Coinductive Programming and Proving in Agda

Lecture 3: Coinduction case studies

Jesper Cockx

21 January 2026

Technical University Delft

Outline

Lecture plan

1. The delay monad
2. Stream processors
3. Formal languages

The delay monad

The delay monad embeds potentially non-terminating computations in Agda:

```
mutual
  data Delay (A : Set) : Set where
    now  : A → Delay A
    later : Delay' A → Delay A

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

Delayed values

Classically, any x : Delay A is either never or laters k a for some $k : \mathbb{N}$ and $a : A$:

never : Delay A
never = later (λ where .force $\rightarrow$ never)

laters : $\mathbb{N} \rightarrow A \rightarrow$ Delay A
laters zero a = now a
laters (suc n) a = later
  (λ where .force $\rightarrow$ laters n a)

Exercise. Implement
iter : $(A \rightarrow A \uplus B) \rightarrow A \rightarrow$ Delay B.

Implementing bind for the delay monad

```
_»=_ : Delay A → (A → Delay B) → Delay B
now x »= f = f x
later d »= f = later λ where
  .force → d .force »= f
```

Convergence of delayed values

```
data _⇓_ {A} : Delay A → A → Set where
  now : (x : A) → now x ⇓ x
  later : d .force ⇓ x → later d ⇓ x

⇓-unique : d ⇓ x → d ⇓ y → x ≡ y
⇓-unique (now x) (now y) = refl
⇓-unique (later p) (later q) = ⇓-unique p q
```

Bisimulation of delayed values

```
mutual
  data _~_ {A} : Delay A → Delay A → Set where
    now  : (x : A) → now  x ~ now  x
    later : x ~' y  → later x ~ later y

  record _~'_ (x y : Delay' A) : Set where
    coinductive
    field
      force : x .force ~ y .force
open _~'_ public
```

Monad laws for Delay

```
refl~ : (x : Delay A) → x ~ x
refl~ (now x) = now x
refl~ (later x) = later λ where
  .force → refl~ (x .force)

now-»= : (x : A) (f : A → Delay B)
       → now x »= f ~ f x
now-»= x f = refl~ (f x)
```

Exercise. State and prove the second and third monad laws »=-now and »=-assoc.

Weak bisimilarity

Since we don't really care about the number of laters, we can make more things bisimilar:

```
mutual
  data _~D_ {A} : Delay A → Delay A → Set where
    value : d₁ ⇓ x → d₂ ⇓ x → d₁ ~D d₂
    later  : d₁ ~D' d₂ → later d₁ ~D later d₂

  record _~D'_ (x y : Delay' A) : Set where
    coinductive
    field force : x .force ~D y .force
open _~D'_ public
```

The partiality monad

Instead of making more elements bisimilar, we can quotient the Delay monad by x later x.

This leads to the definition of the partiality monad by Altenkirch, Danielsson & Kraus (FoSSaC 2017) as a *quotient inductive-inductive type* (QIIT).

Outline

Stream processors

A stream processor describes how to transform a stream of *A*s into a stream of *B*s:

```
mutual
  data SP (A B : Set) : Set where
    get : (A → SP A B) → SP A B
    put : B → SP' A B  → SP A B

  record SP' (A B : Set) : Set where
    coinductive
    field force : SP A B
open SP'
```

There can only be a finite number of gets before there must be a put.

Example stream processor: summing elements pairwise

```
sum2by2 : SP ℕ ℕ
sum2by2 =
  get λ x →
  get λ y →
  put (x + y)
  λ where .force → sum2by2
```

Running a stream processor

```
run : SP A B → Stream A → Stream B
run (get f) xs = run (f (xs .head)) (xs .tail)
run (put y sp) xs .head = y
run (put y sp) xs .tail    = run (sp .force) xs

sum2by2-nats :
  take 5 (run sum2by2 nats)
  ≡ (1 ∷ 5 ∷ 9 ∷ 13 ∷ 17 ∷ [])
sum2by2-nats = refl
```

A slightly more interesting example

Question. What does the stream processor below do?

```
mutual
  sums : SP ℕ ℕ
  sums = get λ n → sumN n 0

  sumN : ℕ → ℕ → SP ℕ ℕ
  sumN zero a = put a λ where .force → sums
  sumN (suc n) a = get λ k → sumN n (a + k)
```

Let's run it on nats!

Composing stream processors

If we have a SP A B and a SP B C, we can apply them in sequence to a Stream A to get a Stream C.

Exercise. Do the same with a single processor:

$$\mathsf{compose} : \mathsf{SP}\ A\ B \to \mathsf{SP}\ B\ C \to \mathsf{SP}\ A\ C$$

$$\begin{aligned}&\mathsf{compose\text{-}correct} :\\ &\quad (p1 : \mathsf{SP}\ A\ B)\ (p2 : \mathsf{SP}\ B\ C)\ (s : \mathsf{Stream}\ A) \to\\ &\quad \mathsf{run}\ (\mathsf{compose}\ p1\ p2)\ s \sim \mathsf{run}\ p2\ (\mathsf{run}\ p1\ s)\end{aligned}$$

Outline

Formal languages, coinductively

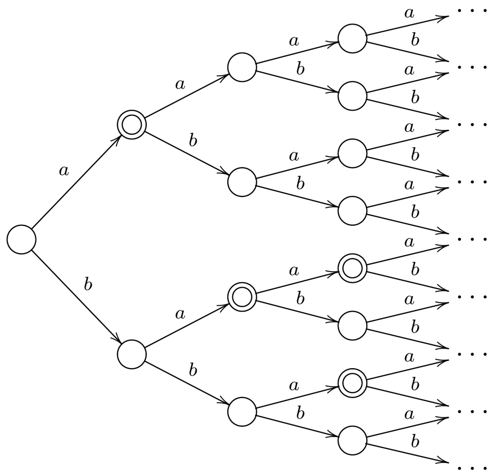
We can describe a formal language l (= a set of strings) over an alphabet A with two pieces of data:

- whether it is nullable
  (= contains the empty string)
- for each $a \in A$, the derivative
  $\delta_a(l) = \{s \mid a \cdot s \in l\}$.

Note that this is a *coinductive* description of formal languages.

Formal languages as (infinite) tries

We can visualize a language as an infinite trie:



Coinductive formal languages in Agda

```
module FormalLanguages
  (A : Set) (_≟_ : DecidableEquality A) where

  record Lang : Set where
    coinductive
    field
      ν : Bool
      δ : A → Lang
  open Lang public
```

Some simple languages

$$\emptyset : \mathsf{Lang}$$
$$\emptyset\ .\nu = \mathsf{false}$$
$$\emptyset\ .\delta = \lambda\ \_ \to \emptyset$$

$$\varepsilon : \mathsf{Lang}$$
$$\varepsilon\ .\nu = \mathsf{true}$$
$$\varepsilon\ .\delta = \lambda\ \_ \to \emptyset$$

$$\mathsf{char} : A \to \mathsf{Lang}$$
$$\mathsf{char}\ a\ .\nu = \mathsf{false}$$
$$\mathsf{char}\ a\ .\delta\ b = \mathsf{if}\ \mathsf{does}\ (a \stackrel{?}{=} b)\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \emptyset$$

Language membership and tabulation

$$\_\ni\_ : \mathsf{Lang} \to \mathsf{List}\ A \to \mathsf{Bool}$$
$$l \ni [] = l\ .\nu$$
$$l \ni (x :: xs) = l\ .\delta\ x \ni xs$$

$$\mathsf{trie} : (\mathsf{List}\ A \to \mathsf{Bool}) \to \mathsf{Lang}$$
$$\mathsf{trie}\ f\ .\nu = f\ []$$
$$\mathsf{trie}\ f\ .\delta\ a = \mathsf{trie}\ (f \circ (a ::\_))$$

Operations on languages

$$
\begin{aligned}
&\mathrm{complement} : \mathrm{Lang} \to \mathrm{Lang} \\
&\mathrm{complement}\ l\ .\nu \quad = \mathrm{not}\ (l\ .\nu) \\
&\mathrm{complement}\ l\ .\delta\ x = \mathrm{complement}\ (l\ .\delta\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\_\cup\_ : \mathrm{Lang} \to \mathrm{Lang} \to \mathrm{Lang} \\
&(l_1 \cup l_2)\ .\nu \quad = l_1\ .\nu \vee l_2\ .\nu \\
&(l_1 \cup l_2)\ .\delta\ x = l_1\ .\delta\ x \cup l_2\ .\delta\ x
\end{aligned}
$$

$$
\begin{aligned}
&\_\cap\_ : \mathrm{Lang} \to \mathrm{Lang} \to \mathrm{Lang} \\
&(l_1 \cap l_2)\ .\nu \quad = l_1\ .\nu \wedge l_2\ .\nu \\
&(l_1 \cap l_2)\ .\delta\ x = l_1\ .\delta\ x \cap l_2\ .\delta\ x
\end{aligned}
$$

Language concatenation

We run into a problem when defining concatenation of languages:

$$
\begin{aligned}
&\_\cdot\_ : \mathsf{Lang} \to \mathsf{Lang} \to \mathsf{Lang} \\
&(l_1 \cdot l_2)\ .\nu \quad = l_1\ .\nu \land l_2\ .\nu \\
&(l_1 \cdot l_2)\ .\delta\ x = (\mathbf{if}\ l_1\ .\nu\ \mathbf{then}\ l_2\ \mathbf{else}\ \emptyset) \cup (l_1\ .\delta\ x \cdot l_2)
\end{aligned}
$$

Error: Termination checking failed for $\_\cdot\_$.
Problematic calls: $l_1\ .\delta\ x\ \cdot\ l_2$

The guardedness is obscured by the call to $\cup$.

Sized types to the rescue

```
record Lang (i : Size) : Set where
  coinductive
  field
    ν : Bool
    δ : {j : Size< i} → A → Lang j
open Lang public
```

Language concatenation with sizes

We can define union to be size-preserving:

$$\_\cup\_ : \mathsf{Lang}\ i \to \mathsf{Lang}\ i \to \mathsf{Lang}\ i$$
$$(l_1 \cup l_2)\ .\nu \quad = l_1\ .\nu \lor l_2\ .\nu$$
$$(l_1 \cup l_2)\ .\delta\ x = l_1\ .\delta\ x \cup l_2\ .\delta\ x$$

This allows the definition of concatenation to pass:

$$\_\cdot\_ : \mathsf{Lang}\ i \to \mathsf{Lang}\ i \to \mathsf{Lang}\ i$$
$$(l_1 \cdot l_2)\ .\nu \quad = l_1\ .\nu \land l_2\ .\nu$$
$$(l_1 \cdot l_2)\ .\delta\ x = (\mathsf{if}\ l_1\ .\nu\ \mathsf{then}\ l_2\ \mathsf{else}\ \emptyset) \cup (l_1\ .\delta\ x \cdot l_2)$$

Definition of Kleene star

$$\_^{*} : \mathsf{Lang}\ i \to \mathsf{Lang}\ i$$
$$(l\ {*})\ .\nu \quad = \mathsf{true}$$
$$(l\ {*})\ .\delta\ x = l\ .\delta\ x \cdot (l\ {*})$$

Arden's rule

Arden's rule states: for a non-nullable language k, if $l = (k \cdot l) \cup m$, then $l = (k\ *) \cdot m$

Question. How do we state this rule in Agda?

[1] or path equality in cubical Agda.

Arden's rule

Arden's rule states: for a non-nullable language k, if $l = (k \cdot l) \cup m$, then $l = (k\,*) \cdot m$

Question. How do we state this rule in Agda?

Answer. Using bisimulation![1]

[1]or path equality in cubical Agda.

Bisimulation of languages

```
record _~⟨_⟩~_
  (l₁ : Lang ∞) (i : Size) (l₂ : Lang ∞) : Set where
  coinductive
  field
    ν : l₁ .ν ≡ l₂ .ν
    δ : {j : Size< i} (x : A) → l₁ .δ x ~⟨ j ⟩~ l₂ .δ x
open _~⟨_⟩~_
```

Arden's rule in Agda

Now we can state Arden's rule:

$$\mathsf{arden} : (k\ l\ m : \mathsf{Lang}\ \infty) \to$$
$$l \sim\!\langle\ \infty\ \rangle\!\sim (k \cdot l) \cup m \to l \sim\!\langle\ \infty\ \rangle\!\sim (k\ {*}) \cdot m$$

For the full proof, see *Equational Reasoning about Formal Languages in Coalgebraic Style* by Andreas Abel (2016).

References

- Abel & Pientka (JFP 2016): *Well-founded recursion with copatterns and sized types.*
- Ghani, Hancock & Pattinson (LMC 2009): *Representations of stream processors using nested fixed points.*
- Capretta (LMC 2005): *General Recursion Via Coinductive Types.*
- Altenkirch, Danielsson & Kraus (FoSSaC 2017): *Partiality, Revisited: The Partiality Monad as a Quotient Inductive-Inductive Type.*
- Abel (2016): *Equational Reasoning about Formal Languages in Coalgebraic Style.*
- Capretta (): *Wander types: A formalization of coinduction-recursion*
- Kidney & Wu (POPL 2025): *Formalising Graph Algorithms with Coinduction*