Cubical Agda[1] 

Paul Tiemeijer Dante van Gemert

Radboud University Nijmegen

20 December 2022

Outline

Outline

What's so Cubicical About Cubical Agda?

What's so Cubicical About Cubical Agda?

- Paths (univalence)

What's so Cubicical About Cubical Agda?

- Paths (univalence)
- HITs

What's so Great About Cubes?

Univalence: The Ultimate ...?

What's so Great About Cubes?

Univalence: The Ultimate ...?

- Transfer programs and proofs between equivalent types
- Prove properties for proof-oriented datatypes using computation-oriented ones.
- Reason about dependently typed programs in terms of inductive families
- Define and reason about quotient types
- Represent topoligical spaces as datatypes and reason about them synthethically
- Treat bisimilar elements of coinductive types as equal

Recall

- Definitional equality: $=$
- Path equality: $\equiv$
- Equivalence: $\simeq$

Paths in Agda

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ
```

Paths in Agda

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} x y = PathP (λ _ → A) x y
```

Paths in Agda

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} x y = PathP (λ _ → A) x y

transport : A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a
```

Paths in Agda

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} x y = PathP (λ _ → A) x y

transport : A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a

refl : {x : A} → x ≡ x
refl {x = x} = λ i → x
```

Paths in Agda

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} x y = PathP (λ _ → A) x y

transport : A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a

refl : {x : A} → x ≡ x
refl {x = x} = λ i → x
```

Implicit arguments

$\{A : Set\ \ell\}$ and $\{x : A\}$ are implicit arguments. In the rest of the presentation, quantification over $\{A : Set\ \ell\}$ and $\{\ell : Level\}$ is implicitly assumed.

Interlude: *It's Really Quite Simple*

According to the authors, proving in Cubical Agda is...

Interlude: *It's Really Quite Simple*

According to the authors, proving in Cubical Agda is...

- "Straightforward" (6x)

Interlude: *It's Really Quite Simple*

According to the authors, proving in Cubical Agda is...

- "Straightforward" (6x)
- "Easy" (10x)

Interlude: *It's Really Quite Simple*

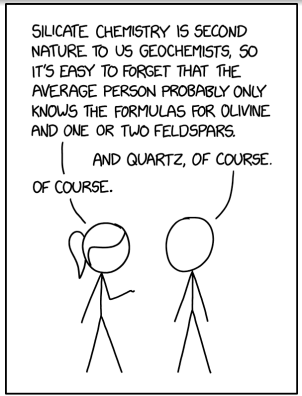
According to the authors, proving in Cubical Agda is...

- "Straightforward" (6x)
- "Easy" (10x)
- "Trivial" (11x)

Interlude: *It's Really Quite Simple*

According to the authors, proving in Cubical Agda is...

- "Straightforward" (6x)
- "Easy" (10x)
- "Trivial" (11x)
- "Simple" (26x)

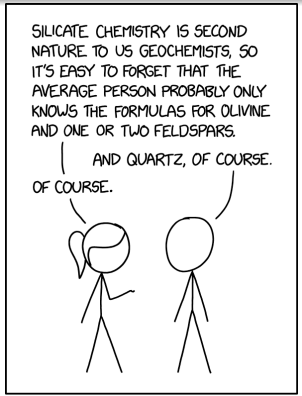


Interlude: *It's Really Quite Simple*

According to the authors, proving in Cubical Agda is...

- "Straightforward" (6x)
- "Easy" (10x)
- "Trivial" (11x)
- "Simple" (26x)

Evaluating these claims is left as an exercise to the reader



Outline

Binary Numbers in Agda

Binary Numbers in Agda

Binary representation of $\mathbb{N}$:

```
data Bin : Set where
    bin0   : Bin
    binPos : Pos → Bin
data Pos : Set where
    pos1 : Pos
    x0   : Pos → Pos
    x1   : Pos → Pos
```

Note

Binary numbers are LSB-first: `binPos (x0 pos1)` is binary `10`.

Addition on Binary Numbers (1/2)

Problem: defining addition on binary numbers is *not* trivial.

Addition on Binary Numbers (1/2)

Problem: defining addition on binary numbers is *not* trivial.
Solution:

```
ℕ≃Bin : ℕ ≃ Bin
ℕ≃Bin = isoToEquiv (iso ℕ→Bin Bin→ℕ Bin→ℕ→Bin ℕ→Bin→ℕ)

ℕ≡Bin : ℕ ≡ Bin
ℕ≡Bin = ua ℕ≃Bin
```

Addition on Binary Numbers (2/2)

A path between $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\mathit{Bin} \to \mathit{Bin} \to \mathit{Bin}$:

```
_+Bin_ : Bin → Bin → Bin
_+Bin_ = transport (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_
```

Addition on Binary Numbers (2/2)

A path between $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $Bin \to Bin \to Bin$:

```
_+Bin_ : Bin → Bin → Bin
_+Bin_ = transport (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_
```

Example

Proof that $100_b + 1_b = 101_b$:

```
_ : (binPos (x0 (x0 pos1))) +Bin (binPos pos1)
    ≡ binPos (x1 (x0 pos1))
_ = refl
```

Associativity of Binary Addition (1/2)

```
addp : PathP (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_ _+Bin_
addp i = transp
           (λ j → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j))
           (~ i) _+_
```

Associativity of Binary Addition (1/2)

```
addp : PathP (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_ _+Bin_
addp i = transp
           (λ j → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j))
           (~ i) _+_
```

This constitutes a path from `_+_` to `_+Bin_`:

At `i0`, `transp (λ j → ℕ≡Bin i0 → ℕ≡Bin i0 → ℕ≡Bin i0) i1 _+_`

At `i1`, `transp (λ j → ℕ≡Bin j → ℕ≡Bin j → ℕ≡Bin j) i0 _+_`

Associativity of Binary Addition (1/2)

```
addp : PathP (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_ _+Bin_
addp i = transp
           (λ j → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j))
           (~ i) _+_
```

This constitutes a path from `_+_` to `_+Bin_`:

At `i0`, `transp (λ j → ℕ → ℕ → ℕ) i1 _+_`

At `i1`, `transp (λ j → ℕ≡Bin j → ℕ≡Bin j → ℕ≡Bin j) i0 _+_`

Associativity of Binary Addition (1/2)

```
addp : PathP (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_ _+Bin_
addp i = transp
           (λ j → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j) → ℕ≡Bin (i ∧ j))
           (~ i) _+_
```

This constitutes a path from `_+_` to `_+Bin_`:

At `i0`, `transp (λ j → ℕ → ℕ → ℕ) i1 _+_`

At `i1`, `transp (λ j → ℕ≡Bin j → ℕ≡Bin j → ℕ≡Bin j) i0 _+_`

Recall the definition of `_+Bin_`:

```
_+Bin_ = transport (λ i → ℕ≡Bin i → ℕ≡Bin i → ℕ≡Bin i) _+_
```

Associativity of Binary Addition (2/2)

```
+Bin-assoc : (m n o : Bin) → m +Bin (n +Bin o) ≡ (m +Bin n) +Bin o
+Bin-assoc = transport
               (λ i → (m n o : ℕ≡Bin i) → addp i m (addp i n o)
                                          ≡ addp i (addp i m n) o)
               +-assoc
```

Associativity of Binary Addition (2/2)

```
+Bin-assoc : (m n o : Bin) → m +Bin (n +Bin o) ≡ (m +Bin n) +Bin o
+Bin-assoc = transport
               (λ i → (m n o : ℕ≡Bin i) → addp i m (addp i n o)
                                          ≡ addp i (addp i m n) o)
               +-assoc
```

We transport the proof that unary addition is associative along the path from

$$(m\,n\,o : \mathbb{N}) \to m + (n + o) \equiv (m + n) + o$$

to

$$(m\,n\,o : \mathit{Bin}) \to m +_{\mathit{Bin}} (n +_{\mathit{Bin}} o) \equiv (m +_{\mathit{Bin}} n) +_{\mathit{Bin}} o$$

Outline

Inductive Families: Vectors

Queue the demo effect!

Outline

Integers as a HIT

We've seen natural numbers, but how do we represent integers?

Integers as a HIT

We've seen natural numbers, but how do we represent integers? In terms of natural numbers:

```
data ℤ : Set where
    pos : (n : ℕ) → ℤ
    neg : (n : ℕ) → ℤ
    posneg : pos 0 ≡ neg 0
```

posneg is a path from `pos 0` to `neg 0` (= a function from the interval type $\mathbb{I}$ to $\mathbb{Z}$).

Let's go from $\mathbb{Z}$ to $\mathbb{Z}$

To construct a path, we need `sucℤ`, `predℤ`, `sucPredℤ` and `predSucℤ`.

Let's go from $\mathbb{Z}$ to $\mathbb{Z}$

To construct a path, we need `sucℤ`, `predℤ`, `sucPredℤ` and `predSucℤ`.

```
sucℤ : ℤ → ℤ
sucℤ (pos n)       = pos (suc n)
sucℤ (neg zero)    = pos 1
sucℤ (neg (suc n)) = neg n
sucℤ (posneg _)    = pos 1
```

`predℤ`, `sucPredℤ`, `predSucℤ` are "trivial", their implementation is left as an exercise.

Let's go from $\mathbb{Z}$ to $\mathbb{Z}$

To construct a path, we need `sucℤ`, `predℤ`, `sucPredℤ` and `predSucℤ`.

```
sucℤ : ℤ → ℤ
sucℤ (pos n)       = pos (suc n)
sucℤ (neg zero)    = pos 1
sucℤ (neg (suc n)) = neg n
sucℤ (posneg _)    = pos 1
```

`predℤ`, `sucPredℤ`, `predSucℤ` are "trivial", their implementation is left as an exercise.

We can now construct a path from $\mathbb{Z}$ to $\mathbb{Z}$:

```
sucPathℤ : ℤ ≡ ℤ
sucPathℤ = isoToPath (iso sucℤ predℤ sucPredℤ predSucℤ)
```

Addition on Integers

To prove that addition is an equivalence, we define an alternative addition function. Consider the following path equality that composes `sucPathℤ` with itself `n` times:

```
addEq : ℕ → ℤ ≡ ℤ
addEq zero    = refl
addEq (suc n) = addEq n · sucPathℤ
```

Addition on Integers

To prove that addition is an equivalence, we define an alternative addition function. Consider the following path equality that composes `sucPathℤ` with itself `n` times:

```
addEq : ℕ → ℤ ≡ ℤ
addEq zero    = refl
addEq (suc n) = addEq n · sucPathℤ
```

By transporting along these paths (`addEq` and the similarly defined `subEq`), we get an addition function:

```
addℤ : ℤ → ℤ → ℤ
addℤ m (pos n)    = transport (addEq n) m
addℤ m (neg n)    = transport (subEq n) m
addℤ m (posneg _) = m
```

Outline

Donut Space 🍩

```
data S¹ : Set where
    base : S¹
    loop : base ≡ base
```

Donut Space 🍩

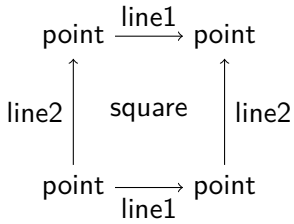
```
data S¹ : Set where
    base : S¹
    loop : base ≡ base

data Torus : Type₀ where
    point : Torus
    line1 : point ≡ point
    line2 : point ≡ point
    square : PathP (λ i → line1 i ≡ line1 i)
                line2 line2
```

Donut Space 🍩

```
data S¹ : Set where
    base : S¹
    loop : base ≡ base

data Torus : Type₀ where
    point : Torus
    line1 : point ≡ point
    line2 : point ≡ point
    square : PathP (λ i → line1 i ≡ line1 i)
                   line2 line2
```



You can visualise the paths on a torus from the image on the right: try to connect *line*1 to *line*1, and then *line*2 to *line*2.

When Life Gives You Two Circles...

In topology, a torus is equivalent to two circles. Using Cubical Agda this is "almost entirely trivial" to prove:

When Life Gives You Two Circles...

In topology, a torus is equivalent to two circles. Using Cubical Agda this is "almost entirely trivial" to prove:

```
t2c : Torus → S¹ × S¹
t2c point        = ( base , base )
t2c (line1 i)    = ( loop i , base )
t2c (line2 j)    = ( base , loop j )
t2c (square i j) = ( loop i , loop j )
```

When Life Gives You Two Circles...

In topology, a torus is equivalent to two circles. Using Cubical Agda this is "almost entirely trivial" to prove:

```
t2c : Torus → S¹ × S¹
t2c point        = ( base , base )
t2c (line1 i)    = ( loop i , base )
t2c (line2 j)    = ( base , loop j )
t2c (square i j) = ( loop i , loop j )

Torus≡S¹×S¹ : Torus ≡ S¹ × S¹
Torus≡S¹×S¹ = isoToPath (iso t2c c2t t2c-c2t c2t-t2c)
```

Outline

All Done

All Done

Bibliography

A. Vezzosi, A. Mörtberg, and A. Abel.
Cubical agda: a dependently typed programming language with univalence and higher inductive types.
Proceedings of the ACM on Programming Languages, 3(ICFP):1–29, 2019.