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

Curry-Howard for propositional logic

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

Propositional logic		**Type system**
conjunction	$P \times Q$	pair type
disjunction	$P \uplus Q$	either type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	empty type

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:

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

However, Agda uses a constructive logic: a proof of $A \lor B$ 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

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:

$$\mathsf{toStream} : \\ (xs : \mathsf{Colist}\ A) \to \mathsf{Infinite}\ xs \to \mathsf{Stream}\ A$$

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?

$$\text{finite-or-infinite} : (xs : \text{Colist}\ A) \rightarrow$$
$$\quad \text{Finite}\ xs \uplus \text{Infinite}\ xs$$

Finite or infinite?

Question. Can we prove the following?

$$\begin{aligned}&\mathsf{finite\text{-}or\text{-}infinite} : (xs : \mathsf{Colist}\ A) \rightarrow \\ &\quad \mathsf{Finite}\ xs \uplus \mathsf{Infinite}\ xs\end{aligned}$$

Answer. No, but we can prove this instead:

$$\begin{aligned}&\mathsf{infinite\text{-}not\text{-}finite} : \mathsf{Infinite}\ xs \rightarrow \neg\,(\mathsf{Finite}\ xs) \\ &\mathsf{not\text{-}finite\text{-}infinite} : \neg\,(\mathsf{Finite}\ xs) \rightarrow \mathsf{Infinite}\ xs\end{aligned}$$

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

Outline

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 $\lambda\ b \to (b \equiv \mathsf{true})$?

1. $\mathsf{Bool} \to \mathsf{Bool}$
2. $\mathsf{Bool} \to \mathsf{Set}$
3. $(b : \mathsf{Bool}) \to b \equiv \mathsf{true}$
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:

$$\begin{aligned}&\mathrm{test}_1 : \mathrm{length}\ (42 :: [\,]) \equiv 1\\&\mathrm{test}_1 = \mathrm{refl}\end{aligned}$$

$$\begin{aligned}&\mathrm{test}_2 : \mathrm{length}\ (\mathrm{map}\ (1 +\_)\ (0 :: 1 :: 2 :: [\,])) \equiv 3\\&\mathrm{test}_2 = \mathrm{refl}\end{aligned}$$

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.

$$
\begin{aligned}
&\mathsf{not\text{-}not} : (b : \mathsf{Bool}) \to \mathsf{not}\,(\mathsf{not}\,b) \equiv b \\
&\mathsf{not\text{-}not}\ \mathsf{true}\ \ = \mathsf{refl} \\
&\mathsf{not\text{-}not}\ \mathsf{false} = \mathsf{refl}
\end{aligned}
$$

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

$$\mathsf{sym} : x \equiv y \to y \equiv x$$
$$\mathsf{sym}\ \mathsf{refl} = \mathsf{refl}$$

$$\mathsf{trans} : x \equiv y \to y \equiv z \to x \equiv z$$
$$\mathsf{trans}\ \mathsf{refl}\ \mathsf{refl} = \mathsf{refl}$$

$$\mathsf{cong} : (f : A \to B) \to x \equiv y \to f\, x \equiv f\, y$$
$$\mathsf{cong}\ f\ \mathsf{refl} = \mathsf{refl}$$

From lists to colists and back

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

$$\begin{aligned}&\mathsf{fromListInv} : (xs : \mathsf{List}\ A)\\ &\quad \to \mathsf{toList}\ (\mathsf{fromList}\ xs)\ (\mathsf{fromListFin}\ xs) \equiv xs\end{aligned}$$

Can we prove the same about fromStream?

Outline

Equational reasoning

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

$$\_\equiv\langle\_\rangle\_ : (x : A) \to x \equiv y \to y \equiv z \to x \equiv z$$
$$x \equiv\langle\ \mathsf{refl}\ \rangle\ q = q$$

$$\_\equiv\langle\rangle\_ : (x : A) \to x \equiv y \to x \equiv y$$
$$x \equiv\langle\rangle\ q = x \equiv\langle\ \mathsf{refl}\ \rangle\ q$$

$$\_\blacksquare : (x : A) \to x \equiv x$$
$$x\ \blacksquare = \mathsf{refl}$$

Equational reasoning example

$$
\begin{aligned}
&\text{reverse} : \text{List } A \to \text{List } A\\
&\text{reverse } [\,] \qquad\qquad\; = [\,]\\
&\text{reverse } (x :: xs) = \text{reverse } xs +\!\!+ (x :: [\,])
\end{aligned}
$$

$$
\begin{aligned}
&\text{reverse-singleton} : \text{reverse } (x :: [\,]) \equiv x :: [\,]\\
&\text{reverse-singleton } \{x = x\} =\\
&\quad \text{reverse } (x :: [\,]) \qquad\quad\; \equiv\langle\rangle\\
&\quad \text{reverse } [\,] +\!\!+ (x :: [\,]) \equiv\langle\rangle\\
&\quad [\,] +\!\!+ (x :: [\,]) \qquad\qquad\;\; \equiv\langle\rangle\\
&\quad (x :: [\,]) \qquad\qquad\qquad\quad\; \blacksquare
\end{aligned}
$$

Equational reasoning + induction

$$
\begin{array}{l}
\text{add-n-zero} : (n : \mathbb{N}) \to n + \text{zero} \equiv n \\
\text{add-n-zero zero} \qquad = \text{refl} \\
\text{add-n-zero} (\text{suc } n) = \\
\quad (\text{suc } n) + \text{zero} \equiv\langle\rangle \\
\quad \text{suc } (n + \text{zero}) \equiv\langle \text{ cong suc } (\text{add-n-zero } n) \rangle \\
\quad \text{suc } n \qquad\qquad \blacksquare
\end{array}
$$

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

Exercise. Prove that *xs* ++ [] = *xs*.

Exercise. Prove associativity of ++.

Example 1: functor laws for `List`

The first functor law for lists:

$$
\begin{array}{l}
\text{map-id} : \{A : \text{Set}\}\ (xs : \text{List}\ A) \to \text{map id}\ xs \equiv xs \\
\text{map-id}\ []\qquad\quad = \text{refl} \\
\text{map-id}\ (x :: xs) = \\
\quad \begin{array}{ll}
\text{map id}\ (x :: xs) & \equiv\langle\rangle \\
\text{id}\ x :: \text{map id}\ xs & \equiv\langle\rangle \\
x :: \text{map id}\ xs & \equiv\langle\ \text{cong}\ (x ::\_)\ (\text{map-id}\ xs)\ \rangle \\
x :: xs & \blacksquare
\end{array}
\end{array}
$$

Exercise

Prove the second functor law for List.

First, we need to define function composition:[1]

$$
\begin{aligned}
&\_\circ\_ : \{A\ B\ C : \mathsf{Set}\} \to \\
&\quad (B \to C) \to (A \to B) \to (A \to C) \\
&f \circ g = \lambda\ x \to f\ (g\ x)
\end{aligned}
$$

Now we can prove that
$\mathsf{map}\ (f \circ g)\ x = (\mathsf{map}\ f \circ \mathsf{map}\ g)\ x$.

[1] Unicode input for ∘: `\circ`

Example 2: verifying optimizations

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

$$
\begin{aligned}
&\mathsf{reverse\text{-}acc} : \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A\\
&\mathsf{reverse\text{-}acc}\ []\ ys = ys\\
&\mathsf{reverse\text{-}acc}\ (x :: xs)\ ys = \mathsf{reverse\text{-}acc}\ xs\ (x :: ys)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reverse'} : \mathsf{List}\ A \to \mathsf{List}\ A\\
&\mathsf{reverse'}\ xs = \mathsf{reverse\text{-}acc}\ xs\ []
\end{aligned}
$$

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

Proving an equality between streams

Let's prove an equality on streams:

$$\begin{aligned}&\mathsf{takeDrop} : (n : \mathbb{N})\ (s : \mathsf{Stream}\ A)\\ &\qquad\qquad \to \mathsf{take}\ n\ s\ \mathsf{++}\ \mathsf{drop}\ n\ s \equiv s\\ &\mathsf{takeDrop}\ \mathsf{zero}\ s = \mathsf{refl}\\ &\mathsf{takeDrop}\ (\mathsf{suc}\ n)\ s = \text{\{! cong (s .head ::S\_) ? !\}}\end{aligned}$$

Error: s .head :: _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
    head : s1 .head ≡ s2 .head
    tail   : s1 .tail   ~  s2 .tail
open _~_ public
```

Proving bisimulation of streams

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

```
refl~ : (s : Stream A) → s ~ s
refl~ s .head = refl
refl~ s .tail = refl~ (s .tail)

takeDrop : (n : ℕ) (s : Stream A)
           → (take n s ++ drop n s) ~ s
takeDrop zero     s = refl~ s
takeDrop (suc n) s .head = refl
takeDrop (suc n) s .tail  = takeDrop n (s .tail)
```

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:

$$
\begin{aligned}
&\mathsf{fromStreamInv} : (xs : \mathsf{Stream}\ A) \\
&\quad \to \mathsf{toStream}\ (\mathsf{fromStream}\ xs)\ (\mathsf{fromStreamInf}\ xs) \\
&\qquad \sim xs
\end{aligned}
$$

Outline

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 i0 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 $f : I \to A$ such that f i0 = x and f i1 = y.

Properties of the path type

$$\mathsf{reflP} : \{x : A\} \to \mathsf{Path}\ A\ x\ x$$
$$\mathsf{reflP}\ \{x = x\}\ i = x$$

$$\mathsf{congP} : (f : A \to B) \to \mathsf{Path}\ A\ x\ y \to \mathsf{Path}\ B\ (f\ x)\ (f\ y)$$
$$\mathsf{congP}\ f\ p\ i = f\ (p\ i)$$

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

In fact, we can prove that $\mathsf{Path}\ A\ x\ y$ 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:

$$\begin{aligned}
\mathsf{funExt} : \{f\ g : A \to B\} \\
&\to ((x : A) \to \mathsf{Path}\ B\ (f\ x)\ (g\ x)) \\
&\to \mathsf{Path}\ (A \to B)\ f\ g
\end{aligned}$$

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

Cubical bisimulation

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

$$
\begin{aligned}
&\mathsf{takeDrop} : (n : \mathbb{N})\ (s : \mathsf{Stream}\ A) \\
&\quad \to \mathsf{Path}\ (\mathsf{Stream}\ A)\ (\mathsf{take}\ n\ s \mathbin{+\!\!+} \mathsf{drop}\ n\ s)\ s \\
&\mathsf{takeDrop}\ \mathsf{zero} \qquad s\ i = s \\
&\mathsf{takeDrop}\ (\mathsf{suc}\ n)\ s\ i\ .\mathsf{head} = s\ .\mathsf{head} \\
&\mathsf{takeDrop}\ (\mathsf{suc}\ n)\ s\ i\ .\mathsf{tail} \quad = \\
&\quad \mathsf{takeDrop}\ n\ (s\ .\mathsf{tail})\ i
\end{aligned}
$$

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 (?)