Karlsruhe Institute of Technology

Formale Systeme II: Theorie

(Co)algebraic Data Types

Summer Semester 2024

Prof. Dr. Bernard Beckert · Dr. Romain Pascual · Dr. Mattias Ulbrich

Algebraic data types

Induction

"An algebraic data type is defined by structural induction."

Induction on $\mathbb{N}$

Principle of induction

Let $P(n)$ be a proposition depending on $n \in \mathbb{N}$. If:

- **Base case:** $P(0)$ is true,
- **Inductive step:** For all $k \in \mathbb{N}$, if $P(k)$ is true, then $P(k+1)$ is true,

then $P(n)$ is true for all $n \in \mathbb{N}$.

Induction on $\mathbb{N}$

Principle of induction

Let $P(n)$ be a proposition depending on $n \in \mathbb{N}$. If:

- **Base case:** $P(0)$ is true,
- **Inductive step:** For all $k \in \mathbb{N}$, if $P(k)$ is true, then $P(k+1)$ is true,

then $P(n)$ is true for all $n \in \mathbb{N}$.

$$\forall P\Big(\big[0 \in P \land \forall k \in \mathbb{N} \big(k \in P \Rightarrow (k+1) \in P\big)\big] \iff \mathbb{N} \subseteq P\Big)$$

Induction on $\mathbb{N}$

Principle of induction

Let $P(n)$ be a proposition depending on $n \in \mathbb{N}$. If:

- **Base case:** $P(0)$ is true,
- **Inductive step:** For all $k \in \mathbb{N}$, if $P(k)$ is true, then $P(k+1)$ is true,

then $P(n)$ is true for all $n \in \mathbb{N}$.

$$\forall P\big(\big[0 \in P \land \forall k \in \mathbb{N}\,\big(k \in P \Rightarrow (k+1) \in P\big)\big] \iff \mathbb{N} \subseteq P\big)$$

$$\forall P\,\big(\big[P(0) \land \forall k\,\big(P(k) \Rightarrow P(k+1)\big)\big] \iff \forall n\; P(n)\big)$$

Induction on $\mathbb{N}$

Principle of induction

Let $P(n)$ be a proposition depending on $n \in \mathbb{N}$. If:

- **Base case:** $P(0)$ is true,
- **Inductive step:** For all $k \in \mathbb{N}$, if $P(k)$ is true, then $P(k+1)$ is true,

then $P(n)$ is true for all $n \in \mathbb{N}$.

$$\forall P\Big(\big[0 \in P \land \forall k \in \mathbb{N}\,\big(k \in P \Rightarrow (k+1) \in P\big)\big] \iff \mathbb{N} \subseteq P\Big)$$

$$\forall P\,\Big(\big[P(0) \land \forall k\,\big(P(k) \Rightarrow P(k+1)\big)\big] \iff \forall n\; P(n)\Big)$$

How can this proof method be generalized to other sets/structures?

Well-founded set

Karlsruhe Institute of Technology

Definition

A set S equipped with a binary relation $\preceq \subseteq S^2$ is **well-founded** if every non-empty subset X of S has a minimal element for $\preceq$.

$$\forall X \subseteq S\,(X \neq \varnothing \Rightarrow \exists m \in X, \forall s \in X\,(s \not\prec m))$$

Example: $\mathbb{N}$ with $\leq_{\mathbb{N}}$

Counter-example: $\mathbb{R}$ with $\leq_{\mathbb{R}}$

Mathematical induction

A poset $(S, \preceq)$ admits the **principle of mathematical induction** if for all propositions P on the elements of S, the two following are equivalent

- $\forall s \in S\ P(s)$
- $\forall s \in S\, (\forall s' \in S,\ s' \preceq s \Rightarrow P(s')) \Rightarrow P(s)$

Theorem

$(S, \preceq)$ *admits the principle of mathematical induction if and only if it is well-founded.*

By Zorn's lemma ($\sim$ AC), $\mathbb{R}$ admits the principle of mathematical induction.

Inductive definition

Definition

An **inductive definition** of a subset S of X is given by :

- the **base elements** B that belong to the set,
- the **construction rules** $F_i \colon X^{n_i} \to X$ that generate new elements from those already in the set.

S is then the smallest set such that $B \subseteq S$ and for all F_i and all $(s_1, \ldots, s_{n_i}) \in S^{n_i}$, $F_i(s_1, \ldots, s_{n_i}) \in S$.

In programming, these sets correspond to **algebraic data types**.

Natural numbers

- 0 is a natural number,
- if n is a natural number, then $\texttt{Succ}(n)$ is a natural number.

Lists of type τ

- `Nil` is a list of type τ,
- if t is a list of type τ and h is of type τ, then $\texttt{Cons}(t, h)$ is a list of type τ.

(Binary) trees of type τ

- `Leaf` is a tree of type τ,
- if l and r are trees of type τ and n is of type τ, then `Branch`(n, l, r) is a tree of type τ.

Structural induction

Structural induction allows to prove properties about algebraic data types.

To prove a property P about an algebraic data type S, it suffices to show:

- **Base case:** $P(b)$ holds for all $b \in B$.
- **Inductive steps:** For each construction rule F_i, if $P(s_1), \ldots,$ $P(s_{n_i})$ hold, then $P(F_i(s_1, \ldots, s_{n_i}))$ holds.

Structural induction is deduced from mathematical induction by considering the order $\preceq$ on terms such that

$$t \preceq t' \iff t' = F_i(\ldots, t, \ldots)$$

for some construction rule F_i.

Algebra on data types

"An algebraic data type is obtained by putting together other types via algebraic manipulations."

Everything is an algebraic data type

```
data Bool = True | False;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
data List a = Nil | Cons a (List a);
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
data List a = Nil | Cons a (List a);
data Tree a =
    | Leaf
    | Branch a (Tree a) (Tree a);
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
data List a = Nil | Cons a (List a);
data Tree a =
    | Leaf
    | Branch a (Tree a) (Tree a);
data Maybe a = Nothing | Just a;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
data List a = Nil | Cons a (List a);
data Tree a =
    | Leaf
    | Branch a (Tree a) (Tree a);
data Maybe a = Nothing | Just a;
data Pair a b = Pair a b;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
data List a = Nil | Cons a (List a);
data Tree a =
    | Leaf
    | Branch a (Tree a) (Tree a);
data Maybe a = Nothing | Just a;
data Pair a b = Pair a b;
data Either a b = Left a | Right b;
```

Everything is an algebraic data type

```
data Bool = True | False;
data Season = Winter | Spring | Summer | Fall;
data Nat = Zero | Succ Nat;
data LinExp =
    | Nat
    | Add LinExp LinExp
    | Mul LinExp LinExp;
data List a = Nil | Cons a (List a);
data Tree a =
    | Leaf
    | Branch a (Tree a) (Tree a);
data Maybe a = Nothing | Just a;
data Pair a b = Pair a b;
data Either a b = Left a | Right b;
```

Algebraic manipulations

Algebra deals with **multiplications** and **additions**.

Algebraic manipulations

Algebra deals with **multiplications** and **additions**.

- **Product** type $\tau = \tau_L \times \tau_R$: `Pair`, product in the category of types and functions, $\sim$ Cartesian product of sets.

  For a type `Pair a b` we assume functions

  - `Fst : Pair a b -> a` giving the first element
  - `Snd : Pair a b -> b` giving the second element

Algebraic manipulations

Algebra deals with **multiplications** and **additions**.

- **Product** type $\tau = \tau_L \times \tau_R$: `Pair`, product in the category of types and functions, $\sim$ Cartesian product of sets.

  For a type `Pair a b` we assume functions
  - `Fst : Pair a b -> a` giving the first element
  - `Snd : Pair a b -> b` giving the second element

- **Sum** type $\tau = \tau_L + \tau_R$: `Either`, coproduct in the category of types and functions, $\sim$ disjoint union of sets.

Algebraic manipulations

Algebra deals with **multiplications** and **additions**.

- **Product** type $\tau = \tau_L \times \tau_R$: `Pair`, product in the category of types and functions, $\sim$ Cartesian product of sets.

  For a type `Pair a b` we assume functions

  - `Fst : Pair a b -> a` giving the first element
  - `Snd : Pair a b -> b` giving the second element

- **Sum** type $\tau = \tau_L + \tau_R$: `Either`, coproduct in the category of types and functions, $\sim$ disjoint union of sets.

- **Unit** type **unit**: `Unit`, terminal object in the category of types and functions, $\sim$ singleton set.

Algebraic manipulations

Algebra deals with **multiplications** and **additions**.

- **Product** type $\tau = \tau_L \times \tau_R$: `Pair`, product in the category of types and functions, $\sim$ Cartesian product of sets.

  For a type `Pair a b` we assume functions

  - `Fst : Pair a b -> a` giving the first element
  - `Snd : Pair a b -> b` giving the second element

- **Sum** type $\tau = \tau_L + \tau_R$: `Either`, coproduct in the category of types and functions, $\sim$ disjoint union of sets.

- **Unit** type **unit**: `Unit`, terminal object in the category of types and functions, $\sim$ singleton set.

- **Zero** type **void**: `Void`, initial object in the category of types and functions, $\sim$ empty set.

Commutativity

For two types τ, τ',

$$\tau \times \tau' = \tau' \times \tau \text{ ?}$$

$$\tau + \tau' = \tau' + \tau \text{ ?}$$

Commutativity

For two types τ, τ',

$$\tau \times \tau' = \tau' \times \tau \ ? \ ✗$$

$$\tau + \tau' = \tau' + \tau \ ? \ ✗$$

Commutativity

For two types τ, τ',

$$\tau \times \tau' = \tau' \times \tau \text{ ? } ✗$$

$$\tau \times \tau' \simeq \tau' \times \tau \text{ ?}$$

$$\tau + \tau' = \tau' + \tau \text{ ? } ✗$$

$$\tau + \tau' \simeq \tau' + \tau \text{ ?}$$

Commutativity

For two types τ, τ',

$$\tau \times \tau' = \tau' \times \tau \text{ ? } ✗$$

$$\tau \times \tau' \simeq \tau' \times \tau \text{ ? } ✓$$

$$\tau + \tau' = \tau' + \tau \text{ ? } ✗$$

$$\tau + \tau' \simeq \tau' + \tau \text{ ? } ✓$$

Commutativity: isomorphisms

For $\tau \times \tau' \simeq \tau' \times \tau$, the isomorphism is given by

```
fun swap : (Pair a b -> Pair b a) =
    p -> Pair (Snd p) (Fst p)
```

For $\tau + \tau' \simeq \tau' + \tau$, the isomorphism is given by

```
fun flip : (Either a b -> Either b a) =
    e -> match e with
          | Left a -> Right a
          | Right b -> Left b
```

Other properties

For any types τ, τ_1, τ_2, and τ_3

Neutral elements

- $\tau \times \textbf{unit} \simeq \tau \simeq \textbf{unit} \times \tau$
- $\tau + \textbf{void} \simeq \tau \simeq \textbf{void} + \tau$

Associativity

- $\tau_1 \times (\tau_2 \times \tau_3) \simeq (\tau_1 \times \tau_2) \times \tau_3$
- $\tau_1 + (\tau_2 + \tau_3) \simeq (\tau_1 + \tau_2) + \tau_3$

Distributivity

- $\tau_1 \times (\tau_2 + \tau_3) \simeq (\tau_1 \times \tau_2) + (\tau_1 \times \tau_3)$

Absorption

- $\tau \times \textbf{void} \simeq \textbf{void} \simeq \textbf{void} \times \tau$

Questions

1. What is the algebraic structure associated with the types?

Questions

1. What is the algebraic structure associated with the types?
A (commutative) semiring.

Questions

1. What is the algebraic structure associated with the types?
   A (commutative) semiring.

2. What is the type 2?

Questions

1. What is the algebraic structure associated with the types?
A (commutative) semiring.

2. What is the type 2?
$2 = 1 + 1$, so **unit** $+$ **unit** $\simeq$ *Bool*.

Questions

1. What is the algebraic structure associated with the types?
A (commutative) semiring.

2. What is the type 2?
$2 = 1 + 1$, so **unit** + **unit** $\simeq$ *Bool*.

3. What is the type associated with

```
data Season = Winter | Spring | Summer | Fall;
```

Questions

1. What is the algebraic structure associated with the types?
A (commutative) semiring.

2. What is the type 2?
$2 = 1 + 1$, so **unit** $+$ **unit** $\simeq$ *Bool*.

3. What is the type associated with

```
data Season = Winter | Spring | Summer | Fall;
```

4.

Questions

1. What is the algebraic structure associated with the types?
A (commutative) semiring.

2. What is the type 2?
$2 = 1 + 1$, so **unit** + **unit** $\simeq$ *Bool*.

3. What is the type associated with

```
data Season = Winter | Spring | Summer | Fall;
```

4.

4. What is the type $1 + a$?

Questions

1. What is the algebraic structure associated with the types?
A (commutative) semiring.

2. What is the type 2?
$2 = 1 + 1$, so **unit** + **unit** $\simeq$ *Bool*.

3. What is the type associated with
```
data Season = Winter | Spring | Summer | Fall;
```
4.

4. What is the type $1 + a$?
```
data Maybe a = Nothing | Just a;.
```

Algebra is about solving equations

Let I be a function such that $I(a) = 1 + a \times I(a)$.

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Let us forget about types for now and do some simple math:

$$l(a) = 1 + a \times l(a)$$

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Let us forget about types for now and do some simple math:

$$
\begin{aligned}
l(a) &= 1 + a \times l(a) \\
l(a) - a \times l(a) &= 1
\end{aligned}
$$

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Let us forget about types for now and do some simple math:

$$\begin{aligned} l(a) &= 1 + a \times l(a) \\ l(a) - a \times l(a) &= 1 \\ l(a) \times (1 - a) &= 1 \end{aligned}$$

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Let us forget about types for now and do some simple math:

$$
\begin{aligned}
l(a) &= 1 + a \times l(a) \\
l(a) - a \times l(a) &= 1 \\
l(a) \times (1 - a) &= 1 \\
l(a) &= \frac{1}{1 - a}
\end{aligned}
$$

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Let us forget about types for now and do some simple math:

$$\begin{aligned} l(a) &= 1 + a \times l(a) \\ l(a) - a \times l(a) &= 1 \\ l(a) \times (1 - a) &= 1 \\ l(a) &= \frac{1}{1 - a} \end{aligned}$$

But, we do not have subtraction or division!

Algebra is about solving equations

Let l be a function such that $l(a) = 1 + a \times l(a)$.

What is the associated type?

Let us forget about types for now and do some simple math:

$$
\begin{aligned}
l(a) &= 1 + a \times l(a) \\
l(a) - a \times l(a) &= 1 \\
l(a) \times (1 - a) &= 1 \\
l(a) &= \frac{1}{1-a}
\end{aligned}
$$

But, we do not have subtraction or division! $l(a) = \sum_{n=0}^{\infty} a^n$

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$I(a) = 1 + a \times I(a)$$

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$\begin{aligned} I(a) &= 1 + a \times I(a) \\ &= 1 + a \times (1 + a \times I(a)) \end{aligned}$$

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$\begin{aligned} I(a) &= 1 + a \times I(a) \\ &= 1 + a \times (1 + a \times I(a)) \\ &= 1 + a + a \times a \times I(a) \end{aligned}$$

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$\begin{aligned} I(a) &= 1 + a \times I(a) \\ &= 1 + a \times (1 + a \times I(a)) \\ &= 1 + a + a \times a \times I(a) \\ &= 1 + a + a \times a \times (1 + a \times I(a)) \end{aligned}$$

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$\begin{aligned} I(a) &= 1 + a \times I(a) \\ &= 1 + a \times (1 + a \times I(a)) \\ &= 1 + a + a \times a \times I(a) \\ &= 1 + a + a \times a \times (1 + a \times I(a)) \\ &= 1 + a + a^2 + a^3 \times I(a) \end{aligned}$$

Solving equations (continued)

How can we interpret $I(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$
\begin{aligned}
I(a) &= 1 + a \times I(a) \\
&= 1 + a \times (1 + a \times I(a)) \\
&= 1 + a + a \times a \times I(a) \\
&= 1 + a + a \times a \times (1 + a \times I(a)) \\
&= 1 + a + a^2 + a^3 \times I(a) \\
&= 1 + a + a^2 + a^3 + a^4 + \ldots
\end{aligned}
$$

Solving equations (continued)

How can we interpret $l(a) = \sum_{n=0}^{\infty} a^n$?

Generating functions and **formal power series**

$$
\begin{aligned}
l(a) &= 1 + a \times l(a) \\
&= 1 + a \times (1 + a \times l(a)) \\
&= 1 + a + a \times a \times l(a) \\
&= 1 + a + a \times a \times (1 + a \times l(a)) \\
&= 1 + a + a^2 + a^3 \times l(a) \\
&= 1 + a + a^2 + a^3 + a^4 + \dots
\end{aligned}
$$

```
data List a = Nil | Cons a (List a)
```

(Initial) Algebras

"An algebraic data type is described by a functor."

Algebra

Algebras are sets with operations.

Example: $(\mathbb{N}, 0, \texttt{Succ})$, with $0 \in \mathbb{N}$ and $\texttt{Succ} : \mathbb{N} \to \mathbb{N}$.

Equivalently,

$$\begin{array}{c} 1 + \mathbb{N} \\ \Big\downarrow {\scriptstyle [\texttt{Zero},\texttt{Succ}]} \\ \mathbb{N} \end{array}$$

where $1 = \{\varnothing\}$ and $\texttt{Zero}(\varnothing) = 0$.

Lists

$(\mathtt{List}(A), [], \mathtt{Cons})$, with

- $[] \in \mathtt{List}(A)$,
- $\mathtt{Cons} : A \times \mathtt{List}(A) \to \mathtt{List}(A)$.

$$\begin{array}{c} 1 + A \times \mathtt{List}(A) \\ \Big\downarrow {\scriptstyle [\mathtt{Nil}, \mathtt{Cons}]} \\ \mathtt{List}(A) \end{array}$$

where $1 = \{\varnothing\}$ and $\mathtt{Nil}(\varnothing) = []$.

Trees

$(\texttt{Tree}(A), [], \texttt{Branch})$, with

- $[] \in \texttt{Tree}(A)$,
- $\texttt{Branch} : A \times \texttt{Tree}(A) \times \texttt{Tree}(A) \to \texttt{Tree}(A)$.

$$\begin{array}{c} 1 + A \times \texttt{Tree}(A) \times \texttt{Tree}(A) \\ \Big\downarrow {\scriptstyle [\texttt{Leaf}, \texttt{Branch}]} \\ \texttt{Tree}(A) \end{array}$$

where $1 = \{\varnothing\}$ and $\texttt{Leaf}(\varnothing) = []$.

Algebras, categorically

For a functor $F\colon \mathcal{C} \to \mathcal{C}$, an F-algebra is a pair (X, α) with

$$\begin{array}{c} F(X) \\ \Big\downarrow{\scriptstyle \alpha} \\ X \end{array}$$

We call F the **type** and α the **structure map** of (X, α).

The structure map α tells us how the elements of X are constructed from other elements in X.

Examples

- $F\colon \mathbf{Set} \to \mathbf{Set};\; X \mapsto 1 + X$ gives $\mathbb{N}$
- $F\colon \mathbf{Set} \to \mathbf{Set};\; X \mapsto 1 + A \times X$ gives $\mathtt{List}(A)$
- $F\colon \mathbf{Set} \to \mathbf{Set};\; X \mapsto 1 + A \times X \times X$ gives $\mathtt{Tree}(A)$

Algebra momorphisms

A morphism of F-algebras is an arrow $f\colon (X,\alpha) \to (Y,\beta)$ such that

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \alpha\big\downarrow & & \big\downarrow\beta \\ X & \xrightarrow[f]{} & Y \end{array}$$

Think functoriality!

Initial algebra

The natural numbers are an **initial** algebra.

- Inductive definitions are based on the existence of $h \colon \mathbb{N} \to A$.
- Inductive proofs are based on the uniqueness of $h \colon \mathbb{N} \to A$.

Coalgebraic Data Types

Motivation

Induction and coinduction

Induction corresponds to

- initiality of an algebra
- least fixed point of a monotone function

Induction and coinduction

Induction corresponds to

- initiality of an algebra
- least fixed point of a monotone function

Coinduction corresponds to

- terminality (also called finality) of an algebra
- greatest fixed point of a monotone function

Constructing lists

```
data List a = Nil | Cons a (List a)
```

Abstracted into $L = \mathbf{1} + A \times L$

Constructors

- `Nil` $\simeq$ `Unit`
- `Cons : Pair a (List a) -> List a`

Constructing lists

```
data List a = Nil | Cons a (List a)
```

Abstracted into $L = \mathbf{1} + A \times L$

Constructors

- `Nil` $\simeq$ `Unit`, equivalent to `Nil` $: \mathbf{1} \to L$
- `Cons : Pair a (List a) -> List a`, equivalent to `Cons` $: A \times L \to L$

Rather than equality, we have $\mathbf{1} + A \times L \to L$

Observing the list

What if we want to observe what is in the list?

Then we need to deconstruct the list!

Destructors

- `Head : List a -> a`
- `Tail : List a -> List a`

Observing the list

What if we want to observe what is in the list?

Then we need to deconstruct the list!

Destructors

- `Head : List a -> a`, equivalent to $\texttt{Head} : L \to A$
- `Tail : List a -> List a`,
  equivalent to $\texttt{Tail} : L \to L$

Rather than equality, we have $L \to \mathbf{1} + A \times L$

Safe head and safe tail

What about $L \to \mathbf{1}$?

Safe head and safe tail

What about $L \to \mathbf{1}$?

```
data Maybe a = Nothing | Just a;
```

Safe head and safe tail

What about $L \to \mathbf{1}$?

```
data Maybe a = Nothing | Just a;
```

The function $L \to \mathbf{1} + A \times L$, corresponds to

```
Maybe (Pair Head Tail)
```

The product $A \times L$ means that the head and tail of a sequence are related: they are selected or observed together

Construction and destruction

$$L = \mathbf{1} + A \times L$$

Construction and destruction

$$L \to \mathbf{1} + A \times L$$

- Construction $\mathbf{1} + A \times L \to L$

Construction and destruction

$$L \leftarrow \mathbf{1} + A \times L$$

- Construction $\mathbf{1} + A \times L \to L$
- Destruction $L \to \mathbf{1} + A \times L$

Construction and destruction

Karlsruhe Institute of Technology

$$L = \mathbf{1} + A \times L$$

- Construction $\mathbf{1} + A \times L \to L$
- Destruction $L \to \mathbf{1} + A \times L$

Coalgebras come from algebras by **duality**

Colgebras, categorically

For a functor $F\colon \mathcal{C} \to \mathcal{C}$, an F-coalgebra is a pair (X, α) with

$$\begin{array}{c} X \\ \downarrow{\scriptstyle \alpha} \\ F(X) \end{array}$$

We call F the **type** and α the **structure map** of (X, α).

The structure map α tells us how the elements of X are observed by deconstruction.

Data stream

$(\texttt{Stream}(A), \texttt{Head}, \texttt{Tail})$, with

- $\texttt{Head} : \texttt{Stream}(A) \to A$,
- $\texttt{Tail} : \texttt{Stream}(A) \to \texttt{Stream}(A)$.

$$\texttt{Stream}(A) \xrightarrow{(\texttt{Head},\texttt{Tail})} A \times \texttt{Stream}(A)$$

$F : \textbf{Set} \to \textbf{Set};\ A \times X \mapsto X$ gives $\texttt{Stream}(A)$

Further reading

- Bart Jacobs and Jan Rutten. "A Tutorial on (Co)Algebras and (Co)Induction". In: **EATCS Bulletin** (1997)
- Jan Rutten. "Universal coalgebra: a theory of systems". In: **Theoretical Computer Science** (2000)
- Jan Rutten. **The Method of Coalgebra: exercises in coinduction**. 2019

“[Coalgebra] aims to be the mathematics of computational dynamics” – Bart Jacobs

Coalgebras as state machines

$\texttt{Stream}(A)$ generates values of type A

Coalgebras as state machines

$\texttt{Stream}(A)$ generates values of type A

$(\texttt{Head}, \texttt{Tail})$ corresponds to a function $X \to A \times X$

For $s, s_1 \in X$, and $a \in A$, we write

$$s \xrightarrow{a} s_1 \quad \text{iff} \quad \texttt{Head}(s) = a \text{ and } \texttt{Tail}(s) = s_1$$

Coalgebras as state machines

$\texttt{Stream}(A)$ generates values of type A

$(\texttt{Head}, \texttt{Tail})$ corresponds to a function $X \to A \times X$

For $s, s_1 \in X$, and $a \in A$, we write

$$s \xrightarrow{a} s_1 \quad \text{iff} \quad \texttt{Head}(s) = a \text{ and } \texttt{Tail}(s) = s_1$$

In the state s, we can observe a and move to the state s_1

We observe s:

$$s \xrightarrow{a} s_1$$

We observe s:

$$s \xrightarrow{a} s_1$$

We observe s_1:

$$s \xrightarrow{a} s_1 \xrightarrow{a_1} s_2$$

We observe s:

$$s \xrightarrow{a} s_1$$

We observe s_1:

$$s \xrightarrow{a} s_1 \xrightarrow{a_1} s_2$$

We observe s_2:

$$s \xrightarrow{a} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3$$

We observe s:

$$s \xrightarrow{a} s_1$$

We observe s_1:

$$s \xrightarrow{a} s_1 \xrightarrow{a_1} s_2$$

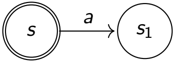
We observe s_2:

$$s \xrightarrow{a} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3$$

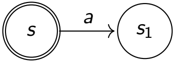
And so on

$$s \xrightarrow{a} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} s_4 \xrightarrow{a_4} \dots$$

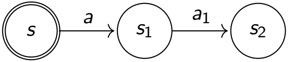
We observe s:



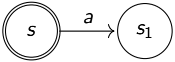
We observe s:



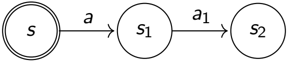
We observe s_1:



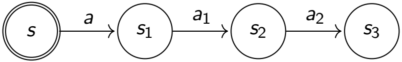
We observe s:



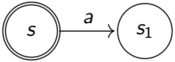
We observe s_1:



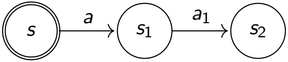
We observe s_2:



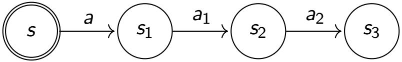
We observe s:



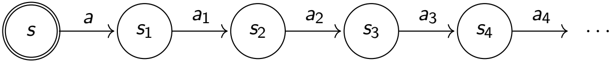
We observe s_1:



We observe s_2:



And so on



Automata?

We can consider

- the sequence $(a, a_1, a_2, a_3, a_4, \ldots)$ as the **trace** of s
- $\{s, s_1, s_2, s_3, s_4, \ldots\}$ as **states**
- $\delta \colon A \times X \to X; (a, s) \mapsto s'$ iff $s \xrightarrow{a} s'$ as a **transition function**

Automata?

We can consider

- the sequence $(a, a_1, a_2, a_3, a_4, \ldots)$ as the **trace** of s
- $\{s, s_1, s_2, s_3, s_4, \ldots\}$ as **states**
- $\delta\colon A \times X \to X; (a, s) \mapsto s'$ iff $s \xrightarrow{a} s'$ as a **transition function**

$$\begin{array}{c} X \\ \alpha \big\downarrow \\ X^A \end{array}$$

Deterministic

Automata?

We can consider

- the sequence $(a, a_1, a_2, a_3, a_4, \ldots)$ as the **trace** of s
- $\{s, s_1, s_2, s_3, s_4, \ldots\}$ as **states**
- $\delta \colon A \times X \to X; (a, s) \mapsto s'$ iff $s \xrightarrow{a} s'$ as a **transition function**

$$X \xrightarrow{\alpha} X^A$$

Deterministic

$$X \xrightarrow{\alpha} \mathscr{P}(X)^A$$

Nondeterministic

Automata?

We can consider

- the sequence $(a, a_1, a_2, a_3, a_4, \dots)$ as the **trace** of s
- $\{s, s_1, s_2, s_3, s_4, \dots\}$ as **states**
- $\delta\colon A \times X \to X; (a, s) \mapsto s'$ iff $s \xrightarrow{a} s'$ as a **transition function**

$$X \xrightarrow{\alpha} X^A$$

Deterministic

$$X \xrightarrow{\alpha} \mathscr{P}(X)^A$$

Nondeterministic

$$X \xrightarrow{\alpha} \mathscr{P}_f(X)^A$$

NDF

Coinductive functions

Coalgebra momorphisms

A morphism of F-coalgebras is an arrow $f\colon (X,\alpha) \to (Y,\beta)$ such that

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ F(X) & \xrightarrow[F(f)]{} & F(Y) \end{array}$$

Think functoriality!

Morphism are maps on the carrier that preserve the dynamics

$$\beta \circ f = \mathcal{F}(f) \circ \alpha$$

$\texttt{Stream}(A)$ as a terminal coalgebra

$\texttt{Stream}(A)$ is a **terminal** coalgebra for $T(X) = A \times X$.

For an arbitrary T-coalgebra U, the unique morphism $f : U \to \texttt{Stream}(A)$ is given by

$$f(u)(n) = \texttt{Head}_U\left(\texttt{Tail}_U^n(u)\right)$$

for all $u \in U$, $n \in \mathbb{N}$.

It satisfies

- $\texttt{Head}_U = \texttt{Head}_{\texttt{Stream}(A)} \circ f$
- $f \circ \texttt{Tail}_U = \texttt{Tail}_{\texttt{Stream}(A)} \circ f$

Uniqueness by induction

$\texttt{Stream}(A)$ as a terminal coalgebra

$\texttt{Stream}(A)$ is a **terminal** coalgebra for $T(X) = A \times X$.

- Coinductive definitions are based on the existence of $h\colon X \to \texttt{Stream}(A)$.
- Coinductive proofs are based on the uniqueness of $h\colon X \to \texttt{Stream}(A)$.

Functions on algebraic data types

An **inductive definition** definition of a function *f* defines values for all constructors.

```
fun Len : (List a -> Nat) =
    l -> match l with
        | Nil -> Zero
        | Cons(a, l') -> Succ (Len l')
```

Functions on coalgebraic data types

A **coinductive definition** definition of a function f defines equations for all destructors.

Functions on coalgebraic data types

A **coinductive definition** definition of a function f defines equations for all destructors.

Let us define a function $\texttt{Odd} : \texttt{Stream}(A) \rightarrow \texttt{Stream}(A)$ that only keeps the elements at odd indices:

Functions on coalgebraic data types

A **coinductive definition** definition of a function f defines equations for all destructors.

Let us define a function $\texttt{Odd} : \texttt{Stream}(A) \to \texttt{Stream}(A)$ that only keeps the elements at odd indices:

$$\begin{cases} \texttt{Head}(\texttt{Odd}(s)) = \texttt{Head}(s) \\ \texttt{Tail}(\texttt{Odd}(s)) = \texttt{Odd}(\texttt{Tail}(\texttt{Tail}(s))) \end{cases}$$

Functions on coalgebraic data types

A **coinductive definition** definition of a function f defines equations for all destructors.

Let us define a function $\texttt{Odd} : \texttt{Stream}(A) \to \texttt{Stream}(A)$ that only keeps the elements at odd indices:

$$\begin{cases} \texttt{Head}(\texttt{Odd}(s)) = \texttt{Head}(s) \\ \texttt{Tail}(\texttt{Odd}(s)) = \texttt{Odd}(\texttt{Tail}(\texttt{Tail}(s))) \end{cases}$$

$\texttt{Odd}$ defines a morphism of T-coalgebras (with $T(X) = A \times X$)

Even and merge

Questions How can we define `Even`?

Even and merge

Questions How can we define `Even`?

$\texttt{Even} = \texttt{Odd} \circ \texttt{Tail}$.

Even and merge

Questions How can we define $\texttt{Even}$?
$\texttt{Even} = \texttt{Odd} \circ \texttt{Tail}$.

$\texttt{Merge} : \texttt{Stream}(A) \times \texttt{Stream}(A) \to \texttt{Stream}(A)$ alternate the elements of the two streams:

Even and merge

Questions How can we define $\texttt{Even}$?
$\texttt{Even} = \texttt{Odd} \circ \texttt{Tail}$.

$\texttt{Merge} : \texttt{Stream}(A) \times \texttt{Stream}(A) \to \texttt{Stream}(A)$ alternate the elements of the two streams:

$$\begin{cases} \texttt{Head}(\texttt{Merge}(s, s')) = \texttt{Head}(s) \\ \texttt{Tail}(\texttt{Merge}(s, s')) = \texttt{Merge}(s', (\texttt{Tail}\, s)) \end{cases}$$

Even and merge

Questions How can we define $\texttt{Even}$?
$\texttt{Even} = \texttt{Odd} \circ \texttt{Tail}.$

$\texttt{Merge} : \texttt{Stream}(A) \times \texttt{Stream}(A) \to \texttt{Stream}(A)$ alternate the elements of the two streams:

$$\begin{cases} \texttt{Head}(\texttt{Merge}(s, s')) = \texttt{Head}(s) \\ \texttt{Tail}(\texttt{Merge}(s, s')) = \texttt{Merge}(s', (\texttt{Tail}\, s)) \end{cases}$$

How do we prove that $\texttt{Merge}(\texttt{Odd}\, s\, \texttt{Even}\, s) = s$ for any stream s?

Fixpoints

Theorem

The operation of an initial (resp. a terminal) algebra is an isomorphism.

If (A, α) is an initial F-algebra, then $\alpha \colon F(A) \to A$ has an inverse $\alpha^{-1} \colon A \to F(A)$.

If (A, α) is a terminal F-algebra, then $\alpha \colon A \to F(A)$ has an inverse $\alpha^{-1} \colon F(A) \to A$.

Proof (case of the initial algebra)

Proof (case of the initial algebra)

1. $(F(A), F(\alpha))$ is an F-algebra

$$\begin{array}{c} F(F(A)) \\ {\scriptstyle F(\alpha)}\big\downarrow \\ F(A) \end{array}$$

Proof (case of the initial algebra)

1. $(F(A), F(\alpha))$ is an F-algebra

$$\begin{array}{c} F(F(A)) \\ {\scriptstyle F(\alpha)}\big\downarrow \\ F(A) \end{array}$$

... But (A, α) is initial!

Proof (case of the initial algebra)

2. By initiality of (A, α), there is a function $a\colon A \to F(A)$ such that the following diagram commutes

$$
\begin{array}{ccc}
F(A) & \xrightarrow{F(a)} & F(F(A)) \\
\alpha \big\downarrow & & \big\downarrow F(\alpha) \\
A & \xrightarrow[a]{} & F(A)
\end{array}
$$

Proof (case of the initial algebra)

2. By initiality of (A, α), there is a function $a\colon A \to F(A)$ such that the following diagram commutes

$$\begin{array}{ccc} F(A) & \xrightarrow{F(a)} & F(F(A)) \\ {\scriptstyle\alpha}\big\downarrow & & \big\downarrow{\scriptstyle F(\alpha)} \\ A & \xrightarrow[a]{} & F(A) \end{array}$$

... But we can compose α and a!

Proof (case of the initial algebra)

3. By composition, $\alpha \circ a$ corresponds to a F-algebra morphism, and the following diagram commutes

$$
\begin{array}{ccccc}
F(A) & \xrightarrow{F(a)} & F(F(A)) & \xrightarrow{F(\alpha)} & F(A) \\
\downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle F(\alpha)} & & \downarrow{\scriptstyle \alpha} \\
A & \xrightarrow[a]{} & F(A) & \xrightarrow[\alpha]{} & A
\end{array}
$$

Proof (case of the initial algebra)

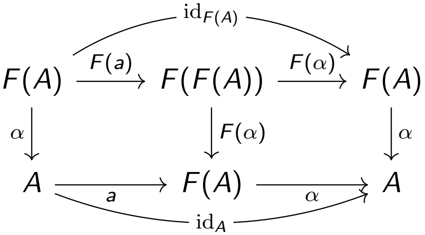
3. By composition, $\alpha \circ a$ corresponds to a F-algebra morphism, and the following diagram commutes

$$
\begin{array}{ccccc}
F(A) & \xrightarrow{F(a)} & F(F(A)) & \xrightarrow{F(\alpha)} & F(A) \\
\downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle F(\alpha)} & & \downarrow{\scriptstyle \alpha} \\
A & \xrightarrow[a]{} & F(A) & \xrightarrow[\alpha]{} & A
\end{array}
$$

... But (A, α) is initial!

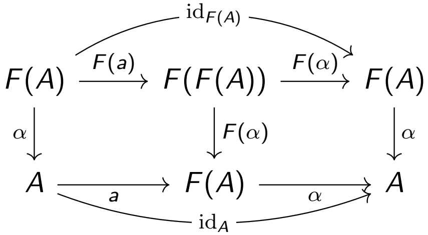
Proof (case of the initial algebra)

4. By initiality, $\alpha \circ a$ is id_A, and the following diagram commutes

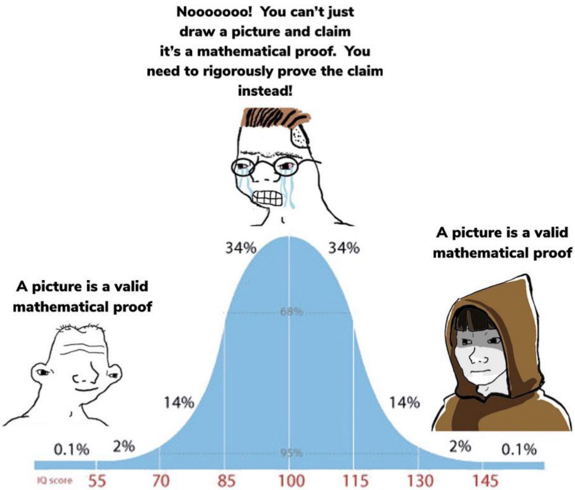


Proof (case of the initial algebra)

4. By initiality, $\alpha \circ a$ is id_A, and the following diagram commutes



Then $a \circ \alpha = F(\alpha) \circ F(a) = F(\alpha \circ a) = F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$,
i.e., $\alpha \colon F(A) \to A$ is an isomorphism with a as its inverse.


Nooooooo! You can't just draw a picture and claim it's a mathematical proof. You need to rigorously prove the claim instead!
A picture is a valid mathematical proof
A picture is a valid mathematical proof
34%
34%
14%
14%
0.1%
2%
2%
0.1%
IQ score
55
70
85
100
115
130
145

Equality?

A coalgebra consists of a carrier set X and a function $\alpha : X \to F(X)$ **out of** X.

Equality?

Karlsruhe Institute of Technology

A coalgebra consists of a carrier set X and a function $\alpha \colon X \to F(X)$ **out of** X.

We do not know how to form elements of X, we only know X through observations, meaning that we have **limited access** to X.

Equality?

A coalgebra consists of a carrier set X and a function $\alpha:$ $X \to F(X)$ **out of** X.

We do not know how to form elements of X, we only know X through observations, meaning that we have **limited access** to X.

Two elements of $\texttt{Stream}(A)$ might be different as elements of $\texttt{Stream}(A)$ while giving rise to the same sequence of elements of A.

Equality?

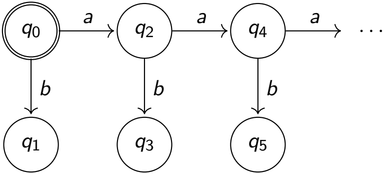
A coalgebra consists of a carrier set X and a function $\alpha: X \to F(X)$ **out of** X.

We do not know how to form elements of X, we only know X through observations, meaning that we have **limited access** to X.

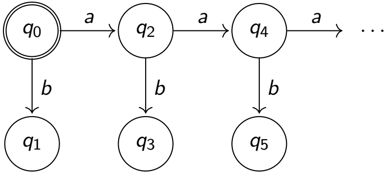
Two elements of $\texttt{Stream}(A)$ might be different as elements of $\texttt{Stream}(A)$ while giving rise to the same sequence of elements of A.

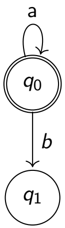
They are **observationally indistinguishable** or **bisimular**.

Bisimulation of automata



Bisimulation of automata





Bisimulation on $\texttt{Stream}(A)$

A **bisimulation** on $\texttt{Stream}(A)$ is a relation $R \subseteq \texttt{Stream}(A) \times \texttt{Stream}(A)$ such that for all $s, s' \in \texttt{Stream}(A)$, $R(s, s')$ implies

- $\texttt{Head}(s) = \texttt{Head}(s')$
- $R(\texttt{Tail}(s), \texttt{Tail}(s'))$

Bisimulation on $\texttt{Stream}(A)$

A **bisimulation** on $\texttt{Stream}(A)$ is a relation $R \subseteq \texttt{Stream}(A) \times \texttt{Stream}(A)$ such that for all $s, s' \in \texttt{Stream}(A)$, $R(s, s')$ implies

- $\texttt{Head}(s) = \texttt{Head}(s')$
- $R(\texttt{Tail}(s), \texttt{Tail}(s'))$

$\texttt{Stream}(A)$ follows the following **coinductive proof principle**: if there is a bisimulation R such that for all $s, s' \in \texttt{Stream}(A)$, $R(s, s')$, then for all $s, s' \in \texttt{Stream}(A)$, $s = s'$.

$\texttt{Merge}(\texttt{Odd}(s), \texttt{Even}(s)) = s$

Consider $R = \{(\texttt{Merge}(\texttt{Odd}(s), \texttt{Even}(s)), s) \mid s \in \texttt{Stream}(A)\}$

$\mathtt{Merge}(\mathtt{Odd}(s), \mathtt{Even}(s)) = s$

Karlsruhe Institute of Technology

Consider $R = \{(\mathtt{Merge}(\mathtt{Odd}(s), \mathtt{Even}(s)), s) \mid s \in \mathtt{Stream}(A)\}$

$$\begin{aligned}
&\mathtt{Head}(\mathtt{Merge}(\mathtt{Odd}(s), \mathtt{Even}(s))) \\
&= \mathtt{Head}(\mathtt{Odd}(s)) \\
&= \mathtt{Head}(s)
\end{aligned}$$

$\texttt{Merge}(\texttt{Odd}(s), \texttt{Even}(s)) = s$

Consider $R = \{(\texttt{Merge}(\texttt{Odd}(s), \texttt{Even}(s)), s) \mid s \in \texttt{Stream}(A)\}$

$$\begin{aligned}
&\texttt{Head}(\texttt{Merge}(\texttt{Odd}(s), \texttt{Even}(s))) \\
&= \texttt{Head}(\texttt{Odd}(s)) \\
&= \texttt{Head}(s)
\end{aligned}$$

$$\begin{aligned}
&\mathit{Tail}(\texttt{Merge}(\texttt{Odd}(s), \texttt{Even}(s))) \\
&= \texttt{Merge}(\texttt{Even}(s), \texttt{Tail}(\texttt{Odd}(s))) \\
&= \texttt{Merge}(\texttt{Odd}(\texttt{Tail}(s)), \texttt{Odd}(\texttt{Tail}(\texttt{Tail}(s)))) \\
&= \texttt{Merge}(\texttt{Odd}(\texttt{Tail}(s)), \texttt{Even}(\texttt{Tail}(s)))
\end{aligned}$$

Bisimulation, formally

Given a functor $F\colon \mathbf{Set} \to \mathbf{Set}$, an F-**bisimulation** between two F-coalgebras (S, α_S) (T, α_T) is an F-coalgebra (R, α_R) such that

- $R \subseteq S \times T$
- the projections $\pi_1\colon R \to S$ and $\pi_2\colon R \to T$ yields F-coalgebra morphisms

$$
\begin{array}{ccccc}
S & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & T \\
\downarrow{\scriptstyle \alpha_S} & & \downarrow{\scriptstyle \alpha_R} & & \downarrow{\scriptstyle \alpha_T} \\
F(S) & \xleftarrow[F(\pi_1)]{} & F(R) & \xrightarrow[F(\pi_2)]{} & F(T)
\end{array}
$$

Coinduction proof principle

If R is a bisimulation between a terminal coalgebra S and itself, then $R \subseteq \{(s, s) \mid s \in S\}$.

Equivalently, For all $s, s' \in S$,

$$R(s, s') \implies s = s'.$$

To prove the equility of two states, if suffices to proove that they are bisimular!