Advanced Computer Algebra: Introduction to AXIOM

Marc Moreno Maza
University of Western Ontario
AM583 – Fall 2003

April 27, 2004

1 A First Session

We give some illustrative examples of Computer Algebra capabilities using the AXIOM system.

- AXIOM was designed at IBM during the 70's and 80's.
- Then AXIOM was maintained at NAG during the 90's.
- AXIOM became in the early 90's the most power computer algebra system, especially thanks to its programming language.
- Then, AXIOM *retired* in 2001 by lack of support, although no other Computer Algebra systems have such powerful programming facilities.
- Today, AXIOM is back and it should become a free software very soon.
- Moreover, AXIOM has a son, ALDOR, born in the early 90's at IBM and growing today at UWO!

For more information about AXIOM please visit:

`http://www.nag.co.uk/symbolic_software.asp`

`http://www.nongnu.org/axiom/`

AXIOM SESSION

```
Starts dribbling to session1.out (Mon Sep 15 12:01:24 2003)


--------------------------------------------
-- There are different types of integers  --
--------------------------------------------

123


   (1)  123
                                                        Type: PositiveInteger
                                                                  Time: 0 sec

0


   (2)  0
                                                     Type: NonNegativeInteger
                                                                  Time: 0 sec

-45


   (3)  - 45
                                                                Type: Integer
                                                                  Time: 0 sec

------------------------------------------
-- There is a type inference mechanism  --
------------------------------------------

123 + (-45)


   (4)  78
                                                        Type: PositiveInteger
                                                                  Time: 0 sec
```

```
123 + (-456)


   (5)  - 333
                                                                   Type: Integer
                                                                     Time: 0 sec

355/113


        355
   (6)  ---
        113
                                                          Type: Fraction Integer
                                                                     Time: 0 sec

355/113.0


   (7)  3.1415929203 539823009
                                                                     Type: Float
                       Time: 0.01 (IN) + 0.01 (OT) + 0.01 (GC) = 0.03 sec

-------------------------------
-- Coercions are also needed --
-------------------------------

(355/113) :: Float


   (8)  3.1415929203 539823009
                                                                     Type: Float
                                                                     Time: 0 sec

(355/113) :: DoubleFloat


   (9)  3.1415929203539825
                                                               Type: DoubleFloat
```

```
                                                            Time: 0 sec

-- Type inference again

n: Integer := 12


   (10)  12
                                                          Type: Integer
                                                            Time: 0 sec

a: Integer := 3


   (11)  3
                                                          Type: Integer
                                                            Time: 0 sec

a^n


   (12)  531441
                                                 Type: Fraction Integer
                                                            Time: 0 sec

n := -n


   (13)  - 12
                                                          Type: Integer
                                              Time: 0.01 (OT) = 0.01 sec

a^n


            1
   (14)  ------
         531441
                                                 Type: Fraction Integer
                                                            Time: 0 sec
```

```
inv(a^n)


   (15)  531441
                                                   Type: Fraction Integer
                                                            Time: 0 sec

f := n * 1.0


   (16)  - 12.0
                                                            Type: Float
                                          Time: 0.02 (OT) = 0.02 sec

a^f


   (17)  0.0000018816 7642315892 07457
                                                            Type: Float
                              Time: 0.01 (IN) + 0.01 (OT) = 0.02 sec

-------------------------------------------
-- Types have an influence on the result --
-------------------------------------------

gcd (123,45)


   (18)  3
                                                  Type: PositiveInteger
                                                            Time: 0 sec

gcd (123,45/1)


   (19)  1
                                                   Type: Fraction Integer
                  Time: 0.03 (IN) + 0.01 (EV) + 0.01 (OT) = 0.05 sec
```

```
x


   (20)  x
                                                          Type: Variable x
                                                              Time: 0 sec

x + 1


   (21)  x + 1
                                                    Type: Polynomial Integer
                                 Time: 0.01 (IN) + 0.01 (GC) = 0.02 sec

x + 1/2


             1
   (22)  x + -
             2
                                           Type: Polynomial Fraction Integer
                                 Time: 0.04 (IN) + 0.01 (OT) = 0.05 sec

p := (2*x + 2) * (x + 2)


           2
   (23)  2x  + 6x + 4
                                                    Type: Polynomial Integer
                                                Time: 0.01 (IN) = 0.01 sec

q := (x + 1) * (x + 3) * 2


           2
   (24)  2x  + 8x + 6
                                                    Type: Polynomial Integer
                                                Time: 0.01 (OT) = 0.01 sec

gcd(p,q)
```

```
   (25)  2x + 2
                                                  Type: Polynomial Integer
                                   Time: 0.01 (OT) + 0.03 (GC) = 0.04 sec

q := (x + 1) * (x + 3) * 2/1


          2
   (26)  2x  + 8x + 6
                                         Type: Polynomial Fraction Integer
             Time: 0.01 (IN) + 0.01 (EV) + 0.01 (OT) + 0.01 (GC) = 0.04 sec

gcd (p,q)


   (27)  x + 1
                                         Type: Polynomial Fraction Integer
                                   Time: 0.02 (IN) + 0.01 (EV) = 0.03 sec

----------------------
-- Specifying types --
----------------------

Z := Integer


   (28)  Integer
                                                              Type: Domain
                                                               Time: 0 sec

a: Z := 12


   (29)  12
                                                             Type: Integer
                                                               Time: 0 sec

b: Z := 45
```

```
   (30)  45
                                                  Type: Integer
                                                    Time: 0 sec

gcd(a,b)

   (31)  3
                                          Type: PositiveInteger
                                                    Time: 0 sec

Q := Fraction Integer

   (32)  Fraction Integer
                                                   Type: Domain
                                                    Time: 0 sec

r: Q := a

   (33)  12
                                         Type: Fraction Integer
                                                    Time: 0 sec

s: Q := b

   (34)  45
                                         Type: Fraction Integer
                                                    Time: 0 sec

gcd(r,s)

   (35)  1
                                         Type: Fraction Integer
                                                    Time: 0 sec
```

```
gcd(12,45)$Q


   (36)  1
                                                       Type: Fraction Integer
                                                                  Time: 0 sec

gcd(12,45)$Z


   (37)  3
                                                                Type: Integer
                                                                  Time: 0 sec

a := inv(a)


   Cannot convert right-hand side of assignment
    1
   --
   12

      to an object of the type Integer of the left-hand side.

--------------------
-- Browsing types --
--------------------

)sh Z

 Integer is a domain constructor.
 Abbreviation for Integer is INT
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/integer.spad to see algebra source code for INT

------------------------------- Operations --------------------------------

 ?*? : (Integer,%) -> %                ?*? : (PositiveInteger,%) -> %
 ?*? : (%,%) -> %                      ?**? : (%,PositiveInteger) -> %
```

```
?+? : (%,%) -> %                       ?-? : (%,%) -> %
-? : % -> %                            ?<? : (%,%) -> Boolean
?<=? : (%,%) -> Boolean                ?=? : (%,%) -> Boolean
?>? : (%,%) -> Boolean                 ?>=? : (%,%) -> Boolean
D : (%,NonNegativeInteger) -> %        D : % -> %
OMwrite : (%,Boolean) -> String        OMwrite : % -> String
1 : () -> %                            0 : () -> %
?^? : (%,PositiveInteger) -> %         abs : % -> %
addmod : (%,%,%) -> %                  associates? : (%,%) -> Boolean
base : () -> %                         binomial : (%,%) -> %
bit? : (%,%) -> Boolean                coerce : % -> OutputForm
coerce : Integer -> %                  coerce : % -> %
convert : % -> DoubleFloat             convert : % -> Float
convert : % -> InputForm               convert : % -> Integer
convert : % -> Pattern Integer         convert : % -> String
copy : % -> %                          dec : % -> %
differentiate : % -> %                 even? : % -> Boolean
factor : % -> Factored %               factorial : % -> %
gcd : List % -> %                      gcd : (%,%) -> %
hash : % -> SingleInteger              hash : % -> %
inc : % -> %                           init : () -> %
invmod : (%,%) -> %                    latex : % -> String
lcm : List % -> %                      lcm : (%,%) -> %
length : % -> %                        mask : % -> %
max : (%,%) -> %                       min : (%,%) -> %
mulmod : (%,%,%) -> %                  negative? : % -> Boolean
odd? : % -> Boolean                    one? : % -> Boolean
permutation : (%,%) -> %               positive? : % -> Boolean
positiveRemainder : (%,%) -> %         powmod : (%,%,%) -> %
prime? : % -> Boolean                  ?quo? : (%,%) -> %
random : % -> %                        random : () -> %
rational : % -> Fraction Integer       rational? : % -> Boolean
recip : % -> Union(%,"failed")         ?rem? : (%,%) -> %
retract : % -> Integer                 sample : () -> %
shift : (%,%) -> %                     sign : % -> Integer
sizeLess? : (%,%) -> Boolean           squareFree : % -> Factored %
squareFreePart : % -> %                submod : (%,%,%) -> %
symmetricRemainder : (%,%) -> %        unit? : % -> Boolean
unitCanonical : % -> %                 zero? : % -> Boolean
?~=? : (%,%) -> Boolean
```

```
 ?*? : (NonNegativeInteger,%) -> %
 ?**? : (%,NonNegativeInteger) -> %
 OMwrite : (OpenMathDevice,%,Boolean) -> Void
 OMwrite : (OpenMathDevice,%) -> Void
 ?^? : (%,NonNegativeInteger) -> %
 characteristic : () -> NonNegativeInteger
 differentiate : (%,NonNegativeInteger) -> %
 divide : (%,%) -> Record(quotient: %,remainder: %)
 euclideanSize : % -> NonNegativeInteger
 expressIdealMember : (List %,%) -> Union(List %,"failed")
 exquo : (%,%) -> Union(%,"failed")
 extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
 extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
 gcdPolynomial : (SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> SparseUnivariatePolynomial %
 multiEuclidean : (List %,%) -> Union(List %,"failed")
 nextItem : % -> Union(%,"failed")
 patternMatch : (%,Pattern Integer,PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%)
 principalIdeal : List % -> Record(coef: List %,generator: %)
 rationalIfCan : % -> Union(Fraction Integer,"failed")
 reducedSystem : Matrix % -> Matrix Integer
 reducedSystem : (Matrix %,Vector %) -> Record(mat: Matrix Integer,vec: Vector Integer)
 retractIfCan : % -> Union(Integer,"failed")
 subtractIfCan : (%,%) -> Union(%,"failed")
 unitNormal : % -> Record(unit: %,canonical: %,associate: %)



)sh Q

 Fraction Integer is a domain constructor.
 Abbreviation for Fraction is FRAC
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/fraction.spad to see algebra source code for FRAC

------------------------------- Operations --------------------------------

 ?*? : (Fraction Integer,%) -> %       ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?*? : (%,Fraction Integer) -> %
 ?*? : (%,Integer) -> %                ?*? : (%,%) -> %
 ?**? : (%,Integer) -> %               ?**? : (%,PositiveInteger) -> %
```

```
?+? : (%,%) -> %
-? : % -> %
?/? : (%,%) -> %
?<=? : (%,%) -> Boolean
?>? : (%,%) -> Boolean
D : (%,List Symbol) -> %
D : (%,Symbol) -> %
OMwrite : (%,Boolean) -> String
1 : () -> %
?^? : (%,Integer) -> %
abs : % -> %
ceiling : % -> Integer
coerce : Fraction Integer -> %
coerce : Symbol -> %
convert : % -> DoubleFloat
convert : % -> InputForm
convert : % -> Pattern Integer
denominator : % -> %
differentiate : % -> %
eval : (%,Equation Integer) -> %
eval : (%,Symbol,Integer) -> %
floor : % -> Integer
gcd : List % -> %
hash : % -> SingleInteger
inv : % -> %
lcm : List % -> %
max : (%,%) -> %
negative? : % -> Boolean
numerator : % -> %
positive? : % -> Boolean
?quo? : (%,%) -> %
recip : % -> Union(%,"failed")
retract : % -> Fraction Integer
retract : % -> Symbol
sign : % -> Integer
squareFree : % -> Factored %
unit? : % -> Boolean
wholePart : % -> Integer
?~=? : (%,%) -> Boolean
?*? : (NonNegativeInteger,%) -> %
?-? : (%,%) -> %
?/? : (Integer,Integer) -> %
?<? : (%,%) -> Boolean
?=? : (%,%) -> Boolean
?>=? : (%,%) -> Boolean
D : (%,NonNegativeInteger) -> %
D : % -> %
OMwrite : % -> String
0 : () -> %
?^? : (%,PositiveInteger) -> %
associates? : (%,%) -> Boolean
coerce : % -> OutputForm
coerce : Integer -> %
coerce : % -> %
convert : % -> Float
convert : % -> Pattern Float
denom : % -> Integer
differentiate : (%,Symbol) -> %
?.? : (%,Integer) -> %
eval : (%,Integer,Integer) -> %
factor : % -> Factored %
fractionPart : % -> %
gcd : (%,%) -> %
init : () -> %
latex : % -> String
lcm : (%,%) -> %
min : (%,%) -> %
numer : % -> Integer
one? : % -> Boolean
prime? : % -> Boolean
random : () -> %
?rem? : (%,%) -> %
retract : % -> Integer
sample : () -> %
sizeLess? : (%,%) -> Boolean
squareFreePart : % -> %
unitCanonical : % -> %
zero? : % -> Boolean
```

```
?**? : (%,NonNegativeInteger) -> %
D : (%,List Symbol,List NonNegativeInteger) -> %
D : (%,(Integer -> Integer),NonNegativeInteger) -> %
D : (%,(Integer -> Integer)) -> %
D : (%,Symbol,NonNegativeInteger) -> %
OMwrite : (OpenMathDevice,%,Boolean) -> Void
OMwrite : (OpenMathDevice,%) -> Void
?^? : (%,NonNegativeInteger) -> %
characteristic : () -> NonNegativeInteger
charthRoot : % -> Union(%,"failed")
conditionP : Matrix % -> Union(Vector %,"failed")
differentiate : (%,List Symbol,List NonNegativeInteger) -> %
differentiate : (%,List Symbol) -> %
differentiate : (%,(Integer -> Integer),NonNegativeInteger) -> %
differentiate : (%,(Integer -> Integer)) -> %
differentiate : (%,NonNegativeInteger) -> %
differentiate : (%,Symbol,NonNegativeInteger) -> %
divide : (%,%) -> Record(quotient: %,remainder: %)
euclideanSize : % -> NonNegativeInteger
eval : (%,List Equation Integer) -> %
eval : (%,List Integer,List Integer) -> %
eval : (%,List Symbol,List Integer) -> %
expressIdealMember : (List %,%) -> Union(List %,"failed")
exquo : (%,%) -> Union(%,"failed")
extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
factorPolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial %
factorSquareFreePolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial %
gcdPolynomial : (SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> SparseUnivariatePolynomial %
map : ((Integer -> Integer),%) -> %
multiEuclidean : (List %,%) -> Union(List %,"failed")
nextItem : % -> Union(%,"failed")
patternMatch : (%,Pattern Float,PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%)
patternMatch : (%,Pattern Integer,PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%)
principalIdeal : List % -> Record(coef: List %,generator: %)
reducedSystem : Matrix % -> Matrix Integer
reducedSystem : (Matrix %,Vector %) -> Record(mat: Matrix Integer,vec: Vector Integer)
retractIfCan : % -> Union(Fraction Integer,"failed")
retractIfCan : % -> Union(Integer,"failed")
retractIfCan : % -> Union(Symbol,"failed")
```

```
 solveLinearPolynomialEquation : (List SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> Union(List SparseUnivariatePolynomial %,"failed")
 squareFreePolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial %
 subtractIfCan : (%,%) -> Union(%,"failed")
 unitNormal : % -> Record(unit: %,canonical: %,associate: %)
```

```
Q has Field
```

```
   (38)  true
                                                                 Type: Boolean
                                                                   Time: 0 sec
```

```
Z has Field
```

```
   (39)  false
                                                                 Type: Boolean
                                                                   Time: 0 sec
```

```
)sh Field
```

```
 Field  is a category constructor
 Abbreviation for Field is FIELD
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/catdef.spad to see algebra source code for FIELD

------------------------------- Operations --------------------------------
 ?*? : (Fraction Integer,%) -> %       ?*? : (%,Fraction Integer) -> %
 ?*? : (%,%) -> %                      ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,Integer) -> %
 ?**? : (%,PositiveInteger) -> %       ?+? : (%,%) -> %
 ?-? : (%,%) -> %                      -? : % -> %
 ?/? : (%,%) -> %                      ?=? : (%,%) -> Boolean
 1 : () -> %                           0 : () -> %
 ?^? : (%,Integer) -> %                ?^? : (%,PositiveInteger) -> %
 associates? : (%,%) -> Boolean        coerce : Fraction Integer -> %
 coerce : % -> %                       coerce : Integer -> %
 coerce : % -> OutputForm              factor : % -> Factored %
```

```
 gcd : List % -> %                     gcd : (%,%) -> %
 hash : % -> SingleInteger             inv : % -> %
 latex : % -> String                   lcm : List % -> %
 lcm : (%,%) -> %                      one? : % -> Boolean
 prime? : % -> Boolean                 ?quo? : (%,%) -> %
 recip : % -> Union(%,"failed")        ?rem? : (%,%) -> %
 sample : () -> %                      sizeLess? : (%,%) -> Boolean
 squareFree : % -> Factored %          squareFreePart : % -> %
 unit? : % -> Boolean                  unitCanonical : % -> %
 zero? : % -> Boolean                  ?~=? : (%,%) -> Boolean
 ?*? : (NonNegativeInteger,%) -> %
 ?**? : (%,NonNegativeInteger) -> %
 ?^? : (%,NonNegativeInteger) -> %
 characteristic : () -> NonNegativeInteger
 divide : (%,%) -> Record(quotient: %,remainder: %)
 euclideanSize : % -> NonNegativeInteger
 expressIdealMember : (List %,%) -> Union(List %,"failed")
 exquo : (%,%) -> Union(%,"failed")
 extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
 extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
 gcdPolynomial : (SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> SparseUnivariatePolynomial %
 multiEuclidean : (List %,%) -> Union(List %,"failed")
 principalIdeal : List % -> Record(coef: List %,generator: %)
 subtractIfCan : (%,%) -> Union(%,"failed")
 unitNormal : % -> Record(unit: %,canonical: %,associate: %)


Z has Ring


   (40)  true
                                                                  Type: Boolean
                                                                    Time: 0 sec

)sh Ring

 Ring  is a category constructor
 Abbreviation for Ring is RING
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/catdef.spad to see algebra source code for RING
```

```
------------------------------- Operations --------------------------------
 ?*? : (%,%) -> %                      ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,PositiveInteger) -> %
 ?+? : (%,%) -> %                      ?-? : (%,%) -> %
 -? : % -> %                           ?=? : (%,%) -> Boolean
 1 : () -> %                           0 : () -> %
 ?^? : (%,PositiveInteger) -> %        coerce : Integer -> %
 coerce : % -> OutputForm              hash : % -> SingleInteger
 latex : % -> String                   one? : % -> Boolean
 recip : % -> Union(%,"failed")        sample : () -> %
 zero? : % -> Boolean                  ?~=? : (%,%) -> Boolean
 ?*? : (NonNegativeInteger,%) -> %
 ?**? : (%,NonNegativeInteger) -> %
 ?^? : (%,NonNegativeInteger) -> %
 characteristic : () -> NonNegativeInteger
 subtractIfCan : (%,%) -> Union(%,"failed")


Z has OrderedRing


   (41)  true
                                                                 Type: Boolean
                                                                   Time: 0 sec

)sh OrderedRing

 OrderedRing  is a category constructor
 Abbreviation for OrderedRing is ORDRING
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/catdef.spad to see algebra source code for ORDRING

------------------------------- Operations --------------------------------
 ?*? : (%,%) -> %                      ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,PositiveInteger) -> %
 ?+? : (%,%) -> %                      -? : % -> %
 ?-? : (%,%) -> %                      ?<? : (%,%) -> Boolean
 ?<=? : (%,%) -> Boolean               ?=? : (%,%) -> Boolean
 ?>? : (%,%) -> Boolean                ?>=? : (%,%) -> Boolean
```

```
 1 : () -> %                            0 : () -> %
 ?^? : (%,PositiveInteger) -> %         abs : % -> %
 coerce : Integer -> %                  coerce : % -> OutputForm
 hash : % -> SingleInteger              latex : % -> String
 max : (%,%) -> %                       min : (%,%) -> %
 negative? : % -> Boolean               one? : % -> Boolean
 positive? : % -> Boolean               recip : % -> Union(%,"failed")
 sample : () -> %                       sign : % -> Integer
 zero? : % -> Boolean                   ?~=? : (%,%) -> Boolean
 ?*? : (NonNegativeInteger,%) -> %
 ?**? : (%,NonNegativeInteger) -> %
 ?^? : (%,NonNegativeInteger) -> %
 characteristic : () -> NonNegativeInteger
 subtractIfCan : (%,%) -> Union(%,"failed")
```

```
Q has OrderedRing


   (42)  true
                                                              Type: Boolean
                                                                Time: 0 sec

Z has GcdDomain


   (43)  true
                                                              Type: Boolean
                                                Time: 0.01 (OT) = 0.01 sec

Q has GcdDomain


   (44)  true
                                                              Type: Boolean
                                                                Time: 0 sec

Z has EuclideanDomain
```

```
   (45)  true
                                                          Type: Boolean
                                              Time: 0.01 (OT) = 0.01 sec

Q has EuclideanDomain


   (46)  true
                                                          Type: Boolean
                                                            Time: 0 sec

--------------------------
-- Programming in AXIOM --
--------------------------

f(u,v) == gcd(u,v)

                                                             Type: Void
                                                            Time: 0 sec

f(a,b)

   Compiling function f with type (Integer,Integer) -> Integer

   (48)  3
                                                  Type: PositiveInteger
                                                            Time: 0 sec

f(r,s)

   Compiling function f with type (Fraction Integer,Fraction Integer)
       -> Fraction Integer

   (49)  1
                                                 Type: Fraction Integer
                                                            Time: 0 sec

g(u,v) == while u ~= v repeat if u < v then v := v - u else u := u - v

                                                             Type: Void
```

```
                                                                 Time: 0 sec

h(u,v) ==
    g(u,v)
    u

                                                                  Type: Void
                                                                 Time: 0 sec

h(12,45)

   Compiling function g with type (PositiveInteger,PositiveInteger) ->
      Void
   Compiling function h with type (PositiveInteger,PositiveInteger) ->
      PositiveInteger

   (52)  12
                                                       Type: PositiveInteger
                                                Time: 0.03 (IN) = 0.03 sec

h(u,v) ==
   while u ~= v repeat if u < v then v := v - u else u := u - v
   u

   Compiled code for h has been cleared.
   1 old definition(s) deleted for function or rule h
                                                                  Type: Void
                                                                 Time: 0 sec

h(12,45)

   Compiling function h with type (PositiveInteger,PositiveInteger) ->
      Integer

+++ |*2;h;1;Session1| redefined

   (54)  3
                                                       Type: PositiveInteger
                                                Time: 0.01 (OT) = 0.01 sec
```

```
-------------------------------
-- Functions returning types --
-------------------------------

3/4


         3
   (55)  -
         4
                                                           Type: Fraction Integer
                                                                      Time: 0 sec

(x + 1)/(x + 2)


         x + 1
   (56)  -----
         x + 2
                                                Type: Fraction Polynomial Integer
                                                       Time: 0.01 (OT) = 0.01 sec

)sh Fraction

 Fraction S: IntegralDomain  is a domain constructor
 Abbreviation for Fraction is FRAC
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/fraction.spad to see algebra source code for FRAC

------------------------------- Operations --------------------------------
 ?*? : (%,S) -> %                      ?*? : (S,%) -> %
 ?*? : (Fraction Integer,%) -> %       ?*? : (%,Fraction Integer) -> %
 ?*? : (%,%) -> %                      ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,Integer) -> %
 ?**? : (%,PositiveInteger) -> %       ?+? : (%,%) -> %
 ?-? : (%,%) -> %                      -? : % -> %
 ?/? : (S,S) -> %                      ?/? : (%,%) -> %
 ?=? : (%,%) -> Boolean                D : (%,(S -> S)) -> %
 D : % -> % if S has DIFRING           1 : () -> %
```

```
0 : () -> %
?^? : (%,PositiveInteger) -> %
ceiling : % -> S if S has INS
coerce : Fraction Integer -> %
coerce : Integer -> %
denom : % -> S
factor : % -> Factored %
gcd : List % -> %
hash : % -> SingleInteger
inv : % -> %
lcm : List % -> %
map : ((S -> S),%) -> %
numerator : % -> %
prime? : % -> Boolean
random : () -> % if S has INS
?rem? : (%,%) -> %
sample : () -> %
squareFree : % -> Factored %
unit? : % -> Boolean
zero? : % -> Boolean
?^? : (%,Integer) -> %
associates? : (%,%) -> Boolean
coerce : S -> %
coerce : % -> %
coerce : % -> OutputForm
denominator : % -> %
floor : % -> S if S has INS
gcd : (%,%) -> %
init : () -> % if S has STEP
latex : % -> String
lcm : (%,%) -> %
numer : % -> S
one? : % -> Boolean
?quo? : (%,%) -> %
recip : % -> Union(%,"failed")
retract : % -> S
sizeLess? : (%,%) -> Boolean
squareFreePart : % -> %
unitCanonical : % -> %
?~=? : (%,%) -> Boolean
?*? : (NonNegativeInteger,%) -> %
?**? : (%,NonNegativeInteger) -> %
?<? : (%,%) -> Boolean if S has ORDSET
?<=? : (%,%) -> Boolean if S has ORDSET
?>? : (%,%) -> Boolean if S has ORDSET
?>=? : (%,%) -> Boolean if S has ORDSET
D : (%,(S -> S),NonNegativeInteger) -> %
D : (%,List Symbol,List NonNegativeInteger) -> % if S has PDRING SYMBOL
D : (%,Symbol,NonNegativeInteger) -> % if S has PDRING SYMBOL
D : (%,List Symbol) -> % if S has PDRING SYMBOL
D : (%,Symbol) -> % if S has PDRING SYMBOL
D : (%,NonNegativeInteger) -> % if S has DIFRING
OMwrite : (OpenMathDevice,%,Boolean) -> Void if S has INS and S has OM
OMwrite : (OpenMathDevice,%) -> Void if S has INS and S has OM
OMwrite : (%,Boolean) -> String if S has INS and S has OM
OMwrite : % -> String if S has INS and S has OM
?^? : (%,NonNegativeInteger) -> %
abs : % -> % if S has OINTDOM
characteristic : () -> NonNegativeInteger
charthRoot : % -> Union(%,"failed") if $ has CHARNZ and S has PFECAT or S has CHARNZ
```

```
coerce : Symbol -> % if S has RETRACT SYMBOL
conditionP : Matrix % -> Union(Vector %,"failed") if $ has CHARNZ and S has PFECAT
convert : % -> DoubleFloat if S has REAL
convert : % -> Float if S has REAL
convert : % -> InputForm if S has KONVERT INFORM
convert : % -> Pattern Float if S has KONVERT PATTERN FLOAT
convert : % -> Pattern Integer if S has KONVERT PATTERN INT
differentiate : (%,(S -> S)) -> %
differentiate : (%,(S -> S),NonNegativeInteger) -> %
differentiate : (%,List Symbol,List NonNegativeInteger) -> % if S has PDRING SYMBOL
differentiate : (%,Symbol,NonNegativeInteger) -> % if S has PDRING SYMBOL
differentiate : (%,List Symbol) -> % if S has PDRING SYMBOL
differentiate : (%,Symbol) -> % if S has PDRING SYMBOL
differentiate : (%,NonNegativeInteger) -> % if S has DIFRING
differentiate : % -> % if S has DIFRING
divide : (%,%) -> Record(quotient: %,remainder: %)
?.? : (%,S) -> % if S has ELTAB(S,S)
euclideanSize : % -> NonNegativeInteger
eval : (%,Symbol,S) -> % if S has IEVALAB(SYMBOL,S)
eval : (%,List Symbol,List S) -> % if S has IEVALAB(SYMBOL,S)
eval : (%,List Equation S) -> % if S has EVALAB S
eval : (%,Equation S) -> % if S has EVALAB S
eval : (%,S,S) -> % if S has EVALAB S
eval : (%,List S,List S) -> % if S has EVALAB S
expressIdealMember : (List %,%) -> Union(List %,"failed")
exquo : (%,%) -> Union(%,"failed")
extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
factorPolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial % if S has PFECAT
factorSquareFreePolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial % if S has PFECAT
fractionPart : % -> % if S has EUCDOM
gcdPolynomial : (SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> SparseUnivariatePolynomial %
max : (%,%) -> % if S has ORDSET
min : (%,%) -> % if S has ORDSET
multiEuclidean : (List %,%) -> Union(List %,"failed")
negative? : % -> Boolean if S has OINTDOM
nextItem : % -> Union(%,"failed") if S has STEP
patternMatch : (%,Pattern Float,PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%) if S has PATMAB FLOAT
patternMatch : (%,Pattern Integer,PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%) if S has PATMAB INT
positive? : % -> Boolean if S has OINTDOM
```

```
 principalIdeal : List % -> Record(coef: List %,generator: %)
 reducedSystem : Matrix % -> Matrix S
 reducedSystem : (Matrix %,Vector %) -> Record(mat: Matrix S,vec: Vector S)
 reducedSystem : (Matrix %,Vector %) -> Record(mat: Matrix Integer,vec: Vector Integer) if S has LINEXP INT
 reducedSystem : Matrix % -> Matrix Integer if S has LINEXP INT
 retract : % -> Integer if S has RETRACT INT
 retract : % -> Fraction Integer if S has RETRACT INT
 retract : % -> Symbol if S has RETRACT SYMBOL
 retractIfCan : % -> Union(Integer,"failed") if S has RETRACT INT
 retractIfCan : % -> Union(Fraction Integer,"failed") if S has RETRACT INT
 retractIfCan : % -> Union(Symbol,"failed") if S has RETRACT SYMBOL
 retractIfCan : % -> Union(S,"failed")
 sign : % -> Integer if S has OINTDOM
 solveLinearPolynomialEquation : (List SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> Union(List SparseUnivariatePolynomial %,"failed") if S has PFECAT
 squareFreePolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial % if S has PFECAT
 subtractIfCan : (%,%) -> Union(%,"failed")
 unitNormal : % -> Record(unit: %,canonical: %,associate: %)
 wholePart : % -> S if S has EUCDOM


)sh Polynomial

 Polynomial R: Ring  is a domain constructor
 Abbreviation for Polynomial is POLY
 This constructor is exposed in this frame.
 Issue )edit /usr/local/share/axiom/axiom2.3//mnt/linuxglibc2.1//../../src/algebra/multpoly.spad to see algebra source code for POLY

------------------------------- Operations --------------------------------
 ?*? : (%,R) -> %                      ?*? : (R,%) -> %
 ?*? : (%,%) -> %                      ?*? : (Integer,%) -> %
 ?*? : (PositiveInteger,%) -> %        ?**? : (%,PositiveInteger) -> %
 ?+? : (%,%) -> %                      ?-? : (%,%) -> %
 -? : % -> %                           ?=? : (%,%) -> Boolean
 D : (%,List Symbol) -> %              D : (%,Symbol) -> %
 1 : () -> %                           0 : () -> %
 ?^? : (%,PositiveInteger) -> %        coefficients : % -> List R
 coerce : Symbol -> %                  coerce : R -> %
 coerce : Integer -> %                 coerce : % -> OutputForm
 differentiate : (%,Symbol) -> %       eval : (%,Symbol,%) -> %
 eval : (%,Symbol,R) -> %              eval : (%,List %,List %) -> %
```

```
eval : (%,%,%) -> %                    eval : (%,Equation %) -> %
eval : (%,List Equation %) -> %        ground : % -> R
ground? : % -> Boolean                 hash : % -> SingleInteger
latex : % -> String                    leadingCoefficient : % -> R
leadingMonomial : % -> %               map : ((R -> R),%) -> %
monomial? : % -> Boolean               monomials : % -> List %
one? : % -> Boolean                    primitiveMonomials : % -> List %
recip : % -> Union(%,"failed")         reductum : % -> %
retract : % -> Symbol                  retract : % -> R
sample : () -> %                       variables : % -> List Symbol
zero? : % -> Boolean                   ?~=? : (%,%) -> Boolean
?*? : (Fraction Integer,%) -> % if R has ALGEBRA FRAC INT
?*? : (%,Fraction Integer) -> % if R has ALGEBRA FRAC INT
?*? : (NonNegativeInteger,%) -> %
?**? : (%,NonNegativeInteger) -> %
?/? : (%,R) -> % if R has FIELD
?<? : (%,%) -> Boolean if R has ORDSET
?<=? : (%,%) -> Boolean if R has ORDSET
?>? : (%,%) -> Boolean if R has ORDSET
?>=? : (%,%) -> Boolean if R has ORDSET
D : (%,List Symbol,List NonNegativeInteger) -> %
D : (%,Symbol,NonNegativeInteger) -> %
?^? : (%,NonNegativeInteger) -> %
associates? : (%,%) -> Boolean if R has INTDOM
binomThmExpt : (%,%,NonNegativeInteger) -> % if R has COMRING
characteristic : () -> NonNegativeInteger
charthRoot : % -> Union(%,"failed") if $ has CHARNZ and R has PFECAT or R has CHARNZ
coefficient : (%,List Symbol,List NonNegativeInteger) -> %
coefficient : (%,Symbol,NonNegativeInteger) -> %
coefficient : (%,IndexedExponents Symbol) -> R
coerce : % -> % if R has INTDOM
coerce : Fraction Integer -> % if R has ALGEBRA FRAC INT or R has RETRACT FRAC INT
conditionP : Matrix % -> Union(Vector %,"failed") if $ has CHARNZ and R has PFECAT
content : (%,Symbol) -> % if R has GCDDOM
content : % -> R if R has GCDDOM
convert : % -> InputForm if Symbol has KONVERT INFORM and R has KONVERT INFORM
convert : % -> Pattern Integer if Symbol has KONVERT PATTERN INT and R has KONVERT PATTERN INT
convert : % -> Pattern Float if Symbol has KONVERT PATTERN FLOAT and R has KONVERT PATTERN FLOAT
degree : (%,List Symbol) -> List NonNegativeInteger
degree : (%,Symbol) -> NonNegativeInteger
```

```
degree : % -> IndexedExponents Symbol
differentiate : (%,List Symbol,List NonNegativeInteger) -> %
differentiate : (%,Symbol,NonNegativeInteger) -> %
differentiate : (%,List Symbol) -> %
discriminant : (%,Symbol) -> % if R has COMRING
eval : (%,List Symbol,List %) -> %
eval : (%,List Symbol,List R) -> %
exquo : (%,%) -> Union(%,"failed") if R has INTDOM
exquo : (%,R) -> Union(%,"failed") if R has INTDOM
factor : % -> Factored % if R has PFECAT
factorPolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial % if R has PFECAT
factorSquareFreePolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial % if R has PFECAT
gcd : (%,%) -> % if R has GCDDOM
gcd : List % -> % if R has GCDDOM
gcdPolynomial : (SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> SparseUnivariatePolynomial % if R has GCDDOM
integrate : (%,Symbol) -> % if R has ALGEBRA FRAC INT
isExpt : % -> Union(Record(var: Symbol,exponent: NonNegativeInteger),"failed")
isPlus : % -> Union(List %,"failed")
isTimes : % -> Union(List %,"failed")
lcm : (%,%) -> % if R has GCDDOM
lcm : List % -> % if R has GCDDOM
mainVariable : % -> Union(Symbol,"failed")
mapExponents : ((IndexedExponents Symbol -> IndexedExponents Symbol),%) -> %
max : (%,%) -> % if R has ORDSET
min : (%,%) -> % if R has ORDSET
minimumDegree : (%,List Symbol) -> List NonNegativeInteger
minimumDegree : (%,Symbol) -> NonNegativeInteger
minimumDegree : % -> IndexedExponents Symbol
monicDivide : (%,%,Symbol) -> Record(quotient: %,remainder: %)
monomial : (%,List Symbol,List NonNegativeInteger) -> %
monomial : (%,Symbol,NonNegativeInteger) -> %
monomial : (R,IndexedExponents Symbol) -> %
multivariate : (SparseUnivariatePolynomial %,Symbol) -> %
multivariate : (SparseUnivariatePolynomial R,Symbol) -> %
numberOfMonomials : % -> NonNegativeInteger
patternMatch : (%,Pattern Integer,PatternMatchResult(Integer,%)) -> PatternMatchResult(Integer,%) if Symbol has PATMAB INT and R has PATMAB INT
patternMatch : (%,Pattern Float,PatternMatchResult(Float,%)) -> PatternMatchResult(Float,%) if Symbol has PATMAB FLOAT and R has PATMAB FLOAT
pomopo! : (%,R,IndexedExponents Symbol,%) -> %
prime? : % -> Boolean if R has PFECAT
primitivePart : (%,Symbol) -> % if R has GCDDOM
```

```
 primitivePart : % -> % if R has GCDDOM
 reducedSystem : Matrix % -> Matrix R
 reducedSystem : (Matrix %,Vector %) -> Record(mat: Matrix R,vec: Vector R)
 reducedSystem : (Matrix %,Vector %) -> Record(mat: Matrix Integer,vec: Vector Integer) if R has LINEXP INT
 reducedSystem : Matrix % -> Matrix Integer if R has LINEXP INT
 resultant : (%,%,Symbol) -> % if R has COMRING
 retract : % -> Integer if R has RETRACT INT
 retract : % -> Fraction Integer if R has RETRACT FRAC INT
 retractIfCan : % -> Union(Symbol,"failed")
 retractIfCan : % -> Union(Integer,"failed") if R has RETRACT INT
 retractIfCan : % -> Union(Fraction Integer,"failed") if R has RETRACT FRAC INT
 retractIfCan : % -> Union(R,"failed")
 solveLinearPolynomialEquation : (List SparseUnivariatePolynomial %,SparseUnivariatePolynomial %) -> Union(List SparseUnivariatePolynomial %,"failed") if R has PFECAT
 squareFree : % -> Factored % if R has GCDDOM
 squareFreePart : % -> % if R has GCDDOM
 squareFreePolynomial : SparseUnivariatePolynomial % -> Factored SparseUnivariatePolynomial % if R has PFECAT
 subtractIfCan : (%,%) -> Union(%,"failed")
 totalDegree : (%,List Symbol) -> NonNegativeInteger
 totalDegree : % -> NonNegativeInteger
 unit? : % -> Boolean if R has INTDOM
 unitCanonical : % -> % if R has INTDOM
 unitNormal : % -> Record(unit: %,canonical: %,associate: %) if R has INTDOM
 univariate : % -> SparseUnivariatePolynomial R
 univariate : (%,Symbol) -> SparseUnivariatePolynomial %


P := Polynomial Q


   (57)  Polynomial Fraction Integer
                                                                  Type: Domain
                                                                   Time: 0 sec

P has EuclideanDomain


   (58)  false
                                                                 Type: Boolean
                                                    Time: 0.01 (IN) = 0.01 sec
```

```
(x + 1)/(y + 2)


         x + 1
   (59)  -----
         y + 2
                                   Type: Fraction Polynomial Integer
                                          Time: 0.01 (IN) = 0.01 sec

U := UnivariatePolynomial(x,Q)


   (60)  UnivariatePolynomial(x,Fraction Integer)
                                                        Type: Domain
                                                         Time: 0 sec

U has EuclideanDomain


   (61)  true
                                                       Type: Boolean
                                                         Time: 0 sec


p1: P := 6 * (x + 1) * (x + 2)


           2
   (62)  6x  + 18x + 12
                                   Type: Polynomial Fraction Integer
                                                         Time: 0 sec

p2: P := 9 * (x + 1) * ( x + 3)


           2
   (63)  9x  + 36x + 27
                                   Type: Polynomial Fraction Integer
                                          Time: 0.01 (IN) = 0.01 sec
```

```
gcd(p1,p2)


   (64)  x + 1
                                          Type: Polynomial Fraction Integer
                                                 Time: 0.01 (EV) = 0.01 sec

u1: U := p1


          2
   (65)  6x  + 18x + 12
                              Type: UnivariatePolynomial(x,Fraction Integer)
                                                 Time: 0.03 (IN) = 0.03 sec

u2: U := p2


          2
   (66)  9x  + 36x + 27
                              Type: UnivariatePolynomial(x,Fraction Integer)
                                                               Time: 0 sec

gcd(u1,u2)


   (67)  x + 1
                              Type: UnivariatePolynomial(x,Fraction Integer)
                                                 Time: 0.01 (IN) = 0.01 sec

gcd( (y^3-1) * p1, (y^2 -1) * p2)


   (68)  (x + 1)y - x - 1
                                          Type: Polynomial Fraction Integer
                                                 Time: 0.02 (EV) = 0.02 sec
Session1 (69) -> )spool

Finished dribbling to session1.out.
```

1.1 Conclusions

- Every AXIOM expression has a unique and precise type
- AXIOM has an inference mechanism to help the user in the desk-calculator mode
- This mechanism uses coercions between types.
- A given mathematical object may have several implementations in AXIOM which one with is different type (for instance multivariate polynomials can be viewed recursively univariate or distributed).
- Types control internal representations and available methods.
- The AXIOM programming language offers *types of types* which are extremely convenient for implementing algebraic structures.
- Types are organized in a hierarchical manner.
- Types of types are called *categories*. Other types are called *domains*.
- An AXIOM function can return a type (or admit a type among its arguments). Such a function is called a *constructor*.

2 A Second Session

2.1 Operations on rational numbers

AXIOM SESSION

```
FermatNumber(n:PositiveInteger):PositiveInteger == 2**2**n + 1

                                                                  Type: Void
                                                                  Time: 0 sec

l := [FermatNumber(i) for i in 1..6]


   (2)  [5,17,257,65537,4294967297,18446744073709551617]
```

```
                                                 Type: List PositiveInteger
                                   Time: 0.01 (IN) + 0.01 (OT) = 0.02 sec

[prime?(n) for n in l]


   (3)  [true,true,true,true,false,false]
                                                         Type: List Boolean
                                   Time: 0.01 (EV) + 0.02 (OT) = 0.03 sec

[factor(n) for n in l]


   (4)  [5,17,257,65537,641 6700417,274177 67280421310721]
                                                Type: List Factored Integer
                                   Time: 0.04 (EV) + 0.01 (OT) = 0.05 sec

-- factor FermatNumber(7)

[factorial(n) for n in 99..101]


   (5)
   [
    9332621544394415268169923885626670049071596826438162146859296389521759999_
     229915608941463976156518286253697920827223758251185210916864000000000000_
     000000000
     ,

    9332621544394415268169923885626670049071596826438162146859296389521759999_
     229915608941463976156518286253697920827223758251185210916864000000000000_
     00000000000
     ,

    9425947759838359420851623124482936749562312794702543768327889353416977599_
     162214765030878615918083469116234900035495995833697063026032640000000000_
     0000000000000
     ]
                                                         Type: List Integer
                                   Time: 0.01 (EV) + 0.01 (OT) = 0.02 sec
```

```
S(n:PositiveInteger):Fraction Integer == reduce(+,[1/k**2 for k in 1..n])

                                                                   Type: Void
                                                                  Time: 0 sec

[S(n) for n in 1..20]

   (7)
      5  49  205  5269  5369  266681  1077749  9778141  1968329  239437889
   [1, -, --, ---, ----, ----, ------, -------, -------, -------, ---------,
      4  36  144  3600  3600  176400   705600  6350400  1270080  153679680
    240505109  40799043101  40931552621  205234915681  822968714749
    ---------, -----------, -----------, ------------, ------------,
    153679680  25971865920  25971865920  129859329600  519437318400
    238357395880861  238820721143261  86364397717734821  17299975731542641
    ---------------, ---------------, -----------------, -----------------]
    150117385017600  150117385017600  54192375991353600  10838475198270720
                                                   Type: List Fraction Integer
                                                    Time: 0.03 (OT) = 0.03 sec
```

2.2 Operations on real numbers

AXIOM SESSION

```
Ran := RealClosure(Fraction Integer)


   (1)  RealClosure Fraction Integer
                                                                 Type: Domain
                                                                  Time: 0 sec

s2 := sqrt(2)$Ran


         +-+
   (2)  \|2
                                           Type: RealClosure Fraction Integer
                  Time: 0.01 (EV) + 0.02 (OT) + 0.03 (GC) = 0.06 sec
```

```
s5 := sqrt(5)$Ran


         +-+
   (3)  \|5
                                         Type: RealClosure Fraction Integer
                                                              Time: 0 sec

s10 := sqrt(10)$Ran


         +--+
   (4)  \|10
                                         Type: RealClosure Fraction Integer
                                                              Time: 0 sec

eq1:=sqrt(s10+3)*sqrt(s5+2) - sqrt(s10-3)*sqrt(s5-2) = sqrt(10*s2+10)


           +----------+ +---------+    +----------+ +---------+   +-----------+
           | +--+       | +-+          | +--+       | +-+         |   +-+
   (5)  - \|\|10  - 3 \|\|5  - 2  + \|\|10  + 3 \|\|5  + 2 = \|10\|2  + 10
                                Type: Equation RealClosure Fraction Integer
                        Time: 0.06 (IN) + 0.03 (EV) + 0.01 (OT) = 0.10 sec

eq1::Boolean


   (6)  true
                                                            Type: Boolean
                                               Time: 0.19 (IN) = 0.19 sec

precision()$Float


   (7)  68
                                                    Type: PositiveInteger
                                               Time: 0.01 (OT) = 0.01 sec
```

```
f2: Float := sqrt(2)


   (8)  1.4142135623 730950488
                                                                   Type: Float
                                                                   Time: 0 sec

f5: Float := sqrt(5)


   (9)  2.2360679774 997896964
                                                                   Type: Float
                                                                   Time: 0 sec

f10: Float := sqrt(10)


   (10)  3.1622776601 68379332
                                                                   Type: Float
                                                                   Time: 0 sec

eq1bis:=sqrt(f10+3)*sqrt(f5+2) - sqrt(f10-3)*sqrt(f5-2) = sqrt(10*f2+10)


   (11)  4.9134647270 262305698= 4.9134647270 262305698
                                                          Type: Equation Float
                                                    Time: 0.03 (IN) = 0.03 sec

eq1bis::Boolean


   (12)  true
                                                                 Type: Boolean
                                                                   Time: 0 sec


r3:Ran:=sqrt(3,5)


         5+-+
```

```
   (13)  \|3
                                              Type: RealClosure Fraction Integer
                                                                 Time: 0 sec
```

```
r25:Ran:=sqrt(1/25,5)
```

```
          +--+
          | 1
   (14)  5|--
         \|25
                                              Type: RealClosure Fraction Integer
                                                                 Time: 0 sec
```

```
r32:Ran:=sqrt(32/5,5)
```

```
          +--+
          |32
   (15)  5|--
         \| 5
                                              Type: RealClosure Fraction Integer
                                                                 Time: 0 sec
```

```
r27:Ran:=sqrt(27/5,5)
```

```
          +--+
          |27
   (16)  5|--
         \| 5
                                              Type: RealClosure Fraction Integer
                                                                 Time: 0 sec
```

```
eq2:=sqrt((r32-r27,3)) = r25*(1+r3-r3**2)
```

```
          +----------------+
          |   +--+    +--+                         +--+
          |   |27     |32       5+-+2   5+-+       | 1
```

```
   (17)  3|- 5|--  + 5|--  = (- \|3   + \|3  + 1) 5|--
         \|  \| 5    \| 5                         \|25
                                  Type: Equation RealClosure Fraction Integer
                                       Time: 0.01 (IN) + 0.05 (EV) = 0.06 sec
```

```
eq2::Boolean
```

```
   (18)  true
                                                                Type: Boolean
                                       Time: 3.14 (IN) + 0.30 (GC) = 3.44 sec
```

```
d3:Float:=nthRoot(3,5)
```

```
   (19)  1.2457309396 15517326
                                                                  Type: Float
                                       Time: 0.01 (EV) + 0.01 (OT) = 0.02 sec
```

```
d25:Float:=nthRoot(1/25,5)
```

```
   (20)  0.5253055608 8075344729
                                                                  Type: Float
                                                   Time: 0.01 (IN) = 0.01 sec
```

```
d32:Float:=nthRoot(32/5,5)
```

```
   (21)  1.4495593273 553910629
                                                                  Type: Float
                                                   Time: 0.01 (EV) = 0.01 sec
```

```
d27:Float:=nthRoot(27/5,5)
```

```
   (22)  1.4011310323 534026984
                                                                  Type: Float
                                                                   Time: 0 sec
```

```
eq2bis:=nthRoot((d32-d27,3)) = d25*(1+d3-d3**2)


   (23)  0.3645018412 1606803517= 0.3645018412 1606803518
                                                       Type: Equation Float
                                                  Time: 0.01 (EV) = 0.01 sec

eq2bis::Boolean


   (24)  false
                                                             Type: Boolean
```

2.3 Operations on matrices

AXIOM SESSION

```
N := PositiveInteger


   (1)  PositiveInteger
                                                              Type: Domain
                                                               Time: 0 sec

D := DoubleFloat


   (2)  DoubleFloat
                                                              Type: Domain
                                                               Time: 0 sec

F := Float


   (3)  Float
                                                              Type: Domain
```

```
                                                                 Time: 0 sec

Q := Fraction Integer


   (4)  Fraction Integer
                                                              Type: Domain
                                                                 Time: 0 sec


HilbertMatrixD(n:N): Matrix  D == _
  matrix([[1.0/(i + j) for i in 1..n] for j in 1..n])

                                                                Type: Void
                                               Time: 0.01 (IN) = 0.01 sec

HilbertMatrixF(n:N): Matrix  F == _
  matrix([[1.0/(i + j) for i in 1..n] for j in 1..n])

                                                                Type: Void
                                               Time: 0.01 (IN) = 0.01 sec

HilbertMatrixQ(n:N): Matrix  Q == _
  matrix([[1/(i + j) for i in 1..n] for j in 1..n])

                                                                Type: Void
                                                                 Time: 0 sec


HilbertMatrixD(3)


        +       0.5         0.333333333333333       0.25       +
        |                                                      |
   (8)  |0.333333333333333       0.25               0.2        |
        |                                                      |
        +      0.25               0.2        0.166666666666667+
                                                  Type: Matrix DoubleFloat
                     Time: 0.04 (IN) + 0.03 (OT) + 0.01 (GC) = 0.08 sec
```

```
HilbertMatrixQ(3)


       +1  1  1+
       |-  -  -|
       |2  3  4|
       |       |
       |1  1  1|
   (9) |-  -  -|
       |3  4  5|
       |       |
       |1  1  1|
       |-  -  -|
       +4  5  6+
                                     Type: Matrix Fraction Integer
                                       Time: 0.02 (OT) = 0.02 sec

HilbertMatrixF(3)


   (10)
   +        0.5           0.3333333333 3333333333          0.25         +
   |                                                                    |
   |0.3333333333 3333333333          0.25                     0.2         |
   |                                                                    |
   +       0.25                     0.2           0.1666666666 6666666667+
                                                     Type: Matrix Float
                     Time: 0.01 (IN) + 0.01 (OT) + 0.01 (GC) = 0.03 sec


[determinant(HilbertMatrixD(5*n)) for n in 1..6]


   (11)
   [1.4878155287776663e-14, 1.1708847390332544e-58, - 7.329551900980472e-127,
    1.8972182407450097e-202, - 1.7767073603611213e-280, 0.0]
                                                 Type: List DoubleFloat
                     Time: 0.43 (IN) + 0.14 (EV) + 0.06 (OT) = 0.63 sec

[determinant(HilbertMatrixF(5*n)) for n in 1..6]
```

```
   (12)
   [0.1487815528 7758289063 E -13, 0.1171371536 8024357091 E -57,
    0.8781685580 3800165504 E -132, 0.4005398450 3281293592 E -224,
    - 0.8672044362 0589913695 E -319, 0.1929750883 9978983829 E -417]
                                                              Type: List Float
                                           Time: 0.48 (EV) + 0.10 (GC) = 0.58 sec
```

```
[determinant(HilbertMatrixQ(5*n)) for n in 1..6]
```

```
   (13)
           1                                      1
   [--------------, ----------------------------------------------------------,
    67212633600000  8537000898240926708833515201784986712482596782080000000000

       1
    /
      146538740183666877481333971725269067959384123502368457086504025547136338_
       72815671732591854778627915571436076400640000000000000000000000
     ,

       1
    /
      327723880133278729476665890222832025204546816520485699917379710978406689_
       527792232144837486511702636314231098574648221249394901642708363125160340_
       152394721953723331337018968272407712485383771764018485657600000000000000_
       000000000000000000000000
     ,

       1
    /
      943443458848441228956693696279760730359903469457936499731851074111615233_
       142555144172252362467760786815226174481301118773695724847968976120659330_
       522376847473280302030846720901675566068651230469157335570350251653672110_
       811610752061204414113553644875364021461843141960577007670520243123626800_
       204426939629251703746714750574133248000000000000000000000000000000000000_
       000000000000000
     ,
```

```
      1
    /
     347678097173631327706181116037484788665140455596751194236374423476934278_
      72336185915903600949727565688512425687551728790953230983654612947880686_
      64034084877513911919412368602956835751833985175217203858528427028478411_
      07420065708261775033449064671483311289619473162541454992338969245601369_
      24601181973306894033338133775362887249764402725843635016814711320344415_
      32672272163948431375908684874930724465894995883925403831087253290653960_
      68715359788388174845939833221273153240082092406289203200000000000000000_
      00000000000000000000000000000000000000
     ]
                                              Type: List Fraction Integer
                                   Time: 0.31 (EV) + 0.01 (OT) = 0.32 sec
```

```
h6F := HilbertMatrixF(6)
```

```
   (14)
   [
     [0.5, 0.3333333333 3333333333, 0.25, 0.2, 0.1666666666 6666666667,
      0.1428571428 5714285714]
     ,

     [0.3333333333 3333333333, 0.25, 0.2, 0.1666666666 6666666667,
      0.1428571428 5714285714, 0.125]
     ,

     [0.25, 0.2, 0.1666666666 6666666667, 0.1428571428 5714285714, 0.125,
      0.1111111111 1111111111]
     ,

     [0.2, 0.1666666666 6666666667, 0.1428571428 5714285714, 0.125,
      0.1111111111 1111111111, 0.1]
     ,

     [0.1666666666 6666666667, 0.1428571428 5714285714, 0.125,
      0.1111111111 1111111111, 0.1, 0.0909090909 0909090909 1]
```

```
     ,

     [0.1428571428 5714285714, 0.125, 0.1111111111 1111111111, 0.1,
      0.0909090909 0909090909 1, 0.0833333333 3333333333 3]
     ]
                                                         Type: Matrix Float
                                                Time: 0.02 (OT) = 0.02 sec
```

```
h6F * inverse(h6F)
```

```
   (15)
   [[1.0,- 0.4 E -14,0.1 E -13,- 0.4 E -13,0.2 E -13,- 0.5 E -14],
    [0.6 E -16,0.9999999999 99999,0.2 E -14,- 0.2 E -13,0.0,- 0.7 E -14],
    [0.6 E -16,- 0.2 E -14,1.0,- 0.2 E -13,0.7 E -14,- 0.4 E -14],
    [0.6 E -16,- 0.2 E -14,0.5 E -14,0.9999999999 9998,0.0,- 0.5 E -14],
    [0.7 E -16,- 0.1 E -14,0.0,- 0.2 E -13,1.0,- 0.4 E -14],
    [0.3 E -16,- 0.9 E -15,0.4 E -14,- 0.1 E -13,0.7 E -14,1.0]]
                                                         Type: Matrix Float
                                  Time: 0.01 (EV) + 0.01 (OT) = 0.02 sec
```

```
h6Q := HilbertMatrixQ(6)
```

```
        +1  1  1  1   1   1 +
        |-  -  -  -   -   - |
        |2  3  4  5   6   7 |
        |                   |
        |1  1  1  1   1   1 |
        |-  -  -  -   -   - |
        |3  4  5  6   7   8 |
        |                   |
        |1  1  1  1   1   1 |
        |-  -  -  -   -   - |
        |4  5  6  7   8   9 |
   (16) |                   |
        |1  1  1  1   1    1|
        |-  -  -  -   -   --|
        |5  6  7  8   9   10|
        |                   |
```

```
         |1  1  1  1    1   1|
         |-  -  -  -   --  --|
         |6  7  8  9   10  11|
         |                   |
         |1  1  1   1   1   1|
         |-  -  -  --  --  --|
         +7  8  9  10  11  12+
                                                 Type: Matrix Fraction Integer
                                                                Time: 0 sec
```

```
h6Q * inverse(h6Q)
```

```
         +1  0  0  0  0  0+
         |                |
         |0  1  0  0  0  0|
         |                |
         |0  0  1  0  0  0|
   (17)  |                |
         |0  0  0  1  0  0|
         |                |
         |0  0  0  0  1  0|
         |                |
         +0  0  0  0  0  1+
                                                 Type: Matrix Fraction Integer
                                                   Time: 0.01 (IN) = 0.01 sec
```

```
[determinant(HilbertMatrixF(5*n)) for n in 1..10];
```

```
                                                           Type: List Float
                                        Time: 3.02 (EV) + 0.29 (GC) = 3.31 sec
```

```
[determinant(HilbertMatrixQ(5*n)) for n in 1..10];
```

```
                                                  Type: List Fraction Integer
                                        Time: 2.96 (EV) + 0.38 (GC) = 3.34 sec
```

```
[determinant(HilbertMatrixF(5*n)) for n in 1..15];


                                                          Type: List Float
                         Time: 0.01 (IN) + 14.09 (EV) + 1.59 (GC) = 15.69 sec

[determinant(HilbertMatrixQ(5*n)) for n in 1..15];


                                               Type: List Fraction Integer
                                     Time: 19.41 (EV) + 2.75 (GC) = 22.16 sec



[determinant(HilbertMatrixF(5*n)) for n in 1..20];


                                                          Type: List Float
                                     Time: 42.99 (EV) + 5.07 (GC) = 48.06 sec

[determinant(HilbertMatrixQ(5*n)) for n in 1..20];


                                               Type: List Fraction Integer
                                    Time: 75.55 (EV) + 11.16 (GC) = 86.71 sec
```

2.4 Operations on univariate polynomials

AXIOM SESSION

```
N := NonNegativeInteger


   (1)  NonNegativeInteger
                                                              Type: Domain
                                                               Time: 0 sec
```

```
Z := Integer


   (2)  Integer
                                                                 Type: Domain
                                                                  Time: 0 sec

UZ := UnivariatePolynomial(x,Z)


   (3)  UnivariatePolynomial(x,Integer)
                                                                 Type: Domain
                                                                  Time: 0 sec


u: UZ  := (3*x-1)**2 * (2*x + 8)


           3      2
   (4)  18x  + 60x  - 46x + 8
                                         Type: UnivariatePolynomial(x,Integer)
                                                    Time: 0.01 (OT) = 0.01 sec

v: UZ := (1 - 6*x + 9*x**2)**2


           4       3      2
   (5)  81x  - 108x  + 54x  - 12x + 1
                                         Type: UnivariatePolynomial(x,Integer)
                                                    Time: 0.01 (OT) = 0.01 sec


u**2 + u*v


             7        6        5         4        3        2
   (6)  1458x  + 3240x  - 7074x  + 10584x  - 9282x  + 4120x  - 878x + 72
                                         Type: UnivariatePolynomial(x,Integer)
                                                    Time: 0.01 (OT) = 0.01 sec
```

```
leadingCoefficient u


   (7)  18
                                                   Type: PositiveInteger
                                                            Time: 0 sec

degree u


   (8)  3
                                                   Type: PositiveInteger
                                                            Time: 0 sec

reductum u


           2
   (9)  60x  - 46x + 8
                                       Type: UnivariatePolynomial(x,Integer)
                                                            Time: 0 sec

gcd(u,v)


           2
   (10)  9x  - 6x + 1
                                       Type: UnivariatePolynomial(x,Integer)
                                                            Time: 0 sec

factor(u)


                         2
   (11)  2(x + 4)(3x - 1)
                              Type: Factored UnivariatePolynomial(x,Integer)
                                                            Time: 0 sec

factor(v)
```

```
                 4
   (12)  (3x - 1)
                              Type: Factored UnivariatePolynomial(x,Integer)
                                                                Time: 0 sec
```

```
PZ := Polynomial Z
```

```
   (13)  Polynomial Integer
                                                               Type: Domain
                                                                Time: 0 sec
```

```
pz: PZ := (4*x**3+2*y**2+1)*(12*x**5-x**3*y+12)
```

```
             3 3       5       2        6    3        8      5      3
   (14)  - 2x y  + (24x  + 24)y  + (- 4x  - x )y + 48x  + 12x  + 48x  + 12
                                                     Type: Polynomial Integer
                                                Time: 0.01 (IN) = 0.01 sec
```

```
factor(pz)
```

```
             3       5         2     3
   (15)  - (x y - 12x  - 12)(2y  + 4x  + 1)
                                            Type: Factored Polynomial Integer
                                                Time: 0.04 (EV) = 0.04 sec
```

```
Q := Fraction Integer
```

```
   (16)  Fraction Integer
                                                               Type: Domain
                                                                Time: 0 sec
```

```
PQ := Polynomial Q


   (17)  Polynomial Fraction Integer
                                                                  Type: Domain
                                                                   Time: 0 sec

pq: PQ := (4*x**3+(2/3)*x**2+1)*(12*x**5-(1/2)*x**3+12)


            8     7     6   35  5   95  3     2
   (18)  48x  + 8x  - 2x  + -- x  + -- x  + 8x  + 12
                             3       2
                                          Type: Polynomial Fraction Integer
                                     Time: 0.03 (IN) + 0.01 (OT) = 0.04 sec

factor(pq)


             3   1  2   1   5    1  3
   (19)  48(x  + - x  + -)(x  - -- x  + 1)
                 6      4       24
                                 Type: Factored Polynomial Fraction Integer
                                     Time: 0.03 (EV) + 0.01 (OT) = 0.04 sec
```

AXIOM SESSION

```
P := UnivariatePolynomial(x,Fraction Integer)


   (1)  UnivariatePolynomial(x,Fraction Integer)
                                                                  Type: Domain
                                                                   Time: 0 sec

euclideanGcd(a,b) ==
         l := [a];
```

```
        a:=unitCanonical a
        b:=unitCanonical b
        while not zero? b repeat
           l := cons(b,l)
           (a,b):= (b,a rem b)
           b:=unitCanonical b
        reverse l
```

```
                                                                    Type: Void
                                                                  Time: 0 sec
```

```
p1: P  := x^8 + x^6  -3*x^4 -3*x^3 +8*x^2 +2*x -5
```

```
         8    6     4     3     2
   (3)  x  + x  - 3x  - 3x  + 8x  + 2x - 5
                              Type: UnivariatePolynomial(x,Fraction Integer)
                                               Time: 0.01 (OT) = 0.01 sec
```

```
q1: P  := 3*x^6 + 5*x^4 - 4*x^2 -9*x +21
```

```
          6     4     2
   (4)  3x  + 5x  - 4x  - 9x + 21
                              Type: UnivariatePolynomial(x,Fraction Integer)
                                   Time: 0.01 (IN) + 0.01 (OT) = 0.02 sec
```

```
euclideanGcd(p1,q1)
```

```
   (5)
     8    6     4     3     2            6   5  4   4  2
   [x  + x  - 3x  - 3x  + 8x  + 2x - 5, x  + - x  - - x  - 3x + 7,
                                             3      3
     4   1  2   3   2   25     49      6150
    x  - - x  + -, x  + -- x - --, x - ----, 1]
         5      5       13     13      4663
                         Type: List UnivariatePolynomial(x,Fraction Integer)
                                   Time: 0.01 (IN) + 0.01 (OT) = 0.02 sec
```

```
p2: P := x^30 + x^19 + 1


        30    19
   (6)  x   + x   + 1
                                Type: UnivariatePolynomial(x,Fraction Integer)
                                                                  Time: 0 sec
q2: P := x^29 - x^17 + 3


        29    17
   (7)  x   - x   + 3
                                Type: UnivariatePolynomial(x,Fraction Integer)
                                                                  Time: 0 sec

euclideanGcd(p2,q2)


   (8)
     30    19       29    17       19    18
   [x   + x   + 1, x   - x   + 3, x   + x   - 3x + 1,

       18    17     11     10     9     8     7     6     5     4     3     2
      x   + x   - 3x   + 4x   - 4x  + 4x  - 4x  + 4x  - 4x  + 4x  - 4x  + 4x
    +
      - 4x - 2
    ,

       12   4  11   4  10   4  9   4  8   4  7   4  6   4  5   4  4   4  3
      x   - - x   + - x   - - x  + - x  - - x  + - x  - - x  + - x  - - x
            3       3       3      3      3      3      3      3      3
    +
      4  2   1     1
      - x  - - x + -
      3      3     3
    ,

       11   8732  10   8732  9   8732  8   10919  7   5816  6   10433  5
      x   - ---- x   + ---- x  - ---- x  + ----- x  - ---- x  + ----- x
            6545       6545      6545      6545       6545      6545
```

+

$$-\frac{676}{595}x^4 + \frac{9164}{6545}x^3 - \frac{8588}{6545}x^2 + \frac{1756}{1309}x + \frac{878}{1309}$$

,

$$x^{10} - x^9 - \frac{19619}{16}x^8 - \frac{6553}{4}x^7 - \frac{15277}{16}x^6 - 729x^5 - 242x^4 - 82x^3 - 26x^2$$

+

$$-\frac{59065}{16}x + \frac{19555}{16}$$

,

$$x^9 + x^8 + \frac{1}{3}x^7 + \frac{1}{3}x^6 + 3x^2 - 2x + \frac{1}{3},$$

$$x^8 + \frac{78620}{58777}x^7 + \frac{45799}{58777}x^6 + \frac{34992}{58777}x^5 + \frac{11616}{58777}x^4 + \frac{4080}{58777}x^3 + \frac{864}{58777}x^2$$

+

$$\frac{177403}{58777}x - \frac{58697}{58777}$$

,

$$x^7 + \frac{170782}{923785}x^6 + \frac{2173617}{3695140}x^5 - \frac{1746351}{3695140}x^4 + \frac{5658021}{3695140}x^3 - \frac{8599611}{3695140}x^2$$

+

$$\frac{2278869}{739028}x - \frac{1232437}{1847570}$$

,

$$x^6 - \frac{2515753}{142893}x^5 + \frac{15269857}{428679}x^4 - \frac{12227447}{428679}x^3 + \frac{2492039}{142893}x^2 - \frac{2531080}{428679}x$$

+

```
     4446728
     -------
      428679
   ,
   5   160644757  4   130375607  3   27115175  2   25927747     49124846
  x  - --------- x  + --------- x  - -------- x  + -------- x - --------,
        73785531       73785531      24595177      73785531     73785531
   4   4553589485  3   1129145127  2   163361143     2886544997
  x  - ---------- x  + ---------- x  + --------- x + ----------,
       7507014112      3753507056      234594191     7507014112
   3   112557717874  2   270014099776      15748937137
  x  - ------------ x  + ------------ x - ------------,
        43571774195      130715322585     130715322585
   2   619536582958      472414012299      19773994490288
  x  - ------------ x + -------------, x + --------------, 1]
       633789802471     2535159209884      34258486832515
                      Type: List UnivariatePolynomial(x,Fraction Integer)
                                          Time: 0.04 (OT) = 0.04 sec
```

2.5 Algebraic systems

AXIOM SESSION

```
N := NonNegativeInteger


   (3)  NonNegativeInteger
                                                         Type: Domain
                                                          Time: 0 sec

P := Polynomial Integer


   (4)  Polynomial Integer
                                                         Type: Domain
                                                          Time: 0 sec
```

```
epsilon := 1/10^21 :: Float


   (5)  0.1 E -20
                                                                  Type: Float
                                                                  Time: 0 sec

syst(n:N): List P == [x^n+y+z-1, x+y^n+z-1, x+y+z^n-1]

                                                                   Type: Void
                                                                  Time: 0 sec


completeExactSolve(syst(1))


   (7)  [[z= - %D - %C + 1,y= %C,x= %D]]
                         Type: List List Equation Fraction Polynomial Integer
                                                  Time: 0.03 (IN) = 0.03 sec


completeExactSolve(syst(2))


   (8)
   [[z= 0,y= 0,x= 1], [z= 0,y= 1,x= 0], [z= 1,y= 0,x= 0],
                2
    [z= x,y= x,x  + 2x - 1= 0]]
                         Type: List List Equation Fraction Polynomial Integer
                                                  Time: 0.09 (EV) = 0.09 sec

realApproximateSolve(syst(2),[x,y,z],epsilon)


   (9)
   [[- 2.4142135623 730950488,- 2.4142135623 730950488,- 2.4142135623 730950488]
     ,
    [0.4142135623 730950488,0.4142135623 730950488,0.4142135623 730950488],
    [1.0,0.0,0.0], [0.0,1.0,0.0], [0.0,0.0,1.0]]
                                                      Type: List List Float
```

```
                                    Time: 0.10 (EV) + 0.02 (OT) = 0.12 sec
```

```
completeExactSolve(syst(3))
```

```
   (10)
   [[z= - 1,y= 1,x= 1], [z= 0,y= 0,x= 1], [z= 1,y= - 1,x= 1], [z= 0,y= 1,x= 0],
    [z= 1,y= 0,x= 0], [z= 1,y= 1,x= - 1],
                 2          2         3
    [z= - y - x,y  + x y + x  - 1= 0,x  - x - 1= 0],
                                       3          3
                3                    - x  + 1    - x  + 1  4    2
    [z= x,y= x,x  + 2x - 1= 0], [z= --------,y= --------,x  - x  - 2x + 3= 0],
                                        2           2
                3          4    3     2
    [z= x,y= - x  - x + 1,x  + x  + 2x  + 1= 0],
           3               4    3     2
    [z= - x  - x + 1,y= x,x  + x  + 2x  + 1= 0]]
                        Type: List List Equation Fraction Polynomial Integer
              Time: 0.01 (IN) + 1.58 (EV) + 0.01 (OT) + 0.13 (GC) = 1.73 sec
```

```
realApproximateSolve(syst(3),[x,y,z],epsilon)
```

```
   (11)
   [[0.4533976515 1640376765,0.4533976515 1640376765,0.4533976515 1640376765],
    [1.0,1.0,- 1.0], [1.0,0.0,0.0], [0.0,1.0,0.0], [1.0,- 1.0,1.0],
    [0.0,0.0,1.0], [- 1.0,1.0,1.0]]
                                                         Type: List List Float
                        Time: 1.56 (EV) + 0.01 (OT) + 0.12 (GC) = 1.69 sec
```

```
completeExactSolve(syst(4))
```

```
   (12)
   [[z= 0,y= 0,x= 1], [z= 0,y= 1,x= 0], [z= 1,y= 0,x= 0],
                4
    [z= x,y= x,x  + 2x - 1= 0],
```

```
         4                11     8     7     5     4     3
 [z= - x  - x + 1,y= x,x   + 2x  - 3x  + 2x  - 4x  + 3x  - 2x + 2= 0],
              4           11     8     7     5     4     3
 [z= x,y= - x  - x + 1,x   + 2x  - 3x  + 2x  - 4x  + 3x  - 2x + 2= 0],

          4           4
       - x  + 1    - x  + 1
  [z= --------, y= --------,
           2           2
    11    10    9    8     7     6     4     3    2
   x   + x   + x  - x  - 4x  - 4x  + 4x  + 7x  - x  - 5x - 7= 0]
  ,

              4      2     4          8    5     4    2
  [z= - y - x  + 1, y  + (x  - 1)y + x  - x  - 2x  + x  + x + 1= 0,
    12     9     8     6     5     4     2
   x   - 2x  - 3x  + 2x  + 4x  + 3x  - 2x  - 2x - 2= 0]
  ]
                     Type: List List Equation Fraction Polynomial Integer
          Time: 0.01 (IN) + 9.63 (EV) + 0.03 (OT) + 0.96 (GC) = 10.63 sec

realApproximateSolve(syst(4),[x,y,z],epsilon)


   (13)
   [[- 0.8755550011 8456161483,- 0.8755550011 8456161483,1.2878847152 284854786]
     ,
    [- 0.8755550011 8456161482,1.2878847152 284854786,- 0.8755550011 8456161482]
     ,
    [1.2878847152 284854786,- 0.8755550011 8456161482,- 0.8755550011 8456161482]
     ,
    [- 1.3953369944 670730188,- 1.3953369944 670730188,- 1.3953369944 670730188]
     ,
    [0.4746266175 6260555033,0.4746266175 6260555033,0.4746266175 6260555033],
    [1.0,0.0,0.0], [0.0,1.0,0.0], [0.0,0.0,1.0]]
                                                  Type: List List Float
                      Time: 9.63 (EV) + 0.01 (OT) + 0.83 (GC) = 10.47 sec


-- completeExactSolve(syst(5))    MORE THAN FIVE MUNITES !!!!
```

2.6 Conclusions

- Observations.
  - Computer algebra algorithms perform symbolic computations: numbers are manipulated by using their mathematical definitions.
  - Hence results are exact and complete.
  - However they can be huge!
  - Moreover intermediate expressions may be much bigger than the input and output.
- Objectives.
  - Our main interest will be here to study the implementation of algorithms
    * that keep the swell of the intermediate expressions under control
    * and offer *optimal* time complexity.
  - Sine this is obviously a vast subject we will focus on univariate polynomials, bivariate polynomials and matrix operations.
  - But in fact these algorithms benefit to many other Computer algebra algorithms.
- Typical (trivial) example.
  - Let $p(x)$ and $q(x)$ be two univariate polynomials over the integer numbers and with degrees n and m respectively such that $n < m$.
  - Suppose we want to decide whether $p(x)$ divides $q(x)$.
  - One can show that the division with remainder of $q(x)$ by $p(x)$ requires at most $(2m+1)(n-m+1)$ operations in the coefficient ring.
  - If $p(x)$ divides $q(x)$ then for a given integer value x_0 of x the integer $p(x_0)$ divides $q(x_0)$.

– Computing $p(x_0)$ and $q(x_0)$ require $2\,n+2\,m+2$ operations in the coefficient ring.

– So trying whether $p(x_0)$ divides $q(x_0)$ (before computing the remainder of $q(x)$ by $p(x)$) will save time (and space) in the average.

3 A review of complexity notions

3.1 Orders of magnitude

Let f, g et h be functions from $\mathbb{N}$ to $\mathbb{R}$

- We say that $g(n)$ is in the <u>ORDER OF MAGNITUDE</u> of $f(n)$ and we write $f(n) \in \Theta(g(n))$ if there exist two (strictly) positive constants c_1 and c_2 such that for n big enough we have

$$0 \;\leq\; c_1\; g(n) \;\leq\; f(n) \;\leq\; c_2\; g(n). \tag{1}$$

- We say that $g(n)$ is an <u>ASYMPTOTIC UPPER BOUND</u> of $f(n)$ and we write $f(n) \in \mathcal{O}(g(n))$ if there exists a (strictly) positive constants c_2 such that for n big enough we have

$$0 \;\leq\; f(n) \;\leq\; c_2\; g(n). \tag{2}$$

- We say that $g(n)$ is an <u>ASYMPTOTIC LOWER BOUND</u> of $f(n)$ and we write $f(n) \in \Omega(g(n))$ if there exists a (strictly) positive constants c_1 such that for n big enough we have

$$0 \;\leq\; c_1\; g(n) \;\leq\; f(n). \tag{3}$$

Let us give some examples.

- With $f(n) = \frac{1}{2}n^2 - 3n$ and $g(n) = n^2$ we have $f(n) \in \Theta(g(n))$. Indeed we have

$$c_1\, n^2 \;\leq\; \frac{1}{2}n^2 - 3n \;\leq\; c_2\, n^2. \tag{4}$$

for $n \geq 12$ with $c_1 = \frac{1}{4}$ and $c_2 = \frac{1}{2}$.

- Assume that there exists a positive integer n_0 such that $f(n) > 0$ and $g(n) > 0$ for every $n \geq n_0$. Then we have

$$max(f(n), g(n)) \in \Theta(f(n) + g(n)). \tag{5}$$

  Indeed we have

$$\frac{1}{2}(f(n) + g(n)) \quad \leq \quad max(f(n), g(n)) \quad \leq \quad (f(n) + g(n)). \tag{6}$$

- Assume a and b are positive real constants. Then we have

$$(n + a)^b \in \Theta(n^b). \tag{7}$$

  Indeed for $n \geq a$ we have

$$0 \quad \leq \quad n^b \quad \leq \quad (n + a)^b \quad \leq \quad (2n)^b. \tag{8}$$

  Hence we can choose $c_1 = 1$ and $c_2 = 2^b$.

We have the following properties.

- $f(n) \in \Theta(g(n))$ holds iff $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$ hold together.
- Each of the predicates $f(n) \in \Theta(g(n))$, $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$ define a reflexive and transitive binary relation among the $\mathbb{N}$-to-$\mathbb{R}$ functions. Moreover $f(n) \in \Theta(g(n))$ is symmetric.
- We have the following TRANSPOSITION FORMULA

$$f(n) \in \mathcal{O}(g(n)) \quad \Longleftrightarrow \quad g(n) \in \Omega(f(n)). \tag{9}$$

In practice $\in$ is replaced by $=$ in each of the expressions $f(n) \in \Theta(g(n))$, $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$. Hence, the following

$$f(n) = h(n) + \Theta(g(n)) \tag{10}$$

means

$$f(n) - h(n) \in \Theta(g(n)). \tag{11}$$

Let us give another fundamental example. Let $p(n)$ be a (univariate) polynomial with degree $d > 0$. Let a_d be its leading coefficient and assume $a_d > 0$. Then we have

(1) if $k \geq d$ then $p(n) \in \mathcal{O}(n^k)$,

(2) if $k \leq d$ then $p(n) \in \Omega(n^k)$,

(3) if $k = d$ then $p(n) \in \Theta(n^k)$.

Exercise: Prove the following

$$\Sigma_{k=1}^{k=n} k \ \in \ \Theta(n^2). \tag{12}$$

3.2 The complexity of divide-and-conquer algorithms

Divide-and-conquer algorithms proceed as follows.

Divide the input problem into sub-problems.

Conquer on the sub-problems by solving them directly if they are small enough or proceed recursively.

Combine the solutions of the sub-problems to obtain the solution of the input problem.

Equation satisfied by $T(n)$. Assume that the size of the input problem increases with an integer n. Let $T(n)$ be the time complexity of a divide-and-conquer algorithm to solve this problem. Then $T(n)$ satisfies an equation of the form:

$$T(n) = a\,T(n/b) + f(n). \tag{13}$$

where $f(n)$ is the cost of the combine-part, $a \geq 1$ is the number of recursively calls and n/b with $b > 1$ is the size of a sub-problem.

Labeled tree associated with the equation. Assume n is a power of b, say $n = b^p$. To *solve* this equation we can associate a labeled tree $\mathcal{A}(n)$ to it as follows.

(1) If $n = 1$, then $\mathcal{A}(n)$ is reduced to a single leaf by labeled $T(1)$.

(2) If $n > 1$, then the root of $\mathcal{A}(n)$ is labeled by $f(n)$ and $\mathcal{A}(n)$ possesses a labeled sub-trees all equal to $\mathcal{A}(n/b)$.

The labeled tree $\mathcal{A}(n)$ associated with $T(n) = a\,T(n/b) + f(n)$ has height $p + 1$. Moreover the sum of its labels is $T(n)$.

Let us give two examples.

- Consider the relation:

$$T(n) = 2\,T(n/2) + n^2. \tag{14}$$

We obtain:

$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots + \frac{n^2}{2^p} + n\,T(1). \tag{15}$$

Hence we have:

$$T(n) \in \Theta(n^2). \tag{16}$$

- Consider the relation:

$$T(n) = 3T(n/3) + n. \tag{17}$$

We obtain:

$$T(n) \in \Theta(\log_3(n)n). \tag{18}$$

A formula to estimate $T(n)$. Let $a > 0$ be an integer and let $S, T : \mathbb{N} \longrightarrow \mathbb{R}_+$ be functions such that

(i) $S(2\,n) \geq 2\,S(n)$ and $S(n) \geq n$.

(ii) If $n = 2^p$ then $T(n) \leq a\,T(n/2) + S(n)$.

Then for $n = 2^p$ we have

(1) if $a = 1$ then

$$T(n) \leq (2 - 2/n)\,S(n) + T(1) \in \mathcal{O}(S(n)), \tag{19}$$

(2) if $a = 2$ then

$$T(n) \leq S(n)\log_2(n) + T(1)\,n \in \mathcal{O}(\log_2(n)\,S(n)), \tag{20}$$

(3) if $a \geq 3$ then

$$T(n) \leq \frac{2}{a-2}\left(n^{\log_2(a)-1} - 1\right) S(n) + T(1)\,n^{\log_2(a)} \in \mathcal{O}(S(n)\,n^{\log_2(a)-1}). \tag{21}$$

Indeed

$$
\begin{array}{rcl}
T(2^p) & \leq & a\,T(2^{p-1}) + S(2^p) \\
& \leq & a\,[a\,T(2^{p-2}) + S(2^{p-1})] + S(2^p) \\
& = & a^2\,T(2^{p-2}) + a\,S(2^{p-1}) + S(2^p) \\
& \leq & a^2\,[a\,T(2^{p-3}) + S(2^{p-2})] + a\,S(2^{p-1}) + S(2^p) \\
& = & a^3\,T(2^{p-3}) + a^2\,S(2^{p-2}) + a\,S(2^{p-1}) + S(2^p) \\
& \leq & a^p\,T(1) + \Sigma_{j=0}^{j=p-1}\,a^j\,S(2^{p-j})
\end{array}
\tag{22}
$$

Moreover

$$
\begin{array}{rcl}
S(2^p) & \geq & 2\,S(2^{p-1}) \\
S(2^p) & \geq & 2^2\,S(2^{p-2}) \\
\vdots & \vdots & \vdots \\
S(2^p) & \geq & 2^j\,S(2^{p-j})
\end{array}
\tag{23}
$$

Thus

$$
\Sigma_{j=0}^{j=p-1}\,a^j\,S(2^{p-j}) \;\leq\; S(2^p)\,\Sigma_{j=0}^{j=p-1}\left(\frac{a}{2}\right)^j. \tag{24}
$$

Hence

$$
T(2^p) \;\leq\; a^p\,T(1) + S(2^p)\,\Sigma_{j=0}^{j=p-1}\left(\frac{a}{2}\right)^j. \tag{25}
$$

For $a = 1$ we obtain

$$
\begin{array}{rcl}
T(2^p) & \leq & T(1) + S(2^p)\,\Sigma_{j=0}^{j=p-1}\,\left(\frac{1}{2}\right)^j \\
& = & T(1) + S(2^p)\,\frac{\frac{1}{2^p}-1}{\frac{1}{2}-1} \\
& = & T(1) + S(n)\,(2 - 2/n).
\end{array}
\tag{26}
$$

For $a = 2$ we obtain

$$
\begin{array}{rcl}
T(2^p) & \leq & 2^p\,T(1) + S(2^p)\,p \\
& = & n\,T(1) + S(n)\log_2(n).
\end{array}
\tag{27}
$$

4 Implementing Computer Algebra: basic ideas

4.1 Which mathematical types do we need to implement?

Obviously we need to implement the various usual sets of numbers:

- the ring of <u>INTEGER NUMBERS</u> $\mathbb{Z}$,

- the field of <u>RATIONAL NUMBERS</u> $\mathbb{Q}$,
- the rings of <u>MODULAR INTEGERS</u> $\mathbb{Z}/n\mathbb{Z}$ (and especially those which are prime fields),
- the different sets of <u>FLOATING POINT NUMBERS</u> (with fixed or arbitrary precision),
- the ordered ring of <u>REAL ALGEBRAIC NUMBERS</u>,
- and many others that we will discover later.

Clearly we need also:

- <u>UNIVARIATE OR MULTIVARIATE POLYNOMIALS</u> over each of these sets of numbers,
- <u>MATRICES</u> over the same sets,
- and also polynomial or matrix rings over any ring we can consider, including polynomial or matrix rings themselves.

So we need infinitely many algebraic types. Let us call these types RINGS.

Moreover we need <u>ALGEBRAIC STRUCTURES</u> (ring, commutative ring, Euclidean domain, field, finite field, prime field, univariate polynomial ring, ...) for the following reasons.

Genericity We can organize our infinitely many RINGS in a finite number of RING types. Then, for instance, we can define the *Euclidean algorithm* once for all for every Euclidean domain. For instance, for any Euclidean domain R we need to say

```
gcd(a,b) ==
  while not zero? b repeat
     (a,b):= (b,a rem b)
  return a
```

Conditional declaration A matrix ring (or polynomial ring) may possess an operation (inverse, gcd, ...) that another does not and this fact depends on its coefficient ring. So our MATRIX RINGS and POLYNOMIAL RINGS should depend on the properties of their coefficient RING. For instance we need to say:

if R is a field then the ring $R[x]$ of univariate polynomials over R is an Euclidean domain.

Conditional implementation The implementation of an operation defined in several univariate polynomial rings like

$$(p, n) \longmapsto p^n \tag{28}$$

may depend on the properties of the coefficient RING. Let R be a ring and $p(X) = aX + b$ be a univariate polynomial in X with coefficient a, b in R. If R is a non-commutative ring then

$$(aX+b)^3 \;=\; a^3X^3+(a^2b+aba+ba^2)X^2+(abb+bba+bab)X+b^3. \tag{29}$$

If R is any commutative ring then

$$(aX + b)^3 \;=\; a^3X^3 + 3\,a^2bX^2 + 3\,ab^2X + b^3. \tag{30}$$

If R is $\mathbb{Z}/3\mathbb{Z}$ then

$$(aX + b)^3 \;=\; a^3X^3 + b^3. \tag{31}$$

Specialization Among all RINGS of a given type one may have a much better algorithm for a given RING. So for this particular one, we may wish to replace the *generic definition* of an operation. For instance for the Euclidean domain of the integers we may wish to say (with some optimizations ...)

```
gcd(a,b) ==
  while a ~= b repeat
     if a < b then b := b-a else a := a-b
  return a
```

4.2 How to encode the elements of these types?

We will discuss here the case of numbers and polynomials. In this whole section we assume that we are using a machine with N-bit words. (Typically $N = 32$ or $N = 64$.)

4.2.1 Integers.

One machine word can contain a single precision integer in the range $[0, 2^N - 1]$. To represent integers outside of this range, so called multiprecision integer, we use arrays of N-bit words. To be precise we consider the 2^N-ary (or radix 2^N) expansion of a nonzero integer:

$$a \quad = \quad (-1)^s \ \Sigma_{0 \leq \imath \leq n} a_i 2^{Ni} \tag{32}$$

where

- every $a_i \in [0, 2^N - 1]$ is a digit in the 2^N-ary expansion of a,
- $s \in \{0, 1\}$ determines the sign of a,
- $n + 1 \in [1, 2^N]$ is the number of 2^N-digits in the 2^N-ary expansion of a,
- $a_n \neq 0$.

Then the integer a with $(n+1)$ 2^N-digits can be represented by an array (of N-bit words) with length $n+3$ since we need

- one word for s and
- one word for n

The reasonable unit operation for integers is the the word operation. One can easily check that

- the addition of two integers with at most n 2^N-digits requires $\mathcal{O}(n)$ word operations
- the multiplication of two integers with at most n 2^N-digits requires $\mathcal{O}(n^2)$ word operations.

4.2.2 Rational numbers.

It is natural to represent the rational number n/d as of the record of the two integers n and d such that

- $d > 0$ and
- $\gcd(n, d) = 1$.

4.2.3 Modular Integers.

There are many ways to implement the rings $\mathbb{Z}/m\mathbb{Z}$ and even more for the fields $\mathbb{Z}/p\mathbb{Z}$ (where p is a prime). Let us restrict to the fields $\mathbb{Z}/p\mathbb{Z}$. Here are some examples.

- For p in the range $[2, 2^N - 1]$ we can use single integers. Multiplication and addition can be made in $\mathcal{O}(1)$ word operations. **N.B.** multiplication requires a two single integers buffer unless p in the range $[2, 2^{N-1} - 1]$. Exponentiation and computation of inverses require a bit more of works but are still in $\mathcal{O}(1)$ word operations. For instance here's an ALDOR code for the division of two integers modulo a third one

```
mod_/(i: %, j: %, n: %): % == {
          local c0, d0, c1, d1: %;
          (c0, d0) := (j, n);
          (c1, d1) := (1, 0);
          while not zero? d0 repeat {
                  q := c0 quo d0;
                  (c0, d0) := (d0, c0 - q*d0);
                  (c1, d1) := (d1, c1 - q*d1)
          }
          if c0 ~= 1 then error "inverse does not exist";
          if c1 < 0  then c1 := c1 + n;
          mod_*(i, c1, n);
```

- The set of the nonzero elements of $\mathbb{Z}/p\mathbb{Z}$ is a cyclic group. In other words, there exists a generator $g \in \mathbb{Z}/p\mathbb{Z}$ such that every nonzero $x \in \mathbb{Z}/p\mathbb{Z}$ is a power of g called the discrete logarithm of x. Hence we can use discrete logarithms to represent nonzero elements of $\mathbb{Z}/p\mathbb{Z}$. Then exponentiation and computation of inverses become much more easy (for inverses we use Little Fermat Theorem) than with the previous encoding. However addition requires the use of tables of logarithms and exponentiations, for efficiency. Again all arithmetic operations are in $\mathcal{O}(1)$ word operations.

4.2.4 Real algebraic numbers.

Will be discussed during later.

4.2.5 Univariate polynomials.

Let R be a ring. The univariate polynomial ring $R[x]$ can be implemented using different data types.

Expression trees. All arithmetic operations are in $\mathcal{O}(1)$. But the representation is **not canonical**: two different expression trees can encode the same polynomial. Therefore operations like DEGREE, LEADING COEFFICIENT, REDUCTUM are in $\mathcal{O}(n)$ where n is the degree of the polynomial. Even worst the equality-test is $\mathcal{O}(n^2)$ (for non-balanced trees).

Arrays of coefficients. The polynomial

$$p(x) \quad = \quad a_n x^n + \cdots + a_1 x + a_0 \tag{33}$$

is coded by a record consisting of

- a single integer s,
- a single integer $d \leq s + 1$,
- an array of size s such that $a_0 + ... + a_n x^n$ is represented by $[a_0, ..., a_n, ...]$ and $d = n$.

This representation is said *dense* becasue all a_i are coded, even those which are null. This representation is canonical. Hence operations like DEGREE, LEADING COEFFICIENT, REDUCTUM are in $\mathcal{O}(1)$. Addition and equality-test are in $\mathcal{O}(n)$ and multiplication is in $\mathcal{O}(n^2)$. This representation is especially good when the ring of coefficients is a small prime field, i.e. $\mathbb{Z}/p\mathbb{Z}$ with p prime and in the range $[2, 2^N - 1]$.

List of terms. The polynomial

$$p(x) \quad = \quad a_n x^n + \cdots + a_1 x + a_0 \tag{34}$$

is coded by the list L of records $[a_i, i]$ where a_i is a nonzero coefficient and such that L is sorted decreasingly w.r.t. i. This representation is said *sparse*, since only the nonzero a_i are coded. This representation is also canonical. Hence operations like DEGREE, LEADING COEFFICIENT, REDUCTUM are in $\mathcal{O}(1)$. Moreover the operation REDUCTUM does not require coefficient duplication (on the contrary of the previous representation). Addition and equality-test are in $\mathcal{O}(n)$ and multiplication is in $\mathcal{O}(n^2)$. This representation is especially good when the ring of coefficients is itself a ring of sparse polynomials.

4.2.6 Multivariate polynomials.

Let R be a ring and $X = \{x_1, \ldots, x_n\}$ be a finite set of variables. The multivariate polynomial ring $R[X]$ can be implemented in different ways.
<u>RECURSIVELY.</u>

- If $n = 1$ we can use a univariate representation
- otherwise we can view $R[X]$ as a univariate polynomial ring with a multivariate polynomial ring as coefficient ring, say for instance $R[x_1, \ldots, x_{n-1}][x_n]$.

This implies to choose an ordering on the variables and a representation for univariate polynomials.

<u>AS LINEAR COMBINATIONS OF MONOMIALS.</u>

- By *monomial* we mean any product of variables.
- The product of variable is assumed to be commutative and induces a commutative multiplication for monomials.
- Each polynomial can be viewed as a linear combination of monomials (with coefficients in R).
- Then the polynomial

  $$p \ = \ a_1 m_1 + \cdots + a_p m_p. \qquad (35)$$

  where the m_i are pairwise different monomials and the a_i are nonzero coefficients, can be represented as an aggregate of terms $[a_i, m_i]$.
- This provides us with a canonical representation.
- To have a fast equality-test, the aggregate of terms must be *linear*. In other words, there should be a *first* term, a *second* term, ... Thus this aggregate should better be a *list* or an *array* and the monomials should be totally ordered.
- Two types of monomial orderings are frequently used.
  - The lexicographical ordering. With $X = \{x > y > z\}$ we have

    $$1 < z < \cdots < z^n \cdots < y < yz < \cdots < yz^n \cdots < y^2 < y^2 z < \cdots < y^2 z^n \cdots \qquad (36)$$

– The degree-lexicographical ordering. With $X = \{x > y > z\}$ we have

$$1 < z < y < x < z^2 < zy < y^2 < zx < xy < x^2 < \cdots < \qquad (37)$$

4.3 What are our requirements for the language?

4.3.1 Bounded genericity with dependent types

Recall that the RINGS R in Computer Algebra are obtained by applying rules like

(1) $R = \mathbb{Z}$,

(2) $n \longmapsto \mathbb{Z}/n\mathbb{Z}$,

(3) $R \longmapsto R[X]$,

(4) $(R, \mathcal{I}) \longmapsto R/\mathcal{I}$

(5) $R \longmapsto \mathbf{Fr}(R)$,

(6) $(R_1, R_2) \longmapsto R_1 \times R_2$.

To implement the above rules we would like to do the following constructions in our programming language.

- To define functions which return a type (as in rule **(2)**).
- To define functions that can be parametrized by a type (as in rule **(3)**). Hence we need <u>GENERICITY</u>.
- To handle <u>INTERFACE</u> (or type of types). Indeed in **(3)** we need R to be a ring. Hence we need <u>BOUNDED</u> <u>GENERICITY</u>.
- In rule **(4)** we need to say that $\mathcal{I}$ is an ideal of R. Hence we need <u>BOUNDED</u> <u>GENERICITY</u> <u>WITH</u> <u>DEPENDENT</u> <u>TYPES</u>.
- We should be able to have functions as parameters of other functions. For instance an ordered abelian free monoid genetated by a set S depends on a total ordering on S. Hence functions must have types too. Finally, both functions and types must have types and must be possible parameters for functions. Hence functions and types must be <u>TREATED</u> <u>AS</u> <u>VALUES</u>.

4.3.2 Inheritance

Recall that we need to implement algebraic structures like: ring, commutative ring, euclidean domain, field, ... Moreover we want to say that *an euclidean domain is a commutative ring*. Hence we want INHERITANCE BETWEEN INTERFACES. Of course we need also INHERITANCE BETWEEN CONCRETE CLASSES. For instance:

- A prime finite field $\mathbb{Z}/p\mathbb{Z}$ is a particular ring of modular integers (whose characteristic is a prime). Hence its implementation should inherit from that of the modular integer ring $\mathbb{Z}/m\mathbb{Z}$ (where m is not necessarily a prime).

- A multivariate polynomial ring $R[x_1, x_2, \ldots, x_n]$ is a particular univariate polynomial ring $(R[x_1, x_2, \ldots, x_{n-1}])[x_n]$. Hence its implementation should be able to reuse that of a *generic* univariate polynomial ring.

4.3.3 Post-facto extensions

We need a mechanism to enhance a given type. For instance,

- to define the *monoid* of non-negative integers we need the type MONOID and

- to define the exponentiation $(p, n) \longmapsto p^n$ when defining monoids, we need the set of non-negative integers to be defined already.

The solution is to be able

1. to define first the set of non-negative integers without viewing it as a MONOID,

2. then to define the interface MONOID

3. and finally to enhance the concrete class of non-negative integers by adding to it the property of being a MONOID.

4.3.4 Efficiency

In any MONOID exponentiation can de defined by the binary powering algorithm

```
(^)(x: %, n: NonNegativeInteger): % ==
        (n < 0) =>
                error "power:  negative exponent";
        n = 0 => 1;
        n = 1 => x;
        n = 2 => x*x;

        -- General code.
        i: SingleInteger := 0;
        l: SingleInteger := length n;
        u: % := 1;
        repeat {
                if bit(n, i) then u := u * x;
                if i >= l then break;
                x := x * x;
                i := i + 1;
        }
        u
}
```

Assume that in the interface Monoid the exponentiation has been defined as above. In the Monoid $\mathbb{Z}/p\mathbb{Z}$ (where p is a prime) we can do better because of little Fermat theorem which states that for $x \neq 0$ we have $x^{p-1} = 1$. Hence we need to overwrite the above definition for $\mathbb{Z}/p\mathbb{Z}$ by something like

```
(^)(x: %, n: NonNegativeInteger): % == {
    n := positiveRemainder(n,p);
    zero? n => 1;
    one? x => x;
    local acc: %;
    if odd? n then acc := x else acc := 1;
    repeat {
        n := shift(n,-1);
        zero? n => return acc;
        x := (x * x) mod p;
        if odd? n then acc := (acc * x) mod p;
    }
}
```

This will be made possible by <u>INHERITANCE.</u> In a polynomial ring $R[x]$ one can do better too in some special cases depending on the coefficient ring, as we have seen before. This will be made possible by <u>CONDITIONAL IMPLEMENTATION.</u>

`http://www-math.uni-paderborn.de/mca`.

References

[vzGG99] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.