The conatural numbers form an exponential commutative semiring

Szumi Xie Viktor Bense

Eötvös Loránd University (ELTE)

TyDe 2025, Singapore

Natural numbers

```
data Nat : Type where
  zero :         Nat
  suc  : Nat → Nat
```

Natural numbers

```
data Maybe (A : Type) : Type where
  nothing :     Maybe A
  just    : A → Maybe A

data Nat : Type where
  zero-or-suc : Maybe Nat → Nat
```

Conatural numbers

```
data Maybe (A : Type) : Type where
  nothing :     Maybe A
  just    : A → Maybe A

record Conat : Type where
  coinductive
  field pred : Maybe Conat
```

Conatural numbers

```
data Maybe (A : Type) : Type where
  nothing :     Maybe A
  just    : A → Maybe A

record Conat : Type where
  coinductive
  field pred : Maybe Conat

pred : Conat → Maybe Conat
```

Examples of conatural numbers

$$\mathsf{zero} : \mathsf{Conat}$$
$$\mathsf{pred}\ \mathsf{zero} = \mathsf{nothing}$$

$$\mathsf{suc} : \mathsf{Conat} \to \mathsf{Conat}$$
$$\mathsf{pred}\ (\mathsf{suc}\ x) = \mathsf{just}\ x$$

$$\infty : \mathsf{Conat}$$
$$\mathsf{pred}\ \infty = \mathsf{just}\ \infty$$

Exponential commutative semirings

$$(x + y) + z = x + (y + z)$$
$$x + y = y + x$$
$$0 + x = x$$

$$(xy)z = x(yz)$$
$$xy = yx$$
$$1x = x$$
$$(x + y)z = xz + yz$$
$$0x = 0$$

$$x^{yz} = (x^y)^z$$
$$x^1 = x$$
$$x^{y+z} = x^y x^z$$
$$x^0 = 1$$
$$(xy)^z = x^z y^z$$
$$1^x = 1$$

Rest of the talk

Naïve attempt

Defining good internal states

Using embedded languages

Quotienting the language

Conclusion

Naïve attempt

Addition

```
_+_ : Conat → Conat → Conat
pred (x + y) with pred x
... | nothing = pred y
... | just x-1  = just (x-1 + y)
```

Multiplication (naïve)

```
_×_ : Conat → Conat → Conat
pred (x × y) with pred x | pred y
... | nothing | _       = nothing
... | just x-1 | nothing = nothing
... | just x-1 | just y-1 = just (y-1 + x-1 × y)
```

Multiplication (naïve)

```
_×_ : Conat → Conat → Conat
pred (x × y) with pred x | pred y
... | nothing | _         = nothing
... | just x-1 | nothing = nothing
... | just x-1 | just y-1 = just (y-1 + x-1 × y)
```

"Termination checking failed"

Commutativity of addition (naïve)

```
+-comm : ∀ x y → x + y ≡ y + x
pred (+-comm x y i) with pred x | pred y
... | just x-1 | just y-1 =
    ...
    x-1 + suc y-1   ≡⟨ ... ⟩
    suc (x-1 + y-1) ≡⟨ cong suc (+-comm x-1 y-1) ⟩
    suc (y-1 + x-1) ≡⟨ ... ⟩
    y-1 + suc x-1
```

“Termination checking failed”

Defining good internal states

The corecursor

```
corec : (S → Maybe S) → S → Conat
pred (corec f s) with f s
... | nothing = nothing
... | just s'    = just (corec f s')
```

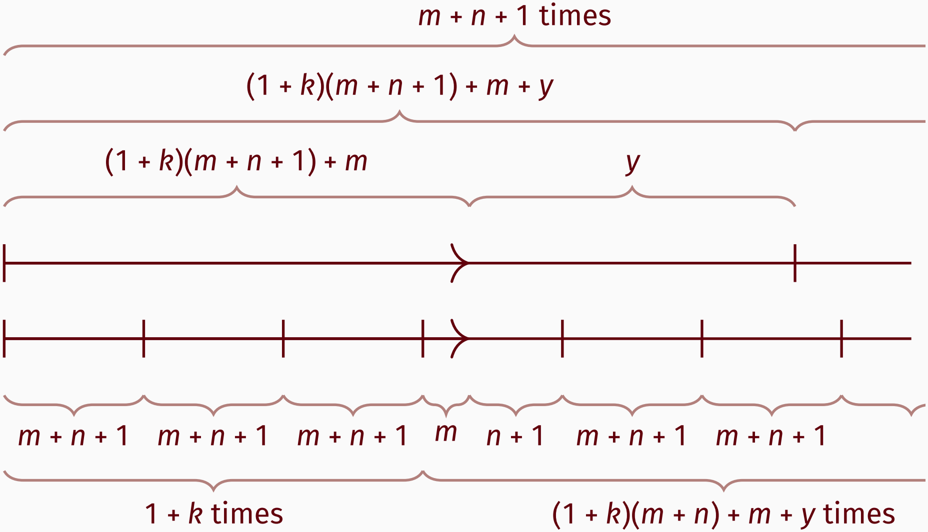
Multiplication using the corecursor

$S := \text{Conat} \times \text{Conat}$

e.g. 4×3:

$$\begin{aligned} &(3,2) \to (3,1) \to (3,0) \to (2,2) \to (2,1) \to (2,0) \to \\ &\to (1,2) \to (1,1) \to (1,0) \to (0,2) \to (0,1) \to (0,0) \end{aligned}$$

Commutativity of multiplication using bisimulation



Using embedded languages

A language with addition

Nils Anders Danielsson (2010), “Beating the Productivity Checker Using Embedded Languages”

```
data Expr : Type where
  embed : NExpr → Expr
  _+_   : Expr → Expr → Expr

record NExpr : Type where
  coinductive
  field pred : Maybe Expr
```

This is a mixed inductive/coinductive type

Multiplication in the language

```
fromConat : Conat → NExpr
pred (fromConat x) with pred x
... | nothing = nothing
... | just x-1 = just (embed (fromConat x-1))

_×'_ : Conat → Conat → NExpr
pred (x ×' y) with pred x | pred y
... | nothing | _          = nothing
... | just x-1  | nothing = nothing
... | just x-1  | just y-1  = just (embed (fromConat y-1) + embed (x-1 ×' y))
```

Interpretation of the language

```
predE : Expr → Maybe Expr
predE (embed x) = pred x
predE (x + y) with predE x
... | nothing = predE y
... | just x-1 = just (x-1 + y)

interp : Expr → Conat
pred (interp x) with predE x
... | nothing = nothing
... | just x-1 = just (interp x-1)

_×_ : Conat → Conat → Conat
x × y = interp (embed (x ×' y))
```

Trace of multiplication using the language

4×3:

$$4 \times' 3 \to 2 + 3 \times' 3 \to 1 + 3 \times' 3 \to 0 + 3 \times' 3 \to 2 + 2 \times' 3 \to \ldots$$

A language for equations

```
data _≈_ : Conat → Conat → Type where
  embed  : x ≈ℕ y → x ≈ y
  refl   : x ≈ x
  sym    : x ≈ y → y ≈ x
  trans  : x ≈ y → y ≈ z → x ≈ z
  cong-+ : x ≈ y → z ≈ y → x + z ≈ y + w

  ...
```

Quotienting the language

A quotiented language

```
record NExpr where
  coinductive
  field pred : Maybe Expr

data Expr where
  _+_        : Expr → Expr → Expr
  _×_        : Expr → Expr → Expr
  zero       : Expr
  +-assoc    : ∀ x y z → (x + y) + z ≡ x + (y + z)
  +-comm     : ∀ x y → x + y ≡ y + x
  ...
  embed      : NExpr → Expr
  embed-zero : ∀ x     → pred x ≡ nothing → embed x ≡ zero
  embed-suc  : ∀ x x-1 → pred x ≡ just x-1  → embed x ≡ one + x-1
```

This is a mixed higher-inductive/coinductive type

Predecessor function for the quotiented language

$\mathrm{pred}_E : \mathrm{Expr} \to \mathrm{Maybe\ Expr}$

$\mathrm{pred}_E\ (x + y)$ with $\mathrm{pred}_E\ x$
$\ldots \mid \mathrm{nothing} = \mathrm{pred}_E\ y$
$\ldots \mid \mathrm{just}\ x\text{-}1 = \mathrm{just}\ (x\text{-}1 + y)$

$\mathrm{pred}_E\ (x \times y)$ with $\mathrm{pred}_E\ x \mid \mathrm{pred}_E\ y$
$\ldots \mid \mathrm{nothing} \mid \_ = \mathrm{nothing}$
$\ldots \mid \mathrm{just}\ x\text{-}1 \mid \mathrm{nothing} = \mathrm{nothing}$
$\ldots \mid \mathrm{just}\ x\text{-}1 \mid \mathrm{just}\ y\text{-}1 = \mathrm{just}\ (y\text{-}1 + x\text{-}1 \times y)$

…

Interpretation of the quotiented language

```
interp : Expr → Conat
pred (interp x) with pred_E x
... | nothing = nothing
... | just x-1 = just (interp x-1)

embedConat  : Conat → Expr
interp-embed : ∀ x → interp (embedConat x) ≡ x
embed-interp : ∀ x → embedConat (interp x) ≡ x

Expr≅Conat : Expr ≅ Conat
```

Deriving the operations and equations for conatural numbers

```
record ECSemiring (A : Type) : Type where
  field
    _+_     : A → A → A
    _×_     : A → A → A
    zero    : A
    +-assoc : ∀ x y z → (x + y) + z ≡ x + (y + z)
    +-comm  : ∀ x y → x + y ≡ y + x
     ...

ExprECSemiring : ECSemiring Expr

ConatECSemiring : ECSemiring Conat
ConatECSemiring =
  subst ECSemiring (univalence Expr≅Conat) ExprECSemiring
```

Conclusion

Conclusion

We formalized that conatural numbers form an exponential commutative semiring in Cubical Agda

We tried different approaches:

- Using the corecursor and bisimulation: we proved that Conats form a commutative semiring, but it doesn't scale to the equations involving exponentiation
- Using a quotiented embedded language: we proved that Conats form an exponential commutative semiring, but it's not modular.

Unresolved questions

Is there a way to prove this that is both simple and modular?

How to define more complicated functions, such as tetration?

What's the semantic justification of mixed higher-inductive/coinductive types?