

Coinductive Programming and Proving in Agda

Lecture 2: Coinductive proving in Agda

Jesper Cockx

21 January 2026

Technical University Delft

Lecture plan

- 1 Curry-Howard
- 2 Properties of coinductive types
- 3 The identity type
- 4 Equational reasoning
- 5 Bisimulation
- 6 Cubical bisimulation

Outline

- 1 Curry-Howard
- 2 Properties of coinductive types
- 3 The identity type
- 4 Equational reasoning
- 5 Bisimulation
- 6 Cubical bisimulation

Curry-Howard for propositional logic

We can interpret logical propositions as the **types** of all their possible proofs.

Propositional logic	Type system
conjunction	$P \times Q$
disjunction	$\text{Either } P Q$
implication	$P \rightarrow Q$
truth	\top
falsity	\perp

To prove a proposition, we just implement a function of the corresponding type!

Constructive logic

In classical logic we can prove certain ‘non-constructive’ statements:

- $P \vee (\neg P)$ (excluded middle)
- $\neg\neg P \Rightarrow P$ (double negation elimination)

However, Agda uses a **constructive logic**: a proof of $A \vee B$ gives us a **decision procedure** to tell whether A or B holds.

When P is unknown, it’s impossible to decide whether P or $\neg P$ holds, so the excluded middle is **unprovable** in Agda.

From classical to constructive logic

Consider the proposition P (“ P is true”) vs. $\neg\neg P$ (“It would be absurd if P were false”).

Classical logic can't tell the difference between the two, but constructive logic can.

Theorem (Gödel and Gentzen). P is provable in classical logic if and only if $\neg\neg P$ is provable in constructive logic.

Exercise. Prove that the double negation of the excluded middle holds in Agda.

Defining predicates

We can define a predicate on type A as a **dependent type** with base type A. For example:

```
data IsEven : ℕ → Set where
```

```
  e-zero : IsEven zero
```

```
  e-suc2 : IsEven n → IsEven (suc (suc n))
```

```
two-is-even : IsEven 2
```

```
two-is-even = e-suc2 e-zero
```

```
five-is-not-even : IsEven 5 → ⊥
```

```
five-is-not-even (e-suc2 (e-suc2 ()))
```

Induction in Agda

In Agda, a **proof by induction** is simply a function using pattern matching and recursion:

`double : ℕ → ℕ`

`double zero = zero`

`double (suc m) = suc (suc (double m))`

`double-even : (n : ℕ) → IsEven (double n)`

`double-even zero = e-zero`

`double-even (suc m) = e-suc2 (double-even m)`

Proving things about programs

General rule of thumb: A proof about a function often follows the same structure as that function:

- To prove something about a function by pattern matching, the proof will also use pattern matching (= **proof by cases**)
- To prove something about a recursive function, the proof will also be recursive (= **proof by induction**)

Outline

- 1 Curry-Howard
- 2 Properties of coinductive types
- 3 The identity type
- 4 Equational reasoning
- 5 Bisimulation
- 6 Cubical bisimulation

Finite colists

Finite is an **inductive** predicate on a (mixed) coinductive type:

```
data Finite {A : Set} : Colist A → Set where
  [] : Finite []
  -∷_ : Finite (xs .force) → Finite (x :: xs)
```

```
fromListFin : (xs : List A) → Finite (fromList xs)
fromListFin [] = []
fromListFin (x :: xs) = -∷ fromListFin xs
```

Converting back to a list

We can convert a finite colist back to a list:

`toList : (xs : Colist A) → Finite xs → List A`

`toList [] [] = []`

`toList (x :: xs) (‐:: fin) = x :: toList (xs .force) fin`

Question. Is this function using induction or coinduction?

Infinite colists

Infinite is a coinductive predicate on colists:

mutual

```
data Infinite {A : Set} : Colist A → Set where
  -∷_ : Infinite' xs → Infinite (x :: xs)
```

```
record Infinite' (xs : Colist' A) : Set where
```

coinductive

```
  field force : Infinite (xs .force)
```

```
open Infinite' public
```

Exercise. Prove that fromStream always produces an infinite colist.

Converting back to a stream

Exercise. Implement the following function:

`toStream :`

`(xs : Colist A) → Infinite xs → Stream A`

Question. What should the function do in the case of an empty colist?

Question. Is this function using induction or coinduction?

Finite or infinite?

Question. Can we prove the following?

`finite-or-infinite : (xs : Colist A) →
 Finite xs ⊕ Infinite xs`

Finite or infinite?

Question. Can we prove the following?

`finite-or-infinite : (xs : Colist A) →
 Finite xs ⊕ Infinite xs`

Answer. No, but we can prove this instead:

`infinite-not-finite : Infinite xs → ¬(Finite xs)
not-finite-infinite : ¬(Finite xs) → Infinite xs`

where $\neg A = A \rightarrow \perp$. **Exercise.** Do it!

Outline

- 1 Curry-Howard**
- 2 Properties of coinductive types**
- 3 The identity type**
- 4 Equational reasoning**
- 5 Bisimulation**
- 6 Cubical bisimulation**

The identity type

The **identity type** $x \equiv y$ says x and y are equal:

```
data _≡_ {A : Set} : A → A → Set where  
  refl : x ≡ x
```

The constructor **refl** proves that two terms are equal if they have the same normal form:

```
one-plus-one : 1 + 1 ≡ 2  
one-plus-one = refl
```

Quiz question

Question. What is the type of the Agda expression $\lambda b \rightarrow (b \equiv \text{true})$?

1. $\text{Bool} \rightarrow \text{Bool}$
2. $\text{Bool} \rightarrow \text{Set}$
3. $(b : \text{Bool}) \rightarrow b \equiv \text{true}$
4. It is not a well-typed expression

Application of the identity type: Writing test cases

One use case of the identity type is for writing test cases:

`test1 : length (42 :: []) ≡ 1`

`test1 = refl`

`test2 : length (map (1 + _) (0 :: 1 :: 2 :: [])) ≡ 3`

`test2 = refl`

The test cases are run each time the file is loaded!

Proving correctness of functions

We can use the identity type to prove the correctness of functional programs.

Example.

`not-not : (b : Bool) → not (not b) ≡ b`

`not-not true = refl`

`not-not false = refl`

Pattern matching on `refl`

If we have a proof of $x \equiv y$ as input, we can pattern match on the constructor `refl` to show Agda that x and y are equal:

```
castVec : m ≡ n → Vec A m → Vec A n
```

```
castVec refl xs = xs
```

When you pattern match on `refl`, Agda applies unification to the two sides of the equality.

Properties of equality

`sym : $x \equiv y \rightarrow y \equiv x$`

`sym refl = refl`

`trans : $x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z$`

`trans refl refl = refl`

`cong : ($f : A \rightarrow B$) \rightarrow x \equiv y \rightarrow fx \equiv fy`

`cong f refl = refl`

From lists to colists and back

Exercise. Prove that converting a list to a colist and back is the identity:

`fromListInv : (xs : List A)`

$\rightarrow \text{toList}(\text{fromList } xs) (\text{fromListFin } xs) \equiv xs$

Can we prove the same about `fromStream`?

Outline

- 1 Curry-Howard**
- 2 Properties of coinductive types**
- 3 The identity type**
- 4 Equational reasoning**
- 5 Bisimulation**
- 6 Cubical bisimulation**

Equational reasoning

We can write more readable identity proofs by using equational reasoning operators:

$_ \equiv \langle _ \rangle _ : (x : A) \rightarrow x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z$

$x \equiv \langle \text{refl} \rangle q = q$

$_ \equiv \langle \rangle _ : (x : A) \rightarrow x \equiv y \rightarrow x \equiv y$

$x \equiv \langle \rangle q = x \equiv \langle \text{refl} \rangle q$

$_ \blacksquare : (x : A) \rightarrow x \equiv x$

$x \blacksquare = \text{refl}$

Equational reasoning example

reverse : List A → List A

reverse [] = []

reverse (x :: xs) = reverse xs ++ (x :: [])

reverse-singleton : reverse (x :: []) ≡ x :: []

reverse-singleton {x = x} =

reverse (x :: []) ≡ ()

reverse [] ++ (x :: []) ≡ ()

[] ++ (x :: []) ≡ ()

(x :: []) ■

Equational reasoning + induction

add-n-zero : $(n : \mathbb{N}) \rightarrow n + \text{zero} \equiv n$

add-n-zero zero = refl

add-n-zero (suc n) =

(suc n) + zero $\equiv \langle \rangle$

suc (n + zero) $\equiv \langle \text{cong suc (add-n-zero } n \text{)} \rangle$

suc n ■

Here we have to provide an explicit proof that $\text{suc}(n + \text{zero}) = \text{suc } n$, using the IH.

Exercise. Prove that $\text{xs} ++ [] = \text{xs}$.

Exercise. Prove associativity of $++$.

Example 1: functor laws for List

The first functor law for lists:

$\text{map-id} : \{A : \text{Set}\} (\text{xs} : \text{List } A) \rightarrow \text{map id xs} \equiv \text{xs}$

$\text{map-id } [] = \text{refl}$

$\text{map-id } (x :: \text{xs}) =$

$\text{map id } (x :: \text{xs}) \equiv \langle \rangle$

$\text{id } x :: \text{map id xs} \equiv \langle \rangle$

$x :: \text{map id xs} \equiv \langle \text{cong } (x :: _) (\text{map-id xs}) \rangle$

$x :: \text{xs}$



Exercise

Prove the second functor law for **List**.

First, we need to define function composition:¹

$$\begin{aligned}\underline{\circ} : \{A\ B\ C : \text{Set}\} \rightarrow \\ (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\ f \circ g = \lambda x \rightarrow f(g x)\end{aligned}$$

Now we can prove that

$$\text{map } (f \circ g) x = (\text{map } f \circ \text{map } g) x.$$

¹Unicode input for \circ : \circ

Example 2: verifying optimizations

A faster version of reverse in $O(n)$:

`reverse-acc : List A → List A → List A`

`reverse-acc [] ys = ys`

`reverse-acc (x :: xs) ys = reverse-acc xs (x :: ys)`

`reverse' : List A → List A`

`reverse' xs = reverse-acc xs []`

Equivalence of reverse and reverse'

reverse-acc-lemma : (xs ys : List A)

→ reverse-acc xs ys ≡ reverse xs ++ ys

reverse-acc-lemma [] ys = refl

reverse-acc-lemma (x :: xs) ys =

reverse-acc (x :: xs) ys ≡ ⟨ ⟩

reverse-acc xs (x :: ys)

≡ ⟨ reverse-acc-lemma xs (x :: ys) ⟩

reverse xs ++ (x :: ys)

≡ ⟨ sym (append-assoc (reverse xs) (x :: []) ys) ⟩

(reverse xs ++ (x :: [])) ++ ys ≡ ⟨ ⟩

reverse (x :: xs) ++ ys ■

Exercise. Use this to prove that reverse and reverse' are equivalent.

Outline

- 1 Curry-Howard**
- 2 Properties of coinductive types**
- 3 The identity type**
- 4 Equational reasoning**
- 5 Bisimulation**
- 6 Cubical bisimulation**

Proving an equality between streams

Let's prove an equality on streams:

`takeDropS : (n : \mathbb{N}) (s : Stream A)`

$\rightarrow \text{takeS } n s ++ \text{dropS } n s \equiv s$

`takeDropS zero s = refl`

`takeDropS (suc n) s = {! cong (s .headS ::S_) ? !}`

Error: `s .headS ::S _y_296 != s`

Bisimulation of streams

Streams are coinductive, but the identity type is **inductive**: Agda needs to ‘see’ both sides are equal in a finite number of steps.

We need a **coinductive** relation instead:

```
record _~_ {A : Set} (s1 s2 : Stream A) : Set where
  coinductive
  field
    heads : s1 .heads ≡ s2 .heads
    tails  : s1 .tails ~ s2 .tails
  open _~_ public
```

Proving bisimulation of streams

Proving bisimulation is just defining a coinductive value of the right type:

$\text{refl}\sim : (s : \text{Stream } A) \rightarrow s \sim s$

$\text{refl}\sim s.\text{headS} = \text{refl}$

$\text{refl}\sim s.\text{tailS} = \text{refl}\sim (s.\text{tailS})$

$\text{takeDropS} : (n : \mathbb{N}) (s : \text{Stream } A)$

$\rightarrow (\text{takeS } n s ++ \text{dropS } n s) \sim s$

$\text{takeDropS zero } s = \text{refl}\sim s$

$\text{takeDropS (suc } n) s.\text{headS} = \text{refl}$

$\text{takeDropS (suc } n) s.\text{tailS} = \text{takeDropS } n (s.\text{tailS})$

From streams to colists and back

Exercise. Prove that converting a stream to a colist and back results in a stream that is bisimilar to the original one:

`fromStreamInv : (xs : Stream A)`

$\rightarrow \text{toStream } (\text{fromStream } xs) \text{ (fromStreamInv } xs\text{)}$

$\sim XS$

Outline

- 1 Curry-Howard
- 2 Properties of coinductive types
- 3 The identity type
- 4 Equational reasoning
- 5 Bisimulation
- 6 Cubical bisimulation

Cubical Agda

Cubical Agda is an extension of Agda with primitives from **cubical type theory**, a version of homotopy type theory (HoTT).

In particular, it provides the cubical path type, a version of **observational equality**.

Cubical Agda also provides other primitives like **glue** and **hcomp**, which are needed to prove the principle of **univalence**.

The cubical path type

The cubical **interval** type **I** is a type with two elements **io** and **i1** that *cannot be distinguished* from inside Agda.

The cubical path type **Path A x y** (sometimes also written $x \equiv y$) is the type of functions $f : I \rightarrow A$ such that $f \text{ io} = x$ and $f \text{ i1} = y$.

Properties of the path type

`reflP : {x : A} → Path A x x`

`reflP {x = x} i = x`

`congP : (f : A → B) → Path A x y → Path B (fx) (fy)`

`congP f p i = f (p i)`

`symP` and `transP` need additional cubical primitives, let's not worry about it for now.

In fact, we can prove that `Path A x y` is **isomorphic** to the inductive identity type!

Functional extensionality

Functional extensionality states that two functions are equal if they give equal outputs on every input.

For inductive identity, functional extensionality is consistent but unprovable.

For the cubical path type, it is trivial:

$$\begin{aligned}\text{funExt} : \{f g : A \rightarrow B\} \\ \rightarrow ((x : A) \rightarrow \text{Path } B (fx) (gx)) \\ \rightarrow \text{Path } (A \rightarrow B) fg\end{aligned}$$

$$\text{funExt } h i x = h x i$$

Cubical bisimulation

The cubical path type serves as a general bisimulation relation for any coinductive type:

`takeDropS : (n : \mathbb{N}) (s : Stream A)`

$\rightarrow \text{Path } (\text{Stream } A) (\text{takeS } n s ++ \text{dropS } n s) s$

`takeDropS zero s i = s`

`takeDropS (suc n) s i .heads = s .heads`

`takeDropS (suc n) s i .tails =`

`takeDropS n (s .tails) i`

Exercise. Prove that bisimilarity of streams implies path equality.

Next time: coinduction case studies

- The delay monad
- Stream processors
- Formal languages
- Wander types
- Coinductive graphs (?)