The Conatural Numbers Form an Exponential Commutative Semiring

Szumi Xie
Eötvös Loránd University
Budapest, Hungary
szumi@inf.elte.hu

Viktor Bense
Eötvös Loránd University
Budapest, Hungary
bense.viktor@inf.elte.hu

Abstract

Conatural numbers are a coinductive type dual to the inductively defined natural numbers. The conatural numbers can represent all natural numbers and an extra element for infinity, this can be useful for representing the amount of steps taken by a possibly non-terminating program. We can define functions on conatural numbers by corecursion, however proof assistants such as Agda require the corecursive definitions to be guarded to make sure that they are productive. This requirement is often too restrictive, as it disallows the corecursive occurrence to appear under previously defined operations. In this paper, we explore some methods to solving this issue using the running examples of multiplication and the commutativity of addition on conatural numbers, then we give comparisons between these methods. As the main result, this is the first proof that conatural numbers form an exponential commutative semiring in cubical type theory without major extensions.

CCS Concepts: • **Theory of computation** → **Type theory**; **Logic and verification**; **Constructive mathematics**; **Program verification**; • **Software and its engineering** → Recursion; Data types and structures.

Keywords: conatural number, corecursion, coinduction, productivity, guardedness, termination, semiring, cubical type theory, univalence

ACM Reference Format:
Szumi Xie and Viktor Bense. 2025. The Conatural Numbers Form an Exponential Commutative Semiring. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3759538.3759654

1 Introduction

In dependent type theory, natural numbers are represented as an inductive type with two constructors, one for zero and the other for the successor of a natural number. In the Agda proof assistant, we write it as follows:

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

Categorically, natural numbers are the initial object in the category of algebras over the $\mathbb{1} + -$ (or the "Maybe") endofunctor. So an equivalent representation of natural numbers is an inductive type with a single constructor containing a Maybe of a natural number.

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

data ℕ : Type where
  con : Maybe ℕ → ℕ
```

We can dualise natural numbers to get *conatural numbers*, which are the terminal object in the category of coalgebras over the $\mathbb{1} + -$ endofunctor [13]. In Agda, it is a coinductive record type with one destructor into Maybe of conatural numbers.

```
record ℕ∞ : Type where
  coinductive
  field
    pred : Maybe ℕ∞
```

This destructor is the predecessor function which either fails or returns another conatural number.

We can define elements of ℕ∞ by copattern matching [3], that is, we specify what the predecessor is for a particular element. As examples, we give the definitions of zero, successor, and infinity. Zero does not have a predecessor:

```
zero : ℕ∞
pred zero = nothing
```

The predecessor of a successor of a number is just that number:

```
suc : ℕ∞ → ℕ∞
pred (suc x) = just x
```

The predecessor of infinity is just infinity:

```
∞ : ℕ∞
pred ∞ = just ∞
```

TyDe '25, Singapore, Singapore

ACM ISBN 979-8-4007-2163-2/25/10
https://doi.org/10.1145/3759538.3759654

The above definition for ∞ is not structurally recursive, but it is *guarded*, that is, the recursive occurrence is after an instance of copattern matching (pred), under only constructors (just) [8]. Agda uses guardedness to check whether a corecursive defintion is productive. A definition is *productive* when one can compute the application of any finite amount of destructors on a corecursive value in a finite amount of steps. Guardedness is sufficient to show productivity but it is not necessary, thus Agda is too conservative about which corecursive definitions it allows.

The conatural numbers can represent all natural numbers and an extra element for infinity, however, it is not constructively isomorphic to $\mathbb{N} + \mathbb{1}$, because all functions out of the conatural numbers must be continuous [10]. Computationally, finite amount of output can only depend on finite amount of input. Topologically, ∞ is the limit of $0, 1, 2, \ldots$, which must be preserved. We can visualise the topological space as follows:

As a result of this restriction, we cannot define a function that decides if an element is equal to ∞.

Our contribution in this paper is to prove that the conatural numbers form an exponential commutative semiring by guarded corecursion, along the way showing methods to keep the corecursion guarded. An *exponential commutative semiring* is a commutative semiring with a binary operation for exponentiation satisfying the following equations:

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

In Section 2, we show how to define addition on conatural numbers and prove that it is associative, at the same time, we show that if we naïvely define multiplication and prove that addition is commutative, they get rejected by Agda because they are not guarded. In the following sections, we show three ways to avoid this issue, using multiplication and commutativity of addition as running examples.

In section 3, we directly use corecursion to define multiplication and prove the commutativity of addition, avoiding reusing any previous definitions to keep guardedness.

In section 4, we use Nils Anders Danielsson's method [9] to use an embedded language to define multiplication, and another language to the prove commutativity of addition.

In section 5, we adapt the previous method to Cubical Agda, making use of mixed higher-inductive/coinductive types and the univalence principle to define and prove all the operations and equations of an exponential commutative semiring at once.

1.1 Formalisation

In vanilla Agda, it is not possible to prove non-trivial equations about conatural numbers using the Martin-Löf identity type [15]. One needs to either postulate the coinduction principle, or instead use a coinductively defined equivalence relation, in which case one would need to manually prove that operations preserve this relation. As such, in this paper we use Cubical Agda, where the coinduction principle and other equations can be directly proved thanks to the interaction between copattern matching and the interval [21]. As an example, the proof below that the predecessor function is injective cannot be done in vanilla Agda:

```
pred-inj : ∀ {x y} → pred x ≡ pred y → x ≡ y
pred (pred-inj p i) = p i
```

With the method in Section 3, we have formalised that the conatural numbers form a commutative semiring, and defined the exponentiation operation without proving the equations. With the method in Section 4 we have proved the properties of addition. With the last method, in Section 5, we have a full formalisation that the conatural numbers form an exponential commutative semiring.

For the details of our proofs, see our formalisation, which is available at https://github.com/szumixie/conat. It depends on the Cubical Agda Library [6].

1.2 Related Work

There are some extensions to type theory that can make corecursive definitions easier to define.

Using sized types, one can attach ordinals to constructors, which allows corecursion to be done by well-founded recursion on the sizes [14, 2]. This bypasses the guardedness restriction, however, the current implementation of sized types in Agda is inconsistent [1].

Guarded recursion adds a modality to type theory and a fixed point operation which uses the modality to avoid allowing non-terminating programs [17, 7]. There is an experimental extension of Agda that implements a version of this [20].

There is an equivalent non-coinductive representation of conatural numbers, which are monotonically decreasing (or increasing) Boolean sequences [10].

$$(f : \mathbb{N} \to 2) \times ((m\, n : \mathbb{N}) \to n \leq m \to f(m) \leq f(n))$$

Naïm Camille Favier proved that conatural numbers form a semiring with meet using this representation, avoiding corecursion [11].

2 Naïve Corecursion

In this section, we attempt to define operations and prove equations in the most straightforward way with corecursion, and show that some of these fail to be guarded.

We can define addition of conatural numbers by guarded corecursion, using a helper function +-match to pattern match on the predecessor of x. Note that we use a helper function instead of Agda's with-abstraction [18], because with-abstractions make later proofs about the function more

complicated, as there is no way to refer to the function generated by the with-abstraction.

```
_+_ : ℕ∞ → ℕ∞ → ℕ∞
pred (x + y) = +-match (pred x) y

+-match : Maybe ℕ∞ → ℕ∞ → Maybe ℕ∞
+-match nothing y = pred y
+-match (just x') y = just (x' + y)
```

In the first case, we say that the predecessor of $0 + y$ is the predecessor of y. In the second case, which is when x is non-zero, we say that the predecessor of $x + y$ is $x' + y$ where x' is the predecessor of x.

If we naïvely try to define multiplication by using addition, then Agda will reject it, unless we use the unsafe TERMINATING pragma.

```
{-# TERMINATING #-}
_×_ : ℕ∞ → ℕ∞ → ℕ∞
pred (x × y) = ×-match (pred x) y (pred y)

×-match :
  Maybe ℕ∞ → ℕ∞ → Maybe ℕ∞ → Maybe ℕ∞
×-match nothing y y'       = nothing
×-match (just x') y nothing = nothing
×-match (just x') y (just y') = just (y' + x' × y)
```

Here we match on both x and y. If neither are zero, then the predecessor of $x \times y$ is $y' + x' \times y$, where $x - 1 = x'$ and $y - 1 = y'$. This definition is rejected by Agda, because it is not guarded (even though it is productive), as the recursive call to _×_ is under _+_, which is not a constructor.

We can also prove some of the usual properties of addition over the conatural numbers, like the associativity of the addition.

```
+-assoc : ∀ x y z → (x + y) + z ≡ x + (y + z)
pred (+-assoc x y z i) = +-assoc-match (pred x) y z i

+-assoc-match :
  ∀ x' y z →
  +-match (+-match x' y) z ≡ +-match x' (y + z)
+-assoc-match nothing y z = refl
+-assoc-match (just x') y z = cong just (+-assoc x' y z)
```

This is again a nice instance of guarded corecusion and Agda happily accepts this as a proof. However, the proof that addition is commutative is problematic. Let us assume that we have the following equation to commute suc (whose definition is problematic for the same reason):

```
+-suc : ∀ x y → x + suc y ≡ suc (x + y)
```

Then we can use it to attempt to prove the commutativity of addition. We make +-comm-match take equations to remember that the arguments we match on are equal to the original values. We only show the case where neither x nor y are zero:

```
{-# TERMINATING #-}
+-comm : ∀ x y → x + y ≡ y + x
pred (+-comm x y i) =
  +-comm-match x (pred x) y (pred y) refl refl i

+-comm-match :
  ∀ x x' y y' → pred x ≡ x' → pred y ≡ y' →
  +-match x' y ≡ +-match y' x
+-comm-match x (just x') y (just y') px py =
  cong just
    ( x' + y        ≡⟨ cong (x' +_) (pred-inj py) ⟩
      x' + suc y'   ≡⟨ +-suc x' y' ⟩
      suc (x' + y') ≡⟨ cong suc (+-comm x' y') ⟩
      suc (y' + x') ≡⟨ sym (+-suc y' x') ⟩
      y' + suc x'   ≡⟨ cong (y' +_) (pred-inj (sym px)) ⟩
      y' + x        ∎)
```

The same problem arises here as in the definition of multiplication above. The definition here is productive but not guarded, because +-comm is used in the equational reasoning chain. The chain is composed of the transitivity operation for equations. However, the transitivity operation is not a constructor.

3 Direct Corecusion

In this section, we show some examples of how to do corecursion without running into the guardedness issue by coming up with specific internal states for each corecursion, which allows us to define the functions in a guarded form.

To avoid the guardedness issue in Section 2, we define multiplication from scratch instead of reusing addition. First, we match to check whether either of the arguments is zero.

```
_×_ : ℕ∞ → ℕ∞ → ℕ∞
pred (x × y) = ×-match (pred x) (pred y)

×-match : Maybe ℕ∞ → Maybe ℕ∞ → Maybe ℕ∞
×-match nothing y'       = nothing
×-match (just x') nothing = nothing
×-match (just x') (just y') = just (x' ×' y')
```

For the non-zero case, we define a separate operation _×'_ such that $x \times' y = (x + 1) \times (y + 1) - 1$. The idea for the function is to count $y + 1$ steps $x + 1$ times. This means that after each $y + 1$ steps, we have to reset the counter. To achieve this, we define a helper function which keeps track of the original y which we call y_0.

```
_×'_ : ℕ∞ → ℕ∞ → ℕ∞
x ×' y = ×'-helper x y y

×'-helper : ℕ∞ → ℕ∞ → ℕ∞ → ℕ∞
pred (×'-helper x y y₀) =
  ×'-helper-match x (pred x) (pred y) y₀

×'-helper-match :
```

```
  ℕ∞ → Maybe ℕ∞ → Maybe ℕ∞ → ℕ∞ →
  Maybe ℕ∞
×'-helper-match x x' (just y') y₀ =
  just (×'-helper x y' y₀)
×'-helper-match x (just x') nothing y₀ =
  just (×'-helper x' y₀ y₀)
×'-helper-match x nothing nothing y₀ =
  nothing
```

If y is not zero, we continue by decreasing it by one. If y is zero but x is non-zero, then we decrease x by one and reset y to y_0. If both are zero then we stop. As an example, if we compute $3 \times' 2$, then we get the following trace for (x, y):

$$(3,2) \to (3,1) \to (3,0) \to (2,2) \to \ldots \to (0,1) \to (0,0)$$

It takes $4 \times 3 - 1 = 11$ steps for it to terminate.

Exponentiation can be defined as well. We can first define a `_^'_` such that $x \mathbin{\hat{}'} y = (x+1)^{y+1} - 1$, then define a helper function with the following type corecursively:

```
_^'_ : ℕ∞ → ℕ∞ → ℕ∞
x ^' y = ^'-helper (y :: replicate y x) x ℕ.zero

^'-helper : NEList∞ ℕ∞ → ℕ∞ → ℕ → ℕ∞
```

In the helper function, `NEList∞` is a nonempty colist (potentially infinite list), the first argument is an iterated version of what was done when defining multiplication and the second argument is the resetting value. The following is an example trace of $2 \mathbin{\hat{}'} 2$

$$[2,2,2] \to [1,2,2] \to [0,2,2] \to [2,1,2] \to \ldots \to [1,0,0] \to [0,0,0]$$

When some prefix of the colist is filled with zeroes, we need to recurse into the colist to find the next number to decrease, but because colist is coinductive and potentially infinite, as the last argument we keep an inductive natural number to track how deep we can go and use it to recurse into the colist.

To prove the commutativity of addition, we keep track of how many steps we have taken so far in the internal state of the corecursion using a finite natural number. For this, we introduce an operation for adding a finite natural number to a conatural number.

```
infixl 6 _+ₗ_
_+ₗ_ : ℕ → ℕ∞ → ℕ∞
ℕ.zero +ₗ x = x
ℕ.suc n +ₗ x = suc (n +ₗ x)
```

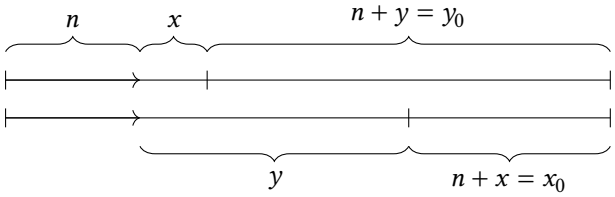
We also prove that the successor operation can be moved from the right side of addition to the front of the whole exression by simple induction on the natural number.

```
+ₗ-suc : ∀ n x → n +ₗ suc x ≡ suc (n +ₗ x)
```

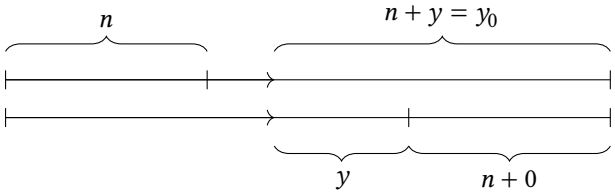
Now we prove the commutativity of addition which has the following type signature:

```
+-comm : ∀ x y → x + y ≡ y + x
```

We illustrate the intuition of the proof using the diagram below. The top line is the left hand side of `+-comm`, and the bottom line is the right hand side. We use x_0 and y_0 to denote the starting x and y values. At the beginning, we set n to be 0, then at each step, if neither x or y has ended, we decrease both of them and increase n by one. The diagram illustrates the middle of this process. As we go through x and y, we learn that both of them must be at least n, and during the proof we keep the proof that adding n to x and y gives us back the original values (x_0 and y_0).



When we reach the end of x, we learn that it is finite and equals to some n. We continue stepping through y on both sides.



When we reach the end of y, we only have n on both sides, which we can immediately prove equal.

In the formalisation, we have two helper lemmas that correspond to the two diagrams. During corecursion, we generalise the resulting equation and add equality arguments to avoid using the transitivity operation after the corecursive call, since transitivity does not preserve guardedness in Agda. We omit the uninteresting equality proofs below for readability, abbreviating them with their type (e.g. `y₀≡x₀` is a proof with type $y_0 \equiv x_0$).

```
+-comm x y = +-comm-helper₁ ℕ.zero x x y y refl refl

+-comm-helper₁ :
  ∀ n x x₀ y y₀ → n +ₗ x ≡ x₀ → n +ₗ y ≡ y₀ →
  x + y₀ ≡ y + x₀
pred (+-comm-helper₁ n x x₀ y y₀ nx ny i) =
  +-comm-helper₁-match n x (pred x) x₀ y (pred y) y₀
    refl refl nx ny i

+-comm-helper₁-match :
  ∀ n x x' x₀ y y' y₀ →
  pred x ≡ x' → pred y ≡ y' →
  n +ₗ x ≡ x₀ → n +ₗ y ≡ y₀ →
  +-match x' y₀ ≡ +-match y' x₀
+-comm-helper₁-match
```

```
  n x nothing x₀ y nothing y₀ px py nx ny =
    cong pred y₀≡x₀
+-comm-helper₁-match
  n x nothing x₀ y (just y') y₀ px py nx ny =
    cong pred y₀≡1+n+y' •
    cong just
      (+-comm-helper₂ n y' (n +L y') refl • y'+n+0≡y'+x₀)
+-comm-helper₁-match
  n x (just x') x₀ y (just y') y₀ px py nx ny =
    cong just
      (+-comm-helper₁ (ℕ.suc n) x' x₀ y' y₀
        1+n+x'≡x₀ 1+n+y'≡y₀)
```

In the first branch, both x and y reach zero at the same time, at which point we can already prove that both sides are equal. In the second branch, x reaches zero first, which is why we switch to the other lemma. There is a symmetric case where y reaches zero first which we omit. The last case is where we have the corecursive call, stepping through both x and y.

In the second lemma, we similarly have to generalise the equation to avoid the transitivity operation. The second branch is an impossible case where we have a contradiction from the equation assumptions.

```
+-comm-helper₂ :
  ∀ n y y₀ → n +L y ≡ y₀ → y₀ ≡ y + (n +L zero)
pred (+-comm-helper₂ n y y₀ ny i) =
  +-comm-helper₂-match
    n y (pred y) y₀ (pred y₀) refl refl ny i

+-comm-helper₂-match :
  ∀ n y y' y₀ y₀' →
  pred y ≡ y' → pred y₀ ≡ y₀' → n +L y ≡ y₀ →
  y₀' ≡ +-match y' (n +L zero)
+-comm-helper₂-match n y nothing y₀ y₀' py py₀ ny =
  sym py₀ • cong pred y₀≡n+0
+-comm-helper₂-match n y (just y') y₀ nothing py py₀ ny
  = impossible
+-comm-helper₂-match n y (just y') y₀ (just y₀') py py₀ ny
  = cong just (+-comm-helper₂ n y' y₀' n+y'≡y₀')
```

The commutativity of multiplication can be proved similarly by tracking the steps taken using $\_+_L\_$.

Defining and proving things directly by guarded corecursion is difficult, since we cannot reuse earlier definitions, such as addition in the case of the definition of multiplication. Defining the corecursion and coinduction principles and then using them can help with managing the proofs. Using the inductively defined identity type instead of the cubical path type for equality also helps, as we can pattern match on them.

4 The Embedded Languages Approach

To avoid the difficulty of working with guarded definitions, we use Nils Anders Danielsson's method [9] of defining an embedded language in this section to keep the definitions guarded. The only difference is that we define coinductive types using coinductive records instead of the deprecated "musical notation" [5].

4.1 Defining Multiplication

We define an embedded language where addition is a constructor, this way we can define multiplication in a similar way as in Section 2 and have Agda accept it without enabling unsafe features. In this language, we also add a way to embed conatural numbers and head normal expressions (NExpr).

```
data Expr : Type where
  embedℕ∞ : ℕ∞ → Expr
  embed   : NExpr → Expr
  _`+_    : Expr → Expr → Expr
```

These head normal expressions are either zero or a successor of an expression. They are defined coinductively and mutually with Expr to allow the definition of multiplication to be defined corecursively.

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

Since Expr is inductive and NExpr is coinductive, this is an instance of a mixed inductive/coinductive definition [12].

Given this language, we can define multiplication. The structure of the definition is similar to the one in Section 2, but here it is guarded, we just need to add some extra embeds:

```
_`×_ : ℕ∞ → ℕ∞ → NExpr
pred (x `× y) = `×-match (pred x) y (pred y)

`×-match :
  Maybe ℕ∞ → ℕ∞ → Maybe ℕ∞ → Maybe Expr
`×-match nothing y y'       = nothing
`×-match (just x') y nothing = nothing
`×-match (just x') y (just y') =
  just (embedℕ∞ y' `+ embed (x' `× y))
```

However, this multiplication exists only as an expression, so we need to interpret this multiplication into actual conatural numbers.

First, we need to define a predecessor function on expressions by recursion.

```
pred_E : Expr → Maybe Expr
pred_E (embedℕ∞ x) = pred_E-embedℕ∞-match (pred x)
pred_E (embed x)   = pred x
pred_E (x `+ y)    = pred_E-+-match (pred_E x) y

pred_E-embedℕ∞-match : Maybe ℕ∞ → Maybe Expr
```

```
pred_E-embedℕ∞-match nothing = nothing
pred_E-embedℕ∞-match (just x') = just (embedℕ∞ x')

pred_E-+-match : Maybe Expr → Expr → Maybe Expr
pred_E-+-match nothing y = pred_E y
pred_E-+-match (just x') y = just (x' `+ y)
```

Then we can interpret the expressions into conatural numbers corecursively via the predecessor function above.

```
interp : Expr → ℕ∞
pred (interp x) = interp-match (pred_E x)

interp-match : Maybe Expr → Maybe ℕ∞
interp-match nothing = nothing
interp-match (just x') = just (interp x')
```

Finally, we can define multiplication on conatural numbers by using the multiplication defined in the language, then interpret it back to conatural numbers.

```
_×_ : ℕ∞ → ℕ∞ → ℕ∞
x × y = interp (embed (x `× y))
```

4.2 Proving the Commutativity of Addition

To prove the commutativity of addition, we also define an embedded language (based on Danielsson [9] and Agda standard library's Stream reasoning combinators [4]) to make the equational reasoning possible by adding the symmetry and transitivity operations on equations as constructors in this language, including the possibility to use corecursive steps in a proof.

We define the language as an inductive relation indexed by conatural numbers. This relation corresponds to the Expr in Section 4.1.

```
data _≈_ : ℕ∞ → ℕ∞ → Type where
  `eq    : x ≡ y → x ≈ y
  `step  : x `≈ y → x ≈ y
  `sym   : x ≈ y → y ≈ x
  `trans : x ≈ y → y ≈ z → x ≈ z
```

We also mutually define a coinductive relation corresponding to NExpr in Section 4.1.

```
record _`≈_ (n k : ℕ∞) : Type where
  coinductive
  field
    pred : Maybe~ _≈_ (pred n) (pred k)
```

We use the pointwise relation on Maybe values defined as follows:

```
data Maybe~ (A~ : A → A → Type) :
  Maybe A → Maybe A → Type
  where
    refl-nothing : Maybe~ A~ nothing nothing
    cong-just : A~ a a' → Maybe~ A~ (just a) (just a')
```

We define the predecessor over the new _≈_ relation and the interpretation into equality analogously to the ones in Section 4.1.

```
pred_E : x ≈ y → Maybe~ _≈_ (pred x) (pred y)
interp : x ≈ y → x ≡ y
interp-match : Maybe~ _≈_ x y → x ≡ y
```

Let us set up the notation for equational reasoning using Agda's standard library's notation [4].

```
pattern _`↺⟨_⟩_ x xy yz = `trans {x} (`step xy) yz
pattern _`≡⟨_⟩_ x xy yz = `trans {x} (`eq xy) yz

_`∎ : ∀ a → a ≈ a
x `∎ = `eq {x} refl
```

With this we show the usage of the language and equational reasoning to prove commutativity of addition over conatural numbers.

First, we prove that suc commutes with addition, in which we already need to use transitivity, thus we prove it in the language. We omit the proof here for brevity.

```
`+-suc : ∀ x y → x + suc y `≈ suc (x + y)
```

We then interpret the proof in the language into an equality.

```
+-suc : ∀ x y → x + suc y ≡ suc (x + y)
+-suc x y = interp (`step (`+-suc x y))
```

With everything set up, we are now able to prove commutativity. As in the definition in Section 4.1, the proof here has almost the same structure as the one in Section 2, but now it is guarded. We omit the non-recursive cases in `+-comm-match.

```
`+-comm : ∀ x y → x + y `≈ y + x
pred (`+-comm x y) =
  `+-comm-match x (pred x) y (pred y) refl refl

`+-comm-match :
  ∀ x x' y y' → x' ≡ pred x → y' ≡ pred y →
  Maybe~ _≈_ (+-match x' y) (+-match y' x)
`+-comm-match x (just x') y (just y') eq1 eq2 =
  cong-just
    ( x' + y        `≡⟨ cong (x' +_) (pred-inj (sym eq2)) ⟩
      x' + suc y'   `↺⟨ `+-comm x' (suc y') ⟩
      suc y' + x'   `≡⟨ pred-inj refl ⟩
      suc (y' + x') `≡⟨ sym (+-suc y' x') ⟩
      y' + suc x'   `≡⟨ cong (y' +_) (pred-inj eq1) ⟩
      y' + x        `∎)
```

There is one difference in this proof compared to the one in Section 2, which is that we need to swap two steps. First, we need to use commutativity to switch the operands of _+_ before making suc the outermost function. This is because we do not have congruence over suc in our language, but it could also be added to the language as a constructor.

The proof only exists in our language so again we convert it to an equality the same way we did with the +-suc property.

```
+-comm : ∀ x y → x + y ≡ y + x
+-comm x y = interp (`step (`+-comm x y))
```

This way, Agda sees that the equational reasoning is guarded, hence it accepts the proof.

We have seen how we can define functions and prove equations using the method by Danielsson [9]. However, this method can result in code duplication, as can be seen in the definition of pred_E and pred_E-+-match in Section 4.1, where we had to duplicate the definition of addition. If we want to use this definition of multiplication and prove properties about it, we would need to separately prove how the addition within the language is related to the addition on conatural numbers.

A way to avoid the code duplication is to define addition using interp and the _`+_ constructor. However, if a coinductive proof depends on the congruence of an operation such as addition, then we need to add the congruence of that operation to the proof language, which means that when interpreting the language into equality, we will have to explicitly prove the congruence of that operation. This proof of congruence duplicates the structure of the definition of the operation. In the next section, we merge operations and equality proofs into a single language to avoid this problem.

5 A Quotiented Embedded Language

Since we are working in Cubical Agda, which has higher inductive types, that is datatypes with equality (path) constructors, we can adapt the approach in Section 4 to add all the algebraic operations and equations that we are interested in into a single embedded language.

5.1 Commutative Semiring

We will first show that the conatural numbers form a commutative semiring.

We mutually define expressions as a higher inductive type and head normal expressions as a coinductive type. This is similar to the types in Section 4.1, except we add equations as constructors for Expr.

```
data Expr : Type
record NExpr : Type
```

Head normal expressions are represented as a coinductive record from which we can extract the predecessor.

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

In Expr, we include all of the commutative semiring operations, constants, and equations.

```
data Expr where
  _+_       : Expr → Expr → Expr
  +-assoc   : ∀ x y z → (x + y) + z ≡ x + (y + z)
  +-comm    : ∀ x y → x + y ≡ y + x

  zero      : Expr
  +-idL     : ∀ x → zero + x ≡ x

  _×_       : Expr → Expr → Expr
  ×-assoc   : ∀ x y z → (x × y) × z ≡ x × (y × z)
  ×-comm    : ∀ x y → x × y ≡ y × x

  one       : Expr
  ×-idL     : ∀ x → one × x ≡ x

  ×-distL-+ : ∀ x y z → (x + y) × z ≡ x × z + y × z
  ×-annihL  : ∀ x → zero × x ≡ zero
```

Since we are working in Cubical Agda, where equations/paths can have content, we need to also say that the equations of Expr are proof irrelevant, that is, it is a set.

```
  isSetExpr : isSet Expr
```

We add a constructor to embed head normal expressions into expressions as in Section 4.1.

```
  embed : NExpr → Expr
```

In addition, we add equations to relate taking the predecessor on head normal expression. Note that here we depend on the destructor pred of NExpr.

```
  embed-zero :
    ∀ x → pred x ≡ nothing → embed x ≡ zero
  embed-suc  :
    ∀ x x' → pred x ≡ just x' → embed x ≡ one + x'
```

Finally, a constructor for transitivity is added because the general transitivity function does not preserve guardedness.

```
  trans : ∀ {x y z} → x ≡ y → y ≡ z → x ≡ z
```

This is a mixed higher-inductive/coinductive definition.

We introduce an abbreviation for adding one to expressions, though it could also be added as a constructor and an equation.

```
suc : Expr → Expr
suc x = one + x
```

We define an inductive predicate IsPred x' x for convenience, which represents the pred $x \equiv x'$ equation, but it can be pattern matched on so it simplifies the proofs.

```
data IsPred : Maybe Expr → Expr → Type where
  nothing :          IsPred nothing zero
  just    : ∀ x' → IsPred (just x') (suc x')
```

We want to define the predecessor function on expressions by induction. We define the motive and methods (terminology from McBride [16]), so the termination checker does not get in the way. If we do recursion by pattern matching over

a higher inductive type, some implicit arguments get solved to an expression that is not reduced and the termination checker complains. For the motive (the type for the result of the induction), we want the predecessor of an expression and a proof that it is indeed the predecessor in terms of the semiring operations.

```
record Pred (x : Expr) : Type where
  field
    pred   : Maybe Expr
    isPred : IsPred pred x
```

We need Pred to be a set to be able to eliminate into it. Fortunately, this fact is easily provable using the combinators provided by the Agda Cubical library [6].

```
isSetPred : ∀ x → isSet (Pred x)
```

We can define the methods (the cases of the induction) for each constructor in Expr. The one below is the method for addition:

```
infixl 6 _+ₚ_
_+ₚ_ : ∀ {x y} → Pred x → Pred y → Pred (x + y)
pred   (_+ₚ_ {y} xₚ yₚ) = +-pred (pred xₚ) y (pred yₚ)
isPred (_+ₚ_     xₚ yₚ) = +-isPred (isPred xₚ) (isPred yₚ)
```

If we know the predecessor of x and y, then we can define the predecessor of $x + y$.

```
+-pred :
  Maybe Expr → Expr → Maybe Expr → Maybe Expr
+-pred nothing y y' = y'
+-pred (just x') y y' = just (x' + y)
```

We then need to prove that it is actually the predecessor, which we do by using the equality constructors.

```
+-isPred :
  ∀ {x x' y y'} → IsPred x' x → IsPred y' y →
  IsPred (+-pred x' y y') (x + y)
+-isPred {y} nothing p = subst (IsPred _) (sym (+-idₗ _)) p
+-isPred {y} (just x') p =
  subst (IsPred _) (sym (+-assoc _ _ _)) (just (x' + y))
```

We can do the same for multiplication as well, here we show how to define the predecessor. The structure is again the same as the one in Section 2 as we can reuse the existing constructors.

```
×-pred :
  Maybe Expr → Expr → Maybe Expr → Maybe Expr
×-pred nothing y y'       = nothing
×-pred (just x') y nothing = nothing
×-pred (just x') y (just y') = just (y' + x' × y)
```

Below we show the method for the commutativity of addition, which has the same structure as the proof in Section 2, and it has less steps due to pattern matching on IsPred. Note that we do not need to define the isPred component because IsPred is a proposition. We also omit the less interesting cases.

```
+-comm-pred :
  ∀ {x x' y y'} → IsPred x' x → IsPred y' y →
  +-pred x' y y' ≡ +-pred y' x x'
+-comm-pred (just x') (just y') =
  congS just
    ( x' + suc y'   ≡⟨ +-sucᵣ _ _ ⟩
      suc (x' + y') ≡⟨ cong suc (+-comm _ _) ⟩
      suc (y' + x') ≡⟨ sym (+-sucᵣ _ _) ⟩
      y' + suc x'   ∎)
```

We can define the rest of the methods in a similar manner as the ones above. Using these, we can recursively eliminate from expressions into Pred.

```
⟦_⟧ₚ : (x : Expr) → Pred x
```

With this, we can extract the familiar pred_E in Section 4.1, but we also get a proof that it is the predecessor.

```
predₑ : Expr → Maybe Expr
predₑ x = pred ⟦ x ⟧ₚ

isPredₑ : ∀ x → IsPred (predₑ x) x
isPredₑ x = isPred ⟦ x ⟧ₚ
```

Using the new predecessor function, we can define a head normalisation function.

```
interpₙ : Expr → NExpr
pred (interpₙ x) = predₑ x
```

Then we can prove that it is an inverse of embed and that Expr and NExpr are isomorphic using isPred_E and the embed-zero and embed-suc constructors that we added to the language.

```
interpₙ-embed : ∀ x → interpₙ (embed x) ≡ x
embed-interpₙ : ∀ x → embed (interpₙ x) ≡ x
ExprIsoNExpr : Iso Expr NExpr
```

Now we define a function that embeds conatural numbers into expressions using the embed constructor.

```
embedℕ∞ : ℕ∞ → Expr
embedℕ∞ x = embed (embedℕ∞ₙ x)

embedℕ∞ₙ : ℕ∞ → NExpr
pred (embedℕ∞ₙ x) = embedℕ∞-match (pred x)

embedℕ∞-match : Maybe ℕ∞ → Maybe Expr
embedℕ∞-match nothing = nothing
embedℕ∞-match (just x') = just (embedℕ∞ x')
```

We can also interpret expressions into conatural numbers using pred_E as in Section 4.1.

```
interp : Expr → ℕ∞
pred (interp x) = interp-match (predₑ x)

interp-match : Maybe Expr → Maybe ℕ∞
```

```
interp-match nothing = nothing
interp-match (just x') = just (interp x')
```

These turn out to be inverses. If we embed a conatural number in the language and then interpret it back to a conatural number, we get back the conatural number we started with.

```
interp-embedℕ∞ : ∀ x → interp (embedℕ∞ x) ≡ x
pred (interp-embedℕ∞ x i) =
  interp-embedℕ∞-match (pred x) i

interp-embedℕ∞-match :
  ∀ x' → interp-match (embedℕ∞-match x') ≡ x'
interp-embedℕ∞-match nothing = refl
interp-embedℕ∞-match (just x') =
  cong just (interp-embedℕ∞ x')
```

If we interpret an expression from the language to conatural numbers and then we embed it back to the language, then we get an expression equal to the one we started with. Here is where we had to use the transitivity constructor, because otherwise it would not be guarded.

```
embedℕ∞-interp : ∀ x → embedℕ∞ (interp x) ≡ x
embedℕ∞-interp x =
  trans
    (cong embed (embedℕ∞-interpₙ x))
    (embed-interpₙ x)

embedℕ∞-interpₙ :
  ∀ x → embedℕ∞ₙ (interp x) ≡ interpₙ x
pred (embedℕ∞-interpₙ x i) =
  embedℕ∞-interp-match (predₑ x) i

embedℕ∞-interp-match :
  ∀ x' → embedℕ∞-match (interp-match x') ≡ x'
embedℕ∞-interp-match nothing = refl
embedℕ∞-interp-match (just x') =
  cong just (embedℕ∞-interp x')
```

With the roundtrips, we get an isomorphism between Expr and ℕ∞.

```
ExprIsoℕ∞ : Iso Expr ℕ∞
```

Now that we have an isomorphism, we get all the operations and equations in Expr for ℕ∞ by the univalence principle [19]. To do that, we define a record with all the operations and equations of a commutative semiring.

```
record ℕ∞Str (A : Type) (pred : A → Maybe A) : Type
```

For example, we add the following fields (the rest is omitted for brevity):

```
field
  isSetA  : isSet A
  _+_     : A → A → A
  +-assoc : ∀ x y z → (x + y) + z ≡ x + (y + z)
  +-comm  : ∀ x y → x + y ≡ y + x
```

We can also add equations that relate the predecessor function with the semiring operations.

```
field
  pred-zero   : pred zero ≡ nothing
  pred-suc    : ∀ x → pred (one + x) ≡ just x
  unpred-pred :
    ∀ x → matchMaybe zero (one +_) (pred x) ≡ x
```

With the record defined, we can easily define it for Expr.

```
ℕ∞StrExpr : ℕ∞Str Expr predₑ
```

Then, by transporting over the isomorphism using univalence, we get the same record for ℕ∞, and thus getting every single operation and equation in the record for conatural numbers at once. Note that it is possible to derive this without univalence, but with more work for every single operation and equation.

```
ℕ∞Strℕ∞ : ℕ∞Str ℕ∞ pred
ℕ∞Strℕ∞ =
  transport
    (cong₂ ℕ∞Str (isoToPath ExprIsoℕ∞) predₑ≡pred)
    ℕ∞StrExpr
```

Hence we have shown that conatural numbers form a commutative semiring.

5.2 Exponentiation

Now we modify the proof to add exponentiation and its related equations. We have to extend our previous language by adding them as constructors.

```
data Expr where
  _^_          : Expr → Expr → Expr
  ^-assocᵣ-×   : ∀ x y z → x ^ (y × z) ≡ (x ^ z) ^ y
  ^-idᵣ        : ∀ x → x ^ one ≡ x

  ^-distᵣ-+    : ∀ x y z → x ^ (y + z) ≡ x ^ z × x ^ y
  ^-annihᵣ     : ∀ x → x ^ zero ≡ one

  ^-distₗ-×    : ∀ x y z → (x × y) ^ z ≡ x ^ z × y ^ z
  ^-annihₗ     : ∀ x → one ^ x ≡ one
```

However, the current method requires us to specify what the predecessor of exponentiating two numbers is. To do this, we create a new operation that represents a "block" of digits, essentially representing y number of the digit x in base $x+1$ (so x is always the last digit in the base). This gives us the following equation:

$$\text{block}\ x\ y = \sum_{i=0}^{y-1} x(1+x)^i = (1+x)^y - 1$$

For example, the predecessor of 10^3 can be represented using block.

$$\text{block } 9\ 3 = 9 \times 10^0 + 9 \times 10^1 + 9 \times 10^2 = 999$$

We add this block operation as a constructor to the language, and some equations that correspond to the ones for exponentiation.

```
block : Expr → Expr → Expr
block-assoc_R-× :
  ∀ x y z → block x (y × z) ≡ block (block x z) y
block-id_R : ∀ x → block x one ≡ x

block-dist_R-+ :
  ∀ x y z →
  block x (y + z) ≡ block x y + block x z × suc x ^ y
block-annih_R : ∀ x → block x zero ≡ zero

block-dist_L-× :
  ∀ x y z →
  block (y + x × suc y) z ≡
  block y z + block x z × suc y ^ z
block-annih_L : ∀ x → block zero x ≡ zero
```

We also add the equation that relates exponentiation and block.

```
^-suc_L : ∀ x y → suc x ^ y ≡ suc (block x y)
```

Finally, we can add infinity as a constructor and add some properties about it. For example, the last equation allows us to prove $x^\infty = \infty$ for $x \geq 2$.

```
∞ : Expr
+-annih_L : ∀ x → ∞ + x ≡ ∞
block-∞_R : ∀ x → block (suc x) ∞ ≡ ∞
```

We then add the methods for the constructor we added. Here we define the predecessor of exponentiation using the aforementioned block constructor.

```
^-pred :
  Maybe Expr → Expr → Maybe Expr → Maybe Expr
^-pred x'      y nothing  = just zero
^-pred nothing y (just y') = nothing
^-pred (just x') y (just y') = just (block x' y)
```

We can also define the predecessor of block itself using addition and multiplication from the langauge. For example, for the predecessor of block 9 3 = 999 we would get $(9 - 1) + 99 * (1 + 9) = 8 + 99 * 10$.

```
block-pred :
  Expr → Maybe Expr → Maybe Expr → Maybe Expr
block-pred x nothing y'       = nothing
block-pred x (just x') nothing = nothing
block-pred x (just x') (just y') =
  just (x' + block x y' × suc x)
```

The operations and equations that we added to Expr are enough to define the rest of the methods.

The rest of the proof remains the same, we just add the new operations and equations to the ℕ∞Str record. The result is that we have the proof that ℕ∞ forms an exponential commutative semiring. Since we are using Cubical Agda, the usage of the univalence principle computes [21], the only problem is that there are some extraneous transports in the result of the computation, but they are invisible if one only uses the interface provided by the exponential commutative semiring and the predecessor. Here we show an example of defining 2^∞ and proving that $2^\infty = \infty$.

```
example : ℕ∞
example = suc (suc zero) ^ ∞

_ : example ≡ ∞
_ =
  suc (suc zero) ^ ∞          ≡⟨ ^-suc_L _ _ ⟩
  suc (block (suc zero) ∞)    ≡⟨ cong suc (block-∞_R _) ⟩
  suc ∞                       ≡⟨ sym ∞-suc ⟩
  ∞                           ∎
```

And below is another example which proves that the predecessor of one is zero, this shows that one can also reason about the predecessor of conatural numbers using the interface.

```
_ : pred one ≡ just zero
_ =
  pred one          ≡⟨ cong pred (sym (+-id_R one)) ⟩
  pred (one + zero) ≡⟨ pred-suc _ ⟩
  just zero         ∎
```

In this section, we have seen how to add exponentiation to the proof, however, we had to modify the original Expr datatype. As such, this method is not modular, as it is inconvenient to define and prove more operations and equations without modifying the original datatype.

6 Conclusion

We studied three methods of reasoning about coinductive types to circumvent the guardedness issue in Agda. We used cubical type theory because reasoning about coinductive types is simpler in it. The third method in Section 5 is the one where we successfully formalised that conatural numbers form an exponential commutative semiring. To conclude, we give a short comparison between the methods.

In Section 3 we directly used corecursion with specific internal states. Using this method, we did not have to consider creating an intermediate language. However, for every operation and proof we have to come up with a new state that can accurately describe the predecessor as the step of the proofs.

In Section 4.1 we followed Danielsson's method [9] and created a domain speific language to define the multiplication operation and later in Section 4.2 we defined a language for proofs to obtain equational reasoning. The problem is we have to define a new language for every new operation we want in which we have to include every operation we want to reuse as a constructor in order to make Agda see the definition as guarded which quickly leads to code duplication.

In Section 5 we extended our language using the idea that we can include equations themselves using a mixed higher-inductive/coinductive definition, which is only allowed in Cubical Agda. This method allows us to reuse any other algebraic operation and equation that we are defining at the same time. With it, we easily proved that conatural numbers form an exponential commutative semiring, showcasing an application of the univalence principle. The only problem with this method is that it is not modular. If we want to include more operations or properties that we define and proofs, then we have to extend the already existing datatype of expressions.

We believe the last method can be generalised to definitions and proofs about other coinductive types, so that one can easily reason about infinite programs.

7 Future Work

Some questions remain as future work. The method in Section 4 can introduce code duplication, while the method in Section 5 is not modular. We wish to find a modular method of doing corecursion that is as convenient as these, where one can reuse other operations and equations, but does not necessitate code duplication.

Another question is how one can define some more complicated operations, such as tetration or the Ackermann function, on conatural numbers. Since conatural numbers are a coinductive type, one defines operations by specifying the predecessor, however, it is not clear how one can specify the predecessors of the aforementioned operations.

Lastly, in Section 5, we used a mixed higher-inductive/coinductive definition to specify the language of expressions, though it is not clear what its semantic justification is, even though Agda supports it. Unlike mixed inductive/coinductive types [12], we cannot directly write it as the terminal coalgebra of some functor. If we were to parametrise Expr with NExpr and pred, then we would have pred : Maybe (Expr NExpr pred), where the type of pred depends on itself.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions, and Thorsten Altenkirch for originally posing this problem to us as a challenge.

This work was co-funded by the European Union (ERC, HOTT, 101170308). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Andreas Abel. 2017. Equality is incompatible with sized types. Retrieved June 23, 2025 from https://github.com/agda/agda/issues/2820.

[2] Andreas Abel and James Chapman. 2014. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014* (EPTCS). Paul Blain Levy and Neel Krishnaswami, (Eds.) Vol. 153, 51–67. doi:10.4204/EPTCS.153.4.

[3] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy, January 23–25, 2013*. Roberto Giacobazzi and Radhia Cousot, (Eds.) ACM, 27–38. doi:10.1145/2429069.2429075.

[4] [SW] The Agda Community, Agda Standard Library version 2.2, 2025. URL: https://github.com/agda/agda-stdlib.

[5] The Agda Community. 2024. *Coinduction*. Retrieved June 23, 2025 from https://agda.readthedocs.io/en/v2.7.0.1/language/coinduction.html.

[6] [SW] The Agda Community, Cubical Agda Library version 0.8, 2025. URL: https://github.com/agda/cubical.

[7] Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25–27, 2013*. Greg Morrisett and Tarmo Uustalu, (Eds.) ACM, 197–208. doi:10.1145/2500365.2500597.

[8] Thierry Coquand. 1993. Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers* (Lecture Notes in Computer Science). Henk Barendregt and Tobias Nipkow, (Eds.) Vol. 806. Springer, 62–78. doi:10.1007/3-540-58085-9_72.

[9] Nils Anders Danielsson. 2010. Beating the productivity checker using embedded languages. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010* (EPTCS). Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, (Eds.) Vol. 43, 29–48. doi:10.4204/EPTCS.43.3.

[10] Martín Escardó. 2013. Infinite sets that satisfy the principle of omniscience in any variety of constructive mathematics. *J. Symb. Log.*, 78, 3, 764–784. doi:10.2178/JSL.7803040.

[11] Naïm Camille Favier. 2023. CoNat.agda. Retrieved June 23, 2025 from https://github.com/ncfavier/graded-type-theory/blob/d778c49ffe69602d803fbd0c34bac4b08ee355d5/Graded/Modality/Instances/CoNat.agda.

[12] Neil Ghani, Peter G. Hancock, and Dirk Pattinson. 2009. Representations of stream processors using nested fixed points. *Log. Methods Comput. Sci.*, 5, 3. doi:10.2168/LMCS-5(3:9)2009.

[13] Tatsuya Hagino. 1987. *A Categorical Programming Language*. Ph.D. Dissertation. University of Edinburgh. doi:10.48550/arXiv.2010.05167.

[14] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21–24, 1996*. Hans-Juergen

Boehm and Guy L. Steele Jr., (Eds.) ACM Press, 410–423. doi:10.1145/237721.240882.

[15] Conor McBride. 2009. Let's see how things unfold: reconciling the infinite with the intensional (extended abstract). In *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7–10, 2009. Proceedings* (Lecture Notes in Computer Science). Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, (Eds.) Vol. 5728. Springer, 113–126. doi:10.1007/978-3-642-03741-2_9.

[16] Conor McBride and James McKinna. 2004. The view from the left. *J. Funct. Program.*, 14, 1, 69–111. doi:10.1017/S0956796803004829.

[17] Hiroshi Nakano. 2000. A modality for recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26–29, 2000*. IEEE Computer Society, 255–266. doi:10.1109/LICS.2000.855774.

[18] Ulf Norell. 2008. Dependently typed programming in agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures* (Lecture Notes in Computer Science). Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, (Eds.) Vol. 5832. Springer, 230–266. doi:10.1007/978-3-642-04652-0_5.

[19] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study. doi:10.48550/arXiv.1308.0729.

[20] Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020*. Jasmin Blanchette and Catalin Hritcu, (Eds.) ACM, 270–283. doi:10.1145/3372885.3373814.

[21] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.*, 31, e8. doi:10.1017/S0956796821000034.

Received 2025-06-23; accepted 2025-07-23