A single number type for Math education in Type Theory

Yves Bertot
Thomas Portet

April 2026

The context

- Point of view: young students in mathematics have a single type of numbers in mind
- Type theory proposes a variety of type numbers
  - Inductive types come with a builtin capability for computation
  - Inductive types come with a direct notion of reasoning by induction
  - Real numbers live a different part of the world
- Experimenting with using only one type: real numbers
  - Use subsets for natural numbers
  - Recover recursive computation
  - Make `ring` and `field` more comfortable

Plan

- The subset of R that contains natural numbers
- Recursive computation with real numbers
- A poor man's calc tactic
- Deep ring and field tactics
- Showcase examples
  - Trigonometry up to Viète's formula
  - Fibonacci upto a closed formula
  - Binomials as fractions of recursively defined factorials

Subsets instead of types

- Avoid using the type `nat`, but use the concept
- Natural numbers are definable as a subset, inductively
- Integers can also be defined inductively, but it is minor

Inductive predicate in Rocq

```
Require Import Reals.
Open Scope R_scope.

Inductive Rnat : R -> Prop :=
  Rnat0 : Rnat 0
| Rnat_succ : forall n, Rnat n -> Rnat (n + 1).
```

Generated induction principle:

```
nat_ind
     : forall P : R -> Prop,
       P 0 ->
       (forall n : R, Rnat n -> P n -> P (n + 1)) ->
       forall r : R, Rnat r -> P r
```

Ad hoc proofs of membership

- When $n, m \in \mathbb{N}$, $(n + m), (n \times m) \in \mathbb{N}$
- Recover information normally given by typing
- When $n, m \in \mathbb{N}$ and $m \leq n$, $(n - m) \in \mathbb{N}$ can also be proved
  - This requires an explicit proof
  - Probably good in a training context for students
  - Similar for division

Computation

- Notations are so that 12 (as a real number) is actually `IZR 13`
- `IZR : Z -> R` is a recursive function
- The interaction with `Compute` is not pleasant

```
Compute 12.
 = (R1 + R1) * ((R1 + R1) * (R1 + (R1 + R1)))
     : R
```

- We designed a new command called `R_compute`.

```
R_compute (12 + 13).
 = 25
```

- Does not cover λ-calculus or let-binders
- Works by using a database of registered functions
- Can produce a lemma stating the result as an equality

Recursive definitions, following the `nat` recursive scheme

- We provide a command that mimicks recursive definitions on `Rnat`
- Only "well-defined" for elements of `Rnat`
- The command produces two objects
  - The function of type `R -> R`
  - The proof of the logical statement for that function

```
Recursive (def fib such that
             fib 0 = 0 /\
             fib 1 = 1 /\
             forall n : R, Rnat (n - 2) ->
                fib n = fib (n - 2) + fib (n - 1)).
```

Note: no pattern matching, but condition `Rnat (n - 2)`

Recursive definitions and R_compute

- When values produced by the function can be inferred to be in $\mathbb{Z}$
- The function can be added to the database for R_compute
- This requires defining a similar function of type Z -> Z

```
Elpi mirror_recursive_definition fib.

R_compute (fib 42).
 = 267914296
```

- The answer is instantaneous, because compilation changes the computation sutructure

Recursive definitions of functions with several arguments

- Some functions take more than one argument such as binomial coefficients :

```
Recursive (def bin such that
  bin 0 = (fun m : R => at_x 0 1 0 m) /\
  forall n, Rnat (n - 1) ->
  bin n = fun m : R => at_x 0 1
      (bin (n - 1) (m - 1) + bin (n - 1) m) m).
```

Recursive definitions of functions with several arguments

- To handle functions with an arbitrary number of arguments, we introduce a *new family of types*: `ty_ T n`.

```
Fixpoint ty_ (T : Type) (n : nat) : Type :=
    match n with
    0 => T
    |S p => (T -> ty_ T p)
     end.
Fixpoint cst_ (T : Type) (k : T) (n : nat) : (ty_ T n):
    match n with
    0 => k
    |S p => (fun _ => (cst_ T k p))
    end.
```

A poor man's `calc` tactic

- Lean has popularized a style of proofs called calculational
- Clever use of transitivity of equality and orders
- For our experiments, we only cover equality
- `start_with` *formula*
  - Checks that the goal has the form $F = G$
  - Checks that the *formula* and F are convertible
  - Changes the goal into *formula* $= G$
- `calc_LHS` *formula*
  - Checks the goal has the form $F = G$
  - Creates two new goals $F =$ *formula* and *formula* $= F$
- Should be read as ... $=$ *formula*
  - The intent is to make proofs declarative (as in *waterproof*)
  - Each step after one instance of `calc_LHS` should be solved in an easy step
- `end_calculate` : checks that the intended equality is "obvious"

Supercharging `ring` and `field`

- Tactics `ring` and `field` only work from the root
- Any application of an *unknown* function is treated as an atom
- Arguments are not modified by `ring` or `field`
- Example `x + sqrt(x + 1) = sqrt (1 + x) + x` is not covered
- ring treats `sqrt (x + 1)` and `sqrt (1 + x)` as two different atoms

A strategy that we propose

- Use `ring_simplify` or `field_simplify`
  - Normalize formulas given as argument, replace them
  - Can handle several formulas at a time (with the same order)
- normalize expressions under unknown functions
- repeat until stability
- On example: normalize `(x + 1)`, `1 + x`, `x + sqrt (1 + x)` and `sqrt (1 + x) + x`
- This may yield `x + sqrt (x + 1) = sqrt (x + 1) + x`
- Next normalization : `x + sqrt (x + 1) = x + sqrt (x + 1)`
- We call the new tactics `deep_ring` and `deep_field`
- We also extend `calc_LHS` to include an attempt to solve with the deep tactics

Example of proof made possible

```
Lemma cos_sub_Pi_half x : cos (x - Pi / 2) = sin x.
Proof.
start_with (cos (x - Pi / 2)).
calc_LHS (cos x * cos (Pi / 2) + sin x * sin (Pi / 2)).
  now rewrite cos_sub.
calc_LHS (cos x * 0 + sin x * sin (Pi / 2)).
  now rewrite cos_Pi_half.
calc_LHS (cos x * 0 + sin x * 1).
  now rewrite sin_Pi_half.
end_calculate.
Qed.
```

Formule de Viète pour le calcul de π

$$\pi = 2 \cdot \frac{2}{\sqrt{2}} \cdots \frac{2}{\sqrt{2+\cdots\sqrt{2}}} \cdots$$

- Use of recursive definition to define $\sqrt{2+\sqrt{\cdots+\sqrt{2}}}$
- Use of iterated products to define the main formula
- **Key point 1**: $\sin \frac{\pi}{2^{n+1}} = \frac{\sin \frac{\pi}{2^n}}{2 \cos \frac{\pi}{2^{n+1}}}$ et $\sin \frac{\pi}{2} = 1$
- **Key point 2**: $\cos \frac{\pi}{2^n} = \frac{\sqrt{2+\sqrt{\ldots+\sqrt{2}}}}{2}$
- **Key point 3**: $\lim_{n->\infty} \frac{\frac{\pi}{2^n}}{\sin \frac{\pi}{2^n}} = 1$

Context for the formal proof of Viète's formula

- `intro_trigo.v`: following notes of J. Puydt
- `trinom` tactics: solutions to 2nd degree equations
- Limits are axiomatized in the same spirit as the trigonometry
- Limits of sequences are also about function from `R` to `R`, restricted to `Rnat`