
and $diag$: $isDiagonal\ B$ **and** $i\text{-}notin$: $i \notin set\ xs$ **and** i: $i<min\ (nrows\ A)\ (ncols\ A)$
shows $B' = diagonal\text{-}to\text{-}Smith\text{-}i\ xs\ B\ i\ bezout$
⟨*proof*⟩

lemma *diagonal-to-Smith-i-PQ'*:
assumes $P'B'Q'$: $(P',B',Q') = diagonal\text{-}to\text{-}Smith\text{-}i\text{-}PQ\ xs\ i\ bezout\ (P,B,Q)$
and xs: $\forall x.\ x \in set\ xs \longrightarrow x < min\ (nrows\ A)\ (ncols\ A)$
and $diag$: $isDiagonal\ B$ **and** $i\text{-}notin$: $i \notin set\ xs$ **and** i: $i<min\ (nrows\ A)\ (ncols\ A)$
shows $B' = P'{**}A{**}Q' \wedge invertible\ P' \wedge invertible\ Q'$
⟨*proof*⟩

corollary *diagonal-to-Smith-i-PQ*:
assumes $P'B'Q'$: $(P',B',Q') = diagonal\text{-}to\text{-}Smith\text{-}i\text{-}PQ\ xs\ i\ bezout\ (P,B,Q)$
and xs: $\forall x.\ x \in set\ xs \longrightarrow x < min\ (nrows\ A)\ (ncols\ A)$
and $diag$: $isDiagonal\ B$ **and** $i\text{-}notin$: $i \notin set\ xs$ **and** i: $i<min\ (nrows\ A)\ (ncols\ A)$
shows $B' = P'{**}A{**}Q' \wedge invertible\ P' \wedge invertible\ Q' \wedge B' = diagonal\text{-}to\text{-}Smith\text{-}i\ xs\ B\ i\ bezout$
⟨*proof*⟩

lemma *Diagonal-to-Smith-row-i-PQ-eq*:
assumes $P'B'Q'$: $(P',B',Q') = Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\text{-}PQ\ i\ bezout\ (P,B,Q)$
and $diag$: $isDiagonal\ B$ **and** i: $i < min\ (nrows\ A)\ (ncols\ A)$
shows $B' = Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ B\ i\ bezout$
⟨*proof*⟩

lemma *Diagonal-to-Smith-row-i-PQ'*:

assumes $P'B'Q'$: $(P',B',Q') = Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\text{-}PQ\ i\ bezout\ (P,B,Q)$
and *diag*: *isDiagonal* B **and** i: $i < min\ (nrows\ A)\ (ncols\ A)$
shows $B' = P'{**}A{**}Q' \wedge invertible\ P' \wedge invertible\ Q'$
⟨*proof*⟩

lemma *Diagonal-to-Smith-row-i-PQ*:
assumes $P'B'Q'$: $(P',B',Q') = Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\text{-}PQ\ i\ bezout\ (P,B,Q)$
and *diag*: *isDiagonal* B **and** i: $i < min\ (nrows\ A)\ (ncols\ A)$
shows $B' = P'{**}A{**}Q' \wedge invertible\ P' \wedge invertible\ Q' \wedge B' = Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\ B\ i\ bezout$
⟨*proof*⟩

end

context
fixes A::$'a$::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
and B::$'a$::{*bezout-ring*}^*cols*::*mod-type*^*rows*::*mod-type*
and P **and** Q
and *bezout*::$'a$ *bezout*
assumes PAQ: $P{**}A{**}Q = B$
and P: *invertible* P **and** Q: *invertible* Q
and *ib*: *is-bezout-ext bezout*
begin

lemma *diagonal-to-Smith-aux-PQ*:
assumes $P'B'Q'$: $(P',B',Q') = diagonal\text{-}to\text{-}Smith\text{-}aux\text{-}PQ\ [0..<k]\ bezout\ (P,B,Q)$
and *diag*: *isDiagonal* B **and** k:$k<min\ (nrows\ A)\ (ncols\ A)$
shows $B' = P'{**}A{**}Q' \wedge invertible\ P' \wedge invertible\ Q' \wedge B' = diagonal\text{-}to\text{-}Smith\text{-}aux\ B\ [0..<k]\ bezout$
⟨*proof*⟩

end

fun *diagonal-to-Smith-PQ*
where *diagonal-to-Smith-PQ A bezout*
$= diagonal\text{-}to\text{-}Smith\text{-}aux\text{-}PQ\ [0..<min\ (nrows\ A)\ (ncols\ A) - 1]\ bezout\ (mat\ 1, A, mat\ 1)$

declare *diagonal-to-Smith-PQ.simps*[*simp del*]
lemmas *diagonal-to-Smith-PQ-def* = *diagonal-to-Smith-PQ.simps*

lemma *diagonal-to-Smith-PQ*:
fixes A::$'a$::{*bezout-ring*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes A: *isDiagonal* A **and** *ib*: *is-bezout-ext bezout*
assumes PBQ: $(P,B,Q) = diagonal\text{-}to\text{-}Smith\text{-}PQ\ A\ bezout$
shows $B = P{**}A{**}Q \wedge invertible\ P \wedge invertible\ Q \wedge B = diagonal\text{-}to\text{-}Smith\ A\ bezout$
⟨*proof*⟩

lemma *diagonal-to-Smith-PQ-exists*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *A*: *isDiagonal A*
shows ∃ *P Q*.
invertible (*P*::*'a*^*'rows*::{*mod-type*}^*'rows*::{*mod-type*})
∧ *invertible* (*Q*::*'a*^*'cols*::{*mod-type*}^*'cols*::{*mod-type*})
∧ *Smith-normal-form* (*P*∗∗*A*∗∗*Q*)
⟨*proof*⟩

2.6 The final soundness theorem

lemma *diagonal-to-Smith-PQ'*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *A*: *isDiagonal A* **and** *ib*: *is-bezout-ext bezout*
assumes *PBQ*: (*P*,*S*,*Q*) = *diagonal-to-Smith-PQ A bezout*
shows *S* = *P*∗∗*A*∗∗*Q* ∧ *invertible P* ∧ *invertible Q* ∧ *Smith-normal-form S*
⟨*proof*⟩

end

3 A new bridge to convert theorems from JNF to HOL Analysis and vice-versa, based on the *mod-type* class

theory *Mod-Type-Connect*
imports
Perron-Frobenius.HMA-Connect
Rank-Nullity-Theorem.Mod-Type
Gauss-Jordan.Elementary-Operations
begin

Some lemmas on *Mod-Type.to-nat* and *Mod-Type.from-nat* are added to have them with the same names as the analogous ones for *Bij-Nat.to-nat* and *Bij-Nat.to-nat*.

lemma *inj-to-nat*: *inj to-nat* ⟨*proof*⟩
lemmas *from-nat-inj* = *from-nat-eq-imp-eq*
lemma *range-to-nat*: *range* (*to-nat* :: *'a* :: *mod-type* ⇒ *nat*) = {*0* ..< *CARD*(*'a*)}
⟨*proof*⟩

This theory is an adaptation of the one presented in *Perron-Frobenius.HMA-Connect*, but for matrices and vectors where indexes have the *mod-type* class restriction.

It is worth noting that some definitions still use the old abbreviation for HOL Analysis (HMA, from HOL Multivariate Analysis) instead of HA. This

is done to be consistent with the existing names in the Perron-Frobenius development

context includes *vec.lifting*
begin
end

definition *from-hma$_v$* :: *'a ^ 'n :: mod-type ⇒ 'a Matrix.vec* **where**
from-hma$_v$ v = Matrix.vec CARD('n) (λ i. v $h from-nat i)

definition *from-hma$_m$* :: *'a ^ 'nc :: mod-type ^ 'nr :: mod-type ⇒ 'a Matrix.mat* **where**
from-hma$_m$ a = Matrix.mat CARD('nr) CARD('nc) (λ (i,j). a $h from-nat i $h from-nat j)

definition *to-hma$_v$* :: *'a Matrix.vec ⇒ 'a ^ 'n :: mod-type* **where**
to-hma$_v$ v = (χ i. v $v to-nat i)

definition *to-hma$_m$* :: *'a Matrix.mat ⇒ 'a ^ 'nc :: mod-type ^ 'nr :: mod-type* **where**
to-hma$_m$ a = (χ i j. a $$ (to-nat i, to-nat j))

lemma *to-hma-from-hma$_v$[simp]*: *to-hma$_v$ (from-hma$_v$ v) = v*
⟨*proof*⟩

lemma *to-hma-from-hma$_m$[simp]*: *to-hma$_m$ (from-hma$_m$ v) = v*
⟨*proof*⟩

lemma *from-hma-to-hma$_v$[simp]*:
v ∈ carrier-vec (CARD('n)) ⟹ from-hma$_v$ (to-hma$_v$ v :: 'a ^ 'n :: mod-type) = v
⟨*proof*⟩

lemma *from-hma-to-hma$_m$[simp]*:
A ∈ carrier-mat (CARD('nr)) (CARD('nc)) ⟹ from-hma$_m$ (to-hma$_m$ A :: 'a ^ 'nc :: mod-type ^ 'nr :: mod-type) = A
⟨*proof*⟩

lemma *from-hma$_v$-inj[simp]*: *from-hma$_v$ x = from-hma$_v$ y ⟷ x = y*
⟨*proof*⟩

lemma *from-hma$_m$-inj[simp]*: *from-hma$_m$ x = from-hma$_m$ y ⟷ x = y*
⟨*proof*⟩

definition *HMA-V* :: *'a Matrix.vec ⇒ 'a ^ 'n :: mod-type ⇒ bool* **where**
HMA-V = (λ v w. v = from-hma$_v$ w)

definition *HMA-M* :: *'a Matrix.mat ⇒ 'a ^ 'nc :: mod-type ^ 'nr :: mod-type ⇒ bool* **where**
HMA-M = (λ a b. a = from-hma$_m$ b)

definition *HMA-I* :: *nat* ⇒ *'n* :: *mod-type* ⇒ *bool* **where**
HMA-I = (λ *i a*. *i* = *to-nat a*)

context includes *lifting-syntax*
begin

lemma *Domainp-HMA-V* [*transfer-domain-rule*]:
Domainp (*HMA-V* :: *'a Matrix.vec* ⇒ *'a* ^ *'n* :: *mod-type* ⇒ *bool*) = (λ *v*. *v* ∈ *carrier-vec* (*CARD*(*'n*)))
⟨*proof*⟩

lemma *Domainp-HMA-M* [*transfer-domain-rule*]:
Domainp (*HMA-M* :: *'a Matrix.mat* ⇒ *'a* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ *bool*)
= (λ *A*. *A* ∈ *carrier-mat CARD*(*'nr*) *CARD*(*'nc*))
⟨*proof*⟩

lemma *Domainp-HMA-I* [*transfer-domain-rule*]:
Domainp (*HMA-I* :: *nat* ⇒ *'n* :: *mod-type* ⇒ *bool*) = (λ *i*. *i* < *CARD*(*'n*)) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *bi-unique-HMA-V* [*transfer-rule*]: *bi-unique HMA-V left-unique HMA-V right-unique HMA-V*
⟨*proof*⟩

lemma *bi-unique-HMA-M* [*transfer-rule*]: *bi-unique HMA-M left-unique HMA-M right-unique HMA-M*
⟨*proof*⟩

lemma *bi-unique-HMA-I* [*transfer-rule*]: *bi-unique HMA-I left-unique HMA-I right-unique HMA-I*
⟨*proof*⟩

lemma *right-total-HMA-V* [*transfer-rule*]: *right-total HMA-V*
⟨*proof*⟩

lemma *right-total-HMA-M* [*transfer-rule*]: *right-total HMA-M*
⟨*proof*⟩

lemma *right-total-HMA-I* [*transfer-rule*]: *right-total HMA-I*
⟨*proof*⟩

lemma *HMA-V-index* [*transfer-rule*]: (*HMA-V* ===> *HMA-I* ===> (=)) ($*v*) ($*h*)
⟨*proof*⟩

lemma *HMA-M-index* [*transfer-rule*]:
(*HMA-M* ===> *HMA-I* ===> *HMA-I* ===> (=)) (λ *A i j*. *A* $$ (*i*,*j*))
index-hma
⟨*proof*⟩

lemma *HMA-V-0* [*transfer-rule*]: *HMA-V* (*0*$_v$ *CARD*('*n*)) (*0* :: '*a* :: *zero* ^ '*n*:: *mod-type*)
⟨*proof*⟩

lemma *HMA-M-0* [*transfer-rule*]:
HMA-M (*0*$_m$ *CARD*('*nr*) *CARD*('*nc*)) (*0* :: '*a* :: *zero* ^ '*nc*:: *mod-type* ^ '*nr* :: *mod-type*)
⟨*proof*⟩

lemma *HMA-M-1*[*transfer-rule*]:
HMA-M (*1*$_m$ (*CARD*('*n*))) (*mat 1* :: '*a*::{*zero*,*one*}^'*n*:: *mod-type*^'*n*:: *mod-type*)
⟨*proof*⟩

lemma *from-hma*$_v$*-add*: *from-hma*$_v$ *v* + *from-hma*$_v$ *w* = *from-hma*$_v$ (*v* + *w*)
⟨*proof*⟩

lemma *HMA-V-add* [*transfer-rule*]: (*HMA-V* ===> *HMA-V* ===> *HMA-V*) (+) (+)
⟨*proof*⟩

lemma *from-hma*$_v$*-diff*: *from-hma*$_v$ *v* − *from-hma*$_v$ *w* = *from-hma*$_v$ (*v* − *w*)
⟨*proof*⟩

lemma *HMA-V-diff* [*transfer-rule*]: (*HMA-V* ===> *HMA-V* ===> *HMA-V*) (−) (−)
⟨*proof*⟩

lemma *from-hma*$_m$*-add*: *from-hma*$_m$ *a* + *from-hma*$_m$ *b* = *from-hma*$_m$ (*a* + *b*)
⟨*proof*⟩

lemma *HMA-M-add* [*transfer-rule*]: (*HMA-M* ===> *HMA-M* ===> *HMA-M*) (+) (+)
⟨*proof*⟩

lemma *from-hma*$_m$*-diff*: *from-hma*$_m$ *a* − *from-hma*$_m$ *b* = *from-hma*$_m$ (*a* − *b*)
⟨*proof*⟩

lemma *HMA-M-diff* [*transfer-rule*]: (*HMA-M* ===> *HMA-M* ===> *HMA-M*) (−) (−)
⟨*proof*⟩

lemma *scalar-product*: **fixes** $v :: 'a :: semiring\text{-}1\ \hat{}\ 'n :: mod\text{-}type$
shows *scalar-prod* (*from-hma*$_v$ v) (*from-hma*$_v$ w) = *scalar-product* v w
⟨*proof*⟩

lemma [*simp*]:
from-hma$_m$ ($y :: 'a\ \hat{}\ 'nc :: mod\text{-}type\ \hat{}\ 'nr :: mod\text{-}type$) ∈ *carrier-mat* (*CARD*($'nr$)) (*CARD*($'nc$))
dim-row (*from-hma*$_m$ ($y :: 'a\ \hat{}\ 'nc :: mod\text{-}type\ \hat{}\ 'nr :: mod\text{-}type$)) = *CARD*($'nr$)
dim-col (*from-hma*$_m$ ($y :: 'a\ \hat{}\ 'nc :: mod\text{-}type\ \hat{}\ 'nr :: mod\text{-}type$)) = *CARD*($'nc$)
⟨*proof*⟩

lemma [*simp*]:
from-hma$_v$ ($y :: 'a\ \hat{}\ 'n :: mod\text{-}type$) ∈ *carrier-vec* (*CARD*($'n$))
dim-vec (*from-hma*$_v$ ($y :: 'a\ \hat{}\ 'n :: mod\text{-}type$)) = *CARD*($'n$)
⟨*proof*⟩

lemma *HMA-scalar-prod* [*transfer-rule*]:
(*HMA-V* ===> *HMA-V* ===> (=)) *scalar-prod scalar-product*
⟨*proof*⟩

lemma *HMA-row* [*transfer-rule*]: (*HMA-I* ===> *HMA-M* ===> *HMA-V*) (λ *i a*. *Matrix.row a i*) *row*
⟨*proof*⟩

lemma *HMA-col* [*transfer-rule*]: (*HMA-I* ===> *HMA-M* ===> *HMA-V*) (λ *i a*. *col a i*) *column*
⟨*proof*⟩

lemma *HMA-M-mk-mat*[*transfer-rule*]: ((*HMA-I* ===> *HMA-I* ===> (=)) ===> *HMA-M*)
(λ *f*. *Matrix.mat* (*CARD*($'nr$)) (*CARD*($'nc$)) (λ (*i*,*j*). *f i j*))
(*mk-mat* :: (($'nr$ ⇒ $'nc$ ⇒ $'a$) ⇒ $'a\ \hat{}\ 'nc :: mod\text{-}type\ \hat{}\ 'nr :: mod\text{-}type$))
⟨*proof*⟩

lemma *HMA-M-mk-vec*[*transfer-rule*]: ((*HMA-I* ===> (=)) ===> *HMA-V*)
(λ *f*. *Matrix.vec* (*CARD*($'n$)) (λ *i*. *f i*))
(*mk-vec* :: (($'n$ ⇒ $'a$) ⇒ $'a\ \hat{}\ 'n :: mod\text{-}type$))
⟨*proof*⟩

lemma *mat-mult-scalar*: A ∗∗ B = *mk-mat* (λ *i j*. *scalar-product* (*row i A*) (*column j B*))
⟨*proof*⟩

lemma *mult-mat-vec-scalar*: A ∗*v* v = *mk-vec* (λ *i*. *scalar-product* (*row i A*) v)
⟨*proof*⟩

lemma *dim-row-transfer-rule*:
HMA-M A (*A'* :: *'a* ^ *'nc*:: *mod-type* ^ *'nr*:: *mod-type*) ⟹ (=) (*dim-row A*) (*CARD*(*'nr*))
⟨*proof*⟩

lemma *dim-col-transfer-rule*:
HMA-M A (*A'* :: *'a* ^ *'nc*:: *mod-type* ^ *'nr*:: *mod-type*) ⟹ (=) (*dim-col A*) (*CARD*(*'nc*))
⟨*proof*⟩

lemma *HMA-M-mult* [*transfer-rule*]: (*HMA-M* ===> *HMA-M* ===> *HMA-M*) (∗) (∗∗)
⟨*proof*⟩

lemma *HMA-V-smult* [*transfer-rule*]: ((=) ===> *HMA-V* ===> *HMA-V*) (·$_v$) (∗*s*)
⟨*proof*⟩

lemma *HMA-M-mult-vec* [*transfer-rule*]: (*HMA-M* ===> *HMA-V* ===> *HMA-V*) (∗$_v$) (∗*v*)
⟨*proof*⟩

lemma *HMA-det* [*transfer-rule*]: (*HMA-M* ===> (=)) *Determinant.det*
(*det* :: *'a* :: *comm-ring-1* ^ *'n* :: *mod-type* ^ *'n* :: *mod-type* ⇒ *'a*)
⟨*proof*⟩

lemma *HMA-mat*[*transfer-rule*]: ((=) ===> *HMA-M*) (λ *k*. *k* ·$_m$ 1$_m$ *CARD*(*'n*))

(*Finite-Cartesian-Product.mat* :: *'a*::*semiring-1* ⇒ *'a*^*'n* :: *mod-type*^*'n* :: *mod-type*)
⟨*proof*⟩

lemma *HMA-mat-minus*[*transfer-rule*]: (*HMA-M* ===> *HMA-M* ===> *HMA-M*)

(λ *A B*. *A* + *map-mat uminus B*) ((−) :: *'a* :: *group-add* ^*'nc*:: *mod-type*^*'nr*:: *mod-type*
⇒ *'a*^*'nc*:: *mod-type*^*'nr*:: *mod-type* ⇒ *'a*^*'nc*:: *mod-type*^*'nr*:: *mod-type*)
⟨*proof*⟩

lemma *HMA-transpose-matrix* [*transfer-rule*]:
(*HMA-M* ===> *HMA-M*) *transpose-mat transpose*
⟨*proof*⟩

lemma *HMA-invertible-matrix-mod-type*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'n* :: *mod-type* ^ *'n* ::

mod-type
⇒ -) ===> (=)) *invertible-mat invertible*
⟨*proof*⟩

end

Some transfer rules for relating the elementary operations are also proved.

context
includes *lifting-syntax*
begin

lemma *HMA-swaprows*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nr* :: *mod-type* ⇒ -)
===> *Mod-Type-Connect.HMA-M*)
(λ*A a b*. *swaprows a b A*) *interchange-rows*
⟨*proof*⟩

lemma *HMA-swapcols*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nc* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nc* :: *mod-type* ⇒ -)
===> *Mod-Type-Connect.HMA-M*)
(λ*A a b*. *swapcols a b A*) *interchange-columns*
⟨*proof*⟩

lemma *HMA-addrow*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nr* :: *mod-type* ⇒ -)
===> (=)
===> *Mod-Type-Connect.HMA-M*)
(λ*A a b q*. *addrow q a b A*) *row-add*
⟨*proof*⟩

lemma *HMA-addcol*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nc* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nc* :: *mod-type* ⇒ -)
===> (=)
===> *Mod-Type-Connect.HMA-M*)
(λ*A a b q*. *addcol q a b A*) *column-add*
⟨*proof*⟩

lemma *HMA-multrow*[*transfer-rule*]:
(*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nr* :: *mod-type* ⇒ -)
===> (=)
===> *Mod-Type-Connect.HMA-M*)
(λ*A i q*. *multrow i q A*) *mult-row*
⟨*proof*⟩

lemma *HMA-multcol*[*transfer-rule*]:
(*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I* :: - ⇒*'nc* :: *mod-type* ⇒ -)
===> (=)
===> *Mod-Type-Connect.HMA-M*)
(λ*A i q*. *multcol i q A*) *mult-column*
⟨*proof*⟩

end

fun *HMA-M3* **where**
HMA-M3 (*P*,*A*,*Q*)
(*P'* :: *'a* :: *comm-ring-1* ^ *'nr* :: *mod-type* ^ *'nr* :: *mod-type*,
A' :: *'a* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type*,
Q' :: *'a* ^ *'nc* :: *mod-type* ^ *'nc* :: *mod-type*) =
(*Mod-Type-Connect.HMA-M P P'* ∧ *Mod-Type-Connect.HMA-M A A'* ∧ *Mod-Type-Connect.HMA-M Q Q'*)

lemma *HMA-M3-def*:
HMA-M3 A B = (*Mod-Type-Connect.HMA-M* (*fst A*) (*fst B*)
∧ *Mod-Type-Connect.HMA-M* (*fst* (*snd A*)) (*fst* (*snd B*))
∧ *Mod-Type-Connect.HMA-M* (*snd* (*snd A*)) (*snd* (*snd B*)))
⟨*proof*⟩

context
includes *lifting-syntax*
begin

lemma *Domainp-HMA-M3* [*transfer-domain-rule*]:
Domainp (*HMA-M3* :: -⇒(-×(*'a*::*comm-ring-1*^*'nc*::*mod-type*^*'nr*::*mod-type*)×-)⇒-)

= (λ(*P*,*A*,*Q*). *P* ∈ *carrier-mat CARD*(*'nr*) *CARD*(*'nr*) ∧ *A* ∈ *carrier-mat CARD*(*'nr*) *CARD*(*'nc*)
∧ *Q* ∈ *carrier-mat CARD*(*'nc*) *CARD*(*'nc*))
⟨*proof*⟩

lemma *bi-unique-HMA-M3* [*transfer-rule*]: *bi-unique HMA-M3 left-unique HMA-M3*

right-unique HMA-M3
⟨*proof*⟩

lemma *right-total-HMA-M3* [*transfer-rule*]: *right-total HMA-M3*
⟨*proof*⟩

end

end

4 Missing results

theory *SNF-Missing-Lemmas*
imports
Hermite.Hermite
Mod-Type-Connect
Jordan-Normal-Form.DL-Rank-Submatrix
List−Index.List-Index
begin

This theory presents some missing lemmas that are required for the Smith normal form development. Some of them could be added to different AFP entries, such as the Jordan Normal Form AFP entry by René Thiemann and Akihisa Yamada.

However, not all the lemmas can be added directly, since some imports are required.

hide-const (**open**) *C*
hide-const (**open**) *measure*

4.1 Miscellaneous lemmas

lemma *sum-two-rw*: $(\sum i = 0..<2.\ f\ i) = (\sum i \in \{0,1::nat\}.\ f\ i)$
⟨*proof*⟩

lemma *sum-common-left*:
fixes $f::'a \Rightarrow 'b::comm\text{-}ring\text{-}1$
assumes *finite A*
shows $sum\ (\lambda i.\ c * f\ i)\ A = c * sum\ f\ A$
⟨*proof*⟩

lemma *prod3-intro*:
assumes $fst\ A = a$ **and** $fst\ (snd\ A) = b$ **and** $snd\ (snd\ A) = c$
shows $A = (a,b,c)$ ⟨*proof*⟩

4.2 Transfer rules for the HMA_Connect file of the Perron-Frobenius development

hide-const (**open**) *HMA-M HMA-I to-hma$_m$ from-hma$_m$*
hide-fact (**open**) *from-hma$_m$-def from-hma-to-hma$_m$ HMA-M-def HMA-I-def dim-row-transfer-rule dim-col-transfer-rule*

context
includes *lifting-syntax*
begin

lemma *HMA-invertible-matrix*[*transfer-rule*]:
((*HMA-Connect.HMA-M* :: - ⇒ 'a :: *comm-ring-1* ^ 'n ^ 'n ⇒ -) ===> (=))
invertible-mat invertible
⟨*proof*⟩
end

4.3 Lemmas obtained from HOL Analysis using local type definitions

thm *Cartesian-Space.invertible-mult*
thm *invertible-iff-is-unit*
thm *det-non-zero-imp-unit*
thm *mat-mult-left-right-inverse*

lemma *invertible-mat-zero*:
assumes *A*: *A* ∈ *carrier-mat 0 0*
shows *invertible-mat A*
⟨*proof*⟩

lemma *invertible-mult-JNF*:
fixes *A*::'*a*::*comm-ring-1 mat*
assumes *A*: *A*∈*carrier-mat n n* **and** *B*: *B*∈*carrier-mat n n*
and *inv-A*: *invertible-mat A* **and** *inv-B*: *invertible-mat B*
shows *invertible-mat* (*A*∗*B*)
⟨*proof*⟩

lemma *invertible-iff-is-unit-JNF*:
assumes *A*: *A* ∈ *carrier-mat n n*
shows *invertible-mat A* ⟷ (*Determinant.det A*) *dvd 1*
⟨*proof*⟩

4.4 Lemmas about matrices, submatrices and determinants

thm *mat-mult-left-right-inverse*
lemma *mat-mult-left-right-inverse*:
fixes *A* :: '*a*::*comm-ring-1 mat*
assumes *A*: *A* ∈ *carrier-mat n n*
and *B*: *B* ∈ *carrier-mat n n* **and** *AB*: *A* ∗ *B* = *1$_m$ n*
shows *B* ∗ *A* = *1$_m$ n*

⟨*proof*⟩

context *comm-ring-1*
begin

lemma *col-submatrix-UNIV*:
assumes $j < card\ \{i.\ i < dim\text{-}col\ A \wedge i \in J\}$
shows $col\ (submatrix\ A\ UNIV\ J)\ j = col\ A\ (pick\ J\ j)$
⟨*proof*⟩

lemma *submatrix-split2*: $submatrix\ A\ I\ J = submatrix\ (submatrix\ A\ I\ UNIV)\ UNIV\ J$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *submatrix-mult*:
$submatrix\ (A*B)\ I\ J = submatrix\ A\ I\ UNIV * submatrix\ B\ UNIV\ J$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *det-singleton*:
assumes *A*: $A \in carrier\text{-}mat\ 1\ 1$
shows $det\ A = A\ \$\$\ (0,0)$
⟨*proof*⟩

lemma *submatrix-singleton-index*:
assumes *A*: $A \in carrier\text{-}mat\ n\ m$
and *an*: $a < n$ **and** *bm*: $b < m$
shows $submatrix\ A\ \{a\}\ \{b\}\ \$\$\ (0,0) = A\ \$\$\ (a,b)$
⟨*proof*⟩
end

lemma *det-not-inj-on*:
assumes *not-inj-on*: $\neg\ inj\text{-}on\ f\ \{0..<n\}$
shows $det\ (mat_r\ n\ n\ (\lambda i.\ Matrix.row\ B\ (f\ i))) = 0$
⟨*proof*⟩

lemma *mat-row-transpose*: $(mat_r\ nr\ nc\ f)^T = mat\ nc\ nr\ (\lambda(i,j).\ vec\text{-}index\ (f\ j)\ i)$
⟨*proof*⟩

lemma *obtain-inverse-matrix*:
assumes *A*: $A \in carrier\text{-}mat\ n\ n$ **and** *i*: *invertible-mat A*
obtains *B* **where** *inverts-mat A B* **and** *inverts-mat B A* **and** $B \in carrier\text{-}mat\ n\ n$
⟨*proof*⟩

lemma *invertible-mat-smult-mat*:
fixes $A :: 'a::comm\text{-}ring\text{-}1\ mat$
assumes *inv-A*: *invertible-mat A* **and** *k*: *k dvd 1*
shows *invertible-mat* $(k \cdot_m A)$
⟨*proof*⟩

lemma *invertible-mat-one*[*simp*]: *invertible-mat* $(1_m\ n)$
⟨*proof*⟩

lemma *four-block-mat-dim0*:
assumes *A*: $A \in carrier\text{-}mat\ n\ n$
and *B*: $B \in carrier\text{-}mat\ n\ 0$
and *C*: $C \in carrier\text{-}mat\ 0\ n$
and *D*: $D \in carrier\text{-}mat\ 0\ 0$
shows *four-block-mat A B C D* = *A*
⟨*proof*⟩

lemma *det-four-block-mat-lower-right-id*:
assumes *A*: $A \in carrier\text{-}mat\ m\ m$
and *B*: $B = 0_m\ m\ (n-m)$
and *C*: $C = 0_m\ (n-m)\ m$
and *D*: $D = 1_m\ (n-m)$
and $n>m$
shows *Determinant.det* (*four-block-mat A B C D*) = *Determinant.det A*
⟨*proof*⟩

lemma *mult-eq-first-row*:
assumes *A*: $A \in carrier\text{-}mat\ 1\ n$
and *B*: $B \in carrier\text{-}mat\ m\ n$
and *m0*: $m \neq 0$
and *r*: *Matrix.row A 0* = *Matrix.row B 0*
shows *Matrix.row* $(A * V)\ 0$ = *Matrix.row* $(B * V)\ 0$
⟨*proof*⟩

lemma *smult-mat-mat-one-element*:
assumes *A*: $A \in carrier\text{-}mat\ 1\ 1$ **and** *B*: $B \in carrier\text{-}mat\ 1\ n$
shows $A * B = A\ \$\$\ (0,0) \cdot_m B$
⟨*proof*⟩

lemma *determinant-one-element*:
assumes *A*: $A \in carrier\text{-}mat\ 1\ 1$ **shows** *Determinant.det A* = $A\ \$\$\ (0,0)$
⟨*proof*⟩

lemma *invertible-mat-transpose*:
assumes *inv-A*: *invertible-mat* (*A*::'*a*::*comm-ring-1 mat*)
shows *invertible-mat* A^T
⟨*proof*⟩

lemma *dvd-elements-mult-matrix-left*:
assumes *A*: (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat m n*
and *P*: *P* ∈ *carrier-mat m m*
and *x*: (∀ *i j*. *i*<*m* ∧ *j*<*n* ⟶ *x dvd A*$$(*i*,*j*))
shows (∀ *i j*. *i*<*m* ∧ *j*<*n* ⟶ *x dvd* (*P*∗*A*)$$(*i*,*j*))
⟨*proof*⟩

lemma *dvd-elements-mult-matrix-right*:
assumes *A*: (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat m n*
and *Q*: *Q* ∈ *carrier-mat n n*
and *x*: (∀ *i j*. *i*<*m* ∧ *j*<*n* ⟶ *x dvd A*$$(*i*,*j*))
shows (∀ *i j*. *i*<*m* ∧ *j*<*n* ⟶ *x dvd* (*A*∗*Q*)$$(*i*,*j*))
⟨*proof*⟩

lemma *dvd-elements-mult-matrix-left-right*:
assumes *A*: (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat m n*
and *P*: *P* ∈ *carrier-mat m m*
and *Q*: *Q* ∈ *carrier-mat n n*
and *x*: (∀ *i j*. *i*<*m* ∧ *j*<*n* ⟶ *x dvd A*$$(*i*,*j*))
shows (∀ *i j*. *i*<*m* ∧ *j*<*n* ⟶ *x dvd* (*P*∗*A*∗*Q*)$$(*i*,*j*))
⟨*proof*⟩

definition *append-cols* :: '*a* :: *zero mat* ⇒ '*a mat* ⇒ '*a mat* (**infixr** ‹@$_c$› *65*)**where**
A @$_c$ *B* = *four-block-mat A B* (0$_m$ *0* (*dim-col A*)) (0$_m$ *0* (*dim-col B*))

lemma *append-cols-carrier*[*simp*,*intro*]:
A ∈ *carrier-mat n a* ⟹ *B* ∈ *carrier-mat n b* ⟹ (*A* @$_c$ *B*) ∈ *carrier-mat n* (*a*+*b*)
⟨*proof*⟩

lemma *append-cols-mult-left*:
assumes *A*: *A* ∈ *carrier-mat n a*
and *B*: *B* ∈ *carrier-mat n b*
and *P*: *P* ∈ *carrier-mat n n*
shows *P* ∗ (*A* @$_c$ *B*) = (*P*∗*A*) @$_c$ (*P*∗*B*)
⟨*proof*⟩

lemma *append-cols-mult-right-id*:
assumes *A*: (*A*::'*a*::*semiring-1 mat*) ∈ *carrier-mat n 1*
and *B*: *B* ∈ *carrier-mat n* (*m*−*1*)
and *C*: *C* = *four-block-mat* (1$_m$ *1*) (0$_m$ *1* (*m* − *1*)) (0$_m$ (*m* − *1*) *1*) *D*

and D: $D \in$ *carrier-mat* $(m-1)$ $(m-1)$
shows $(A$ @$_c$ $B) * C = A$ @$_c$ $(B * D)$
⟨*proof*⟩

lemma *append-cols-mult-right-id2*:
assumes A: $(A::'a::$*semiring-1 mat*$) \in$ *carrier-mat* n a
and B: $B \in$ *carrier-mat* n b
and C: $C =$ *four-block-mat* D $(0_m$ a $b)$ $(0_m$ b $a)$ $(1_m$ $b)$
and D: $D \in$ *carrier-mat* a a
shows $(A$ @$_c$ $B) * C = (A * D)$ @$_c$ B
⟨*proof*⟩

lemma *append-cols-nth*:
assumes A: $A \in$ *carrier-mat* n a
and B: $B \in$ *carrier-mat* n b
and i: $i<n$ **and** j: $j < a + b$
shows $(A$ @$_c$ $B)$ $$ $(i, j) =$ (*if* $j <$ *dim-col* A *then* A $$$(i,j)$ *else* $B$$$(i,j-a))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *append-cols-split*:
assumes d: *dim-col* $A > 0$
shows $A =$ *mat-of-cols* (*dim-row* A) [*col* A *0*] @$_c$
mat-of-cols (*dim-row* A) (*map* (*col* A) [*1*..<*dim-col* A]) (**is** *?lhs = ?A1* @$_c$ *?A2*)
⟨*proof*⟩

lemma *append-rows-nth*:
assumes A: $A \in$ *carrier-mat* a n
and B: $B \in$ *carrier-mat* b n
and i: $i<a+b$ **and** j: $j < n$
shows $(A$ @$_r$ $B)$ $$ $(i, j) =$ (*if* $i <$ *dim-row* A *then* A $$$(i,j)$ *else* $B$$$(i-a,j))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *append-rows-split*:
assumes k: $k \leq$ *dim-row* A
shows $A =$ (*mat-of-rows* (*dim-col* A) [*Matrix.row* A i. $i \leftarrow$ [*0*..<k]]) @$_r$
(*mat-of-rows* (*dim-col* A) [*Matrix.row* A i. $i \leftarrow$ [k..<*dim-row* A]]) (**is** *?lhs = ?A1* @$_r$ *?A2*)
⟨*proof*⟩

lemma *transpose-mat-append-rows*:
assumes A: $A \in$ *carrier-mat* a n **and** B: $B \in$ *carrier-mat* b n

shows $(A\ @_r\ B)^T = A^T\ @_c\ B^T$
⟨*proof*⟩

lemma *transpose-mat-append-cols*:
assumes *A*: $A \in$ *carrier-mat n a* **and** *B*: $B \in$ *carrier-mat n b*
shows $(A\ @_c\ B)^T = A^T\ @_r\ B^T$
⟨*proof*⟩

lemma *append-rows-mult-right*:
assumes *A*: $(A::'a::$*comm-semiring-1 mat*$) \in$ *carrier-mat a n* **and** *B*: $B \in$ *carrier-mat b n*
and *Q*: $Q \in$ *carrier-mat n n*
shows $(A\ @_r\ B) * Q = (A * Q)\ @_r\ (B*Q)$
⟨*proof*⟩

lemma *append-rows-mult-left-id*:
assumes *A*: $(A::'a::$*comm-semiring-1 mat*$) \in$ *carrier-mat 1 n*
and *B*: $B \in$ *carrier-mat* $(m-1)$ *n*
and *C*: $C =$ *four-block-mat* $(1_m\ 1)\ (0_m\ 1\ (m-1))\ (0_m\ (m-1)\ 1)\ D$
and *D*: $D \in$ *carrier-mat* $(m-1)\ (m-1)$
shows $C * (A\ @_r\ B) = A\ @_r\ (D * B)$
⟨*proof*⟩

lemma *append-rows-mult-left-id2*:
assumes *A*: $(A::'a::$*comm-semiring-1 mat*$) \in$ *carrier-mat a n*
and *B*: $B \in$ *carrier-mat b n*
and *C*: $C =$ *four-block-mat D* $(0_m\ a\ b)\ (0_m\ b\ a)\ (1_m\ b)$
and *D*: $D \in$ *carrier-mat a a*
shows $C * (A\ @_r\ B) = (D * A)\ @_r\ B$
⟨*proof*⟩

lemma *four-block-mat-preserves-column*:
assumes *A*: $(A::'a::$*semiring-1 mat*$) \in$ *carrier-mat n m*
and *B*: $B =$ *four-block-mat* $(1_m\ 1)\ (0_m\ 1\ (m-1))\ (0_m\ (m-1)\ 1)\ C$
and *C*: $C \in$ *carrier-mat* $(m-1)\ (m-1)$
and *i*: $i<n$ **and** *m*: $0<m$
shows $(A*B)$ \$\$ $(i,0) = A$ \$\$ $(i,0)$
⟨*proof*⟩

definition *lower-triangular A* $= (\forall\, i\ j.\ i < j \wedge i <$ *dim-row A* $\wedge\ j <$ *dim-col A* $\longrightarrow A$ \$\$ $(i,j) = 0)$

lemma *lower-triangular-index*:
assumes *lower-triangular A* $i<j$ $i <$ *dim-row A* $j <$ *dim-col A*
shows *A* \$\$ $(i,j) = 0$
⟨*proof*⟩

lemma *commute-multiples-identity*:

assumes *A*: (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat n n*
shows *A* ∗ (*k* ·$_m$ (*1*$_m$ *n*)) = (*k* ·$_m$ (*1*$_m$ *n*)) ∗ *A*
⟨*proof*⟩

lemma *det-2*:
assumes *A*: *A* ∈ *carrier-mat 2 2*
shows *Determinant.det A* = *A*$$(*0*,*0*) ∗ *A* $$ (*1*,*1*) − *A*$$(*0*,*1*)∗*A*$$(*1*,*0*)
⟨*proof*⟩

lemma *mat-diag-smult*: *mat-diag n* (λ *x*. (*k*::'*a*::*comm-ring-1*)) = (*k* ·$_m$ *1*$_m$ *n*)
⟨*proof*⟩

lemma *invertible-mat-four-block-mat-lower-right*:
assumes *A*: (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat n n* **and** *inv-A*: *invertible-mat A*
shows *invertible-mat* (*four-block-mat* (*1*$_m$ *1*) (*0*$_m$ *1 n*) (*0*$_m$ *n 1*) *A*)
⟨*proof*⟩

lemma *invertible-mat-four-block-mat-lower-right-id*:
assumes *A*: (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat m m* **and** *B*: *B* = *0*$_m$ *m* (*n*−*m*) **and** *C*: *C* = *0*$_m$ (*n*−*m*) *m*
and *D*: *D* = *1*$_m$ (*n*−*m*) **and** *n*>*m* **and** *inv-A*: *invertible-mat A*
shows *invertible-mat* (*four-block-mat A B C D*)
⟨*proof*⟩

lemma *split-block4-decreases-dim-row*:
assumes *E*: (*A*,*B*,*C*,*D*) = *split-block E 1 1*
and *E1*: *dim-row E* > *1* **and** *E2*: *dim-col E* > *1*
shows *dim-row D* < *dim-row E*
⟨*proof*⟩

lemma *inv-P'PAQQ'*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *P*: *P* ∈ *carrier-mat n n*
and *inv-P*: *inverts-mat P' P*
and *inv-Q*: *inverts-mat Q Q'*
and *Q*: *Q* ∈ *carrier-mat n n*
and *P'*: *P'* ∈ *carrier-mat n n*
and *Q'*: *Q'* ∈ *carrier-mat n n*
shows (*P'*∗(*P*∗*A*∗*Q*)∗*Q'*) = *A*
⟨*proof*⟩

lemma
assumes *U* ∈ *carrier-mat 2 2* **and** *V* ∈ *carrier-mat 2 2* **and** *A* = *U* ∗ *V*
shows *mat-mult2-00*: *A* $$ (*0*,*0*) = *U* $$ (*0*,*0*)∗*V* $$ (*0*,*0*) + *U* $$ (*0*,*1*)∗*V* $$ (*1*,*0*)

and *mat-mult2-01*: $A\ \$\$\ (0,1) = U\ \$\$\ (0,0) * V\ \$\$\ (0,1) + U\ \$\$\ (0,1) * V\ \$\$\ (1,1)$
and *mat-mult2-10*: $A\ \$\$\ (1,0) = U\ \$\$\ (1,0) * V\ \$\$\ (0,0) + U\ \$\$\ (1,1) * V\ \$\$\ (1,0)$
and *mat-mult2-11*: $A\ \$\$\ (1,1) = U\ \$\$\ (1,0) * V\ \$\$\ (0,1) + U\ \$\$\ (1,1) * V\ \$\$\ (1,1)$
⟨*proof*⟩

4.5 Lemmas about *sorted lists*, *insort* and *pick*

lemma *sorted-distinct-imp-sorted-wrt*:
assumes *sorted xs* **and** *distinct xs*
shows *sorted-wrt* (<) *xs*
⟨*proof*⟩

lemma *sorted-map-strict*:
assumes *strict-mono-on* {*0*..<*n*} *g*
shows *sorted* (*map g* [*0*..<*n*])
⟨*proof*⟩

lemma *sorted-list-of-set-map-strict*:
assumes *strict-mono-on* {*0*..<*n*} *g*
shows *sorted-list-of-set* (*g* ‘ {*0*..<*n*}) = *map g* [*0*..<*n*]
⟨*proof*⟩

lemma *sorted-nth-strict-mono*:
sorted xs ⟹ *distinct xs* ⟹*i* < *j* ⟹ *j* < *length xs* ⟹ *xs*!*i* < *xs*!*j*
⟨*proof*⟩

lemma *sorted-list-of-set-0-LEAST*:
assumes *finI*: *finite I* **and** *I*: *I* ≠ {}
shows *sorted-list-of-set I* ! *0* = (*LEAST n*. *n*∈*I*)
⟨*proof*⟩

lemma *sorted-list-of-set-eq-pick*:
assumes *i*: *i* < *length* (*sorted-list-of-set I*)
shows *sorted-list-of-set I* ! *i* = *pick I i*
⟨*proof*⟩

b is the position where we add, *a* the element to be added and *i* the position that is checked

lemma *insort-nth'*:
assumes ∀*j*<*b*. *xs* ! *j* < *a* **and** *sorted xs* **and** *a* ∉ *set xs*
and *i* < *length xs* + *1* **and** *i* < *b*
and *xs* ≠ [] **and** *b* < *length xs*

shows *insort a xs* ! *i* = *xs* ! *i*
⟨*proof*⟩

lemma *insort-nth*:
assumes *sorted xs* **and** *a* ∉ *set xs*
and *i* < *index* (*insort a xs*) *a*
and *xs* ≠ []
shows *insort a xs* ! *i* = *xs* ! *i*
⟨*proof*⟩

lemma *insort-nth2*:
assumes *sorted xs* **and** *a* ∉ *set xs*
and *i* < *length xs* **and** *i* ≥ *index* (*insort a xs*) *a*
and *xs* ≠ []
shows *insort a xs* ! (*Suc i*) = *xs* ! *i*
⟨*proof*⟩

lemma *pick-index*:
assumes *a*: *a* ∈ *I* **and** *a'-card*: *a'* < *card I*
shows (*pick I a'* = *a*) = (*index* (*sorted-list-of-set I*) *a* = *a'*)
⟨*proof*⟩

end

5 The Cauchy–Binet formula

theory *Cauchy-Binet*
imports
Diagonal-To-Smith
SNF-Missing-Lemmas
begin

5.1 Previous missing results about *pick* and *insert*

lemma *pick-insert*:
assumes *a-notin-I*: *a* ∉ *I* **and** *i2*: *i* < *card I*
and *a-def*: *pick* (*insert a I*) *a'* = *a*
and *ia'*: *i* < *a'*
and *a'-card*: *a'* < *card I* + *1*
shows *pick* (*insert a I*) *i* = *pick I i*
⟨*proof*⟩

lemma *pick-insert2*:
assumes *a-notin-I*: *a* ∉ *I* **and** *i2*: *i* < *card I*
and *a-def*: *pick* (*insert a I*) *a'* = *a*
and *ia'*: *i* ≥ *a'*
and *a'-card*: *a'* < *card I* + *1*

shows *pick* (*insert a I*) *i* < *pick I i*
⟨*proof*⟩

lemma *pick-insert3*:
assumes *a-notin-I*: *a* ∉ *I* **and** *i2*: *i* < *card I*
and *a-def*: *pick* (*insert a I*) *a′* = *a*
and *ia′*: *i* ≥ *a′*
and *a′-card*: *a′* < *card I* + *1*
shows *pick* (*insert a I*) (*Suc i*) = *pick I i*
⟨*proof*⟩

lemma *pick-insert-index*:
assumes *Ik*: *card I* = *k*
and *a-notin-I*: *a* ∉ *I*
and *ik*: *i* < *k*
and *a-def*: *pick* (*insert a I*) *a′* = *a*
and *a′k*: *a′* < *card I* + *1*
shows *pick* (*insert a I*) (*insert-index a′ i*) = *pick I i*
⟨*proof*⟩

5.2 Start of the proof

definition *strict-from-inj n f* = (λ*i*. *if i*∈{*0*..<*n*} *then* (*sorted-list-of-set* (*f*‘{*0*..<*n*}))
! *i else i*)

lemma *strict-strict-from-inj*:
fixes *f*::*nat* ⇒ *nat*
assumes *inj-on f* {*0*..<*n*} **shows** *strict-mono-on* {*0*..<*n*} (*strict-from-inj n f*)
⟨*proof*⟩

lemma *strict-from-inj-image′*:
assumes *f*: *inj-on f* {*0*..<*n*}
shows *strict-from-inj n f* ‘ {*0*..<*n*} = *f*‘{*0*..<*n*}
⟨*proof*⟩

definition *Z* (*n*::*nat*) (*m*::*nat*) = {(*f*,*π*)|*f* *π*. *f* ∈ {*0*..<*n*} → {*0*..<*m*}
∧ (∀ *i*. *i* ∉ {*0*..<*n*} ⟶ *f i* = *i*)
∧ *π permutes* {*0*..<*n*}}

lemma *Z-alt-def*: *Z n m* = {*f*. *f* ∈ {*0*..<*n*} → {*0*..<*m*} ∧ (∀ *i*. *i* ∉ {*0*..<*n*} ⟶
f i = *i*)} × {*π*. *π permutes* {*0*..<*n*}}
⟨*proof*⟩

lemma *det-mul-finsum-alt*:
assumes *A*: $A \in$ *carrier-mat n m*
and *B*: $B \in$ *carrier-mat m n*
shows *det* $(A*B) =$ *det* $(mat_r\ n\ n\ (\lambda i.$ *finsum-vec TYPE*$('a::$*comm-ring-1*$)\ n$
$(\lambda k.\ B\ \$\$\ (k,\ i)\ \cdot_v$ *Matrix.col A k*$)\ \{0..<m\}))$
⟨*proof*⟩

lemma *det-cols-mul*:
assumes *A*: $A \in$ *carrier-mat n m*
and *B*: $B \in$ *carrier-mat m n*
shows *det* $(A*B) = (\sum f \mid (\forall i{\in}\{0..<n\}.\ f\ i \in \{0..<m\}) \wedge (\forall i.\ i \notin \{0..<n\}$
$\longrightarrow f\ i = i).$
$(\prod i = 0..<n.\ B\ \$\$\ (f\ i,\ i)) *$ *Determinant.det* $(mat_r\ n\ n\ (\lambda i.$ *col A* $(f\ i))))$
⟨*proof*⟩

lemma *det-cols-mul'*:
assumes *A*: $A \in$ *carrier-mat n m*
and *B*: $B \in$ *carrier-mat m n*
shows *det* $(A*B) = (\sum f \mid (\forall i{\in}\{0..<n\}.\ f\ i \in \{0..<m\}) \wedge (\forall i.\ i \notin \{0..<n\}$
$\longrightarrow f\ i = i).$
$(\prod i = 0..<n.\ A\ \$\$\ (i,\ f\ i)) *$ *det* $(mat_r\ n\ n\ (\lambda i.$ *row B* $(f\ i))))$
⟨*proof*⟩

lemma
assumes *F*: $F = \{f.\ f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)\}$
and *p*: π *permutes* $\{0..<n\}$
shows $(\sum f{\in}F.\ (\prod i = 0..<n.\ B\ \$\$\ (f\ i,\ \pi\ i))) = (\sum f{\in}F.\ (\prod i = 0..<n.\ B\ \$\$$
$(f\ i,\ i)))$
⟨*proof*⟩

lemma *detAB-Znm-aux*:
assumes *F*: $F = \{f.\ f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)\}$
shows$(\sum \pi \mid \pi$ *permutes* $\{0..<n\}.\ (\sum f{\in}F.$ *prod* $(\lambda i.\ B\ \$\$\ (f\ i,\ i))\ \{0..<n\}$
$*$ (*signof* $\pi *$ $(\prod i = 0..<n.\ A\ \$\$\ (\pi\ i,\ f\ i)))))$
$= (\sum \pi \mid \pi$ *permutes* $\{0..<n\}.\ \sum f{\in}F.\ (\prod i = 0..<n.\ B\ \$\$\ (f\ i,\ \pi\ i))$
$*$ (*signof* $\pi *$ $(\prod i = 0..<n.\ A\ \$\$\ (i,\ f\ i))))$
⟨*proof*⟩

lemma *detAB-Znm*:
assumes *A*: $A \in$ *carrier-mat n m*
and *B*: $B \in$ *carrier-mat m n*
shows *det* $(A*B) = (\sum (f,\ \pi){\in}Z\ n\ m.$ *signof* $\pi *$ $(\prod i = 0..<n.\ A\ \$\$\ (i,\ f\ i) *$
$B\ \$\$\ (f\ i,\ \pi\ i)))$
⟨*proof*⟩

context
  fixes n m **and** A B::'a::*comm-ring-1 mat*
  assumes A: $A \in$ *carrier-mat n m*
    and B: $B \in$ *carrier-mat m n*
begin

private definition *Z-inj* = ({f. $f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$
  $\wedge$ *inj-on f* $\{0..<n\}$} $\times$ {π. π *permutes* $\{0..<n\}$})

private definition *Z-not-inj* = ({f. $f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$
  $\wedge \neg$ *inj-on f* $\{0..<n\}$} $\times$ {π. π *permutes* $\{0..<n\}$})

private definition *Z-strict* = ({f. $f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$
  $\wedge$ *strict-mono-on* $\{0..<n\}$ f} $\times$ {π. π *permutes* $\{0..<n\}$})

private definition *Z-not-strict* = ({f. $f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$
  $\wedge \neg$ *strict-mono-on* $\{0..<n\}$ f} $\times$ {π. π *permutes* $\{0..<n\}$})

private definition *weight f* π
  = (*signof* π) $*$ (*prod* ($\lambda i.$ *A*$$(*i*,*f i*) $*$ *B* $$ (*f i*, π *i*)) $\{0..<n\}$)

private definition *Z-good g* = ({f. $f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$
  $\wedge$ *inj-on f* $\{0..<n\} \wedge (f`\{0..<n\} = g`\{0..<n\})$} $\times$ {π. π *permutes* $\{0..<n\}$})

private definition *F-strict* = {f. $f \in \{0..<n\} \to \{0..<m\}$
  $\wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i) \wedge$ *strict-mono-on* $\{0..<n\}$ f}

private definition *F-inj* = {f. $f \in \{0..<n\} \to \{0..<m\}$
  $\wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i) \wedge$ *inj-on f* $\{0..<n\}$}

private definition *F-not-inj* = {f. $f \in \{0..<n\} \to \{0..<m\}$
  $\wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i) \wedge \neg$ *inj-on f* $\{0..<n\}$}

private definition F = {f. $f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$}

The Cauchy–Binet formula is proven in https://core.ac.uk/download/pdf/82475020.pdf In that work, they define $\sigma \equiv inv\ \varphi \circ \pi$. I had problems following this proof in Isabelle, since I was demanded to show that such permutations commute, which is false. It is a notation problem of the $\circ$ operator, the author means $\sigma \equiv \pi \circ inv\ \varphi$ using the Isabelle notation (i.e., $\sigma\ x = \pi\ ((inv\ \varphi)\ x)$).

lemma *step-weight*:

fixes φ π
defines $\sigma \equiv \pi \circ$ *Hilbert-Choice.inv* φ
assumes *f-inj*: $f \in$ *F-inj* **and** *gF*: $g \in F$ **and** *pi*: π *permutes* $\{0..<n\}$
and *phi*: φ *permutes* $\{0..<n\}$ **and** *fg-phi*: $\forall x \in \{0..<n\}. f\ x = g\ (\varphi\ x)$
shows *weight* $f\ \pi$ = (*signof* φ) $*$ ($\prod i = 0..<n.$ A \$\$ $(i, g\ (\varphi\ i))$)
$*$ (*signof* σ) $*$ ($\prod i = 0..<n.$ B \$\$ $(g\ i, \sigma\ i)$)
⟨*proof*⟩

lemma *Z-good-fun-alt-sum*:
fixes g
defines *Z-good-fun* $\equiv \{f.\ f \in \{0..<n\} \to \{0..<m\} \wedge (\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$
$\wedge$ *inj-on* f $\{0..<n\} \wedge (f`\{0..<n\} = g`\{0..<n\})\}$
assumes g: $g \in$ *F-inj*
shows $(\sum f \in$ *Z-good-fun*. $P\ f) = (\sum \pi \in \{\pi.\ \pi$ *permutes* $\{0..<n\}\}.\ P\ (g \circ \pi))$
⟨*proof*⟩

lemma *F-injI*:
assumes $f \in \{0..<n\} \to \{0..<m\}$
and $(\forall i.\ i \notin \{0..<n\} \longrightarrow f\ i = i)$ **and** *inj-on* f $\{0..<n\}$
shows $f \in$ *F-inj* ⟨*proof*⟩

lemma *F-inj-composition-permutation*:
assumes *phi*: φ *permutes* $\{0..<n\}$
and g: $g \in$ *F-inj*
shows $g \circ \varphi \in$ *F-inj*
⟨*proof*⟩

lemma *F-strict-imp-F-inj*:
assumes f: $f \in$ *F-strict*
shows $f \in$ *F-inj*
⟨*proof*⟩

lemma *one-step*:
assumes *g1*: $g \in$ *F-strict*
shows *det* (*submatrix* A *UNIV* $(g`\{0..<n\}))$ $*$ *det* (*submatrix* $B$ $(g`\{0..<n\})$ *UNIV*)
$= (\sum (x, y) \in$ *Z-good* g. *weight* $x\ y)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *gather-by-strictness*:
sum $(\lambda g.$ *sum* $(\lambda(f,\pi).$ *weight* $f\ \pi)$ (*Z-good* $g))$ *F-strict*
= *sum* $(\lambda g.$ *det* (*submatrix* A *UNIV* $(g`\{0..<n\}))$ $*$ *det* (*submatrix* $B$ $(g`\{0..<n\})$ *UNIV*)) *F-strict*

⟨*proof*⟩

lemma *finite-Z-strict*[*simp*]: *finite Z-strict*
⟨*proof*⟩

lemma *finite-Z-not-strict*[*simp*]: *finite Z-not-strict*
⟨*proof*⟩

lemma *finite-Znm*[*simp*]: *finite* (*Z n m*)
⟨*proof*⟩

lemma *finite-F-inj*[*simp*]: *finite F-inj*
⟨*proof*⟩

lemma *finite-F-strict*[*simp*]: *finite F-strict*
⟨*proof*⟩

lemma *nth-strict-mono*:
fixes *f*::*nat* ⇒ *nat*
assumes *strictf*: *strict-mono f* **and** *i*: *i*<*n*
shows *f i* = (*sorted-list-of-set* (*f*‘{*0*..<*n*})) ! *i*
⟨*proof*⟩

lemma *nth-strict-mono-on*:
fixes *f*::*nat* ⇒ *nat*
assumes *strictf*: *strict-mono-on* {*0*..<*n*} *f* **and** *i*: *i*<*n*
shows *f i* = (*sorted-list-of-set* (*f*‘{*0*..<*n*})) ! *i*
⟨*proof*⟩

lemma *strict-fun-eq*:
assumes *f*: *f* ∈ *F-strict* **and** *g*: *g* ∈ *F-strict* **and** *fg*: *f*‘{*0*..<*n*} = *g*‘{*0*..<*n*}
shows *f* = *g*
⟨*proof*⟩

lemma *strict-from-inj-preserves-F*:
assumes *f*: *f* ∈ *F-inj*
shows *strict-from-inj n f* ∈ *F*
⟨*proof*⟩

lemma *strict-from-inj-F-strict*: *strict-from-inj n xa* ∈ *F-strict*
if *xa*: *xa* ∈ *F-inj* **for** *xa*
⟨*proof*⟩

lemma *strict-from-inj-image*:
assumes *f*: *f*∈ *F-inj*
shows *strict-from-inj n f* ‘ {*0*..<*n*} = *f*‘{*0*..<*n*}
⟨*proof*⟩

lemma *Z-good-alt*:
assumes *g*: $g \in$ *F-strict*
shows *Z-good* $g = \{x \in$ *F-inj. strict-from-inj* $n\ x = g\} \times \{\pi.\ \pi$ *permutes* $\{0..<n\}\}$
⟨*proof*⟩

lemma *weight-0*: $(\sum (f,\ \pi) \in$ *Z-not-inj. weight* $f\ \pi) = 0$
⟨*proof*⟩

5.3 Final theorem

lemma *Cauchy-Binet1*:
shows *det* $(A*B) =$
sum $(\lambda f.$ *det* (*submatrix A UNIV* (f‘$\{0..<n\}$)) $*$ *det* (*submatrix B* (f‘$\{0..<n\}$) *UNIV*)) *F-strict*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Cauchy-Binet*:
det $(A*B) = (\sum I \in \{I.\ I \subseteq \{0..<m\} \wedge$ *card* $I=n\}.$ *det* (*submatrix A UNIV I*) $*$ *det* (*submatrix B I UNIV*))
⟨*proof*⟩
end

end

6 Definition of Smith normal form in JNF

theory *Smith-Normal-Form-JNF*
imports
SNF-Missing-Lemmas
begin

Now, we define diagonal matrices and Smith normal form in JNF

definition *isDiagonal-mat* $A = (\forall\, i\ j.\ i \neq j \wedge i <$ *dim-row* $A \wedge j <$ *dim-col* $A \longrightarrow$ *A*$$(*i*,*j*) = 0$)

definition *Smith-normal-form-mat* $A =$
(
$(\forall\, a.\ a + 1 <$ *min* (*dim-row A*) (*dim-col A*) $\longrightarrow$ *A* $$ (*a*,*a*) *dvd A* $$ (*a*+*1*,*a*+*1*))
$\wedge$ *isDiagonal-mat A*
)

lemma *SNF-first-divides*:
assumes *SNF-A*: *Smith-normal-form-mat A* **and** (*A*::(′*a*::*comm-ring-1*) *mat*) $\in$ *carrier-mat n m*
and *i*: $i <$ *min* (*dim-row A*) (*dim-col A*)

shows *A* $$ (*0*,*0*) *dvd A* $$ (*i*,*i*)
⟨*proof*⟩

lemma *Smith-normal-form-mat-intro*:
assumes (∀ *a*. *a* + *1* < *min* (*dim-row A*) (*dim-col A*) ⟶ *A* $$ (*a*,*a*) *dvd A* $$ (*a+1*,*a+1*))
and *isDiagonal-mat A*
shows *Smith-normal-form-mat A*
⟨*proof*⟩

lemma *Smith-normal-form-mat-m0*[*simp*]:
assumes *A*: *A*∈*carrier-mat m 0*
shows *Smith-normal-form-mat A*
⟨*proof*⟩

lemma *Smith-normal-form-mat-0m*[*simp*]:
assumes *A*: *A*∈*carrier-mat 0 m*
shows *Smith-normal-form-mat A*
⟨*proof*⟩

lemma *S00-dvd-all-A*:
assumes *A*: (*A*::*'a*::*comm-ring-1 mat*) ∈ *carrier-mat m n*
and *P*: *P* ∈ *carrier-mat m m*
and *Q*: *Q* ∈ *carrier-mat n n*
and *inv-P*: *invertible-mat P*
and *inv-Q*: *invertible-mat Q*
and *S-PAQ*: *S* = *P*∗*A*∗*Q*
and *SNF-S*: *Smith-normal-form-mat S*
and *i*: *i*<*m* **and** *j*: *j*<*n*
shows *S*$$(*0*,*0*) *dvd A* $$ (*i*,*j*)
⟨*proof*⟩

lemma *SNF-first-divides-all*:
assumes *SNF-A*: *Smith-normal-form-mat A* **and** *A*: (*A*::(*'a*::*comm-ring-1*) *mat*) ∈ *carrier-mat m n*
and *i*: *i* < *m* **and** *j*: *j*<*n*
shows *A* $$ (*0*,*0*) *dvd A* $$ (*i*,*j*)
⟨*proof*⟩

lemma *SNF-divides-diagonal*:
fixes *A*::*'a*::*comm-ring-1 mat*
assumes *A*: *A* ∈ *carrier-mat n m*
and *SNF-A*: *Smith-normal-form-mat A*
and *j*: *j* < *min n m*
and *ij*: *i*≤*j*
shows *A*$$(*i*,*i*) *dvd A*$$(*j*,*j*)
⟨*proof*⟩

lemma *Smith-zero-imp-zero*:
fixes *A*::*'a*::*comm-ring-1 mat*
assumes *A*: *A* ∈ *carrier-mat m n*
and *SNF*: *Smith-normal-form-mat A*
and *Aii*: *A$$(i,i) = 0*
and *j*: *j*<*min m n*
and *ij*: *i*≤*j*
shows *A$$(j,j) = 0*
⟨*proof*⟩

lemma *SNF-preserved-multiples-identity*:
assumes *S*: *S* ∈ *carrier-mat m n* **and** *SNF*: *Smith-normal-form-mat* (*S*::*'a*::*comm-ring-1 mat*)
shows *Smith-normal-form-mat* (*S*∗(*k* ·$_m$ *1*$_m$ *n*))
⟨*proof*⟩

end

7 Some theorems about rings and ideals

theory *Rings2-Extended*
imports
Echelon-Form.Rings2
HOL−Types-To-Sets.Types-To-Sets
begin

7.1 Missing properties on ideals

lemma *ideal-generated-subset2*:
assumes ∀ *b*∈*B*. *b* ∈ *ideal-generated A*
shows *ideal-generated B* ⊆ *ideal-generated A*
⟨*proof*⟩

context *comm-ring-1*
begin

lemma *ideal-explicit*: *ideal-generated S*
= {*y*. ∃*f U*. *finite U* ∧ *U* ⊆ *S* ∧ (∑ *i*∈*U*. *f i* ∗ *i*) = *y*}
⟨*proof*⟩
end

lemma *ideal-generated-minus*:
assumes *a*: *a* ∈ *ideal-generated* (*S*−{*a*})
shows *ideal-generated S* = *ideal-generated* (*S*−{*a*})
⟨*proof*⟩

lemma *ideal-generated-dvd-eq*:
assumes *a-dvd-b*: *a dvd b*

and *a*: $a \in S$
and *a-not-b*: $a \neq b$
shows *ideal-generated* S = *ideal-generated* $(S - \{b\})$
⟨*proof*⟩

lemma *ideal-generated-dvd-eq-diff-set*:
assumes *i-in-I*: $i \in I$ **and** *i-in-J*: $i \notin J$ **and** *i-dvd-j*: $\forall j \in J.$ *i dvd j*
and *f*: *finite J*
shows *ideal-generated I* = *ideal-generated* $(I - J)$
⟨*proof*⟩

context *comm-ring-1*
begin

lemma *ideal-generated-singleton-subset*:
assumes *d*: $d \in$ *ideal-generated S* **and** *fin-S*: *finite S*
shows *ideal-generated* $\{d\} \subseteq$ *ideal-generated S*
⟨*proof*⟩

lemma *ideal-generated-singleton-dvd*:
assumes *i*: *ideal-generated S* = *ideal-generated* $\{d\}$ **and** *x*: $x \in S$
shows *d dvd x*
⟨*proof*⟩

lemma *ideal-generated-UNIV-insert*:
assumes *ideal-generated S* = *UNIV*
shows *ideal-generated* (*insert a S*) = *UNIV* ⟨*proof*⟩

lemma *ideal-generated-UNIV-union*:
assumes *ideal-generated S* = *UNIV*
shows *ideal-generated* $(A \cup S)$ = *UNIV*
⟨*proof*⟩

lemma *ideal-explicit2*:
assumes *finite S*
shows *ideal-generated S* = $\{y.\ \exists f.\ (\sum i \in S.\ f\ i * i) = y\}$
⟨*proof*⟩

lemma *ideal-generated-unit*:
assumes *u*: *u dvd 1*
shows *ideal-generated* $\{u\}$ = *UNIV*
⟨*proof*⟩

lemma *ideal-generated-dvd-subset*:
assumes *x*: $\forall x \in S.$ *d dvd x* **and** *S*: *finite S*
shows *ideal-generated S* $\subseteq$ *ideal-generated* $\{d\}$
⟨*proof*⟩

lemma *ideal-generated-mult-unit*:
assumes *f*: *finite S* **and** *u*: *u dvd 1*
shows *ideal-generated* ((λ*x*. *u*∗*x*)‘ *S*) = *ideal-generated S*
⟨*proof*⟩

corollary *ideal-generated-mult-unit2*:
assumes *u*: *u dvd 1*
shows *ideal-generated* {*u*∗*a*,*u*∗*b*} = *ideal-generated* {*a*,*b*}
⟨*proof*⟩

lemma *ideal-generated-1*[*simp*]: *ideal-generated* {*1*} = *UNIV*
⟨*proof*⟩

lemma *ideal-generated-pair*: *ideal-generated* {*a*,*b*} = {*p*∗*a*+*q*∗*b* | *p q*. *True*}
⟨*proof*⟩

lemma *ideal-generated-pair-exists-pq1*:
assumes *i*: *ideal-generated* {*a*,*b*} = (*UNIV*::*'a set*)
shows ∃ *p q*. *p*∗*a* + *q*∗*b* = *1*
⟨*proof*⟩

lemma *ideal-generated-pair-UNIV*:
assumes *sa-tb-u*: *s*∗*a*+*t*∗*b* = *u* **and** *u*: *u dvd 1*
shows *ideal-generated* {*a*,*b*} = *UNIV*
⟨*proof*⟩

lemma *ideal-generated-pair-exists*:
assumes *l*: (*ideal-generated* {*a*,*b*} = *ideal-generated* {*d*})
shows (∃ *p q*. *p*∗*a*+*q*∗*b* = *d*)
⟨*proof*⟩

lemma *obtain-ideal-generated-pair*:
assumes *c* ∈ *ideal-generated* {*a*,*b*}
obtains *p q* **where** *p*∗*a*+*q*∗*b*=*c*
⟨*proof*⟩

lemma *ideal-generated-pair-exists-UNIV*:
shows (*ideal-generated* {*a*,*b*} = *ideal-generated* {*1*}) = (∃ *p q*. *p*∗*a*+*q*∗*b* = *1*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

corollary *ideal-generated-UNIV-obtain-pair*:
assumes *ideal-generated* {*a*,*b*} = *ideal-generated* {*1*}
shows (∃ *p q*. *p*∗*a*+*q*∗*b* = *d*)
⟨*proof*⟩

lemma *sum-three-elements*:
shows $\exists x\ y\ z{::}'a.\ (\sum i{\in}\{a,b,c\}.\ f\ i * i) = x * a + y * b + z * c$
⟨*proof*⟩

lemma *sum-three-elements'*:
shows $\exists f{::}'a{\Rightarrow}'a.\ (\sum i{\in}\{a,b,c\}.\ f\ i * i) = x * a + y * b + z * c$
⟨*proof*⟩

lemma *ideal-generated-triple-pair-rewrite*:
assumes *i1*: *ideal-generated* $\{a,\ b,\ c\}$ = *ideal-generated* $\{d\}$
and *i2*: *ideal-generated* $\{a,\ b\}$ = *ideal-generated* $\{d'\}$
shows *ideal-generated*$\{d',c\}$ = *ideal-generated* $\{d\}$
⟨*proof*⟩

lemma *ideal-generated-dvd*:
assumes *i*: *ideal-generated* $\{a,b{::}'a\}$ = *ideal-generated*$\{d\}$
and *a*: d' *dvd a* **and** *b*: d' *dvd b*
shows d' *dvd d*
⟨*proof*⟩

lemma *ideal-generated-dvd2*:
assumes *i*: *ideal-generated S* = *ideal-generated*$\{d{::}'a\}$
and *finite S*
and *x*: $\forall x{\in}S.\ d'$ *dvd x*
shows d' *dvd d*
⟨*proof*⟩

end

7.2 An equivalent characterization of Bézout rings

The goal of this subsection is to prove that a ring is Bézout ring if and only if every finitely generated ideal is principal.

definition *finitely-generated-ideal I* = (*ideal I* ∧ (∃ *S*. *finite S* ∧ *ideal-generated S* = *I*))

context
assumes *SORT-CONSTRAINT*('*a*::*comm-ring-1*)
begin

lemma *sum-two-elements'*:
fixes $d{::}'a$
assumes *s*: $(\sum i{\in}\{a,b\}.\ f\ i * i) = d$

obtains p **and** q **where** $d = p * a + q * b$
⟨*proof*⟩

This proof follows Theorem 6-3 in "First Course in Rings and Ideals" by Burton

lemma *all-fin-gen-ideals-are-principal-imp-bezout*:
assumes *all*: $\forall I::'a\ set.\ finitely\text{-}generated\text{-}ideal\ I \longrightarrow principal\text{-}ideal\ I$
shows *OFCLASS* ($'a$, *bezout-ring-class*)
⟨*proof*⟩
end

context *bezout-ring*
begin

lemma *exists-bezout-extended*:
assumes *S*: *finite S* **and** *ne*: $S \neq \{\}$
shows $\exists f\ d.\ (\sum a{\in}S.\ f\ a * a) = d \wedge (\forall a{\in}S.\ d\ dvd\ a) \wedge (\forall d'.\ (\forall a{\in}S.\ d'\ dvd\ a) \longrightarrow d'\ dvd\ d)$
⟨*proof*⟩

end

lemma *ideal-generated-empty*: *ideal-generated* $\{\} = \{0\}$
⟨*proof*⟩

lemma *bezout-imp-all-fin-gen-ideals-are-principal*:
fixes $I::'a$ :: *bezout-ring set*
assumes *fin*: *finitely-generated-ideal I*
shows *principal-ideal I*
⟨*proof*⟩

Now we have the required lemmas to prove the theorem that states that a ring is Bézout ring if and only if every finitely generated ideal is principal. They are the following ones.

- *all-fin-gen-ideals-are-principal-imp-bezout*
- *bezout-imp-all-fin-gen-ideals-are-principal*

However, in order to prove the final lemma, we need the lemmas with no type restrictions. For instance, we need a version of theorem *bezout-imp-all-fin-gen-ideals-are-principal* as
OFCLASS($'a$,*bezout-ring*) $\Longrightarrow$ the theorem with generic types (i.e., $'a$ with no type restrictions)
or as

class.bezout-ring - - - - $\Longrightarrow$ the theorem with generic types (i.e., $'a$ with no type restrictions)

Thanks to local type definitions, we can obtain it automatically by means of *internalize−sort*.

lemma *bezout-imp-all-fin-gen-ideals-are-principal-unsatisfactory*:
assumes *a1*: *class.bezout-ring* $(*)$ $(1::'b::comm\text{-}ring\text{-}1)$ $(+)$ *0* $(-)$ *uminus*
shows $\forall I::'b\ set.$ *finitely-generated-ideal I* $\longrightarrow$ *principal-ideal I*
⟨*proof*⟩

The standard library does not connect *OFCLASS* and *class.bezout-ring* in both directions. Here we show that *OFCLASS* $\Longrightarrow$ *class.bezout-ring*.

lemma *OFCLASS-bezout-ring-imp-class-bezout-ring*:
assumes *OFCLASS*($'a::comm\text{-}ring\text{-}1$,*bezout-ring-class*)
shows *class.bezout-ring* $((*)::'a\Rightarrow'a\Rightarrow'a)$ *1* $(+)$ *0* $(-)$ *uminus*
⟨*proof*⟩

The other implication can be obtained by thm *bezout-ring.intro-of-class*

thm *bezout-ring.intro-of-class*

Final theorem (with OFCLASS)

lemma *bezout-ring-iff-fin-gen-principal-ideal*:
$(\bigwedge I::'a::comm\text{-}ring\text{-}1\ set.$ *finitely-generated-ideal I* $\Longrightarrow$ *principal-ideal I*$)$
$\equiv$ *OFCLASS*($'a$, *bezout-ring-class*)
⟨*proof*⟩

Final theorem (with *class.bezout-ring*)

lemma *bezout-ring-iff-fin-gen-principal-ideal2*:
$(\forall I::'a::comm\text{-}ring\text{-}1\ set.$ *finitely-generated-ideal I* $\longrightarrow$ *principal-ideal I*$)$
$=$ (*class.bezout-ring* $((*)::'a\Rightarrow'a\Rightarrow'a)$ *1* $(+)$ *0* $(-)$ *uminus*)
⟨*proof*⟩

end

8 Connection between *mod-ring* and *mod-type*

This file shows that the type *mod-ring*, which is defined in the Berlekamp–Zassenhaus development, is an instantiation of the type class *mod-type*.

theory *Finite-Field-Mod-Type-Connection*
imports
Berlekamp-Zassenhaus.Finite-Field
Rank-Nullity-Theorem.Mod-Type
begin

instantiation *mod-ring* :: (*finite*) *ord*
begin

definition *less-eq-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *bool*
where *less-eq-mod-ring x y* = (*to-int-mod-ring x* ≤ *to-int-mod-ring y*)

definition *less-mod-ring* :: *'a mod-ring* ⇒ *'a mod-ring* ⇒ *bool*
where *less-mod-ring x y* = (*to-int-mod-ring x* < *to-int-mod-ring y*)

instance ⟨*proof*⟩
end

instantiation *mod-ring* :: (*finite*) *linorder*
begin
instance ⟨*proof*⟩
end

instance *mod-ring* :: (*finite*) *wellorder*
⟨*proof*⟩

lemma *strict-mono-to-int-mod-ring*: *strict-mono to-int-mod-ring*
⟨*proof*⟩

instantiation *mod-ring* :: (*nontriv*) *mod-type*
begin
definition *Rep-mod-ring* :: *'a mod-ring* ⇒ *int*
where *Rep-mod-ring x* = *to-int-mod-ring x*

definition *Abs-mod-ring* :: *int* ⇒ *'a mod-ring*
where *Abs-mod-ring x* = *of-int-mod-ring x*

instance
⟨*proof*⟩
end
end

9 Generality of the Algorithm to transform from diagonal to Smith normal form

theory *Admits-SNF-From-Diagonal-Iff-Bezout-Ring*
imports
Diagonal-To-Smith
Rings2-Extended
Smith-Normal-Form-JNF
Finite-Field-Mod-Type-Connection
begin

hide-const (**open**) *mat*

This section provides a formal proof on the generality of the algorithm that transforms a diagonal matrix into its Smith normal form. More concretely, we prove that all diagonal matrices with coefficients in a ring R admit Smith normal form if and only if R is a Bézout ring.

Since our algorithm is defined for Bézout rings and for any matrices (including non-square and singular ones), this means that it does not exist another algorithm that performs the transformation in a more abstract structure.

Firstly, we hide some definitions and facts, since we are interested in the ones developed for the *mod-type* class.

hide-const (**open**) *Bij-Nat.to-nat Bij-Nat.from-nat Countable.to-nat Countable.from-nat*

hide-fact (**open**) *Bij-Nat.to-nat-from-nat-id Bij-Nat.to-nat-less-card*

definition *admits-SNF-HA* (A::$'a$::*comm-ring-1*$^{\sim}{}'n$::{*mod-type*}$^{\sim}{}'n$::{*mod-type*}) =
(*isDiagonal A*
$\longrightarrow$ ($\exists$ *P Q*. *invertible* ((*P*::$'a$::*comm-ring-1*$^{\sim}{}'n$::{*mod-type*}$^{\sim}{}'n$::{*mod-type*}))
$\wedge$ *invertible* (*Q*::$'a$::*comm-ring-1*$^{\sim}{}'n$::{*mod-type*}$^{\sim}{}'n$::{*mod-type*}) $\wedge$ *Smith-normal-form*
(*P****A****Q*)))

definition *admits-SNF-JNF A* = (*square-mat* (*A*::$'a$::*comm-ring-1 mat*) $\wedge$ *isDiagonal-mat A*
$\longrightarrow$ ($\exists$ *P Q*. *P* $\in$ *carrier-mat* (*dim-row A*) (*dim-row A*) $\wedge$ *Q* $\in$ *carrier-mat*
(*dim-row A*) (*dim-row A*)
$\wedge$ *invertible-mat P* $\wedge$ *invertible-mat Q* $\wedge$ *Smith-normal-form-mat* (*P***A***Q*)))

9.1 Proof of the $\Longleftarrow$ implication in HA.

lemma *exists-f-PAQ-Aii'*:
fixes *A*::$'a$::{*comm-ring-1*}$^{\sim}{}'n$::{*mod-type*}$^{\sim}{}'n$::{*mod-type*}
assumes *diag-A*: *isDiagonal A*
shows $\exists f$. (*P****A****Q*) $h *i* $h *i* = ($\sum$ *i*$\in$(*UNIV*::$'n$ *set*). *f i* * *A* $h *i* $h *i*)
⟨*proof*⟩

We apply *internalize-sort* to the lemma that we need

lemmas *diagonal-to-Smith-PQ-exists-internalize-sort*
= *diagonal-to-Smith-PQ-exists*[*internalize-sort* $'a$ :: *bezout-ring*]

We get the $\Longleftarrow$ implication in HA.

lemma *bezout-ring-imp-diagonal-admits-SNF*:
assumes *of*: *OFCLASS*($'a$::*comm-ring-1*, *bezout-ring-class*)
shows $\forall$ *A*::$'a^{\sim}{}'n$::{*mod-type*}$^{\sim}{}'n$::{*mod-type*}. *isDiagonal A*
$\longrightarrow$ ($\exists$ *P Q*.
invertible (*P*::$'a^{\sim}{}'n$::*mod-type*$^{\sim}{}'n$::*mod-type*) $\wedge$
invertible (*Q*::$'a^{\sim}{}'n$::*mod-type*$^{\sim}{}'n$::*mod-type*) $\wedge$
Smith-normal-form (*P****A****Q*))
⟨*proof*⟩

9.2 Trying to prove the $\Longrightarrow$ implication in HA.

There is a problem: we need to define a matrix with a concrete dimension, which is not possible in HA (the dimension depends on the number of elements on a set, and Isabelle/HOL does not feature dependent types)

lemma
assumes $\forall A::'a::comm\text{-}ring\text{-}1\hat{}'n::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}.\ admits\text{-}SNF\text{-}HA\ A$
shows $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class)$ ⟨*proof*⟩

9.3 Proof of the $\Longrightarrow$ implication in JNF.

lemma *exists-f-PAQ-Aii*:
assumes *diag-A*: $isDiagonal\text{-}mat\ (A::'a::\ comm\text{-}ring\text{-}1\ mat)$
and *P*: $P \in carrier\text{-}mat\ n\ n$
and *A*: $A \in carrier\text{-}mat\ n\ n$
and *Q*: $Q \in carrier\text{-}mat\ n\ n$
and *i*: $i < n$

shows $\exists f.\ (P*A*Q)\ \$\$\ (i,\ i) = (\sum i{\in}set\ (diag\text{-}mat\ A).\ f\ i * i)$
⟨*proof*⟩

Proof of the $\Longrightarrow$ implication in JNF.

lemma *diagonal-admits-SNF-imp-bezout-ring-JNF*:
assumes *admits-SNF*: $\forall A\ n.\ (A::'a\ mat) \in carrier\text{-}mat\ n\ n \wedge isDiagonal\text{-}mat\ A \longrightarrow (\exists P\ Q.\ P \in carrier\text{-}mat\ n\ n \wedge Q \in carrier\text{-}mat\ n\ n \wedge invertible\text{-}mat\ P \wedge invertible\text{-}mat\ Q \wedge Smith\text{-}normal\text{-}form\text{-}mat\ (P*A*Q))$
shows $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class)$
⟨*proof*⟩

corollary *diagonal-admits-SNF-imp-bezout-ring-JNF-alt*:
assumes *admits-SNF*: $\forall A.\ square\text{-}mat\ (A::'a\ mat) \wedge isDiagonal\text{-}mat\ A \longrightarrow (\exists P\ Q.\ P \in carrier\text{-}mat\ (dim\text{-}row\ A)\ (dim\text{-}row\ A) \wedge Q \in carrier\text{-}mat\ (dim\text{-}row\ A)\ (dim\text{-}row\ A) \wedge invertible\text{-}mat\ P \wedge invertible\text{-}mat\ Q \wedge Smith\text{-}normal\text{-}form\text{-}mat\ (P*A*Q))$
shows $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class)$
⟨*proof*⟩

9.4 Trying to transfer the $\Longrightarrow$ implication to HA.

We first hide some constants defined in *Mod-Type-Connect* in order to use the ones presented in *Perron-Frobenius.HMA-Connect* by default.

context

includes *lifting-syntax*
begin

lemma *to-nat-mod-type-Bij-Nat*:
fixes *a*::*'n*::*mod-type*
obtains *b*::*'n* **where** *mod-type-class.to-nat a* = *Bij-Nat.to-nat b*
⟨*proof*⟩

lemma *inj-on-Bij-nat-from-nat*: *inj-on* (*Bij-Nat.from-nat*::*nat* ⇒ *'a*) {*0*..<*CARD*(*'a*::*finite*)}

⟨*proof*⟩

This lemma only holds if a and b have the same type. Otherwise, it is possible that *Bij-Nat.to-nat a* = *Bij-Nat.to-nat b*

lemma *Bij-Nat-to-nat-neq*:
fixes *a b* ::*'n*::*mod-type*
assumes *to-nat a* ≠ *to-nat b*
shows *Bij-Nat.to-nat a* ≠ *Bij-Nat.to-nat b*
⟨*proof*⟩

The following proof (a transfer rule for diagonal matrices) is weird, since it does not hold *Bij-Nat.to-nat a* = *mod-type-class.to-nat a*.

At first, it seems possible to obtain the element a' that satisfies *Bij-Nat.to-nat* a' = *mod-type-class.to-nat a* and then continue with the proof, but then we cannot prove *HMA-I* (*Bij-Nat.to-nat a'*) *a*.

This means that we must use the previous lemma *Bij-Nat-to-nat-neq*, but this imposes the matrix to be square.

lemma *HMA-isDiagonal*[*transfer-rule*]: (*HMA-M* ===> (=))
isDiagonal-mat (*isDiagonal*::(*'a*::{*zero*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} => *bool*))
⟨*proof*⟩

Indeed, we can prove the transfer rules with the new connection based on the *mod-type* class, which was developed in the *Mod-Type-Connect* file

This is the same lemma as the one presented above, but now using the *to-nat* function defined in the *mod-type* class and then we can prove it for non-square matrices, which is very useful since our algorithms are not restricted to square matrices.

lemma *HMA-isDiagonal-Mod-Type*[*transfer-rule*]: (*Mod-Type-Connect.HMA-M* ===> (=))
isDiagonal-mat (*isDiagonal*::(*'a*::{*zero*}^*'n*::{*mod-type*}^*'m*::{*mod-type*} => *bool*))
⟨*proof*⟩

We state the transfer rule using the relations developed in the new bride of the file *Mod-Type-Connect*.

lemma *HMA-SNF*[*transfer-rule*]: (*Mod-Type-Connect.HMA-M* ===> (=)) *Smith-normal-form-mat*

(*Smith-normal-form*::'*a*::{*comm-ring-1*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}⇒*bool*)
⟨*proof*⟩

lemma *HMA-admits-SNF* [*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ 'a :: *comm-ring-1* ^ 'n::{*mod-type*} ^ 'n::{*mod-type*}
⇒ -) ===> (=))
admits-SNF-JNF admits-SNF-HA
⟨*proof*⟩
end

Here we have a problem when trying to apply local type definitions

lemma *diagonal-admits-SNF-imp-bezout-ring*:
assumes *admits-SNF*: ∀ *A*::'*a*::*comm-ring-1*^'*n*::{*mod-type*}^'*n*::{*mod-type*}. *isDiagonal A*
⟶ (∃ *P Q*. *invertible* (*P*::'*a*::*comm-ring-1*^'*n*::{*mod-type*}^'*n*::{*mod-type*})
∧ *invertible* (*Q*::'*a*::*comm-ring-1*^'*n*::{*mod-type*}^'*n*::{*mod-type*})
∧ *Smith-normal-form* (*P****A****Q*))
shows *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*)
⟨*proof*⟩

This means that the ⟹ implication cannot be proven in HA, since we cannot quantify over type variables in Isabelle/HOL. We then prove both implications in JNF.

9.5 Transfering the ⟸ implication from HA to JNF using transfer rules and local type definitions

lemma *bezout-ring-imp-diagonal-admits-SNF-mod-ring*:
assumes *of*: *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*)
shows ∀ *A*::'*a*^'*n*::*nontriv mod-ring*^'*n*::*nontriv mod-ring*. *isDiagonal A*
⟶ (∃ *P Q*.
invertible (*P*::'*a*^'*n*::*nontriv mod-ring*^'*n*::*nontriv mod-ring*) ∧
invertible (*Q*::'*a*^'*n*::*nontriv mod-ring*^'*n*::*nontriv mod-ring*) ∧
Smith-normal-form (*P****A****Q*))
⟨*proof*⟩

lemma *bezout-ring-imp-diagonal-admits-SNF-mod-ring-admits*:
assumes *of*: *class.bezout-ring* (∗) (*1*::'*a*::*comm-ring-1*) (+) *0* (−) *uminus*
shows ∀ *A*::'*a*^'*n*::*nontriv mod-ring*^'*n*::*nontriv mod-ring*. *admits-SNF-HA A*
⟨*proof*⟩

I start here to apply local type definitions

context
fixes *p*::*nat*
assumes *local-typedef*: ∃ (*Rep* :: ('*b* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*p* :: *int*}

and *p*: *p>1*
begin

lemma *type-to-set*:
shows *class.nontriv TYPE('b)* (**is** *?a*) **and** *p=CARD('b)* (**is** *?b*)
⟨*proof*⟩

I transfer the lemma from HA to JNF, substituting $CARD('n)$ by p. I apply *internalize−sort* to $'n$ and get rid of the *nontriv* restriction.

lemma *bezout-ring-imp-diagonal-admits-SNF-mod-ring-admits-aux*:
assumes *class.bezout-ring* (∗) (*1*::*'a*::*comm-ring-1*) (+) *0* (−) *uminus*
shows *Ball* {*A*::*'a*::*comm-ring-1 mat*. *A* ∈ *carrier-mat p p*} *admits-SNF-JNF*
⟨*proof*⟩
end

The ⟸ implication in JNF

Since *nontriv* imposes the type to have more than one element, the cases $n = 0$ ($A \in$ *carrier-mat 0 0*) and $n = 1$ ($A \in$ *carrier-mat 1 1*) must be treated separately.

lemma *bezout-ring-imp-diagonal-admits-SNF-mod-ring-admits-aux2*:
assumes *of*: *class.bezout-ring* (∗) (*1*::*'a*::*comm-ring-1*) (+) *0* (−) *uminus*
shows ∀ (*A*::*'a mat*)∈*carrier-mat n n*. *admits-SNF-JNF A*
⟨*proof*⟩

Alternative statements

lemma *bezout-ring-imp-diagonal-admits-SNF-JNF*:
assumes *of*: *class.bezout-ring* (∗) (*1*::*'a*::*comm-ring-1*) (+) *0* (−) *uminus*
shows ∀ *A*::*'a mat*. *admits-SNF-JNF A*
⟨*proof*⟩

lemma *admits-SNF-JNF-alt-def*:
(∀ *A*::*'a*::*comm-ring-1 mat*. *admits-SNF-JNF A*)
= (∀ *A n*. (*A*::*'a mat*) ∈ *carrier-mat n n* ∧ *isDiagonal-mat A*
⟶ (∃ *P Q*. *P* ∈ *carrier-mat n n* ∧ *Q* ∈ *carrier-mat n n* ∧ *invertible-mat P* ∧ *invertible-mat Q*
∧ *Smith-normal-form-mat* (*P*∗*A*∗*Q*))) (**is** *?a* = *?b*)
⟨*proof*⟩

9.6 Final theorem in JNF

Final theorem using *class.bezout-ring*

theorem *diagonal-admits-SNF-iff-bezout-ring*:
shows *class.bezout-ring* (∗) (*1*::*'a*::*comm-ring-1*) (+) *0* (−) *uminus*
⟷ (∀ *A*::*'a mat*. *admits-SNF-JNF A*) (**is** *?a* ⟷ *?b*)
⟨*proof*⟩

Final theorem using *OFCLASS*

theorem *diagonal-admits-SNF-iff-bezout-ring'*:
shows *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*) ≡ (⋀*A*::'*a mat*. *admits-SNF-JNF A*)
⟨*proof*⟩

end

10 Uniqueness of the Smith normal form

theory *SNF-Uniqueness*
imports
Cauchy-Binet
Smith-Normal-Form-JNF
Admits-SNF-From-Diagonal-Iff-Bezout-Ring
begin

lemma *dvd-associated1*:
fixes *a*::'*a*::*comm-ring-1*
assumes ∃ *u*. *u dvd 1* ∧ *a* = *u*∗*b*
shows *a dvd b* ∧ *b dvd a*
⟨*proof*⟩

This is a key lemma. It demands the type class to be an integral domain. This means that the uniqueness result will be obtained for GCD domains, instead of rings.

lemma *dvd-associated2*:
fixes *a*::'*a*::*idom*
assumes *ab*: *a dvd b* **and** *ba*: *b dvd a* **and** *a*: *a*≠*0*
shows ∃ *u*. *u dvd 1* ∧ *a* = *u*∗*b*
⟨*proof*⟩

corollary *dvd-associated*:
fixes *a*::'*a*::*idom*
assumes *a*≠*0*
shows (*a dvd b* ∧ *b dvd a*) = (∃ *u*. *u dvd 1* ∧ *a* = *u*∗*b*)
⟨*proof*⟩

lemma *exists-inj-ge-index*:
assumes *S*: *S* ⊆ {*0*..<*n*} **and** *Sk*: *card S* = *k*
shows ∃*f*. *inj-on f* {*0*..<*k*} ∧ *f*‘{*0*..<*k*} = *S* ∧ (∀ *i*∈{*0*..<*k*}. *i* ≤ *f i*)
⟨*proof*⟩

10.1 More specific results about submatrices

lemma *diagonal-imp-submatrix0*:
assumes *dA*: *diagonal-mat A* **and** *A-carrier*: *A*∈ *carrier-mat n m*

and *Ik*: *card I* = *k* **and** *Jk*: *card J* = *k*
and *r*: ∀ *row-index* ∈ *I*. *row-index* < *n*
and *c*: ∀ *col-index* ∈ *J*. *col-index* < *m*
and *a*: *a*<*k* **and** *b*: *b*<*k*
shows *submatrix A I J* $$ (*a*, *b*) = *0* ∨ *submatrix A I J* $$ (*a*,*b*) = *A* $$(*pick I a*, *pick I a*)
⟨*proof*⟩

lemma *diagonal-imp-submatrix-element-not0*:
assumes *dA*: *diagonal-mat A*
and *A-carrier*: *A* ∈ *carrier-mat n m*
and *Ik*: *card I* = *k* **and** *Jk*: *card J* = *k*
and *I*: *I* ⊆ {*0*..<*n*}
and *J*: *J* ⊆ {*0*..<*m*}
and *b*: *b* < *k*
and *ex-not0*: ∃ *i*. *i*<*k* ∧ *submatrix A I J* $$ (*i*, *b*) ≠ *0*
shows ∃!*i*. *i*<*k* ∧ *submatrix A I J* $$ (*i*, *b*) ≠ *0*
⟨*proof*⟩

lemma *submatrix-index-exists*:
assumes *A-carrier*: *A*∈ *carrier-mat n m*
and *Ik*: *card I* = *k* **and** *Jk*: *card J* = *k*
and *a*: *a* ∈ *I* **and** *b*: *b* ∈ *J* **and** *k*: *k* > *0*
and *I*: *I* ⊆ {*0*..<*n*} **and** *J*: *J* ⊆ {*0*..<*m*}
shows ∃ *a′ b′*. *a′* < *k* ∧ *b′* < *k* ∧ *submatrix A I J* $$ (*a′*,*b′*) = *A* $$ (*a*,*b*)
∧ *a* = *pick I a′* ∧ *b* = *pick J b′*
⟨*proof*⟩

lemma *mat-delete-submatrix-insert*:
assumes *A-carrier*: *A* ∈ *carrier-mat n m*
and *Ik*: *card I* = *k* **and** *Jk*: *card J* = *k*
and *I*: *I* ⊆ {*0*..<*n*} **and** *J*: *J* ⊆ {*0*..<*m*}
and *a*: *a* < *n* **and** *b*: *b* < *m*
and *k*: *k* < *min n m*
and *a-notin-I*: *a* ∉ *I* **and** *b-notin-J*: *b* ∉ *J*
and *a′k*: *a′* < *Suc k* **and** *b′k*: *b′* < *Suc k*
and *a-def*: *pick* (*insert a I*) *a′* = *a*
and *b-def*: *pick* (*insert b J*) *b′* = *b*
shows *mat-delete* (*submatrix A* (*insert a I*) (*insert b J*)) *a′ b′* = *submatrix A I J*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

10.2 On the minors of a diagonal matrix

lemma *det-minors-diagonal*:

assumes *dA*: *diagonal-mat A* **and** *A-carrier*: $A \in$ *carrier-mat n m*
and *Ik*: *card I* $= k$ **and** *Jk*: *card J* $= k$
and *r*: $I \subseteq \{0..<n\}$
and *c*: $J \subseteq \{0..<m\}$ **and** *k*: $k>0$
shows *det* (*submatrix A I J*) $= 0$
$\vee$ ($\exists$ *xs*. (*det* (*submatrix A I J*) $=$ *prod-list xs* $\vee$ *det* (*submatrix A I J*) $=$ $-$ *prod-list xs*)
$\wedge$ *set xs* $\subseteq \{A\$\$(i,i) | i.\ i<min\ n\ m \wedge A\$\$(i,i) \neq 0\} \wedge$ *length xs* $= k$)
⟨*proof*⟩

definition *minors A k* $= \{$*det* (*submatrix A I J*)$|$ *I J*. $I \subseteq \{0..<$*dim-row A*$\}$
$\wedge$ $J \subseteq \{0..<$*dim-col A*$\} \wedge$ *card I* $= k \wedge$ *card J* $= k\}$

lemma *Gcd-minors-dvd*:
fixes *A*::*'a*::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes *PAQ-B*: $P * A * Q = B$
and *P*: $P \in$ *carrier-mat m m*
and *A*: $A \in$ *carrier-mat m n*
and *Q*: $Q \in$ *carrier-mat n n*
and *I*: $I \subseteq \{0..<$*dim-row A*$\}$ **and** *J*: $J \subseteq \{0..<$*dim-col A*$\}$
and *Ik*: *card I* $= k$ **and** *Jk*: *card J* $= k$
shows *Gcd* (*minors A k*) *dvd det* (*submatrix B I J*)
⟨*proof*⟩

lemma *det-minors-diagonal2*:
assumes *dA*: *diagonal-mat A* **and** *A-carrier*: $A \in$ *carrier-mat n m*
and *Ik*: *card I* $= k$ **and** *Jk*: *card J* $= k$
and *r*: $I \subseteq \{0..<n\}$
and *c*: $J \subseteq \{0..<m\}$ **and** *k*: $k>0$
shows *det* (*submatrix A I J*) $= 0 \vee (\exists S.\ S \subseteq \{0..<min\ n\ m\} \wedge$ *card S* $= k \wedge$
$S=I \wedge$
(*det* (*submatrix A I J*) $= (\prod i \in S.\ A\ \$\$\ (i,i)) \vee$ *det* (*submatrix A I J*) $=$ $-$ $(\prod i \in S.\ A\ \$\$\ (i,i))))$
⟨*proof*⟩

10.3 Relating minors and GCD

lemma *diagonal-dvd-Gcd-minors*:
fixes *A*::*'a*::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes *A*: $A \in$ *carrier-mat n m*
and *SNF-A*: *Smith-normal-form-mat A*
shows $(\prod i=0..<k.\ A\ \$\$\ (i,i))$ *dvd Gcd* (*minors A k*)
⟨*proof*⟩

lemma *Gcd-minors-dvd-diagonal*:

fixes A::'a::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes A: $A \in$ *carrier-mat n m*
and *SNF-A*: *Smith-normal-form-mat A*
and k: $k \leq$ *min n m*
shows *Gcd* (*minors A k*) *dvd* ($\prod$ i=*0*..<k. *A* $$ (i,i))
⟨*proof*⟩

lemma *Gcd-minors-A-dvd-Gcd-minors-PAQ*:
fixes A::'a::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes A: $A \in$ *carrier-mat m n*
and P: $P \in$ *carrier-mat m m* **and** Q: $Q \in$ *carrier-mat n n*
shows *Gcd* (*minors A k*) *dvd Gcd* (*minors* (*P*∗*A*∗*Q*) *k*)
⟨*proof*⟩

lemma *Gcd-minors-PAQ-dvd-Gcd-minors-A*:
fixes A::'a::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes A: $A \in$ *carrier-mat m n*
and P: $P \in$ *carrier-mat m m*
and Q: $Q \in$ *carrier-mat n n*
and *inv-P*: *invertible-mat P*
and *inv-Q*: *invertible-mat Q*
shows *Gcd* (*minors* (*P*∗*A*∗*Q*) *k*) *dvd Gcd* (*minors A k*)
⟨*proof*⟩

lemma *Gcd-minors-dvd-diag-PAQ*:
fixes *P A Q*::'a::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes A: $A \in$ *carrier-mat m n*
and P: $P \in$ *carrier-mat m m*
and Q: $Q \in$ *carrier-mat n n*
and *SNF*: *Smith-normal-form-mat* (*P*∗*A*∗*Q*)
and k: $k \leq$*min m n*
shows *Gcd* (*minors A k*) *dvd* ($\prod$ i=*0*..<k. (*P* ∗ *A* ∗ *Q*) $$ (i,i))
⟨*proof*⟩

lemma *diag-PAQ-dvd-Gcd-minors*:
fixes *P A Q*::'a::{*semiring-Gcd*,*comm-ring-1*} *mat*
assumes A: $A \in$ *carrier-mat m n*
and P: $P \in$ *carrier-mat m m*
and Q: $Q \in$ *carrier-mat n n*
and *inv-P*: *invertible-mat P*
and *inv-Q*: *invertible-mat Q*
and *SNF*: *Smith-normal-form-mat* (*P*∗*A*∗*Q*)
shows ($\prod$ i=*0*..<k. (*P* ∗ *A* ∗ *Q*) $$ (i,i)) *dvd Gcd* (*minors A k*)
⟨*proof*⟩

lemma *Smith-prod-zero-imp-last-zero*:
fixes *A*::*'a*::{*semidom*,*comm-ring-1*} *mat*
assumes *A*: *A* ∈ *carrier-mat m n*
and *SNF*: *Smith-normal-form-mat A*
and *prod-0*: (∏ *j*=*0*..<*Suc i*. *A* $$ (*j*,*j*)) = *0*
and *i*: *i*<*min m n*
shows *A* $$(*i*,*i*) = *0*
⟨*proof*⟩

10.4 Final theorem

lemma *Smith-normal-form-uniqueness-aux*:
fixes *P A Q*::*'a*::{*idom*,*semiring-Gcd*} *mat*
assumes *A*: *A* ∈ *carrier-mat m n*

and *P*: *P* ∈ *carrier-mat m m*
and *Q*: *Q* ∈ *carrier-mat n n*
and *inv-P*: *invertible-mat P*
and *inv-Q*: *invertible-mat Q*
and *PAQ-B*: *P*∗*A*∗*Q* = *B*
and *SNF*: *Smith-normal-form-mat B*

and *P'*: *P'* ∈ *carrier-mat m m*
and *Q'*: *Q'* ∈ *carrier-mat n n*
and *inv-P'*: *invertible-mat P'*
and *inv-Q'*: *invertible-mat Q'*
and *P'AQ'-B'*: *P'*∗*A*∗*Q'* = *B'*
and *SNF-B'*: *Smith-normal-form-mat B'*
and *k*: *k*<*min m n*
shows ∀ *i*≤*k*. *B*$$(*i*,*i*) *dvd B'*$$(*i*,*i*) ∧ *B'*$$(*i*,*i*) *dvd B*$$(*i*,*i*)
⟨*proof*⟩

lemma *Smith-normal-form-uniqueness*:
fixes *P A Q*::*'a*::{*idom*,*semiring-Gcd*} *mat*
assumes *A*: *A* ∈ *carrier-mat m n*

and *P*: *P* ∈ *carrier-mat m m*
and *Q*: *Q* ∈ *carrier-mat n n*
and *inv-P*: *invertible-mat P*
and *inv-Q*: *invertible-mat Q*
and *PAQ-B*: *P*∗*A*∗*Q* = *B*
and *SNF*: *Smith-normal-form-mat B*

and *P'*: *P'* ∈ *carrier-mat m m*
and *Q'*: *Q'* ∈ *carrier-mat n n*
and *inv-P'*: *invertible-mat P'*

and *inv-Q'*: *invertible-mat Q'*
and *P'AQ'-B'*: *P'*A*Q' = B'*
and *SNF-B'*: *Smith-normal-form-mat B'*
and *i*: *i < min m n*
shows ∃ *u*. *u dvd 1* ∧ *B $$ (i,i) = u * B' $$ (i,i)*
⟨*proof*⟩

The final theorem, moved to HOL Analysis

lemma *Smith-normal-form-uniqueness-HOL-Analysis*:
fixes *A*::*'a*::{*idom*,*semiring-Gcd*}^*'m*::*mod-type*^*'n*::*mod-type*
and *P P'*::*'a*^*'n*::*mod-type*^*'n*::*mod-type*
and *Q Q'*::*'a*^*'m*::*mod-type*^*'m*::*mod-type*
assumes

inv-P: *invertible P*
and *inv-Q*: *invertible Q*
and *PAQ-B*: *P**A**Q = B*
and *SNF*: *Smith-normal-form B*

and *inv-P'*: *invertible P'*
and *inv-Q'*: *invertible Q'*
and *P'AQ'-B'*: *P'**A**Q' = B'*
and *SNF-B'*: *Smith-normal-form B'*
and *i*: *i < min (nrows A) (ncols A)*
shows ∃ *u*. *u dvd 1* ∧ *B $h Mod-Type.from-nat i $h Mod-Type.from-nat i*
*= u * B' $h Mod-Type.from-nat i $h Mod-Type.from-nat i*
⟨*proof*⟩

10.5 Uniqueness fixing a complete set of non-associates

definition *Smith-normal-form-wrt A 𝒬 = (*
(∀ a b. Mod-Type.to-nat a = Mod-Type.to-nat b ∧ Mod-Type.to-nat a + 1 < nrows A
∧ Mod-Type.to-nat b + 1 < ncols A ⟶ A $h a $h b dvd A $h (a+1) $h (b+1))
∧ isDiagonal A ∧ Complete-set-non-associates 𝒬
∧ (∀ a b. Mod-Type.to-nat a = Mod-Type.to-nat b ∧ Mod-Type.to-nat a < min (nrows A) (ncols A)
∧ Mod-Type.to-nat b < min (nrows A) (ncols A) ⟶ A $h a $h b ∈ 𝒬)
)

lemma *Smith-normal-form-wrt-uniqueness-HOL-Analysis*:
fixes *A*::*'a*::{*idom*,*semiring-Gcd*}^*'m*::*mod-type*^*'n*::*mod-type*
and *P P'*::*'a*^*'n*::*mod-type*^*'n*::*mod-type*
and *Q Q'*::*'a*^*'m*::*mod-type*^*'m*::*mod-type*
assumes

P: *invertible P*
and *Q*: *invertible Q*

and *PAQ-S*: $P{*}{*}A{*}{*}Q = S$
and *SNF*: *Smith-normal-form-wrt* S $\mathcal{Q}$

and P': *invertible* P'
and Q': *invertible* Q'
and *P'AQ'-S'*: $P'{*}{*}A{*}{*}Q' = S'$
and *SNF-S'*: *Smith-normal-form-wrt* S' $\mathcal{Q}$
shows $S = S'$
⟨*proof*⟩

end

11 The Cauchy–Binet formula in HOL Analysis

theory *Cauchy-Binet-HOL-Analysis*
imports
Cauchy-Binet
Perron-Frobenius.HMA-Connect
begin

11.1 Definition of submatrices in HOL Analysis

definition *submatrix-hma* :: *'a^'nc^'nr⇒nat set⇒nat set⇒('a^'nc2^'nr2)*
where *submatrix-hma A I J* = (*χ a b. A $h* (*from-nat* (*pick I* (*to-nat a*))) *$h* (*from-nat* (*pick J* (*to-nat b*))))

context includes *lifting-syntax*
begin

context
fixes *I*::*nat set* **and** *J*::*nat set*
assumes *I*: *card* {*i*. *i* < *CARD*('*nr*::*finite*) ∧ *i* ∈ *I*} = *CARD*('*nr2*::*finite*)
assumes *J*: *card* {*i*. *i* < *CARD*('*nc*::*finite*) ∧ *i* ∈ *J*} = *CARD*('*nc2*::*finite*)
begin

lemma *HMA-submatrix*[*transfer-rule*]: (*HMA-M* ===> *HMA-M*) (λ*A*. *submatrix A I J*)
((λ*A*. *submatrix-hma A I J*):: *'a^'nc^'nr* ⇒ *'a^'nc2^'nr2*)
⟨*proof*⟩

end
end

11.2 Transferring the proof from JNF to HOL Analysis

lemma *Cauchy-Binet-HOL-Analysis*:
fixes *A*::*'a*::*comm-ring-1^'m^'n* **and** *B*::*'a^'n^'m*

shows *Determinants.det* $(A{*}{*}B) = (\sum I{\in}\{I.\ I{\subseteq}\{0..{<}ncols\ A\} \wedge card\ I{=}nrows\ A\}.$
Determinants.det $((submatrix\text{-}hma\ A\ UNIV\ I){::}'a\hat{}'n\hat{}'n)\ *$
Determinants.det $((submatrix\text{-}hma\ B\ I\ UNIV){::}'a\hat{}'n\hat{}'n))$
⟨*proof*⟩

end

12 Diagonalizing matrices in JNF and HOL Analysis

theory *Diagonalize*
imports *Admits-SNF-From-Diagonal-Iff-Bezout-Ring*
begin

This section presents a *locale* that assumes a sound operation to make a matrix diagonal. Then, the result is transferred to HOL Analysis.

12.1 Diagonalizing matrices in JNF

We assume a *diagonalize-JNF* operation in JNF, which is applied to matrices over a Bézout ring. However, probably a more restrictive type class is required.

locale *diagonalize* =
fixes *diagonalize-JNF* :: $'a{::}bezout\text{-}ring\ mat \Rightarrow 'a\ bezout \Rightarrow ('a\ mat \times 'a\ mat \times 'a\ mat)$
assumes *soundness-diagonalize-JNF*:
$\forall A\ bezout.\ A \in carrier\text{-}mat\ m\ n \wedge is\text{-}bezout\text{-}ext\ bezout \longrightarrow$
$(case\ diagonalize\text{-}JNF\ A\ bezout\ of\ (P,S,Q) \Rightarrow$
$P \in carrier\text{-}mat\ m\ m \wedge Q \in carrier\text{-}mat\ n\ n \wedge S \in carrier\text{-}mat\ m\ n$
$\wedge\ invertible\text{-}mat\ P \wedge invertible\text{-}mat\ Q \wedge isDiagonal\text{-}mat\ S \wedge S = P{*}A{*}Q)$
begin

lemma *soundness-diagonalize-JNF'*:
fixes $A{::}'a\ mat$
assumes *is-bezout-ext bezout* **and** $A \in carrier\text{-}mat\ m\ n$
and *diagonalize-JNF A bezout* $= (P,S,Q)$
shows $P \in carrier\text{-}mat\ m\ m \wedge Q \in carrier\text{-}mat\ n\ n \wedge S \in carrier\text{-}mat\ m\ n$
$\wedge\ invertible\text{-}mat\ P \wedge invertible\text{-}mat\ Q \wedge isDiagonal\text{-}mat\ S \wedge S = P{*}A{*}Q$
⟨*proof*⟩

12.2 Implementation and soundness result moved to HOL Analysis.

definition *diagonalize* :: $'a{::}bezout\text{-}ring\ \hat{}\ 'nc :: mod\text{-}type\ \hat{}\ 'nr :: mod\text{-}type$
$\Rightarrow 'a\ bezout \Rightarrow$
$(('a\ \hat{}\ 'nr :: mod\text{-}type\ \hat{}\ 'nr :: mod\text{-}type)$

$\times$ ('a ^ 'nc :: *mod-type* ^ 'nr :: *mod-type*)
$\times$ ('a ^ 'nc :: *mod-type* ^ 'nc :: *mod-type*))
where *diagonalize A bezout* = (
let (*P*,*S*,*Q*) = *diagonalize-JNF* (*Mod-Type-Connect.from-hma*$_m$ *A*) *bezout*
in (*Mod-Type-Connect.to-hma*$_m$ *P*,*Mod-Type-Connect.to-hma*$_m$ *S*,*Mod-Type-Connect.to-hma*$_m$ *Q*)
)

lemma *soundness-diagonalize*:
assumes *b*: *is-bezout-ext bezout*
and *d*: *diagonalize A bezout* = (*P*,*S*,*Q*)
shows *invertible P* $\wedge$ *invertible Q* $\wedge$ *isDiagonal S* $\wedge$ *S* = *P*$**$*A*$**$*Q*
⟨*proof*⟩
end

end

13 Smith normal form algorithm based on two steps in HOL Analysis

theory *SNF-Algorithm-Two-Steps*
imports *Diagonalize*
begin

This file contains an algorithm to transform a matrix to its Smith normal form, based on two steps: first it is converted into a diagonal matrix and then transformed from diagonal to Smith.

We assume the existence of a diagonalize operation, and then we just have to connect it to the existing algorithm (in HOL Analysis) to transform a diagonal matrix into its Smith normal form.

13.1 The implementation

context *diagonalize*
begin

definition *Smith-normal-form-of A bezout* = (
let (*P''*,*D*,*Q''*) = *diagonalize A bezout*;
(*P'*,*S*,*Q'*) = *diagonal-to-Smith-PQ D bezout*
in (*P'*$**$*P''*,*S*,*Q''*$**$*Q'*)
)

13.2 Soundness in HOL Analysis

lemma *Smith-normal-form-of-soundness*:
fixes *A*::'*a*::{*bezout-ring*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *b*: *is-bezout-ext bezout*

assumes *PSQ*: $(P,S,Q) =$ *Smith-normal-form-of A bezout*
shows $S = P{*}{*}A{*}{*}Q \wedge$ *invertible P* $\wedge$ *invertible Q* $\wedge$ *Smith-normal-form S*
⟨*proof*⟩

end
end

14 Algorithm to transform a diagonal matrix into its Smith normal form in JNF

theory *Diagonal-To-Smith-JNF*
imports *Admits-SNF-From-Diagonal-Iff-Bezout-Ring*
begin

In this file, we implement an algorithm to transform a diagonal matrix into its Smith normal form, using the JNF library.

There are, at least, three possible options:

1. Implement and prove the soundness of the algorithm from scratch in JNF

2. Implement it in JNF and connect it to the HOL Analysis version by means of transfer rules. Thus, we could obtain the soundness lemma in JNF.

3. Implement it in JNF, with calls to the HOL Analysis version by means of the functions *from-hma*$_m$ and *to-hma*$_m$. That is, transform the matrix to HOL Analysis, apply the existing algorith in HOL Analysis to get the Smith normal form and then transform the output to JNF. Then, we could try to get the soundness theorem in JNF by means of transfer rules and local type definitions.

The first option requires much effort. As we will see, the third option is not possible.

14.1 Attempt with the third option: definitions and conditional transfer rules

context
fixes *A*::*'a*::*bezout-ring mat*
assumes *A* ∈ *carrier-mat CARD*(*'nr*::*mod-type*) *CARD*(*'nc*::*mod-type*)
begin

private definition *diagonal-to-Smith-PQ-JNF' bezout* = (
let A' = *Mod-Type-Connect.to-hma*$_m$ *A*::*'a*^*'nc*::*mod-type*^*'nr*::*mod-type*;
$(P,S,Q) =$ (*diagonal-to-Smith-PQ A' bezout*)

in (*Mod-Type-Connect.from-hma*$_m$ *P*, *Mod-Type-Connect.from-hma*$_m$ *S*, *Mod-Type-Connect.from-hma*$_m$ *Q*))

end

This approach will not work. The type is necessary in the definition of the function. That is, outside the context, the function will be:

diagonal-to-Smith-PQ-JNF' *TYPE*('*nc*) *TYPE*('*nr*) *A bezout*

And we cannot get rid of such *TYPE*('*nc*).

That is, we could get a lemma like:

lemma assumes $A \in$ *carrier-mat m n* **and** $(P,S,Q) =$ *diagonal-to-Smith-PQ-JNF'* *TYPE*('*nr*::*mod-type*) *TYPE*('*nc*::*mod-type*) *A bezout* **shows** *invertible-mat P* $\wedge$ *invertible-mat Q* $\wedge$ $S = P * A * Q \wedge$ *Smith-normal-form-mat S*

But we wouldn't be able to get rid of such types.

14.2 Attempt with the second option: implementation and soundness in JNF

definition *diagonal-step-JNF A i j d v* =
Matrix.mat (*dim-row A*) (*dim-col A*) (λ (*a*,*b*). *if a* = *i* $\wedge$ *b* = *i then d* *else*
if a = *j* $\wedge$ *b* = *j*
then v $*$ (*A* $$ (*j*,*j*)) *else A* $$ (*a*,*b*))

Conditional transfer rules are required, so I prove them within context with assumptions.

context
includes *lifting-syntax*
fixes *i* **and** *j*::*nat*
assumes *i*: *i*<*min* (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
and *j*: *j*<*min* (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
begin

lemma *HMA-diagonal-step*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - $\Rightarrow$ '*a* :: *comm-ring-1* ^ '*nc* :: *mod-type* ^ '*nr* :: *mod-type* $\Rightarrow$ -)
===> (=) ===> (=) ===> *Mod-Type-Connect.HMA-M*)
(λ*A*. *diagonal-step-JNF A i j*) (λ*B*. *diagonal-step B i j*)
⟨*proof*⟩

end

definition *diagonal-step-PQ-JNF* ::
'*a*::{*bezout-ring*} *mat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ '*a bezout* $\Rightarrow$ ('*a mat* $\times$ ('*a mat*))
where *diagonal-step-PQ-JNF A i k bezout* =
(*let* *m* = *dim-row A*; *n* = *dim-col A*;
(*p*, *q*, *u*, *v*, *d*) = *bezout* (*A* $$ (*i*,*i*)) (*A* $$ (*k*,*k*));

P = *addrow* $(-v)$ k i (*swaprows* i k (*addrow* p k i (1_m m)));
Q = *multcol* k (-1) (*addcol* u k i (*addcol* q i k (1_m n)))
in (P,Q)
)

context
includes *lifting-syntax*
fixes i **and** k::*nat*
assumes i: $i < min$ (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
and k: $k < min$ (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
begin

lemma *HMA-diagonal-step-PQ*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M* :: - ⇒ '*a* :: *bezout-ring* ^ '*nc* :: *mod-type* ^ '*nr* :: *mod-type* ⇒ -)
===> (=) ===> *rel-prod Mod-Type-Connect.HMA-M Mod-Type-Connect.HMA-M*)

(λ*A bezout*. *diagonal-step-PQ-JNF A i k bezout*) (λ*A bezout*. *diagonal-step-PQ A i k bezout*)
⟨*proof*⟩

end

fun *diagonal-to-Smith-i-PQ-JNF* ::
nat list ⇒ *nat* ⇒ ('*a*::{*bezout-ring*} *bezout*)
⇒ ('*a mat* × '*a mat* × '*a mat*) ⇒ ('*a mat* × '*a mat* × '*a mat*)
where
diagonal-to-Smith-i-PQ-JNF [] *i bezout* (P,A,Q) = (P,A,Q) |
diagonal-to-Smith-i-PQ-JNF (j#*xs*) *i bezout* (P,A,Q) = (
if A $$ (i,i) *dvd A* $$ (j,j)
then diagonal-to-Smith-i-PQ-JNF xs i bezout (P,A,Q)
else let $(p,\ q,\ u,\ v,\ d)$ = *bezout* (*A* $$ (i,i)) (*A* $$ (j,j));
A' = *diagonal-step-JNF A i j d v*;
(P',Q') = *diagonal-step-PQ-JNF A i j bezout*
in diagonal-to-Smith-i-PQ-JNF xs i bezout $(P'*P,A',Q*Q')$ — Apply the step
)

context
includes *lifting-syntax*
fixes i **and** *xs*
assumes i: $i < min$ (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
and *xs*: ∀ j∈*set xs*. $j < min$ (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
begin

declare *diagonal-step-PQ.simps*[*simp del*]

lemma *HMA-diagonal-to-Smith-i-PQ-aux*: *HMA-M3* (P,A,Q)

(*P'* :: *'a* :: *bezout-ring* ^ *'nr* :: *mod-type* ^ *'nr* :: *mod-type*,
A' :: *'a* :: *bezout-ring* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type*,
Q' :: *'a* :: *bezout-ring* ^ *'nc* :: *mod-type* ^ *'nc* :: *mod-type*)
⟹ *HMA-M3* (*diagonal-to-Smith-i-PQ-JNF xs i bezout* (*P*,*A*,*Q*))
(*diagonal-to-Smith-i-PQ xs i bezout* (*P'*,*A'*,*Q'*))
⟨*proof*⟩

lemma *HMA-diagonal-to-Smith-i-PQ*[*transfer-rule*]:
((=)
===> (*HMA-M3* :: (- ⇒ (-×(*'a* :: *bezout-ring* ^ *'nc* :: *mod-type* ^ *'nr* :: *mod-type*)
× -) ⇒-))
===> *HMA-M3*) (*diagonal-to-Smith-i-PQ-JNF xs i*) (*diagonal-to-Smith-i-PQ*
xs i)
⟨*proof*⟩

end

fun *Diagonal-to-Smith-row-i-PQ-JNF*
where *Diagonal-to-Smith-row-i-PQ-JNF i bezout* (*P*,*A*,*Q*)
= *diagonal-to-Smith-i-PQ-JNF* [*i* + *1*..<*min* (*dim-row A*) (*dim-col A*)] *i bezout*
(*P*,*A*,*Q*)

declare *Diagonal-to-Smith-row-i-PQ-JNF.simps*[*simp del*]
lemmas *Diagonal-to-Smith-row-i-PQ-JNF-def* = *Diagonal-to-Smith-row-i-PQ-JNF.simps*

context
includes *lifting-syntax*
fixes *i*
assumes *i*: *i* < *min* (*CARD*('*nr*::*mod-type*)) (*CARD*('*nc*::*mod-type*))
begin

lemma *HMA-Diagonal-to-Smith-row-i-PQ*[*transfer-rule*]:
((=) ===> (*HMA-M3* :: (- ⇒ (- × ('*a*::*bezout-ring*^'*nc*::*mod-type*^'*nr*::*mod-type*)
× -) ⇒ -)) ===> *HMA-M3*)
(*Diagonal-to-Smith-row-i-PQ-JNF i*) (*Diagonal-to-Smith-row-i-PQ i*)
⟨*proof*⟩

end

fun *diagonal-to-Smith-aux-PQ-JNF*
where
diagonal-to-Smith-aux-PQ-JNF [] *bezout* (*P*,*A*,*Q*) = (*P*,*A*,*Q*) |
diagonal-to-Smith-aux-PQ-JNF (*i*#*xs*) *bezout* (*P*,*A*,*Q*)
= *diagonal-to-Smith-aux-PQ-JNF xs bezout* (*Diagonal-to-Smith-row-i-PQ-JNF*
i bezout (*P*,*A*,*Q*))

context
includes *lifting-syntax*
fixes *xs*

assumes *xs*: $\forall j{\in}set\ xs.\ j < min\ (CARD('nr{::}mod\text{-}type))\ (CARD('nc{::}mod\text{-}type))$
begin

lemma *HMA-diagonal-to-Smith-aux-PQ-JNF*[*transfer-rule*]:
$((=) ===> (HMA\text{-}M3 :: (- \Rightarrow (- \times ('a{::}bezout\text{-}ring\ \hat{}\ 'nc{::}mod\text{-}type\ \hat{}\ 'nr{::}mod\text{-}type) \times -) \Rightarrow -)) ===> HMA\text{-}M3)$
$(diagonal\text{-}to\text{-}Smith\text{-}aux\text{-}PQ\text{-}JNF\ xs)\ (diagonal\text{-}to\text{-}Smith\text{-}aux\text{-}PQ\ xs)$
⟨*proof*⟩

end

fun *diagonal-to-Smith-PQ-JNF*
where *diagonal-to-Smith-PQ-JNF A bezout*
$= diagonal\text{-}to\text{-}Smith\text{-}aux\text{-}PQ\text{-}JNF\ [0..{<}min\ (dim\text{-}row\ A)\ (dim\text{-}col\ A) - 1]$
$bezout\ (1_m\ (dim\text{-}row\ A), A, 1_m\ (dim\text{-}col\ A))$

declare *diagonal-to-Smith-PQ-JNF.simps*[*simp del*]
lemmas *diagonal-to-Smith-PQ-JNF-def* = *diagonal-to-Smith-PQ-JNF.simps*

lemma *diagonal-step-PQ-JNF-dim*:
assumes *A*: $A \in carrier\text{-}mat\ m\ n$
and *d*: $diagonal\text{-}step\text{-}PQ\text{-}JNF\ A\ i\ j\ bezout = (P,Q)$
shows $P \in carrier\text{-}mat\ m\ m \wedge Q \in carrier\text{-}mat\ n\ n$
⟨*proof*⟩

lemma *diagonal-step-JNF-dim*:
assumes *A*: $A \in carrier\text{-}mat\ m\ n$
shows $diagonal\text{-}step\text{-}JNF\ A\ i\ j\ d\ v \in carrier\text{-}mat\ m\ n$
⟨*proof*⟩

lemma *diagonal-to-Smith-i-PQ-JNF-dim*:
assumes $P' \in carrier\text{-}mat\ m\ m \wedge A' \in carrier\text{-}mat\ m\ n \wedge Q' \in carrier\text{-}mat\ n\ n$
and $diagonal\text{-}to\text{-}Smith\text{-}i\text{-}PQ\text{-}JNF\ xs\ i\ bezout\ (P',A',Q') = (P,A,Q)$
shows $P \in carrier\text{-}mat\ m\ m \wedge A \in carrier\text{-}mat\ m\ n \wedge Q \in carrier\text{-}mat\ n\ n$
⟨*proof*⟩

lemma *Diagonal-to-Smith-row-i-PQ-JNF-dim*:
assumes $P' \in carrier\text{-}mat\ m\ m \wedge A' \in carrier\text{-}mat\ m\ n \wedge Q' \in carrier\text{-}mat\ n\ n$
and $Diagonal\text{-}to\text{-}Smith\text{-}row\text{-}i\text{-}PQ\text{-}JNF\ i\ bezout\ (P',A',Q') = (P,A,Q)$
shows $P \in carrier\text{-}mat\ m\ m \wedge A \in carrier\text{-}mat\ m\ n \wedge Q \in carrier\text{-}mat\ n\ n$
⟨*proof*⟩

lemma *diagonal-to-Smith-aux-PQ-JNF-dim*:
assumes $P' \in carrier\text{-}mat\ m\ m \wedge A' \in carrier\text{-}mat\ m\ n \wedge Q' \in carrier\text{-}mat\ n\ n$
and $diagonal\text{-}to\text{-}Smith\text{-}aux\text{-}PQ\text{-}JNF\ xs\ bezout\ (P',A',Q') = (P,A,Q)$
shows $P \in carrier\text{-}mat\ m\ m \wedge A \in carrier\text{-}mat\ m\ n \wedge Q \in carrier\text{-}mat\ n\ n$
⟨*proof*⟩

lemma *diagonal-to-Smith-PQ-JNF-dim*:
assumes $A \in$ *carrier-mat m n*
and *PSQ*: *diagonal-to-Smith-PQ-JNF A bezout* = (*P*,*S*,*Q*)
shows $P \in$ *carrier-mat m m* $\wedge$ $S \in$ *carrier-mat m n* $\wedge$ $Q \in$ *carrier-mat n n*
⟨*proof*⟩

context
includes *lifting-syntax*
begin

lemma *HMA-diagonal-to-Smith-PQ-JNF*[*transfer-rule*]:
((*Mod-Type-Connect.HMA-M*) ===> (=) ===> *HMA-M*³) (*diagonal-to-Smith-PQ-JNF*)
(*diagonal-to-Smith-PQ*)
⟨*proof*⟩

end

14.3 Applying local type definitions

Now we get the soundness lemma in JNF, via the one in HOL Analysis. I need transfer rules and local type definitions.

context
includes *lifting-syntax*
begin

private lemma *diagonal-to-Smith-PQ-JNF-with-types*:
assumes *A*: $A \in$ *carrier-mat CARD*('*nr*::*mod-type*) *CARD*('*nc*::*mod-type*)
and *S*: $S \in$ *carrier-mat CARD*('*nr*) *CARD*('*nc*)
and *P*: $P \in$ *carrier-mat CARD*('*nr*) *CARD*('*nr*)
and *Q*: $Q \in$ *carrier-mat CARD*('*nc*) *CARD*('*nc*)
and *PSQ*: *diagonal-to-Smith-PQ-JNF A bezout* = (*P*, *S*, *Q*)
and *d*:*isDiagonal-mat A* **and** *ib*: *is-bezout-ext bezout*
shows $S = P * A * Q \wedge$ *invertible-mat P* $\wedge$ *invertible-mat Q* $\wedge$ *Smith-normal-form-mat S*
⟨*proof*⟩ **lemma** *diagonal-to-Smith-PQ-JNF-mod-ring-with-types*:
assumes *A*: $A \in$ *carrier-mat CARD*('*nr*::*nontriv mod-ring*) *CARD*('*nc*::*nontriv mod-ring*)
and *S*: $S \in$ *carrier-mat CARD*('*nr mod-ring*) *CARD*('*nc mod-ring*)
and *P*: $P \in$ *carrier-mat CARD*('*nr mod-ring*) *CARD*('*nr mod-ring*)
and *Q*: $Q \in$ *carrier-mat CARD*('*nc mod-ring*) *CARD*('*nc mod-ring*)
and *PSQ*: *diagonal-to-Smith-PQ-JNF A bezout* = (*P*, *S*, *Q*)
and *d*:*isDiagonal-mat A* **and** *ib*: *is-bezout-ext bezout*
shows $S = P * A * Q \wedge$ *invertible-mat P* $\wedge$ *invertible-mat Q* $\wedge$ *Smith-normal-form-mat S*
⟨*proof*⟩

thm *diagonal-to-Smith-PQ-JNF-mod-ring-with-types*[*unfolded CARD-mod-ring*, *internalize-sort 'nr::nontriv*]

private lemma *diagonal-to-Smith-PQ-JNF-internalized-first*:
class.nontriv TYPE('a::type) ⟹
A ∈ *carrier-mat CARD('a) CARD('nc::nontriv)* ⟹
S ∈ *carrier-mat CARD('a) CARD('nc)* ⟹
P ∈ *carrier-mat CARD('a) CARD('a)* ⟹
Q ∈ *carrier-mat CARD('nc) CARD('nc)* ⟹
diagonal-to-Smith-PQ-JNF A bezout = (*P*, *S*, *Q*) ⟹
isDiagonal-mat A ⟹ *is-bezout-ext bezout* ⟹
S = *P* ∗ *A* ∗ *Q* ∧ *invertible-mat P* ∧ *invertible-mat Q* ∧ *Smith-normal-form-mat S*
⟨*proof*⟩ **lemma** *diagonal-to-Smith-PQ-JNF-internalized*:
class.nontriv TYPE('c::type) ⟹
class.nontriv TYPE('a::type) ⟹
A ∈ *carrier-mat CARD('a) CARD('c)* ⟹
S ∈ *carrier-mat CARD('a) CARD('c)* ⟹
P ∈ *carrier-mat CARD('a) CARD('a)* ⟹
Q ∈ *carrier-mat CARD('c) CARD('c)* ⟹
diagonal-to-Smith-PQ-JNF A bezout = (*P*, *S*, *Q*) ⟹
isDiagonal-mat A ⟹ *is-bezout-ext bezout* ⟹
S = *P* ∗ *A* ∗ *Q* ∧ *invertible-mat P* ∧ *invertible-mat Q* ∧ *Smith-normal-form-mat S*
⟨*proof*⟩

context
fixes *m*::*nat* **and** *n*::*nat*
assumes *local-typedef1*: ∃(*Rep* :: ('*b* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*m* :: *int*}
assumes *local-typedef2*: ∃(*Rep* :: ('*c* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*n* :: *int*}
and *m*: *m*>*1*
and *n*: *n*>*1*
begin

lemma *type-to-set1*:
shows *class.nontriv TYPE('b)* (**is** *?a*) **and** *m*=*CARD('b)* (**is** *?b*)
⟨*proof*⟩

lemma *type-to-set2*:
shows *class.nontriv TYPE('c)* (**is** *?a*) **and** *n*=*CARD('c)* (**is** *?b*)
⟨*proof*⟩

lemma *diagonal-to-Smith-PQ-JNF-local-typedef*:
assumes *A*: *isDiagonal-mat A* **and** *ib*: *is-bezout-ext bezout*
and *A-dim*: *A* ∈ *carrier-mat m n*
assumes *PSQ*: (*P*,*S*,*Q*) = *diagonal-to-Smith-PQ-JNF A bezout*

shows $S = P*A*Q \wedge$ *invertible-mat* $P \wedge$ *invertible-mat* $Q \wedge$ *Smith-normal-form-mat* S
$\wedge\ P \in$ *carrier-mat* $m\ m \wedge S \in$ *carrier-mat* $m\ n \wedge Q \in$ *carrier-mat* $n\ n$
⟨*proof*⟩
end
end

context
begin
private lemma *diagonal-to-Smith-PQ-JNF-canceled-first*:
$\exists$ *Rep Abs. type-definition Rep Abs* $\{0..<int\ n\} \Longrightarrow \{0..<int\ m\} \neq \{\} \Longrightarrow$
$1 < m \Longrightarrow 1 < n \Longrightarrow$ *isDiagonal-mat* $A \Longrightarrow$ *is-bezout-ext bezout* $\Longrightarrow$
$A \in$ *carrier-mat* $m\ n \Longrightarrow (P,\ S,\ Q) =$ *diagonal-to-Smith-PQ-JNF* A *bezout* $\Longrightarrow$
$S = P * A * Q \wedge$ *invertible-mat* $P \wedge$ *invertible-mat* $Q \wedge$ *Smith-normal-form-mat* S
$\wedge\ P \in$ *carrier-mat* $m\ m \wedge S \in$ *carrier-mat* $m\ n \wedge Q \in$ *carrier-mat* $n\ n$
⟨*proof*⟩ **lemma** *diagonal-to-Smith-PQ-JNF-canceled-both*:
$\{0..<int\ n\} \neq \{\} \Longrightarrow \{0..<int\ m\} \neq \{\} \Longrightarrow 1 < m \Longrightarrow 1 < n \Longrightarrow$
isDiagonal-mat $A \Longrightarrow$ *is-bezout-ext bezout* $\Longrightarrow A \in$ *carrier-mat* $m\ n \Longrightarrow$
$(P,\ S,\ Q) =$ *diagonal-to-Smith-PQ-JNF* A *bezout* $\Longrightarrow S = P * A * Q \wedge$
invertible-mat $P \wedge$ *invertible-mat* $Q \wedge$ *Smith-normal-form-mat* S
$\wedge\ P \in$ *carrier-mat* $m\ m \wedge S \in$ *carrier-mat* $m\ n \wedge Q \in$ *carrier-mat* $n\ n$
⟨*proof*⟩

14.4 The final result

lemma *diagonal-to-Smith-PQ-JNF*:
assumes A: *isDiagonal-mat* A **and** *ib*: *is-bezout-ext bezout*
and $A \in$ *carrier-mat* $m\ n$
and *PBQ*: $(P,S,Q) =$ *diagonal-to-Smith-PQ-JNF* A *bezout*

and n: $n>1$ **and** m: $m>1$
shows $S = P*A*Q \wedge$ *invertible-mat* $P \wedge$ *invertible-mat* $Q \wedge$ *Smith-normal-form-mat* S
$\wedge\ P \in$ *carrier-mat* $m\ m \wedge S \in$ *carrier-mat* $m\ n \wedge Q \in$ *carrier-mat* $n\ n$
⟨*proof*⟩
end
end

15 Smith normal form algorithm based on two steps in JNF

theory *SNF-Algorithm-Two-Steps-JNF*
imports
Diagonalize
Diagonal-To-Smith-JNF
begin

15.1 Moving the result from HOL Analysis to JNF

context *diagonalize*
begin

definition *Smith-normal-form-of-JNF A bezout* = (
  let (*P''*,*D*,*Q''*) = *diagonalize-JNF A bezout*;
    (*P'*,*S*,*Q'*) = *diagonal-to-Smith-PQ-JNF D bezout*
  in (*P'*∗*P''*,*S*,*Q''*∗*Q'*)
)

lemma *Smith-normal-form-of-JNF-soundness*:
 assumes *b*: *is-bezout-ext bezout* **and** *A*: *A* ∈ *carrier-mat m n*
 and *n*: *1* < *n* **and** *m*: *1* < *m*
 and *PSQ*: *Smith-normal-form-of-JNF A bezout* = (*P*,*S*,*Q*)
shows *S* = *P*∗*A*∗*Q* ∧ *invertible-mat P* ∧ *invertible-mat Q* ∧ *Smith-normal-form-mat S*
 ∧ *P* ∈ *carrier-mat m m* ∧ *S* ∈ *carrier-mat m n* ∧ *Q*∈ *carrier-mat n n*
⟨*proof*⟩

end
end

16 A general algorithm to transform a matrix into its Smith normal form

theory *SNF-Algorithm*
 imports
  Smith-Normal-Form-JNF
begin

This theory presents an executable algorithm to transform a matrix to its Smith normal form.

16.1 Previous definitions and lemmas

definition *is-SNF A R* = (*case R of* (*P*,*S*,*Q*) ⇒
 P ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) ∧
 Q ∈ *carrier-mat* (*dim-col A*) (*dim-col A*)
 ∧ *invertible-mat P* ∧ *invertible-mat Q*
 ∧ *Smith-normal-form-mat S* ∧ *S* = *P* ∗ *A* ∗ *Q*)

lemma *is-SNF-intro*:
 assumes *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*)
 and *Q* ∈ *carrier-mat* (*dim-col A*) (*dim-col A*)
 and *invertible-mat P* **and** *invertible-mat Q*

and *Smith-normal-form-mat S* **and** $S = P * A * Q$
shows *is-SNF A* (*P*,*S*,*Q*) ⟨*proof*⟩

lemma *Smith-1xn-two-matrices*:
fixes *A* :: *'a*::*comm-ring-1 mat*
assumes *A*: $A \in$ *carrier-mat 1 n*
and *PSQ*: (*P*,*S*,*Q*) = (*Smith-1xn A*)
and *is-SNF*: *is-SNF A* (*Smith-1xn A*)
shows ∃ *Smith-1xn'*. *is-SNF A* (*1*$_m$ *1*, (*Smith-1xn'* *A*))
⟨*proof*⟩

lemma *Smith-1xn-two-matrices-all*:
assumes *is-SNF*: ∀ (*A*::*'a*::*comm-ring-1 mat*) ∈ *carrier-mat 1 n*. *is-SNF A* (*Smith-1xn A*)
shows ∃ *Smith-1xn'*. ∀ (*A*::*'a*::*comm-ring-1 mat*) ∈ *carrier-mat 1 n*. *is-SNF A* (*1*$_m$ *1*, (*Smith-1xn'* *A*))
⟨*proof*⟩

16.2 Previous operations

context
assumes *SORT-CONSTRAINT*(*'a*::*comm-ring-1*)
begin

definition *is-div-op* :: (*'a*⇒*'a*⇒*'a*) ⇒*bool*
where *is-div-op div-op* = (∀ *a b*. *b dvd a* ⟶ *div-op a b* * *b* = *a*)

lemma *div-op-SOME*: *is-div-op* (λ*a b*. (*SOME k*. *k* * *b* = *a*))
⟨*proof*⟩

fun *reduce-column-aux* :: (*'a*⇒*'a*⇒*'a*) ⇒ *nat list* ⇒ *'a mat* ⇒ (*'a mat* × *'a mat*) ⇒ (*'a mat* × *'a mat*)
where *reduce-column-aux div-op* [] *H* (*P*,*K*) = (*P*,*K*)
| *reduce-column-aux div-op* (*i*#*xs*) *H* (*P*,*K*) = (
— Reduce the i-th row
let k = *div-op* (*H*$$(*i*,*0*)) (*H* $$ (*0*, *0*));
P' = *addrow-mat* (*dim-row H*) (−*k*) *i 0*;
K' = *addrow* (−*k*) *i 0 K*
in reduce-column-aux div-op xs H (*P'*∗*P*,*K'*)
)

definition *reduce-column div-op H* = *reduce-column-aux div-op* [*2*..<*dim-row H*] *H* (*1*$_m$ (*dim-row H*),*H*)

lemma *reduce-column-aux*:
assumes *H*: $H \in$ *carrier-mat m n*
and *P-init*: *P-init* $\in$ *carrier-mat m m*
and *K-init*: *K-init* $\in$ *carrier-mat m n*
and *P-init-H-K-init*: *P-init* $*$ *H* $=$ *K-init*
and *PK-H*: (*P*,*K*) $=$ *reduce-column-aux div-op xs H* (*P-init*,*K-init*)
and *m*: $0 < m$
and *inv-P*: *invertible-mat P-init*
and *xs*: $0 \notin$ *set xs*
shows $P \in$ *carrier-mat m m* $\wedge$ $K \in$ *carrier-mat m n* $\wedge$ $P * H = K \wedge$ *invertible-mat P*
⟨*proof*⟩

lemma *reduce-column-aux-preserves*:
assumes *H*: $H \in$ *carrier-mat m n*
and *P-init*: *P-init* $\in$ *carrier-mat m m*
and *K-init*: *K-init* $\in$ *carrier-mat m n*
and *P-init-H-K-init*: *P-init* $*$ *H* $=$ *K-init*
and *PK-H*: (*P*,*K*) $=$ *reduce-column-aux div-op xs H* (*P-init*,*K-init*)
and *m*: $0 < m$
and *inv-P*: *invertible-mat P-init*
and *xs*: $0 \notin$ *set xs* **and** *i*: $i \notin$ *set xs* **and** *im*: $i<m$
shows *Matrix.row K i* $=$ *Matrix.row K-init i*
⟨*proof*⟩

lemma *reduce-column-aux-index'*:
assumes *H*: $H \in$ *carrier-mat m n*
and *P-init*: *P-init* $\in$ *carrier-mat m m*
and *K-init*: *K-init* $\in$ *carrier-mat m n*
and *P-init-H-K-init*: *P-init* $*$ *H* $=$ *K-init*
and *PK-H*: (*P*,*K*) $=$ *reduce-column-aux div-op xs H* (*P-init*,*K-init*)
and *m*: $0 < m$
and *inv-P*: *invertible-mat P-init*
and *xs*: $0 \notin$ *set xs*
and $\forall\, x \in$ *set xs*. $x<m$
and *distinct xs*
shows ($\forall\, i \in$ *set xs*. *Matrix.row K i* $=$
Matrix.row (*addrow* ($-$(*div-op* (*H* $ $ (*i*, 0)) (*H* $ $ (0, 0)))) *i 0 K-init*) *i*)
⟨*proof*⟩

corollary *reduce-column-aux-index*:
assumes *H*: $H \in$ *carrier-mat m n*
and *P-init*: *P-init* $\in$ *carrier-mat m m*
and *K-init*: *K-init* $\in$ *carrier-mat m n*
and *P-init-H-K-init*: *P-init* $*$ *H* $=$ *K-init*
and *PK-H*: (*P*,*K*) $=$ *reduce-column-aux div-op xs H* (*P-init*,*K-init*)
and *m*: $0 < m$
and *inv-P*: *invertible-mat P-init*

and *xs*: $0 \notin set\ xs$
and $\forall x{\in}set\ xs.\ x{<}m$
and *distinct xs*
and $i{\in}set\ xs$
shows $Matrix.row\ K\ i =$
$Matrix.row\ (addrow\ (-(div\text{-}op\ (H\ \$\$\ (i,\ 0))\ (H\ \$\$\ (0,\ 0))))\ i\ 0\ K\text{-}init)\ i$
⟨*proof*⟩

corollary *reduce-column-aux-works*:
assumes *H*: $H \in carrier\text{-}mat\ m\ n$
and *PK-H*: $(P,K) = reduce\text{-}column\text{-}aux\ div\text{-}op\ xs\ H\ (1_m\ (dim\text{-}row\ H),\ H)$
and *m*: $0 < m$
and *xs*: $0 \notin set\ xs$
and *xm*: $\forall x \in set\ xs.\ x{<}m$
and *d-xs*: *distinct xs*
and *i*: $i \in set\ xs$
and *dvd*: $H\ \$\$\ (0,\ 0)\ dvd\ H\ \$\$\ (i,\ 0)$
and *j0*: $\forall j{\in}\{1..{<}n\}.\ H\$\$(0,j) = 0$
and *j1n*: $j{\in}\{1..{<}n\}$
and *n*: $0{<}n$
and *id*: *is-div-op div-op*
shows $K\ \$\$\ (i,0) = 0$ **and** $K\$\$(i,j) = H\ \$\$\ (i,j)$
⟨*proof*⟩

lemma *reduce-column*:
assumes *H*: $H \in carrier\text{-}mat\ m\ n$
and *PK-H*: $(P,K) = reduce\text{-}column\ div\text{-}op\ H$
and *m*: $0 < m$
shows $P \in carrier\text{-}mat\ m\ m \wedge K \in carrier\text{-}mat\ m\ n \wedge P * H = K \wedge invertible\text{-}mat\ P$
⟨*proof*⟩

lemma *reduce-column-preserves*:
assumes *H*: $H \in carrier\text{-}mat\ m\ n$
and *PK-H*: $(P,K) = reduce\text{-}column\ div\text{-}op\ H$
and *m*: $0 < m$
and $i{\in}\{0,1\}$
and $i{<}m$
shows $Matrix.row\ K\ i = Matrix.row\ H\ i$
⟨*proof*⟩

lemma *reduce-column-preserves2*:
assumes *H*: $H \in carrier\text{-}mat\ m\ n$
and *PK-H*: $(P,K) = reduce\text{-}column\ div\text{-}op\ H$
and *m*: $0 < m$ **and** *i*: $i{\in}\{0,1\}$ **and** *im*: $i{<}m$ **and** *j*: $j{<}n$
shows $K\ \$\$\ (i,j) = H\ \$\$\ (i,j)$
⟨*proof*⟩

corollary *reduce-column-works*:
assumes *H*: $H \in$ *carrier-mat m n*
and *PK-H*: $(P,K) =$ *reduce-column div-op H*
and *m*: $0 < m$
and *dvd*: *H $$ (0, 0) dvd H $$ (i, 0)*
and *j0*: $\forall j \in \{1..<n\}$. *H $$ (0, j) = 0*
and *j1n*: $j \in \{1..<n\}$
and *n*: $0<n$
and $i \in \{2..<m\}$
and *id*: *is-div-op div-op*
shows *K $$ (i,0) = 0* **and** *K$$(i,j) = H $$ (i,j)*
⟨*proof*⟩

end

16.3 The implementation

We define a locale where we implement the algorithm. It has three fixed operations:

1. an operation to transform any 1×2 matrix into its Smith normal form
2. an operation to transform any 2×2 matrix into its Smith normal form
3. an operation that provides a witness for division (this operation always exists over a commutative ring with unit, but maybe we cannot provide a computable algorithm).

Since we are working in a commutative ring, we can easily get an operation for 2×1 matrices via the 1×2 operation.

locale *Smith-Impl* =
fixes *Smith-1x2* :: ('*a*::*comm-ring-1*) *mat* ⇒ ('*a mat* × '*a mat*)
and *Smith-2x2* :: '*a mat* ⇒ ('*a mat* × '*a mat* × '*a mat*)
and *div-op* :: '*a*⇒'*a*⇒'*a*
assumes *SNF-1x2-works*: $\forall$ (*A*::'*a mat*) ∈ *carrier-mat 1 2*. *is-SNF A* (1_m *1*, (*Smith-1x2 A*))
and *SNF-2x2-works*: $\forall$ (*A*::'*a mat*) ∈ *carrier-mat 2 2*. *is-SNF A* (*Smith-2x2 A*)
and *id*: *is-div-op div-op*
begin

From a 2×2 matrix (the B), we construct the identity matrix of size n with the elements of B placed to modify the first element of a matrix and the element in position (k, k)

definition *make-mat n k* (*B*::'*a mat*) = (*Matrix.mat n n* (λ(*i*,*j*). *if i = 0 ∧ j = 0 then B$$(0,0) else*
if i = 0 ∧ j = k then B$$(0,1) else if i=k ∧ j = 0
then B$$(1,0) else if i=k ∧ j=k then B$$(1,1)

else if i=j then 1 else 0))

lemma *make-mat-carrier*[*simp*]:
shows *make-mat n k B* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *upper-triangular-mat-delete-make-mat*:
shows *upper-triangular* (*mat-delete* (*make-mat n k B*) *0 0*)
⟨*proof*⟩

lemma *upper-triangular-mat-delete-make-mat2*:
assumes *kn*: *k*<*n*
shows *upper-triangular* (*mat-delete* (*mat-delete* (*make-mat n k B*) *0 k*) (*k* − *1*) *0*)
⟨*proof*⟩

corollary *det-mat-delete-make-mat*:
assumes *kn*: *k*<*n*
shows *Determinant.det* (*mat-delete* (*mat-delete* (*make-mat n k B*) *0 k*) (*k* − *1*) *0*) = *1*
⟨*proof*⟩

lemma *swaprows-make-mat*:
assumes *B*: *B* ∈ *carrier-mat 2 2* **and** *k0*: *k*≠*0* **and** *k*: *k*<*n*
shows *swaprows k 0* (*make-mat n k B*) = *make-mat n k* (*swaprows 1 0 B*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cofactor-make-mat-00*:
assumes *k*: *k*<*n* **and** *k0*: *k*≠*0*
shows *cofactor* (*make-mat n k B*) *0 0* = *B* $$ (*1*,*1*)
⟨*proof*⟩

lemma *cofactor-make-mat-0k*:
assumes *kn*: *k*<*n* **and** *k0*: *k*≠*0* **and** *n0*: *1*<*n*
shows *cofactor* (*make-mat n k B*) *0 k* = − *B* $$ (*1*,*0*)
⟨*proof*⟩

lemma *invertible-make-mat*:
assumes *inv-B*: *invertible-mat B* **and** *B*: *B* ∈ *carrier-mat 2 2*
and *kn*: *k*<*n* **and** *k0*: *k*≠*0*
shows *invertible-mat* (*make-mat n k B*)
⟨*proof*⟩

lemma *make-mat-index*:

assumes i: $i<n$ **and** j: $j<n$
shows *make-mat n k B* $$ $(i,j) = (if\ i = 0 \wedge j = 0\ then\ B\$\$(0,0)\ else$
if $i = 0 \wedge j = k$ *then* $B\$\$(0,1)$ *else if* $i=k \wedge j = 0$
then $B\$\$(1,0)$ *else if* $i=k \wedge j=k$ *then* $B\$\$(1,1)$
else if $i=j$ *then 1 else 0*)
⟨*proof*⟩

lemma *make-mat-works*:
assumes A: $A \in carrier\text{-}mat\ m\ n$ **and** *Suc-i-less-n*: $Suc\ i < n$
and *Q-step-def*: *Q-step* = (*make-mat n* (*Suc i*) (*snd* (*Smith-1x2*
(*Matrix.mat 1 2* ($\lambda(a,b)$. *if b = 0 then A* $$ (*0*,*0*) *else A* $$(*0*,*Suc i*))))))
shows *A* $$ (*0*,*0*) ∗ *Q-step* $$ (*0*,(*Suc i*)) + *A* $$ (*0*, *Suc i*) ∗ *Q-step* $$ (*Suc i*, *Suc i*) = *0*
⟨*proof*⟩

16.3.1 Case $1 \times n$

fun *Smith-1xn-aux* :: *nat* ⇒ *'a mat* ⇒ (*'a mat* × *'a mat*) ⇒ (*'a mat* × *'a mat*)
where
Smith-1xn-aux 0 A (*S*,*Q*) = (*S*,*Q*) |
Smith-1xn-aux (*Suc i*) *A* (*S*,*Q*) = (*let*
A-step-1x2 = (*Matrix.mat 1 2* ($\lambda(a,b)$. *if b = 0 then S* $$ (*0*,*0*) *else S* $$(*0*,*Suc i*)));
(*S-step-1x2*, *Q-step-1x2*) = *Smith-1x2 A-step-1x2*;
Q-step = *make-mat* (*dim-col A*) (*Suc i*) *Q-step-1x2*;
S' = *S* ∗ *Q-step*
in Smith-1xn-aux i A (*S'*,*Q*∗*Q-step*))

definition *Smith-1xn A* = (*if dim-col A = 0 then* (*A*,1_m (*dim-col A*))
else Smith-1xn-aux (*dim-col A − 1*) *A* (*A*,1_m (*dim-col A*)))

lemma *Smith-1xn-aux-Q-carrier*:
assumes *r*: (*S'*,*Q'*) = (*Smith-1xn-aux i A* (*S*,*Q*))
assumes *A*: *A* ∈ *carrier-mat 1 n* **and** *Q*: *Q* ∈ *carrier-mat n n*
shows *Q'* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *Smith-1xn-aux-invertible-Q*:
assumes *r*: (*S'*,*Q'*) = (*Smith-1xn-aux i A* (*S*,*Q*))
assumes *A*: *A* ∈ *carrier-mat 1 n* **and** *Q*: *Q* ∈ *carrier-mat n n*
and *i*: $i<n$ **and** *inv-Q*: *invertible-mat Q*
shows *invertible-mat Q'*
⟨*proof*⟩

lemma *Smith-1xn-aux-S'-AQ'*:
assumes *r*: (*S'*,*Q'*) = (*Smith-1xn-aux i A* (*S*,*Q*))
assumes *A*: *A* ∈ *carrier-mat 1 n* **and** *S*: *S* ∈ *carrier-mat 1 n* **and** *Q*: *Q* ∈ *carrier-mat n n*

and *S-AQ*: $S = A*Q$ **and** *i*: $i<n$
shows $S' = A * Q'$
⟨*proof*⟩

lemma *Smith-1xn-aux-S'-works*:
assumes *r*: $(S',Q') =$ (*Smith-1xn-aux i A* (S,Q))
assumes *A*: $A \in$ *carrier-mat 1 n* **and** *S*: $S \in$ *carrier-mat 1 n* **and** *Q*: $Q \in$ *carrier-mat n n*
and *S-AQ*: $S = A*Q$ **and** *i*: $i<n$ **and** *j0*: $0<j$ **and** *jn*: $j<n$
and *all-j-zero*: $\forall j \in \{i+1..<n\}$. $S \$\$(0,j) = 0$
shows $S' \$\$ (0,j) = 0$
⟨*proof*⟩

lemma *Smith-1xn-works*:
assumes *A*: $A \in$ *carrier-mat 1 n*
and *SQ*: $(S,Q) =$ *Smith-1xn A*
shows *is-SNF A* $(1_m\ 1,\ S,Q)$
⟨*proof*⟩

16.3.2 Case $n \times 1$

definition *Smith-nx1 A* =
(*let* $(S,P) =$ (*Smith-1xn-aux* (*dim-row A* − *1*) (*transpose-mat A*) (*transpose-mat A*,1_m (*dim-row A*)))
in (*transpose-mat P*, *transpose-mat S*))

lemma *Smith-nx1-works*:
assumes *A*: $A \in$ *carrier-mat n 1*
and *SQ*: $(P,S) =$ *Smith-nx1 A*
shows *is-SNF A* $(P,\ S,1_m\ 1)$
⟨*proof*⟩

16.3.3 Case $2 \times n$

function *Smith-2xn* :: *'a mat* ⇒ (*'a mat* × *'a mat* × *'a mat*)
where
Smith-2xn A = (
if dim-col A = *0 then* (1_m (*dim-row A*),*A*,1_m *0*) *else*
if dim-col A = *1 then let* $(P,S) =$ *Smith-nx1 A in* $(P,S,\ 1_m$ (*dim-col A*)) *else*
if dim-col A = *2 then Smith-2x2 A*
else
let A1 = *mat-of-cols* (*dim-row A*) [*col A 0*];
A2 = *mat-of-cols* (*dim-row A*) [*col A i*. $i \leftarrow [1..<$*dim-col A*]];
$(P1,D1,Q1) =$ *Smith-2xn A2*;
C = (*P1*∗*A1*) $@_c$ (*P1*∗*A2*∗*Q1*);
D = *mat-of-cols* (*dim-row A*) [*col C 0*, *col C 1*];
E = *mat-of-cols* (*dim-row A*) [*col C i*. $i \leftarrow [2..<$*dim-col A*]];
$(P2,D2,Q2) =$ *Smith-2x2 D*;

$H = (P2{*}D{*}Q2)\ @_c\ (P2 * E)$;
$k = (div\text{-}op\ (H\ \$\$\ (0,2))\ (H\ \$\$\ (0,0)))$;
$H2 = addcol\ (-k)\ 2\ 0\ H$;
$(\text{-},\text{-},\text{-},H2\text{-}DR) = split\text{-}block\ H2\ 1\ 1$;
$(H\text{-}1xn, Q3) = Smith\text{-}1xn\ H2\text{-}DR$;
$S = four\text{-}block\text{-}mat\ (Matrix.mat\ 1\ 1\ (\lambda(a,b).\ H\$\$(0,0)))\ (0_m\ 1\ (dim\text{-}col\ A - 1))\ (0_m\ 1\ 1)\ H\text{-}1xn$;
$Q1' = four\text{-}block\text{-}mat\ (1_m\ 1)\ (0_m\ 1\ (dim\text{-}col\ A - 1))\ (0_m\ (dim\text{-}col\ A - 1)\ 1)\ Q1$;
$Q2' = four\text{-}block\text{-}mat\ Q2\ (0_m\ 2\ (dim\text{-}col\ A - 2))\ (0_m\ (dim\text{-}col\ A - 2)\ 2)\ (1_m\ (dim\text{-}col\ A - 2))$;
$Q\text{-}div\text{-}k = addrow\text{-}mat\ (dim\text{-}col\ A)\ (-k)\ 0\ 2$;
$Q3' = four\text{-}block\text{-}mat\ (1_m\ 1)\ (0_m\ 1\ (dim\text{-}col\ A - 1))\ (0_m\ (dim\text{-}col\ A - 1)\ 1)\ Q3$
in $(P2 * P1, S, Q1' * Q2' * Q\text{-}div\text{-}k * Q3')$)
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *Smith-2xn-0*:
assumes *A*: $A \in carrier\text{-}mat\ 2\ 0$
shows $is\text{-}SNF\ A\ (Smith\text{-}2xn\ A)$
⟨*proof*⟩

lemma *Smith-2xn-1*:
assumes *A*: $A \in carrier\text{-}mat\ 2\ 1$
shows $is\text{-}SNF\ A\ (Smith\text{-}2xn\ A)$
⟨*proof*⟩

lemma *Smith-2xn-2*:
assumes *A*: $A \in carrier\text{-}mat\ 2\ 2$
shows $is\text{-}SNF\ A\ (Smith\text{-}2xn\ A)$
⟨*proof*⟩

lemma *is-SNF-Smith-2xn-n-ge-2*:
assumes *A*: $A \in carrier\text{-}mat\ 2\ n$ **and** *n*: $n>2$
shows $is\text{-}SNF\ A\ (Smith\text{-}2xn\ A)$
⟨*proof*⟩

lemma *is-SNF-Smith-2xn*:
assumes *A*: $A \in carrier\text{-}mat\ 2\ n$
shows $is\text{-}SNF\ A\ (Smith\text{-}2xn\ A)$
⟨*proof*⟩

16.3.4 Case $n \times 2$

definition $Smith\text{-}nx2\ A = (let\ (P,S,Q) = Smith\text{-}2xn\ A^T\ in$
$(Q^T,\ S^T,\ P^T))$

lemma *is-SNF-Smith-nx2*:
assumes *A*: $A \in$ *carrier-mat n 2*
shows *is-SNF A* (*Smith-nx2 A*)
⟨*proof*⟩

16.3.5 Case $m \times n$

declare *Smith-2xn.simps*[*simp del*]

function (*domintros*) *Smith-mxn* :: *'a mat* ⇒ (*'a mat* × *'a mat* × *'a mat*)
where
Smith-mxn A = (
if dim-row A = *0* ∨ *dim-col A* = *0 then* (*1*$_m$ (*dim-row A*),*A*,*1*$_m$ (*dim-col A*))
else if dim-row A = *1 then* (*1*$_m$ *1*, *Smith-1xn A*)
else if dim-row A = *2 then Smith-2xn A*
else if dim-col A = *1 then let* (*P*,*S*) = *Smith-nx1 A in* (*P*,*S*,*1*$_m$ *1*)
else if dim-col A = *2 then Smith-nx2 A*
else
let A1 = *mat-of-row* (*Matrix.row A 0*);
A2 = *mat-of-rows* (*dim-col A*) [*Matrix.row A i*. *i* ← [*1*..<*dim-row A*]];
(*P1*,*D1*,*Q1*) = *Smith-mxn A2*;
C = (*A1*∗*Q1*) @$_r$ (*P1*∗*A2*∗*Q1*);
D = *mat-of-rows* (*dim-col A*) [*Matrix.row C 0*, *Matrix.row C 1*];
E = *mat-of-rows* (*dim-col A*) [*Matrix.row C i*. *i* ← [*2*..<*dim-row A*]];
(*P2*,*F*,*Q2*) = *Smith-2xn D*;
H = (*P2*∗*D*∗*Q2*) @$_r$ (*E*∗*Q2*);
(*P-H2*, *H2*) = *reduce-column div-op H*;
(*H2-UL*, *H2-UR*, *H2-DL*, *H2-DR*) = *split-block H2 1 1*;
(*P3*,*S'*,*Q3*) = *Smith-mxn H2-DR*;
S = *four-block-mat* (*Matrix.mat 1 1* (λ(*a*, *b*). *H* $$ (*0*, *0*))) (*0*$_m$ *1* (*dim-col A* − *1*)) (*0*$_m$ (*dim-row A* − *1*) *1*) *S'*;
P1' = *four-block-mat* (*1*$_m$ *1*) (*0*$_m$ *1* (*dim-row A* − *1*)) (*0*$_m$ (*dim-row A* − *1*) *1*) *P1*;
P2' = *four-block-mat P2* (*0*$_m$ *2* (*dim-row A* − *2*)) (*0*$_m$ (*dim-row A* − *2*) *2*) (*1*$_m$ (*dim-row A* − *2*));
P3' = *four-block-mat* (*1*$_m$ *1*) (*0*$_m$ *1* (*dim-row A* − *1*)) (*0*$_m$ (*dim-row A* − *1*) *1*) *P3*;
Q3' = *four-block-mat* (*1*$_m$ *1*) (*0*$_m$ *1* (*dim-col A* − *1*)) (*0*$_m$ (*dim-col A* − *1*) *1*) *Q3*
in (*P3'* ∗ *P-H2* ∗ *P2'* ∗ *P1'*,*S*, *Q1* ∗ *Q2* ∗ *Q3'*)
)
⟨*proof*⟩

declare *Smith-2xn.simps*[*simp*]

lemma *Smith-mxn-dom-nm-less-2*:
assumes *A*: $A \in$ *carrier-mat m n* **and** *mn*: $n \leq 2 \vee m \leq 2$
shows *Smith-mxn-dom A*
⟨*proof*⟩

lemma *Smith-mxn-pinduct-carrier-less-2*:
assumes *A*: $A \in$ *carrier-mat m n* **and** *mn*: $n \leq 2 \vee m \leq 2$
shows *fst* (*Smith-mxn A*) ∈ *carrier-mat m m*
∧ *fst* (*snd* (*Smith-mxn A*)) ∈ *carrier-mat m n*
∧ *snd* (*snd* (*Smith-mxn A*)) ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *Smith-mxn-pinduct-carrier-ge-2*: ⟦*Smith-mxn-dom A*; $A \in$ *carrier-mat m n*; $m > 2$; $n > 2$⟧ ⟹
fst (*Smith-mxn A*) ∈ *carrier-mat m m*
∧ *fst* (*snd* (*Smith-mxn A*)) ∈ *carrier-mat m n*
∧ *snd* (*snd* (*Smith-mxn A*)) ∈ *carrier-mat n n*
⟨*proof*⟩

corollary *Smith-mxn-pinduct-carrier*: ⟦*Smith-mxn-dom A*; $A \in$ *carrier-mat m n*⟧ ⟹
fst (*Smith-mxn A*) ∈ *carrier-mat m m*
∧ *fst* (*snd* (*Smith-mxn A*)) ∈ *carrier-mat m n*
∧ *snd* (*snd* (*Smith-mxn A*)) ∈ *carrier-mat n n*
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *is-SNF-Smith-mxn-less-2*:
assumes *A*: $A \in$ *carrier-mat m n* **and** *mn*: $n \leq 2 \vee m \leq 2$
shows *is-SNF A* (*Smith-mxn A*)
⟨*proof*⟩

lemma *is-SNF-Smith-mxn-ge-2*:
assumes *A*: $A \in$ *carrier-mat m n* **and** *m*: $m > 2$ **and** *n*: $n > 2$
shows *is-SNF A* (*Smith-mxn A*)
⟨*proof*⟩

16.4 Soundness theorem

theorem *is-SNF-Smith-mxn*:
assumes *A*: $A \in$ *carrier-mat m n*
shows *is-SNF A* (*Smith-mxn A*)
⟨*proof*⟩

declare *Smith-mxn.simps*[*code*]

end

declare *Smith-Impl.Smith-mxn.simps*[*code-unfold*]

definition *T-spec* :: ('*a*::{*comm-ring-1*} ⇒ '*a* ⇒ ('*a* × '*a* × '*a*)) ⇒ *bool*
where *T-spec T* = (∀ *a b*::'*a*. *let* (*a1*,*b1*,*d*) = *T a b in*
a = *a1*∗*d* ∧ *b* = *b1*∗*d* ∧ *ideal-generated* {*a1*,*b1*} = *ideal-generated* {*1*})

definition *D'-spec* :: ('*a*::{*comm-ring-1*} ⇒ '*a* ⇒ '*a* ⇒ ('*a* × '*a*)) ⇒ *bool*
where *D'-spec D'* = (∀ *a b c*::'*a*. *let* (*p*,*q*) = *D' a b c in*
ideal-generated{*a*,*b*,*c*} = *ideal-generated*{*1*}
⟶ *ideal-generated* {*p*∗*a*,*p*∗*b*+*q*∗*c*} = *ideal-generated* {*1*})

end

17 The Smith normal form algorithm in HOL Analysis

theory *SNF-Algorithm-HOL-Analysis*
imports
SNF-Algorithm
Admits-SNF-From-Diagonal-Iff-Bezout-Ring
begin

17.1 Transferring the result from JNF to HOL Anaylsis

definition *Smith-mxn-HMA* :: (('*a*::*comm-ring-1*^2) ⇒ (('*a*^2) × ('*a*^2^2)))
⇒ (('*a*^2^2) ⇒ (('*a*^2^2) × ('*a*^2^2) × ('*a*^2^2))) ⇒ ('*a*⇒'*a*⇒'*a*) ⇒ ('*a*^'*n*::*mod-type*^'*m*::*mod-type*)

⇒ (('*a*^'*m*::*mod-type*^'*m*::*mod-type*)× ('*a*^'*n*::*mod-type*^'*m*::*mod-type*) × ('*a*^'*n*::*mod-type*^'*n*::*mod-type*))
where
Smith-mxn-HMA Smith-1x2 Smith-2x2 div-op A =
(*let Smith-1x2-JNF* = (λ*A'*. *let* (*S'*,*Q'*) = *Smith-1x2* (*Mod-Type-Connect.to-hma*$_v$
(*Matrix.row A' 0*))
in (*mat-of-row* (*Mod-Type-Connect.from-hma*$_v$ *S'*),
Mod-Type-Connect.from-hma$_m$ *Q'*));
Smith-2x2-JNF = (λ*A'*. *let* (*P'*, *S'*,*Q'*) = *Smith-2x2* (*Mod-Type-Connect.to-hma*$_m$
A')
in (*Mod-Type-Connect.from-hma*$_m$ *P'*, *Mod-Type-Connect.from-hma*$_m$
S', *Mod-Type-Connect.from-hma*$_m$ *Q'*));
(*P*,*S*,*Q*) = *Smith-Impl.Smith-mxn Smith-1x2-JNF Smith-2x2-JNF div-op*
(*Mod-Type-Connect.from-hma*$_m$ *A*)
in (*Mod-Type-Connect.to-hma*$_m$ *P*, *Mod-Type-Connect.to-hma*$_m$ *S*, *Mod-Type-Connect.to-hma*$_m$
Q)
)

definition *is-SNF-HMA A R* = (*case R of* (*P*,*S*,*Q*) ⇒
invertible P ∧ *invertible Q*
∧ *Smith-normal-form S* ∧ *S* = *P* ∗∗ *A* ∗∗ *Q*)

17.2 Soundness in HOL Anaylsis

lemma *is-SNF-Smith-mxn-HMA*:
fixes *A*::′*a*::*comm-ring-1* ^ ′*n*::*mod-type* ^ ′*m*::*mod-type*
assumes *PSQ*: (*P*,*S*,*Q*) = *Smith-mxn-HMA Smith-1x2 Smith-2x2 div-op A*
and *SNF-1x2-works*: ∀ *A*. *let* (*S*′,*Q*) = *Smith-1x2 A in S*′ $*h* 1 = 0 ∧ *invertible Q* ∧ *S*′ = *A* *v*∗ *Q*
and *SNF-2x2-works*: ∀ *A*. *is-SNF-HMA A* (*Smith-2x2 A*)
and *d*: *is-div-op div-op*
shows *is-SNF-HMA A* (*P*,*S*,*Q*)
⟨*proof*⟩
end

18 Elementary divisor rings

theory *Elementary-Divisor-Rings*
imports
SNF-Algorithm
Rings2-Extended
begin

This theory contains the definition of elementary divisor rings and Hermite rings, as well as the corresponding relation between both concepts. It also includes a complete characterization for elementary divisor rings, by means of an *if and only if*-statement.

The results presented here follows the article “Some remarks about elementary divisor rings” by Leonard Gillman and Melvin Henriksen.

18.1 Previous definitions and basic properties of Hermite ring

definition *admits-triangular-reduction A* =
(∃ *U*::′*a*::*comm-ring-1 mat*. *U* ∈ *carrier-mat* (*dim-col A*) (*dim-col A*)
∧ *invertible-mat U* ∧ *lower-triangular* (*A*∗*U*))

class *Hermite-ring* =
assumes ∀ (*A*::′*a*::*comm-ring-1 mat*). *admits-triangular-reduction A*

lemma *admits-triangular-reduction-intro*:
assumes *invertible-mat* (*U*::′*a*::*comm-ring-1 mat*)
and *U* ∈ *carrier-mat* (*dim-col A*) (*dim-col A*)
and *lower-triangular* (*A*∗*U*)

shows *admits-triangular-reduction A*
⟨*proof*⟩

lemma *OFCLASS-Hermite-ring-def*:
OFCLASS('*a*::*comm-ring-1*, *Hermite-ring-class*)
≡ (⋀(*A*::'*a*::*comm-ring-1 mat*). *admits-triangular-reduction A*)
⟨*proof*⟩

definition *admits-diagonal-reduction*::'*a*::*comm-ring-1 mat* ⇒ *bool*
where *admits-diagonal-reduction A* = (∃ *P Q*. *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) ∧
Q ∈ *carrier-mat* (*dim-col A*) (*dim-col A*)
∧ *invertible-mat P* ∧ *invertible-mat Q*
∧ *Smith-normal-form-mat* (*P* ∗ *A* ∗ *Q*))

lemma *admits-diagonal-reduction-intro*:
assumes *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*)
and *Q* ∈ *carrier-mat* (*dim-col A*) (*dim-col A*)
and *invertible-mat P* **and** *invertible-mat Q*
and *Smith-normal-form-mat* (*P* ∗ *A* ∗ *Q*)
shows *admits-diagonal-reduction A* ⟨*proof*⟩

lemma *admits-diagonal-reduction-imp-exists-algorithm-is-SNF*:
assumes *A* ∈ *carrier-mat m n*
and *admits-diagonal-reduction A*
shows ∃ *algorithm*. *is-SNF A* (*algorithm A*)
⟨*proof*⟩

lemma *exists-algorithm-is-SNF-imp-admits-diagonal-reduction*:
assumes *A* ∈ *carrier-mat m n*
and ∃ *algorithm*. *is-SNF A* (*algorithm A*)
shows *admits-diagonal-reduction A*
⟨*proof*⟩

lemma *admits-diagonal-reduction-eq-exists-algorithm-is-SNF*:
assumes *A*: *A* ∈ *carrier-mat m n*
shows *admits-diagonal-reduction A* = (∃ *algorithm*. *is-SNF A* (*algorithm A*))
⟨*proof*⟩

lemma *admits-diagonal-reduction-imp-exists-algorithm-is-SNF-all*:
assumes (∀ (*A*::'*a*::*comm-ring-1 mat*) ∈ *carrier-mat m n*. *admits-diagonal-reduction A*)
shows (∃ *algorithm*. ∀ (*A*::'*a mat*) ∈ *carrier-mat m n*. *is-SNF A* (*algorithm A*))
⟨*proof*⟩

lemma *exists-algorithm-is-SNF-imp-admits-diagonal-reduction-all*:
assumes $(\exists\, algorithm.\ \forall\, (A::'a\ mat) \in carrier\text{-}mat\ m\ n.\ is\text{-}SNF\ A\ (algorithm\ A))$
shows $(\forall\, (A::'a::comm\text{-}ring\text{-}1\ mat) \in carrier\text{-}mat\ m\ n.\ admits\text{-}diagonal\text{-}reduction\ A)$
⟨*proof*⟩

lemma *admits-diagonal-reduction-eq-exists-algorithm-is-SNF-all*:
shows $(\forall\, (A::'a::comm\text{-}ring\text{-}1\ mat) \in carrier\text{-}mat\ m\ n.\ admits\text{-}diagonal\text{-}reduction\ A)$
$= (\exists\, algorithm.\ \forall\, (A::'a\ mat) \in carrier\text{-}mat\ m\ n.\ is\text{-}SNF\ A\ (algorithm\ A))$
⟨*proof*⟩

18.2 The class that represents elementary divisor rings

class *elementary-divisor-ring* =
assumes $\forall\, (A::'a::comm\text{-}ring\text{-}1\ mat).\ admits\text{-}diagonal\text{-}reduction\ A$

lemma *dim-row-mat-diag*[*simp*]: $dim\text{-}row\ (mat\text{-}diag\ n\ f) = n$ **and**
dim-col-mat-diag[*simp*]: $dim\text{-}col\ (mat\text{-}diag\ n\ f) = n$
⟨*proof*⟩

18.3 Hermite ring implies Bézout ring

To prove this fact, we make use of the alternative definition for Bézout rings: each finitely generated ideal is principal

lemma *Hermite-ring-imp-Bezout-ring*:
assumes H: $OFCLASS('a::comm\text{-}ring\text{-}1,\ Hermite\text{-}ring\text{-}class)$
shows $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class)$
⟨*proof*⟩

18.4 Elementary divisor ring implies Hermite ring

context
assumes $SORT\text{-}CONSTRAINT('a::comm\text{-}ring\text{-}1)$
begin

lemma *triangularizable-m0*:
assumes A: $A \in carrier\text{-}mat\ m\ 0$
shows $\exists\, U.\ U \in carrier\text{-}mat\ 0\ 0 \wedge invertible\text{-}mat\ U \wedge lower\text{-}triangular\ (A * U)$
⟨*proof*⟩

lemma *triangularizable-0n*:
assumes A: $A \in carrier\text{-}mat\ 0\ n$
shows $\exists\, U.\ U \in carrier\text{-}mat\ n\ n \wedge invertible\text{-}mat\ U \wedge lower\text{-}triangular\ (A * U)$
⟨*proof*⟩

lemma *diagonal-imp-triangular-1x2*:
assumes *A*: $A \in$ *carrier-mat 1 2* **and** *d*: *admits-diagonal-reduction* (A::'*a mat*)
shows *admits-triangular-reduction A*
⟨*proof*⟩

lemma *triangular-imp-diagonal-1x2*:
assumes *A*: $A \in$ *carrier-mat 1 2* **and** *t*: *admits-triangular-reduction* (A::'*a mat*)
shows *admits-diagonal-reduction A*
⟨*proof*⟩

lemma *triangular-eq-diagonal-1x2*:
($\forall A \in$*carrier-mat 1 2*. *admits-triangular-reduction* (A::'*a mat*))
= ($\forall A \in$*carrier-mat 1 2*. *admits-diagonal-reduction* (A::'*a mat*))
⟨*proof*⟩

lemma *admits-triangular-mat-1x1*:
assumes *A*: $A \in$ *carrier-mat 1 1*
shows *admits-triangular-reduction* (A::'*a mat*)
⟨*proof*⟩

lemma *admits-diagonal-mat-1x1*:
assumes *A*: $A \in$ *carrier-mat 1 1*
shows *admits-diagonal-reduction* (A::'*a mat*)
⟨*proof*⟩

lemma *admits-diagonal-imp-admits-triangular-1xn*:
assumes *a*: $\forall A \in$*carrier-mat 1 2*. *admits-diagonal-reduction* (A::'*a mat*)
shows $\forall A \in$*carrier-mat 1 n*. *admits-triangular-reduction* (A::'*a mat*)
⟨*proof*⟩

lemma *admits-diagonal-imp-admits-triangular*:
assumes *a*: $\forall A \in$*carrier-mat 1 2*. *admits-diagonal-reduction* (A::'*a mat*)
shows $\forall A$. *admits-triangular-reduction* (A::'*a mat*)
⟨*proof*⟩

corollary *admits-diagonal-imp-admits-triangular'*:
assumes *a*: $\forall A$. *admits-diagonal-reduction* (A::'*a mat*)
shows $\forall A$. *admits-triangular-reduction* (A::'*a mat*)
⟨*proof*⟩

lemma *admits-triangular-reduction-1x2*:
assumes $\forall A$::'*a mat*. $A \in$ *carrier-mat 1 2* $\longrightarrow$ *admits-triangular-reduction A*

shows $\forall\, C{::}'a\ mat.\ admits\text{-}triangular\text{-}reduction\ C$
⟨*proof*⟩

lemma *Hermite-ring-OFCLASS*:
assumes $\forall\, A \in carrier\text{-}mat\ 1\ 2.\ admits\text{-}triangular\text{-}reduction\ (A{::}'a\ mat)$
shows *OFCLASS*('*a*, *Hermite-ring-class*)
⟨*proof*⟩

lemma *Hermite-ring-OFCLASS'*:
assumes $\forall\, A \in carrier\text{-}mat\ 1\ 2.admits\text{-}diagonal\text{-}reduction\ (A{::}'a\ mat)$
shows *OFCLASS*('*a*, *Hermite-ring-class*)
⟨*proof*⟩

lemma *theorem3-part1*:
assumes *T*: $(\forall\, a\ b{::}'a.\ \exists\ a1\ b1\ d.\ a = a1{*}d \wedge b = b1{*}d$
$\wedge\ ideal\text{-}generated\ \{a1,b1\} = ideal\text{-}generated\ \{1\})$
shows $\forall\, A{::}'a\ mat.\ admits\text{-}triangular\text{-}reduction\ A$
⟨*proof*⟩

lemma *theorem3-part2*:
assumes *1*: $\forall\, A{::}'a\ mat.\ admits\text{-}triangular\text{-}reduction\ A$
shows $\forall\, a\ b{::}'a.\ \exists\ a1\ b1\ d.\ a = a1{*}d \wedge b = b1{*}d \wedge ideal\text{-}generated\ \{a1,b1\} = ideal\text{-}generated\ \{1\}$
⟨*proof*⟩

theorem *theorem3*:
shows $(\forall\, A{::}'a\ mat.\ admits\text{-}triangular\text{-}reduction\ A)$
$= (\forall\, a\ b{::}'a.\ \exists\ a1\ b1\ d.\ a = a1{*}d \wedge b = b1{*}d \wedge ideal\text{-}generated\ \{a1,b1\} = ideal\text{-}generated\ \{1\})$
⟨*proof*⟩

end

context *comm-ring-1*
begin

lemma *lemma4-prev*:
assumes *a*: $a = a1{*}d$ **and** *b*: $b = b1{*}d$
and *i*: $ideal\text{-}generated\ \{a1,b1\} = ideal\text{-}generated\ \{1\}$
shows $ideal\text{-}generated\ \{a,b\} = ideal\text{-}generated\ \{d\}$
⟨*proof*⟩

lemma *lemma4*:
assumes *a*: $a = a1*d$ **and** *b*: $b = b1*d$
and *i*: *ideal-generated* $\{a1,b1\}$ = *ideal-generated* $\{1\}$
and *i2*: *ideal-generated* $\{a,b\}$ = *ideal-generated* $\{d'\}$
shows $\exists\, a1'\ b1'.\ a = a1' * d' \wedge b = b1' * d'$
$\wedge$ *ideal-generated* $\{a1',b1'\}$ = *ideal-generated* $\{1\}$
⟨*proof*⟩

lemma *corollary5*:
assumes *T*: $\forall\, a\ b.\ \exists\, a1\ b1\ d.\ a = a1 * d \wedge b = b1 * d$
$\wedge$ *ideal-generated* $\{a1,\ b1\}$ = *ideal-generated* $\{1::'a\}$
and *i2*: *ideal-generated* $\{a,b,c\}$ = *ideal-generated* $\{d\}$
shows $\exists\ a1\ b1\ c1.\ a = a1 * d \wedge b = b1 * d \wedge c = c1 * d$
$\wedge$ *ideal-generated* $\{a1,b1,c1\}$ = *ideal-generated* $\{1\}$
⟨*proof*⟩

end

context
assumes *SORT-CONSTRAINT*(*'a*::*comm-ring-1*)
begin

lemma *OFCLASS-elementary-divisor-ring-imp-class*:
assumes *OFCLASS*(*'a*::*comm-ring-1*, *elementary-divisor-ring-class*)
shows *class.elementary-divisor-ring TYPE*(*'a*)
⟨*proof*⟩

corollary *Elementary-divisor-ring-imp-Hermite-ring*:
assumes *OFCLASS*(*'a*::*comm-ring-1*, *elementary-divisor-ring-class*)
shows *OFCLASS*(*'a*::*comm-ring-1*, *Hermite-ring-class*)
⟨*proof*⟩

corollary *Elementary-divisor-ring-imp-Bezout-ring*:
assumes *OFCLASS*(*'a*::*comm-ring-1*, *elementary-divisor-ring-class*)
shows *OFCLASS*(*'a*::*comm-ring-1*, *bezout-ring-class*)
⟨*proof*⟩

18.5 Characterization of Elementary divisor rings

lemma *necessity-D'*:
assumes *edr*: $(\forall\, (A::'a\ mat).\ admits\text{-}diagonal\text{-}reduction\ A)$
shows $\forall\, a\ b\ c::'a.$ *ideal-generated* $\{a,b,c\}$ = *ideal-generated*$\{1\}$
$\longrightarrow$ $(\exists\, p\ q.$ *ideal-generated* $\{p*a,p*b+q*c\}$ = *ideal-generated* $\{1\})$

⟨*proof*⟩

lemma *necessity*:
assumes (∀ (*A*::'*a mat*). *admits-diagonal-reduction A*)
shows (∀ (*A*::'*a mat*). *admits-triangular-reduction A*)
and ∀ *a b c*::'*a*. *ideal-generated*{*a*,*b*,*c*} = *ideal-generated*{*1*}
⟶ (∃ *p q*. *ideal-generated* {*p*∗*a*,*p*∗*b*+*q*∗*c*} = *ideal-generated* {*1*})
⟨*proof*⟩

In the article, the authors change the notation and assume $(a, b, c) = (1)$. However, we have to provide here the complete prove. To to this, I obtained a D matrix such that $A' = A * D$ and D is a diagonal matrix with d in the diagonal. Proving that D is left and right commutative, I can follow the reasoning in the article

lemma *sufficiency*:
assumes *hermite-ring*: (∀ (*A*::'*a mat*). *admits-triangular-reduction A*)
and *D'*: ∀ *a b c*::'*a*. *ideal-generated*{*a*,*b*,*c*} = *ideal-generated*{*1*}
⟶ (∃ *p q*. *ideal-generated* {*p*∗*a*,*p*∗*b*+*q*∗*c*} = *ideal-generated* {*1*})
shows (∀ (*A*::'*a mat*). *admits-diagonal-reduction A*)
⟨*proof*⟩

18.6 Final theorem

theorem *edr-characterization*:
(∀ (*A*::'*a mat*). *admits-diagonal-reduction A*) = ((∀ (*A*::'*a mat*). *admits-triangular-reduction A*)
∧ (∀ *a b c*::'*a*. *ideal-generated*{*a*,*b*,*c*} = *ideal-generated*{*1*}
⟶ (∃ *p q*. *ideal-generated* {*p*∗*a*,*p*∗*b*+*q*∗*c*} = *ideal-generated* {*1*})))
⟨*proof*⟩

corollary *OFCLASS-edr-characterization*:
OFCLASS('*a*, *elementary-divisor-ring-class*) ≡ (*OFCLASS*('*a*, *Hermite-ring-class*)

&&& (∀ *a b c*::'*a*. *ideal-generated*{*a*,*b*,*c*} = *ideal-generated*{*1*}
⟶ (∃ *p q*. *ideal-generated* {*p*∗*a*,*p*∗*b*+*q*∗*c*} = *ideal-generated* {*1*}))) (**is** *?lhs* ≡ *?rhs*)
⟨*proof*⟩

corollary *edr-characterization-class*:
class.elementary-divisor-ring TYPE('*a*)
= (*class.Hermite-ring TYPE*('*a*)
∧ (∀ *a b c*::'*a*. *ideal-generated*{*a*,*b*,*c*} = *ideal-generated*{*1*}
⟶ (∃ *p q*. *ideal-generated* {*p*∗*a*,*p*∗*b*+*q*∗*c*} = *ideal-generated* {*1*}))) (**is** *?lhs* = (*?H* ∧ *?D'*))
⟨*proof*⟩

corollary *edr-iff-T-D'*:
shows *class.elementary-divisor-ring TYPE('a) = (*
*(∀ a b::'a. ∃ a1 b1 d. a = a1*d ∧ b = b1*d ∧ ideal-generated {a1,b1} = ideal-generated {1})*
∧ (∀ a b c::'a. ideal-generated{a,b,c} = ideal-generated{1}
*⟶ (∃ p q. ideal-generated {p*a,p*b+q*c} = ideal-generated {1}))*
) (**is** *?lhs = (?T ∧ ?D')*)
⟨*proof*⟩

end
end

19 Executable Smith normal form algorithm over Euclidean domains

theory *SNF-Algorithm-Euclidean-Domain*
imports
Diagonal-To-Smith
Echelon-Form.Examples-Echelon-Form-Abstract

Elementary-Divisor-Rings
Diagonal-To-Smith-JNF

Mod-Type-Connect
Show.Show-Instances
Jordan-Normal-Form.Show-Matrix
Show.Show-Poly
begin

This provides an executable implementation of the verified general algorithm, provinding executable operations over a Euclidean domain.

lemma *zero-less-one-type2*: *(0::2) < 1*
⟨*proof*⟩

19.1 Previous code equations

definition *to-hma$_m$-row A i*
= (vec-lambda (λj. A $$ (Mod-Type.to-nat i, Mod-Type.to-nat j)))

lemma *bezout-matrix-row-code* [*code abstract*]:
vec-nth (to-hma$_m$-row A i) =
(λj. A $$ (Mod-Type.to-nat i, Mod-Type.to-nat j))
⟨*proof*⟩

lemma [*code abstract*]: *vec-nth (Mod-Type-Connect.to-hma$_m$ A) = to-hma$_m$-row A*
⟨*proof*⟩

19.2 An executable algorithm to transform 2×2 matrices into its Smith normal form in HOL Analysis

subclass (**in** *euclidean-ring-gcd*) *bezout-ring-div*
⟨*proof*⟩

context
  fixes *bezout*::('*a*::*euclidean-ring-gcd* ⇒ '*a* ⇒ ('*a*×'*a*×'*a*×'*a*×'*a*))
  assumes *ib*: *is-bezout-ext bezout*
begin

lemma *normalize-bezout-gcd*:
  assumes *b*: (*p*,*q*,*u*,*v*,*d*) = *bezout a b*
  shows *normalize d* = *gcd a b*
⟨*proof*⟩

end

lemma *bezout-matrix-works-transpose1*:
  assumes *ib*: *is-bezout-ext bezout*
  and *a-not-b*: *a* ≠ *b*
shows (*A*∗∗*transpose* (*bezout-matrix* (*transpose A*) *a b i bezout*)) $ *i* $ *a*
    = *snd* (*snd* (*snd* (*snd* (*bezout* (*A* $ *i* $ *a*) (*A* $ *i* $ *b*)))))
⟨*proof*⟩

lemma *invertible-bezout-matrix-transpose*:
 fixes *A*::'*a*::{*bezout-ring-div*}^'*cols*::{*finite*,*wellorder*}^'*rows*
  assumes *ib*: *is-bezout-ext bezout*
  and *a-less-b*: *a* < *b*
  and *aj*: *A* $*h i* $*h a* ≠ *0*
shows *invertible* (*transpose* (*bezout-matrix* (*transpose A*) *a b i bezout*))
⟨*proof*⟩

function *diagonalize-2x2-aux* :: (('*a*::*euclidean-ring-gcd*^*2^*2*) × ('*a*^*2^*2*)×('*a*^*2^*2*))
⇒
                                    (('*a*^*2^*2*) ×('*a*^*2^*2*)×('*a*^*2^*2*))
  where *diagonalize-2x2-aux* (*P*,*A*,*Q*) =
(
  let
      a = *A* $*h 0* $*h 0*;
      b = *A* $*h 0* $*h 1*;
      c = *A* $*h 1* $*h 0*;
      d = *A* $*h 1* $*h 1 in*
     if a≠ *0* ∧ ¬ *a dvd b then let bezout-mat* = *transpose* (*bezout-matrix* (*transpose*
A) *0 1 0 euclid-ext2*) *in*

diagonalize-2x2-aux (*P*, *A****bezout-mat*,*Q****bezout-mat*) *else*
if *a* ≠ *0* ∧ ¬ *a dvd c then let bezout-mat* = *bezout-matrix A 0 1 0 euclid-ext2*
in diagonalize-2x2-aux (*bezout-mat****P*,*bezout-mat****A*,*Q*) *else* — We can divide an get zeros
let Q′ = *column-add* (*Finite-Cartesian-Product.mat 1*) *1 0* (− (*b div a*));
P′ = *row-add* (*Finite-Cartesian-Product.mat 1*) *1 0* (− (*c div a*)) *in*
(*P′****P*,*P′****A****Q′*,*Q****Q′*)
) ⟨*proof*⟩

termination
⟨*proof*⟩

lemma *diagonalize-2x2-aux-works*:
assumes *A* = *P* ** *A-input* ** *Q*
and *invertible P* **and** *invertible Q*
and (*P′*,*D*,*Q′*) = *diagonalize-2x2-aux* (*P*,*A*,*Q*)
and *A* $*h 0* $*h 0* ≠ *0*
shows *D* = *P′* ** *A-input* ** *Q′* ∧ *invertible P′* ∧ *invertible Q′* ∧ *isDiagonal D*
⟨*proof*⟩

definition *diagonalize-2x2 A* =
(*if A* $*h 0* $*h 0* = *0 then*
if A $*h 0* $*h 1* ≠ *0 then*
let A′ = *interchange-columns A 0 1*;
Q′ = *interchange-columns* (*Finite-Cartesian-Product.mat 1*) *0 1 in*
diagonalize-2x2-aux (*Finite-Cartesian-Product.mat 1*, *A′*, *Q′*)
else
if A $*h 1* $*h 0* ≠ *0 then*
let A′ = *interchange-rows A 0 1*;
P′ = *interchange-rows* (*Finite-Cartesian-Product.mat 1*) *0 1 in*
diagonalize-2x2-aux (*P′*, *A′*, *Finite-Cartesian-Product.mat 1*)
else (*Finite-Cartesian-Product.mat 1*,*A*,*Finite-Cartesian-Product.mat 1*)
else diagonalize-2x2-aux (*Finite-Cartesian-Product.mat 1*,*A*,*Finite-Cartesian-Product.mat 1*)
)

lemma *diagonalize-2x2-works*:
assumes *PDQ*: (*P*,*D*,*Q*) = *diagonalize-2x2 A*
shows *D* = *P* ** *A* ** *Q* ∧ *invertible P* ∧ *invertible Q* ∧ *isDiagonal D*
⟨*proof*⟩

definition *diagonalize-2x2-JNF* (*A*::*'a*::*euclidean-ring-gcd mat*)
= (*let* (*P*,*D*,*Q*) = *diagonalize-2x2* (*Mod-Type-Connect.to-hma*$_m$ *A*::*'a*^2^2) *in*
(*Mod-Type-Connect.from-hma*$_m$ *P*,*Mod-Type-Connect.from-hma*$_m$ *D*,*Mod-Type-Connect.from-hma*$_m$

Q))

lemma *diagonalize-2x2-JNF-works*:
assumes *A*: *A* ∈ *carrier-mat 2 2*
and *PDQ*: (*P*,*D*,*Q*) = *diagonalize-2x2-JNF A*
shows *D* = *P* ∗ *A* ∗ *Q* ∧ *invertible-mat P* ∧ *invertible-mat Q* ∧ *isDiagonal-mat D* ∧ *P*∈*carrier-mat 2 2*
∧ *Q* ∈ *carrier-mat 2 2* ∧ *D* ∈ *carrier-mat 2 2*
⟨*proof*⟩

definition *Smith-2x2-eucl A* = (
let (*P*,*D*,*Q*) = *diagonalize-2x2 A*;
(*P*′,*S*,*Q*′) = *diagonal-to-Smith-PQ D euclid-ext2*
in (*P*′ ∗∗ *P*, *S*, *Q* ∗∗ *Q*′))

lemma *Smith-2x2-eucl-works*:
assumes *PBQ*: (*P*,*S*,*Q*) = *Smith-2x2-eucl A*
shows *S* = *P* ∗∗ *A* ∗∗ *Q* ∧ *invertible P* ∧ *invertible Q* ∧ *Smith-normal-form S*
⟨*proof*⟩

19.3 An executable algorithm to transform 2 × 2 matrices into its Smith normal form in JNF

definition *Smith-2x2-JNF-eucl A* = (
let (*P*,*D*,*Q*) = *diagonalize-2x2-JNF A*;
(*P*′,*S*,*Q*′) = *diagonal-to-Smith-PQ-JNF D euclid-ext2*
in (*P*′ ∗ *P*, *S*, *Q* ∗ *Q*′))

lemma *Smith-2x2-JNF-eucl-works*:
assumes *A*: *A* ∈ *carrier-mat 2 2*
and *PBQ*: (*P*,*S*,*Q*) = *Smith-2x2-JNF-eucl A*
shows *is-SNF A* (*P*,*S*,*Q*)
⟨*proof*⟩

19.4 An executable algorithm to transform 1 × 2 matrices into its Smith normal form

definition *Smith-1x2-eucl* (*A*::′*a*::*euclidean-ring-gcd*^2^1) = (
if A $*h 0* $*h 0* = *0* ∧ *A* $*h 0* $*h 1* ≠ *0 then*
let Q = *interchange-columns* (*Finite-Cartesian-Product.mat 1*) *0 1*;
A′ = *interchange-columns A 0 1 in* (*A*′,*Q*)
else
if A $*h 0* $*h 0* ≠ *0* ∧ *A* $*h 0* $*h 1* ≠ *0 then*
let bezout-matrix-right = *transpose* (*bezout-matrix* (*transpose A*) *0 1 0 eu-*

clid-ext2)
      in (*A* ** *bezout-matrix-right*, *bezout-matrix-right*)
    else (*A*, *Finite-Cartesian-Product.mat 1*)
)

lemma *Smith-1x2-eucl-works*:
  assumes *SQ*: (*S*,*Q*) = *Smith-1x2-eucl A*
  shows *S* = *A* ** *Q* ∧ *invertible Q* ∧ *S* $*h 0* $*h 1* = *0*
⟨*proof*⟩

definition *bezout-matrix-JNF* :: *'a*::*comm-ring-1 mat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
    ⇒ (*'a* ⇒ *'a* ⇒ (*'a* × *'a* × *'a* × *'a* × *'a*)) ⇒ *'a mat*
  where
  bezout-matrix-JNF A a b j bezout = *Matrix.mat* (*dim-row A*) (*dim-row A*) (λ(*x*,*y*).

      (*let*
        (*p*, *q*, *u*, *v*, *d*) = *bezout* (*A* $$ (*a*, *j*)) (*A* $$ (*b*, *j*))
       in
        if x = *a* ∧ *y* = *a then p else*
        if x = *a* ∧ *y* = *b then q else*
        if x = *b* ∧ *y* = *a then u else*
        if x = *b* ∧ *y* = *b then v else*
        if x = *y then 1 else 0*))

definition *Smith-1x2-eucl-JNF* (*A*::*'a*::*euclidean-ring-gcd mat*) = (
  if A $$ (*0*, *0*) = *0* ∧ *A* $$ (*0*, *1*) ≠ *0 then*
    let Q = *swaprows-mat 2 0 1*;
        A' = *swapcols 0 1 A*
     in (*A'*,*Q*)
  else
    if A $$ (*0*, *0*) ≠ *0* ∧ *A* $$ (*0*, *1*) ≠ *0 then*
      let bezout-matrix-right = *transpose-mat* (*bezout-matrix-JNF* (*transpose-mat*
A) *0 1 0 euclid-ext2*)
      in (*A* * *bezout-matrix-right*, *bezout-matrix-right*)
    else (*A*, *1*$_m$ *2*)
)

lemma *Smith-1x2-eucl-JNF-works*:
  assumes *A*: *A* ∈ *carrier-mat 1 2*
  and *SQ*: (*S*,*Q*) = *Smith-1x2-eucl-JNF A*
shows *is-SNF A* (*1*$_m$ *1*, (*Smith-1x2-eucl-JNF A*))
⟨*proof*⟩

19.5 The final executable algorithm to transform any matrix into its Smith normal form

global-interpretation *Smith-ED*: *Smith-Impl Smith-1x2-eucl-JNF Smith-2x2-JNF-eucl* (*div*)
  defines *Smith-ED-1xn-aux* = *Smith-ED.Smith-1xn-aux*
    and *Smith-ED-nx1* = *Smith-ED.Smith-nx1*
  and *Smith-ED-1xn* = *Smith-ED.Smith-1xn*
  and *Smith-ED-2xn* = *Smith-ED.Smith-2xn*
  and *Smith-ED-nx2* = *Smith-ED.Smith-nx2*
  and *Smith-ED-mxn* = *Smith-ED.Smith-mxn*
⟨*proof*⟩

end

20 A certified checker based on an external algorithm to compute Smith normal form

theory *Smith-Certified*
  imports
    SNF-Algorithm-Euclidean-Domain
begin

This (unspecified) function takes as input the matrix A and returns five matrices (P, S, Q, P', Q'), which must satisfy $S = PAQ$, S is in Smith normal form, P' and Q' are the inverse matrices of P and Q respectively

The matrices are given in terms of lists for the sake of simplicity when connecting the function to external solvers, like Mathematica or Sage.

consts *external-SNF* ::
  int list list ⇒ *int list list* × *int list list* × *int list list* × *int list list* × *int list list*

We implement the checker by means of the following definition. The checker is implemented in the JNF representation of matrices to make use of the Strassen matrix multiplication algorithm. In case that the certification fails, then the verified Smith normal form algorithm is executed. Thus, we will always get a verified result.

definition *checker-SNF A* = (
    let A' = *mat-to-list A*; *m* = *dim-row A*; *n* = *dim-col A in*
      case external-SNF A' of (*P-ext*,*S-ext*,*Q-ext*,*P'-ext*,*Q'-ext*) ⇒ *let*

P = *mat-of-rows-list m P-ext*;
S = *mat-of-rows-list m S-ext*;
Q = *mat-of-rows-list m Q-ext*;
P' = *mat-of-rows-list m P'-ext*;
Q' = *mat-of-rows-list m Q'-ext in*
(*if dim-row* $P = m \wedge$ *dim-col* $P = m$
$\wedge$ *dim-row* $S = m \wedge$ *dim-col* $S = n$
$\wedge$ *dim-row* $Q = n \wedge$ *dim-col* $Q = n$
$\wedge$ *dim-row* $P' = m \wedge$ *dim-col* $P' = m$
$\wedge$ *dim-row* $Q' = n \wedge$ *dim-col* $Q' = n$
$\wedge$ $P * P' = 1_m\ m \wedge Q * Q' = 1_m\ n$
$\wedge$ *Smith-normal-form-mat* $S \wedge (S = P{*}A{*}Q)$ *then*
(P,S,Q) *else Code.abort* (*STR ''Certification failed''*) (λ -. *Smith-ED-mxn A*))
)

theorem *checker-SNF-soudness*:
assumes *A*: $A \in$ *carrier-mat m n*
and *c*: *checker-SNF* $A = (P,S,Q)$
shows *is-SNF A* (P,S,Q)
⟨*proof*⟩

end
theory *Alternative-Proofs*
imports *Smith-Normal-Form.Admits-SNF-From-Diagonal-Iff-Bezout-Ring*
Smith-Normal-Form.Elementary-Divisor-Rings
begin

Theorem 2: (C) ==> (A)

lemma *diagonal-2x2-admits-SNF-imp-bezout-ring-JNF*:
assumes *admits-SNF*: $\forall A.\ (A::'a\ mat) \in$ *carrier-mat 2 2* $\wedge$ *isDiagonal-mat A*
$\longrightarrow (\exists P\ Q.\ P \in$ *carrier-mat 2 2* $\wedge\ Q \in$ *carrier-mat 2 2* $\wedge$ *invertible-mat P* $\wedge$ *invertible-mat Q*
$\wedge$ *Smith-normal-form-mat* $(P{*}A{*}Q))$
shows *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*)
⟨*proof*⟩

Theorem 2: (A) ==> (C)

lemma *bezout-ring-imp-diagonal-2x2-admits-SNF-JNF*:
assumes *c*: *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*)
shows $\forall A.\ (A::'a\ mat) \in$ *carrier-mat 2 2* $\wedge$ *isDiagonal-mat A*
$\longrightarrow (\exists P\ Q.\ P \in$ *carrier-mat 2 2* $\wedge\ Q \in$ *carrier-mat 2 2*
$\wedge$ *invertible-mat P* $\wedge$ *invertible-mat Q* $\wedge$ *Smith-normal-form-mat* $(P{*}A{*}Q))$
⟨*proof*⟩

Theorem 2: (A) <==> (C)

lemma *diagonal-2x2-admits-SNF-iff-bezout-ring*:
shows *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*)
$\equiv (\bigwedge A::'a\ mat.\ A \in$ *carrier-mat 2 2* $\longrightarrow$ *admits-SNF-JNF A*) (**is** *?lhs* $\equiv$ *?rhs*)
⟨*proof*⟩

Theorem 2: (B) <==> (C)

lemma *diagonal-2x2-admits-SNF-iff-diagonal-admits-SNF*:
shows $(\forall (A::'a::comm\text{-}ring\text{-}1\ mat).\ admits\text{-}SNF\text{-}JNF\ A) =$
$(\forall (A::'a\ mat) \in carrier\text{-}mat\ 2\ 2.\ admits\text{-}SNF\text{-}JNF\ A)$
⟨*proof*⟩

Theorem 2: final statements

theorem *Theorem2-final*:
shows *A-imp-B*: $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class)$
$\Longrightarrow (\forall A::'a\ mat.\ admits\text{-}SNF\text{-}JNF\ A)$
and *B-imp-C*: $(\forall A::'a\ mat.\ admits\text{-}SNF\text{-}JNF\ A) \Longrightarrow$
$(\forall (A::'a\ mat) \in carrier\text{-}mat\ 2\ 2.\ admits\text{-}SNF\text{-}JNF\ A)$
and *C-imp-A*: $(\forall (A::'a\ mat) \in carrier\text{-}mat\ 2\ 2.\ admits\text{-}SNF\text{-}JNF\ A)$
$\Longrightarrow OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class)$
⟨*proof*⟩

theorem *Theorem2-final'*:
shows *A-eq-B*: $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class) \equiv (\bigwedge A::'a\ mat.$
$admits\text{-}SNF\text{-}JNF\ A)$
and *A-eq-C*: $OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class) \equiv$
$(\bigwedge (A::'a\ mat).\ A \in carrier\text{-}mat\ 2\ 2 \longrightarrow admits\text{-}SNF\text{-}JNF\ A)$
and *B-eq-C*: $(\forall (A::'a::comm\text{-}ring\text{-}1\ mat).\ admits\text{-}SNF\text{-}JNF\ A) =$
$(\forall (A::'a\ mat) \in carrier\text{-}mat\ 2\ 2.\ admits\text{-}SNF\text{-}JNF\ A)$
⟨*proof*⟩

Theorem 2: final statement in HA. (A) <==> (C).

theorem *Theorem2-A-eq-C-HA*:
$OFCLASS('a::comm\text{-}ring\text{-}1,\ bezout\text{-}ring\text{-}class) \equiv (\bigwedge (A::'a\hat{}2\hat{}2).\ admits\text{-}SNF\text{-}HA$
$A)$
⟨*proof*⟩

Hermite implies Bezout

Theorem 3, proof for 1x2 matrices

lemma *theorem3-restricted-12-part1*:
assumes *T*: $(\forall\ a\ b::'a::comm\text{-}ring\text{-}1.\ \exists\ a1\ b1\ d.\ a = a1*d \wedge b = b1*d$
$\wedge\ ideal\text{-}generated\ \{a1,b1\} = ideal\text{-}generated\ \{1\})$
shows $\forall (A::'a\ mat) \in carrier\text{-}mat\ 1\ 2.\ admits\text{-}triangular\text{-}reduction\ A$
⟨*proof*⟩

lemma *theorem3-restricted-12-part2*:
assumes *1*: $\forall (A::'a::comm\text{-}ring\text{-}1\ mat) \in carrier\text{-}mat\ 1\ 2.\ admits\text{-}triangular\text{-}reduction$
A
shows $\forall\ a\ b::'a.\ \exists\ a1\ b1\ d.\ a = a1*d \wedge b = b1*d \wedge ideal\text{-}generated\ \{a1,b1\} =$
$ideal\text{-}generated\ \{1\}$
⟨*proof*⟩

lemma *Hermite-ring-imp-Bezout-ring*:
  assumes *H*: *OFCLASS*('*a*::*comm-ring-1*, *Hermite-ring-class*)
  shows *OFCLASS*('*a*::*comm-ring-1*, *bezout-ring-class*)
⟨*proof*⟩

end