Provable programming of algebra: arithmetic of fractions.

Sergei D. Meshveliani [1]

Program Systems Institute of Russian Academy of sciences,
Pereslavl-Zalessky, Russia. http://botik.ru/PSI

September 29, 2017

[1] This work is supported by the FANO project of the Russian Academy of Sciences, the project registration No AAAA-A16-116021760039-0.

Adequate programming of mathematics

- Constructive mathematics, constructive logic for CA.
- A functional language with dependent types. `Agda` (do not confuse with `Ada` !).
- Representing an algebraic domain depending on a dynamic parameter.
- Mathematical definitions and formal proofs are a part of a program, understood by the compiler and automatically checked *before* run-time.

- Termination proof is required.
- Exclusive “or” is applicable — for a decidable relation.
- Performance is not damaged.
- Full formal constructive proofs required or ‘`postulate`’.
- DoCon-A 2.00.

About the Agda syntax

Example: `n≤5+2` is an identifier,

and in `n ≤ 5 + 2`, `≤` and `+` are operators.

`S → T` means the type of functions from `S` to `T`.

A statement `P` is expressed as a type family (`T`).

A proof for `P` is any value in `T`.

`P => Q` is expressed as `S → T`.

A proof of a theorem is any function (algorithm) that returns a value in the corresponding type.

Fraction field (`Fraction R`)

The three versions are considered:

integral ring `R`, GCD-ring, Unique factorization ring.

The elements of `Q = Fraction R` are represented as

$$n/d, \quad n, d \in R, \quad d \neq 0.$$

Equality and arithmetic:

n/d =' n'/d' = n * d' $\approx$ n' * d

n/d *' n'/d' = (n * n') / (d * d')

n/d +' n'/d' = (n * d' + n' * d) / (d * d')

divide n/d n'/d' = n/d *' d'/n' for n' $\not\approx$ 0

For an *integral ring* R,

this domain satisfies the properties of a *field*.

Cancel by gcd !

Example 1:

$$\Sigma_{k=1}^{n}\ 1/k,$$

Example 2:

put to a matrix 10×10 $n/1$ with random $0 < n < 10$

and compute determinant by Gauss method.

`DoCon-A 2.00` applies eager cancelling by gcd,

and uses a fraction representation with `Coprime num denom`.

But there are needed

- GCDRing R,
- machine-checked proofs for `Field (Fraction R)`

  for the corresponding operations.

Naive definition and methods

```
record Prefraction : Set where
       constructor preFr
       field
         num      :  C
         denom    :  C
         denom#0  :  denom # 0#

_='_ :  Rel Prefraction _
f =' g =  (num f * denom g) ≈ (num g * denom f)
```

```
_*'_ : Op₂ Prefraction
(preFr n d d≁0) *' (preFr n' d' d'≁0) =
                                preFr (n * n') (d * d') dd'≁0
                                where
                                dd'≁0 = nz*nz d≁0 d'≁0

_+'_ : Op₂ Prefraction
(preFr n d d≁0) +' (preFr n' d' d'≁0) =
                          preFr ((n * d') + (n' * d)) (d * d')
                                                (nz*nz d≁0 d'≁0)
```

Half-optimized arithmetic

The division relation in a semigroup:

```
_|_  : Rel C _
x | y =  ∃ \q → x • q ≈ y
```

The coprimality notion for any monoid:

```
Coprime :  Rel C _
Coprime a b =  (c : C) → c | a → c | b → c | ε
```

GCD, GCD-ring

```
record GCD (a b : C) :  Set
  where
  constructor gcd′
  field
    proper    : C                    -- proper gcd value
    divides₁  : proper ∣ a
    divides₂  : proper ∣ b
    greatest :
              ∀ {d} → (d ∣ a) → (d ∣ b) → (d ∣ proper)

gcd :  (a b : C) → GCD a b
```

The fraction notion for any GCD-ring

```
record Fraction : Set where
       constructor fr′
       field  num      : C
              denom    : C
              denom≉0  : denom ≉ 0#
              coprime  : Coprime num denom

fraction :  (a b : C) → b ≉ 0# → Fraction

_*₁_ : Op₂ Fraction
(fr′ n d d≉0 _) *₁ (fr′ n' d' d'≉0 _) =
                   fraction (n * n') (d * d') dd'≉0
                   where
                   dd'≉0 = nz*nz d≉0 d'≉0
```

```
_+₁_ :  Op₂ Fraction
(fr′ n d d≠0 _) +₁ (fr′ n’ d’ d’≠0 _) =
       fraction ((n * d’) + (n’ * d)) (d * d’) dd’≠0
       where
       dd’≠0 = nz*nz d≠0 d’≠0
```

Optimized arithmetic

Its correctness requires an unique factorization domain.

```
n1/d1 + n2/d2 =  ((n1d2' + n2d1')/'g1) / (d1'd2'(g/'g1))
                 where
                 g   = gcd d1 d2
                 d1' = d1 /' g
                 d2' = d2 /' g
                 g1  = gcd (n1d2' + n2d1') g
```

```
_+fr_ :  Op₂ Fraction
(fr′ n₁ d₁ d₁≉0 coprime-n₁d₁) +fr
                  (fr′ n₂ d₂ d₂≉0 coprime-n₂d₂) =
                  fr′ s' ddg' ddg'≉0 coprime-s'-ddg'
   where
   -- (FSum)
   g   = gcd d₁ d₂;
   d₁' = d₁ /' g;                    d₂'  = d₂ /' g
   s   = n₁ * d₂' + n₂ * d₁';        g₁   = gcd s g
   s'  = s /' g₁;                    g'   = g /' g₁
   dd  = d₁' * d₂';                  ddg' = dd * g'
```

The three main points to prove:

$d_1'd_2'g' \not\approx 0$,

$s' / (d_1'd_2'g') =fr (n_1d_2 + n_2d_1)/(d_1d_2)$ (Corr)

Coprime s' $(d_1'd_2'g')$.

Proof for $d_1'd_2'g' \not\approx 0$:

$d_1 \not\approx 0$, $d_1' g \approx d_1$. Hence $d_1' \not\approx 0$, $g \not\approx 0$ ...

IntegralRing R, $d_1' \not\approx 0$, $d_2' \not\approx 0$, $g' \not\approx 0$.

Hence $(d_1' d_2') g' \not\approx 0$.

Proof for (Corr)

(Corr) means that +fr is the sum of fractions:

$$(s/'g_1)\ d_1d_2 \approx (n_1d_2 + n_2d_1)\ ((d_1/'g)\ (d_2/'g)\ g'),$$

that is it returns a fraction equal to the one returned by +'.

Goal:

$$(s/'g_1)\ d_1d_2 \approx (n_1d_2' + n_2d_1')\ g\ (d_1/'g)\ (d_2/'g)\ g'$$

The right hand side is

$$\begin{aligned}
&s\ g\ (d_1/'g)\ (d_2/'g)\ g' &&\approx \\
&s\ d_1\ (d_2/'g)\ g' &&\approx \\
&s\ d_1\ (d_2/'(g'g_1))\ g' &&\approx \\
&s\ d_1d_2/'g_1 &&\approx \\
&(s/'g_1)\ d_1d_2
\end{aligned}$$

Proof for coprimality

```
-- (FSum)
g   = gcd d1 d2,
d1' = d1 /' g,                      d2'  = d2 /' g,
s   = n1 * d2' + n2 * d1',          g1   = gcd s g,
s'  = s /' g1,                      g'   = g /' g1,
dd  = d1' * d2',                    ddg' = dd * g'

d1 ≉ 0,  d2 ≉ 0,  Coprime n1 d1,  Coprime n2 d2.

Goal:    Coprime  s'  ((d1' * d2') * g')
```

Its proof needs the structure of an Unique Factorization Ring for R.

Describe some details in this plan

```
CoprimeByPrimes :  Rel C _
CoprimeByPrimes a b =  ∀ p → IsPrime p → p | a → p ∤ b
```

From `Prime|split` it is easily derived the multiplicative property:

```
coprimeByPrimesWithProduct :
 Prime|split → (∀ {a b c} → CoprimeByPrimes a b →
                            CoprimeByPrimes a c →
                            CoprimeByPrimes a (b * c))
```

`CoprimeByPrimes s' d₁'` is proved by the function

`coprimeP-s'd₁'`.

It takes

$$\texttt{p : C}, \quad \texttt{prime-p : IsPrime p}, \quad \texttt{p|s' : p} \mid \texttt{s'}$$

and returns $\texttt{p}\nmid\texttt{d}_1\texttt{'} : \texttt{p} \nmid \texttt{d}_1\texttt{'}$,

The latter negation is expressed as a function that maps any value $\texttt{p|d}_1\texttt{'} : \texttt{p} \mid \texttt{d}_1\texttt{'}$ to the empty type $\bot$.

So: there are given the values $\texttt{p}, \texttt{prime-p}, \texttt{p|s'}, \texttt{p|d}_1\texttt{'}$,

and the goal is to build $\bot$ out of this.

This is done by using the equality

$$\texttt{n}_1 \texttt{ * d}_2\texttt{'} \approx \texttt{s - (n}_2 \texttt{ * d}_1\texttt{')} \qquad (1)$$

As p divides s', it also divides $\texttt{s = g}_1 \texttt{ * s'}$.

It is given $\texttt{p}|\texttt{d}_1\texttt{'}$. Hence, by (1), $\texttt{p} \mid \texttt{(n}_1 \texttt{ * d}_2\texttt{')}$.

By Prime|split, it holds $\texttt{p} \mid \texttt{n}_1$ or $\texttt{p} \mid \texttt{d}_2\texttt{'}$.

In the first case, it hold $\texttt{p} \mid \texttt{n}_1$ and $\texttt{p} \mid \texttt{d}_1\texttt{'}$.

Then p divides $\texttt{d}_1 \texttt{ = g * d}_1\texttt{'}$.

By $\texttt{(Coprime n}_1 \texttt{ d}_1\texttt{)}$, it is derived that p is invertible.

This contradicts to primality of p, and produces the value $\bot$.

In a similar style, it is proved all the rest for Goal.

Referring to an expensive algorithm is not expensive

We need to avoid factoring when performing fraction arithmetic.

The program for optimized fraction sum uses that

any common non-invertible factor for `a` and `b`

contains a prime which divides both `a` and `b`.

And its proof refers to the `factor` algorithm.

But the reference to '`factor`' is only in the proof part of the client function.

The program is so that factoring is not performed in the executable code for this client function.

Proof for the *field* laws

There are needed proofs for `Field (Fraction R)`:

- congruence, associativity, commutativity for `*fr` and `+fr`,
- distributivity, the division property, ...

The proof scheme is as follows.

- First these properties are proved for `(Prefraction, +', *')`.
- Then they are proved for `(Fraction, +fr, *fr)` by using the above equality proofs for `+fr`$\overset{\circ}{=}$`+'`, `*fr`$\overset{\circ}{=}$`*'`.

For example, consider the proof for associativity of +':

$$(n_1/d_1 +' n_2/d_2) +' n_3/d_3 \;=' \; n_1/d_1 +' (n_2/d_2 +' n_3/d_3)$$

<==>

$$(n_1d_2 + n_2d_1)/(d_1d_2) +' n_3/d_3 \;=' \; n_1/d_1 +' (n_2d_3 + n_3d_2)/(d_2d_3)$$

<==>

$$((n_1d_2 + n_2d_1)*d_3 + n_3d_1d_2)/(d_1d_2d_3) \;=' \; (n_1d_2d_3 + (n_2d_3 + n_3d_2)*d_1)/(d_1d_2d_3)$$

<==>

$$((n_1d_2 + n_2d_1)*d_3 + n_2(d_1d_2))*(d_1(d_2d_3)) \;\approx\; (n_1d_2d_3 + (n_2d_3 + n_3d_2)*d_1)*((d_1d_2)d_3)$$

...

Simple examples of a formal proof

```
='trans :  Transitive _='_
='trans {preFr n1 d1 _} {preFr n2 d2 d2≉0}
                        {preFr n3 d3 _}
                        n1*d2≈n2*d1 n2*d3≈n3*d2 =  goal
  where
  e0 :  d2 * (n1 * d3) ≈ d2 * (n3 * d1)
  e0 =
   begin
     d2 * (n1 * d3)    ≈[ ≈sym $ *assoc d2 n1 d3 ]
     (d2 * n1) * d3    ≈[ *cong₁ $ *comm d2 n1 ]
     (n1 * d2) * d3    ≈[ *cong₁ n1*d2≈n2*d1 ]
     (n2 * d1) * d3    ≈[ *cong₁ $ *comm n2 d1 ]
     (d1 * n2) * d3    ≈[ *assoc d1 n2 d3 ]
```

```
    d1 * (n2 * d3)     ≈[ *cong₂ n2*d3≈n3*d2 ]
    d1 * (n3 * d2)     ≈[ *comm d1 _ ]
    (n3 * d2) * d1     ≈[ *cong₁ $ *comm n3 d2 ]
    (d2 * n3) * d1     ≈[ *assoc d2 n3 d1 ]
    d2 * (n3 * d1)
```

□

```
goal : n1 * d3 ≈ n3 * d1
goal =
    cancelNonzeroLFactor d2 (n1 * d3) (n3 * d1) d2≉0 e0
```

Proof for double reverse of a list

```
module _ {a} (A : Set a)
 where
 _++_ : List A → List A → List A      -- concatenation
 []       ++ ys =  ys
 (x :: xs) ++ ys =  x :: (xs ++ ys)

 rev : List A → List A                -- reversing a list
 rev []       = []
 rev (x :: xs) = (rev xs) ++ [ x ]
```

```
-- lemma
rev-append :  ∀ x ys → rev (ys ++ [ x ]) ≡ x :: (rev ys)
rev-append x []       =  PE.refl
rev-append x (y :: ys) =
  ≡begin
    rev ((y :: ys) ++ [ x ])       ≡[ PE.refl ]
    rev (y :: (ys ++ [ x ]))       ≡[ PE.refl ]
    (rev (ys ++ [ x ])) ++ [ y ]
              ≡[ PE.cong (_++ [ y ]) (rev-append x ys) ]
    (x :: (rev ys)) ++ [ y ]       ≡[ PE.refl ]
    x :: ((rev ys) ++ [ y ])       ≡[ PE.refl ]
    x :: (rev (y :: ys))
  ≡end
```

```
revrev : ∀ xs → rev (rev xs) ≡ xs      -- goal theorem

revrev []       =  PE.refl

revrev (x :: xs) =

  ≡begin

   rev (rev (x :: xs))        ≡[ PE.refl ]

   rev ((rev xs) ++ [ x ])  ≡[ rev-append x (rev xs) ]

   x :: (rev (rev xs))     ≡[ PE.cong (x ::_) (revrev xs) ]

   x :: xs

  ≡end
```


S. Lang. *Algebra*. Addison-Wesley Publishing Company, 1965.


A. A. Markov. *On constructive mathematics*,
In Problems of the constructive direction in mathematics. Part 2. Constructive mathematical analysis, Collection of articles, Trudy Mat. Inst. Steklov., 67, Acad. Sci. USSR, Moscow–Leningrad, 1962, 8–14.


Per Martin-Löef. *Intuitionistic Type Theory*.
Bibliopolis. ISBN 88-7088-105-9, 1984.


S. D. Meshveliani.
On dependent types and intuitionism in programming mathematics,
(In Russian). In electronic journal Program systems: theory and applications, 2014, Vol. 5, No 3(21), pp. 27–50,
`http://psta.psiras.ru/read/psta2014_3_27-50.pdf`

 S. D. Meshveliani.

Programming basic computer algebra in a language with dependent types,

In electronic journal Program systems: theory and applications, 2015, 6:4(27), pp. 313–340. (In Russian), `http://psta.psiras.ru/read/psta2015_4_313-340.pdf`

 S. D. Meshveliani.

Programming computer algebra with basing on constructive mathematics. Domains with factorization. In RUSSIAN. In electronic journal Program systems: theory and applications, 2017, Vol 8, No 1, 2017, `http://psta.psiras.ru/read/psta2017_1_3-46.pdf`


S. D. Meshveliani. *DoCon-A — a provable algebraic domain constructor*, the source program and manual, 2016, Pereslavl-Zalessky.
`http://www.botik.ru/pub/local/Mechveliani/docon-A/`


U. Norell, J. Chapman.
Dependently Typed Programming in Agda,
`http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`