Coinductive Programming and Proving in Agda

Lecture 1: Coinductive programming in Agda

Jesper Cockx

21 January 2026

Technical University Delft

Lecture plan

1. Introduction
2. Data types and (co)pattern matching
3. Universes and polymorphism
4. Dependent types
5. Coinductive record types
6. Mixing induction and coinduction
7. Sized types

Outline

Why dependent types?

Dependent types embed formal verification directly inside your type system.

Advantages of dependent types.

- Single syntax for programming and proving
- Editor support for interactive development
- Invariants can be embedded inside programs (= *intrinsic* verification)

Verifying a program should not be more difficult than writing it in the first place!

The Agda language

Agda is a purely functional programming language similar to Haskell.

Unlike Haskell, it has full support for dependent types.

It also supports interactive programming with help from the type checker.

Installing Agda

1. **Agda binary.** Download the binary[1], then run `agda --setup`.
2. **Editor plugin.** Install the VS Code plugin or run `agda --emacs-mode setup`.
3. **Standard library.** Download and unpack[2], then add it to `libraries` and `defaults`.

Detailed instructions:
agda.readthedocs.io/en/v2.8.0/getting-started

[1] github.com/agda/agda/releases/tag/v2.8.0
[2] github.com/agda/agda-stdlib/archive/v2.3.tar.gz

Basic syntax

Names can be any non-reserved sequence of unicode[3] characters, except `. ; { } ( ) "`.

A **type declaration** is written as $b\ :\ A$.

Function types are written $A \to B$ or $(x : A) \to B$. $\lambda\, x \to u$ is an (anonymous) function and $f\ x$ is function application.

An **infix operator** `_+_` is used as `x + y`. `x+y` is a valid name, so use enough spaces.

[3] Supported editors will replace LaTeX-like syntax (e.g. `\to`) with unicode.

Loading an Agda file

You can load an Agda file by pressing `Ctrl+c` followed by `Ctrl+l`.

Once the file is loaded (and there are no errors), other commands become available:

`Ctrl+c Ctrl+d`	Infer type of an expression.
`Ctrl+c Ctrl+n`	Evaluate an expression.

Holes in programs

A hole is an incomplete part of a program. You can create one by writing `?` or `{!!}` and loading the file.

New commands for holes:

`Ctrl+c Ctrl+,`	Get hole information
`Ctrl+c Ctrl+c`	Case split on a variable
`Ctrl+c Ctrl+space`	Fill in the hole

Outline

Declaring new datatypes

```
data Bool : Set where
  true  : Bool
  false : Bool

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}
```

Set is the type of (small) types (see later).

Defining functions by pattern matching

$$\begin{aligned}
&\mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}\\
&\mathsf{not}\ \mathsf{true}\ \ = \mathsf{false}\\
&\mathsf{not}\ \mathsf{false} = \mathsf{true}
\end{aligned}$$

$$\begin{aligned}
&_+_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\
&\mathsf{zero} \qquad + y = y\\
&(\mathsf{suc}\ x) + y = \mathsf{suc}\ (x + y)
\end{aligned}$$

Exercise. Define isEven, _*_, and $\_\leq\_$ on $\mathbb{N}$.

Total functional programming

Agda is a total language: evaluating a function call always returns a result in finite time:

- The **coverage checker** ensures completeness of pattern matches.
- The **termination checker** ensures termination of recursive definitions.

Reasons to care about totality:

- It prevents crashes and infinite loops.
- It is needed for **decidable type checking**.
- It is needed for **logical soundness**.

Record types and projections

```
record Rect : Set where
  constructor rect -- optional
  field
    height : ℕ
    width : ℕ

square : ℕ → Rect
square x = record { height = x ; width = x }
  -- or: rect x x

area : Rect → ℕ
area r = Rect.height r + Rect.width r
```

More record syntax

You can open the record to bring projections into scope:

```
open Rect

perimeter : Rect → ℕ
perimeter r = 2 * height r + 2 * width r
```

You can also use projections as postfix:

```
rotate : Rect → Rect
rotate r = rect (r .width) (r .height)
```

Copattern matching

Where data types are defined by how we construct an element, record types are defined by what we can observe of an element.

This duality is exploited by Agda's copattern syntax:

```
squeeze : Rect → Rect
squeeze r .height = half (r .height)
squeeze r .width  = 2 * r .width
```

We define squeeze r by its projections.

Outline

The type `Set`

In Agda, types such as $\mathbb{N}$ and Bool $\rightarrow$ Bool are themselves expressions of type Set.

We can pass around and return values of type Set just like values of any other type.

Example.

$$\mathsf{id} : (A : \mathsf{Set}) \rightarrow A \rightarrow A$$
$$\mathsf{id}\ A\ x = x$$

A type like Set whose elements are themselves types is called a universe.

Side note: the `Set` hierarchy

Set itself does not have type Set: assuming so leads to inconsistency.[4]

Instead, $\mathsf{Set} = \mathsf{Set}_0$ has type Set_1, which has type Set_2, which has type…

In fact, you can write universe-polymorphic definitions by quantifying over $l : \mathsf{Level}$ and working with universe $\mathsf{Set}\ l$.

[4] See Girard's paradox and Hurken's paradox.

Polymorphic functions in Agda

We can use Set to define polymorphic functions and data types:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

length : {A : Set} → List A → ℕ
length [] = 0
length (_ :: xs) = suc (length xs)
```

The curly braces mark A as implicit.

Exercise. Implement map and _++_.

Variable generalization

We can mark declare a variable to be generalized automatically:

$$\text{variable } A\ B\ C : \mathsf{Set}$$

$$\mathsf{id} : A \to A$$
$$\mathsf{id}\ x = x$$

This is equivalent to writing $\{A : \mathsf{Set}\} \to \ldots$

Polymorphic record types

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
open _×_ public

swap : A × B → B × A
swap (x , y) = y , x
```

If/then/else as a function

We can define if/then/else in Agda as follows:

```
if_then_else_ : Bool → A → A → A
if true  then x else y = x
if false then x else y = y
```

This is an example of a mixfix operator.

Example usage.

```
test : ℕ → ℕ
test x = if (x ≤ 9000) then 0 else 42
```

The empty type and absurd patterns

```
data ⊥ : Set where

absurd : ⊥ → A
absurd ()
```

The absurd pattern `()` indicates that there are no possible constructors.

The disjoint sum type

The Either type from Haskell is called $\_\uplus\_$:

$$
\begin{aligned}
&\mathsf{data}\ \_\uplus\_\ (A\ B : \mathsf{Set}) : \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{inj}_1 : A \to A \uplus B\\
&\quad \mathsf{inj}_2 : B \to A \uplus B
\end{aligned}
$$

$$
\begin{aligned}
&[\_,\_] : (A \to C) \to (B \to C) \to A \uplus B \to C\\
&[\, f , g \,]\ (\mathsf{inj}_1\ x) = f\ x\\
&[\, f , g \,]\ (\mathsf{inj}_2\ y) = g\ y
\end{aligned}
$$

Outline

Dependent types

A dependent type is a family of types, depending on a term of a base type:

```
data Flavour : Set where
  cheesy chocolatey : Flavour

data Food : Flavour → Set where
  pizza  : Food cheesy
  cake   : Food chocolatey
  bread : {f : Flavour} → Food f

amountOfCheese : Food cheesy → ℕ
amountOfCheese pizza  = 100
amountOfCheese bread = 20
```

Agda knows that cake is not a valid input!

Vectors: lists that know their length

Vec *A* *n* is the type of vectors of length *n*:

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A 0
  _::_ : A → Vec A n → Vec A (suc n)

myVec : Vec ℕ 4
myVec = 1 :: 2 :: 3 :: 4 :: []
```

Types will be normalized during type checking:

```
myVec' : Vec ℕ (2 + 2)
myVec' = myVec
```

Side note: Parameters vs. indices

The argument $(A : Set)$ in the definition of Vec is a parameter, and has to be *the same in the type of each constructor*.

The argument of type $\mathbb{N}$ in the definition of Vec is an index, and must be *determined individually for each constructor*.

Quiz question

Question. How many elements are there in the type Vec Bool 3?

Quiz question

Question. How many elements are there in the type Vec Bool 3?

Answer. 8 elements:

- true :: true :: true :: []
- true :: true :: false :: []
- true :: false :: true :: []
- true :: false :: false :: []
- false :: true :: true :: []
- false :: true :: false :: []
- false :: false :: true :: []
- false :: false :: false :: []

Dependent function types

A dependent function type is a type of the form $(x : A) \rightarrow B\ x$ where the *type* of the output depends on the *value* of the input.

Example.

$$
\begin{aligned}
&\mathsf{replicate} : (n : \mathbb{N}) \rightarrow A \rightarrow \mathsf{Vec}\ A\ n \\
&\mathsf{replicate}\ \mathsf{zero}\qquad x = [] \\
&\mathsf{replicate}\ (\mathsf{suc}\ n)\ x = x :: \mathsf{replicate}\ n\ x
\end{aligned}
$$

E.g. $\mathsf{replicate}\ 3\ 0$ has type $\mathsf{Vec}\ \mathbb{N}\ 3$ and evaluates to $0 :: 0 :: 0 :: []$.

Dependent pattern matching

We can pattern match on Vec just like on List:

$$\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ B\ n \\
&\mathsf{map}\ f\ [] \qquad\quad\ \ = [] \\
&\mathsf{map}\ f\ (x :: xs) = f\ x :: \mathsf{map}\ f\ xs
\end{aligned}$$

$$\begin{aligned}
&\mathsf{head} : \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A \\
&\mathsf{head}\ (x :: xs) = x
\end{aligned}$$

$$\begin{aligned}
&\mathsf{tail} : \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to \mathsf{Vec}\ A\ n \\
&\mathsf{tail}\ (x :: xs) = xs
\end{aligned}$$

In head and tail, the cases for [] are impossible!

A safe lookup

To define a total lookup on vectors, we need the type Fin n:

```
data Fin : ℕ → Set where
  zero : Fin (suc n)
  suc  : Fin n → Fin (suc n)

lookupVec : Vec A n → Fin n → A
lookupVec (x :: xs) zero    = x
lookupVec (x :: xs) (suc i) = lookupVec xs i
```

Again, there is no case for the empty vector!

The family of `Fin` types

$$
\begin{tikzcd}
\mathsf{Fin}\ 0 & \mathsf{Fin}\ 1 & \mathsf{Fin}\ 2 & \mathsf{Fin}\ 3 & \mathsf{Fin}\ 4 & \cdots \\
 & \mathsf{zero} & \mathsf{zero} & \mathsf{zero} & \mathsf{zero} & \cdots \\
 & & \mathsf{suc} \arrow[ul] & \mathsf{suc} \arrow[ul] & \mathsf{suc} \arrow[ul] & \cdots \\
 & & & \mathsf{suc} \arrow[ul] & \mathsf{suc} \arrow[ul] & \cdots \\
 & & & & \mathsf{suc} \arrow[ul] & \cdots
\end{tikzcd}
$$

Some more vector functions

Exercise. Implement the following functions:

$$\mathsf{zipVec} : \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ B\ n \to \mathsf{Vec}\ (A \times B)\ n$$
$$\mathsf{updateVecAt} : \mathsf{Fin}\ n \to A \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ n$$

Outline

Streams as a coinductive record

A stream consists of a head and a tail (= another stream):

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail   : Stream A
open Stream public
```

The coinductive keyword indicates that we want the largest such type.

Functions on streams

We can use the projections to define functions on streams:

$$\mathsf{firstTwo} : \mathsf{Stream}\ A \to A \times A$$
$$\mathsf{firstTwo}\ s = s\ .\mathsf{head}\ ,\ s\ .\mathsf{tail}\ .\mathsf{head}$$

$$\mathsf{drop} : \mathbb{N} \to \mathsf{Stream}\ A \to \mathsf{Stream}\ A$$
$$\mathsf{drop}\ \mathsf{zero}\quad s = s$$
$$\mathsf{drop}\ (\mathsf{suc}\ n)\ s = \mathsf{drop}\ n\ (s\ .\mathsf{tail})$$

Defining a new stream

Defining a new stream with a record constructor fails the termination check:

```
zeroes : Stream ℕ
zeroes = record { head = 0 ; tail = zeroes }
```

Termination checking failed for zeroes

Allowing this would violate strong normalization of Agda!

Defining a new stream

To define a new stream, we have to use copatterns instead:

```
zeroes : Stream ℕ
zeroes .head = 0
zeroes .tail    = zeroes
```

zeroes only reduces when projections are applied to it, thus preserving strong normalization.

The guardedness criterion

Coinductive values should be productive: applying any (finite) number of projections to them should terminate.

This is enforced by the guardedness criterion: [5] every (co)recursive call needs to appear

1. in a clause with a copattern, *and*
2. either at the top level or as the argument to one or more constructors

[5] Enabled by `{-# OPTION --guardedness #-}`

Limitations of guardedness

$$\text{map} : (A \to B) \to \text{Stream}\ A \to \text{Stream}\ B$$
$$\text{map}\ f\ xs\ .\text{head} = f\ (xs\ .\text{head})$$
$$\text{map}\ f\ xs\ .\text{tail} \quad = \text{map}\ f\ (xs\ .\text{tail})$$

$$\text{nats} : \text{Stream}\ \mathbb{N}$$
$$\text{nats}\ .\text{head} = 0$$
$$\text{nats}\ .\text{tail} = \text{map suc nats}$$

Termination checking failed for nats

Problem. The guardedness checker does not know anything about mapS!

Working around the limitations

```
natsFrom : ℕ → Stream ℕ
natsFrom n .head = n
natsFrom n .tail = natsFrom (suc n)

nats : Stream ℕ
nats = natsFrom 0
```

Alternatively, we can use sized types to provide Agda with more information about mapS (see later).

Exercises

- Define $\mathsf{repeat} : A \to \mathsf{Stream}\ A$.
- Define $\mathsf{lookup} : \mathsf{Stream}\ A \to \mathbb{N} \to A$.
- Define $\mathsf{tabulate} : (\mathbb{N} \to A) \to \mathsf{Stream}\ A$.
- Use tabulate to define $\mathsf{fibonacci} : \mathsf{Stream}\ \mathbb{N}$.
- Define $\mathsf{transpose} : \mathsf{Stream}\ (\mathsf{Stream}\ A) \to \mathsf{Stream}\ (\mathsf{Stream}\ A)$.
- **(Bonus)** Prove that $\mathsf{lookup}\ (\mathsf{lookup}\ (\mathsf{transpose}\ xss)\ i)\ j \equiv \mathsf{lookup}\ (\mathsf{lookup}\ xss\ j)\ i$.

Outline

Colists: potentially infinite lists

```
mutual
  data Colist (A : Set) : Set where
    []   : Colist A
    _::_ : A → Colist' A → Colist A

  record Colist' (A : Set) : Set where
    coinductive
    field
      force : Colist A
open Colist' public
```

Converting a stream to a colist

```
fromStream : Stream A → Colist A
fromStream {A} xs = xs .head :: rest
  where
    rest : Colist' A
    rest .force = fromStream (xs .tail)
```

Alternatively, we use a copattern lambda:

```
fromStream' : Stream A → Colist A
fromStream' {A} xs = xs .head ::
  (λ where .force → fromStream' (xs .tail))
```

Exercise. Define fromList : List A → Colist A.

Another example: co-natural numbers

```
mutual
  data CoℕN : Set where
    zero : Coℕ
    suc  : Coℕ' → Coℕ

  record Coℕ' : Set where
    coinductive
    field force : Coℕ
open Coℕ' public
```

Exercise. Define ∞ : Coℕ, fromℕ : ℕ → Coℕ, and colength : Colist A → Coℕ.

Outline

Sized types

Sized types[6] are an alternative to the syntactic guardedness checker that annotates the size of expressions in their types.

The module `Size` provides:

- $\mathsf{Size} : \mathsf{Set}$
- $\mathsf{Size}\!< : \mathsf{Size} \to \mathsf{Set}$
- $\infty : \mathsf{Size}$.

+ some operators not relevant for coinduction.

[6] Enabled by `{-# OPTION --sized-types #-}`

Sized streams

We can parametrize a stream by its size i
(= the number of elements we can observe):

```
record Stream (A : Set) (i : Size) : Set where
  coinductive
  field
    head : A
    tail  : {j : Size< i} → Stream A j
open Stream
```

The tail can be assigned any size j that is strictly smaller than i.

The infinite size

When *consuming* a stream we can use size ∞:

$$
\begin{aligned}
&\mathsf{take} : \mathbb{N} \to \mathsf{Stream}\ A\ \infty \to \mathsf{List}\ A \\
&\mathsf{take}\ \mathsf{zero}\ s \qquad\quad = [] \\
&\mathsf{take}\ (\mathsf{suc}\ n)\ s = s\ .\mathsf{head} :: \mathsf{take}\ n\ (s\ .\mathsf{tail})
\end{aligned}
$$

Exercise. Define dropS.

Defining sized streams

When *defining* a new stream we should define it at an arbitrary size i:

$$\mathsf{zeroes} : \mathsf{Stream}\ \mathbb{N}\ i$$
$$\mathsf{zeroes}\ .\mathsf{head} = 0$$
$$\mathsf{zeroes}\ .\mathsf{tail} \quad = \mathsf{zeroes}$$

More explicitly:

$$\mathsf{zeroes'} : \{i : \mathsf{Size}\} \to \mathsf{Stream}\ \mathbb{N}\ i$$
$$\mathsf{zeroes'}\ \{i\}\ .\mathsf{head} \quad = 0$$
$$\mathsf{zeroes'}\ \{i\}\ .\mathsf{tail}\ \{j\} = \mathsf{zeroes}\ \{j\}$$

A size-preserving map

We can now give a more precise type to map:

$$
\begin{aligned}
&\mathsf{map} : (A \to B) \to \mathsf{Stream}\ A\ i \to \mathsf{Stream}\ B\ i \\
&\mathsf{map}\ f\ s\ .\mathsf{head} = f\ (s\ .\mathsf{head}) \\
&\mathsf{map}\ f\ s\ .\mathsf{tail} \quad = \mathsf{map}\ f\ (s\ .\mathsf{tail})
\end{aligned}
$$

This allows a direct definition of nats:

$$
\begin{aligned}
&\mathsf{nats} : \mathsf{Stream}\ \mathbb{N}\ i \\
&\mathsf{nats}\ .\mathsf{head} = 0 \\
&\mathsf{nats}\ .\mathsf{tail} \quad = \mathsf{map}\ \mathsf{suc}\ \mathsf{nats}
\end{aligned}
$$

Exercise. Define zipWithS and use it to define fibonacci without using tabulate.

Next time

- The Curry-Howard correspondence
- Equational reasoning
- Bisimulation as a coinductive relation
- Bisimulation as the cubical path type