Intrinsically Recursive Coalgebras

Cass Alexandru Henning Urbat Thorsten Wißmann

2026-01-09

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*
- In the context of *total functional programming* (i.e. algorithms proven to terminate), these arise from *recursive coalgebras*

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*
- In the context of *total functional programming* (i.e. algorithms proven to terminate), these arise from *recursive coalgebras*
- However: Lack of criteria for proving "recursivity" of a coalgebra, in particular amenable to formalization in a dependetly typed fp language (s.a. Agda)

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*
- In the context of *total functional programming* (i.e. algorithms proven to terminate), these arise from *recursive coalgebras*
- However: Lack of criteria for proving "recursivity" of a coalgebra, in particular amenable to formalization in a dependetly typed fp language (s.a. Agda)
- This talk:

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*
- In the context of *total functional programming* (i.e. algorithms proven to terminate), these arise from *recursive coalgebras*
- However: Lack of criteria for proving "recursivity" of a coalgebra, in particular amenable to formalization in a dependetly typed fp language (s.a. Agda)
- This talk:
  - Motivation for Divide-and-Conquer Algorithms, categorically

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*
- In the context of *total functional programming* (i.e. algorithms proven to terminate), these arise from *recursive coalgebras*
- However: Lack of criteria for proving "recursivity" of a coalgebra, in particular amenable to formalization in a dependetly typed fp language (s.a. Agda)
- This talk:
  - Motivation for Divide-and-Conquer Algorithms, categorically
  - How proving partial correctness sets the stage for expressing...

Motivation

- Divide-and-Conquer algorithms are captured by the category-theoretical notion of *coalgebra-to-algebra morphisms*
- In the context of *total functional programming* (i.e. algorithms proven to terminate), these arise from *recursive coalgebras*
- However: Lack of criteria for proving “recursivity” of a coalgebra, in particular amenable to formalization in a dependetly typed fp language (s.a. Agda)
- This talk:
  - Motivation for Divide-and-Conquer Algorithms, categorically
  - How proving partial correctness sets the stage for expressing...
  - our novel categorical criterion for termination of such algorithms!

Structure

Divide and Conquer "Divide and Conquer"

- A D&C algorithm can be split into the following steps:

[1] this is a Chekov's gun

Divide and Conquer "Divide and Conquer"

- A D&C algorithm can be split into the following steps:
  - *Divide* input into "smaller"[1]inputs;

[1]this is a Chekov's gun

Divide and Conquer "Divide and Conquer"

- A D&C algorithm can be split into the following steps:
  - *Divide* input into "smaller"[1] inputs;
  - Recursively apply the algorithm to them;

[1] this is a Chekov's gun

Divide and Conquer “Divide and Conquer”

- A D&C algorithm can be split into the following steps:
  - *Divide* input into “smaller”[1]inputs;
  - Recursively apply the algorithm to them;
  - *Combine* to compute the result.

[1]this is a Chekov’s gun

Case Studies: Quick- and Mergesort

- Algorithms can differ in which step does the "heavy lifting"

Case Studies: Quick- and Mergesort

- Algorithms can differ in which step does the “heavy lifting”
- Quicksort: Main logic in the *divide* step: partition elements around the pivot. Combine step: concatenation.

Case Studies: Quick- and Mergesort

- Algorithms can differ in which step does the “heavy lifting”
- Quicksort: Main logic in the *divide* step: partition elements around the pivot. Combine step: concatenation.
- Mergesort: Business end is the *combine* step: zipping two ordered lists into one. Divide step: Splitting the list in half.

D&CAs as Coalgebra-to-Algebra Morphisms

$$\begin{array}{ccc} FI & \xrightarrow{Fh} & FO \\ {\scriptstyle c}\big\uparrow & & \big\downarrow{\scriptstyle a} \\ I & \overset{h}{\dashrightarrow} & O \end{array}$$

D&CAs as Coalgebra-to-Algebra Morphisms

$$\begin{array}{ccc} FI & \xrightarrow{Fh} & FO \\ {\scriptstyle c}\uparrow & & \downarrow{\scriptstyle a} \\ I & \overset{h}{\dashrightarrow} & O \end{array}$$

- *Coalgebra* c : Divide input up into smaller inputs, the distribution of which is given by a functor F;

D&CAs as Coalgebra-to-Algebra Morphisms

$$\begin{array}{ccc} FI & \xrightarrow{Fh} & FO \\ {\scriptstyle c}\uparrow & & \downarrow{\scriptstyle a} \\ I & \overset{h}{\dashrightarrow} & O \end{array}$$

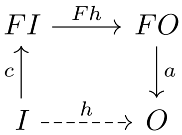
- *Coalgebra* c : Divide input up into smaller inputs, the distribution of which is given by a functor F;
- Fh: Apply h recursively under F;

D&CAs as Coalgebra-to-Algebra Morphisms

$$\begin{array}{ccc} FI & \xrightarrow{Fh} & FO \\ c \uparrow & & \downarrow a \\ I & \overset{h}{\dashrightarrow} & O \end{array}$$

- *Coalgebra* c : Divide input up into smaller inputs, the distribution of which is given by a functor F;
- Fh: Apply h recursively under F;
- *Algebra* a: Combine an F-structure of the results of recursive calls to obtain the output.

D&CAs as Coalgebra-to-Algebra Morphisms



- A coalgebra c is called *recursive* if, for every algebra a, it admits a unique solution to the equation[2]

$$h = c; Fh; a$$

[2] sometimes called the "hylo" equation

D&CAs as Coalgebra-to-Algebra Morphisms

$$
\begin{array}{ccc}
FI & \xrightarrow{Fh} & FO \\
c \uparrow & & \downarrow a \\
I & \overset{h}{\dashrightarrow} & O
\end{array}
$$

- A coalgebra c is called *recursive* if, for every algebra a, it admits a unique solution to the equation[2]

$$h = c; Fh; a$$

- NB: In a language permitting general recursion, the above may be read as a *definition*.

[2] sometimes called the "hylo" equation

Structure

Narrowing our focus

- As mentionend, the *interesting* step of quicksort is the *divide* step, i.e. partitioning a list around a pivot.

Narrowing our focus

- As mentionend, the *interesting* step of quicksort is the *divide* step, i.e. partitioning a list around a pivot.
- We therefore focus on that step from now.

Narrowing our focus

- As mentionend, the *interesting* step of quicksort is the *divide* step, i.e. partitioning a list around a pivot.
- We therefore focus on that step from now.
- partition: $\mathsf{List}A \to 1 + \mathsf{List}A \times A \times \mathsf{List}A \ldots$

Narrowing our focus

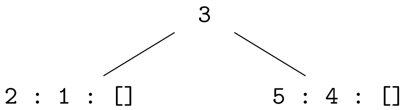
- As mentionend, the *interesting* step of quicksort is the *divide* step, i.e. partitioning a list around a pivot.
- We therefore focus on that step from now.
- partition : $\mathsf{List}A \to 1 + \mathsf{List}A \times A \times \mathsf{List}A$ ...
- ... $\Rightarrow$ Functor F is: $FX = 1 + X \times A \times X$

Example: Growing a BST with partition

`2 : 5 : 4 : 1 : 3 : []`

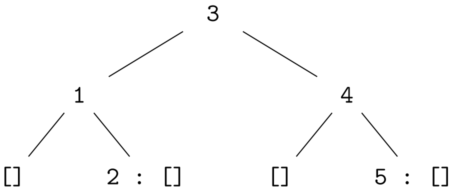
$\mathsf{partition} \colon \mathsf{List}A \to \bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

Example: Growing a BST with partition



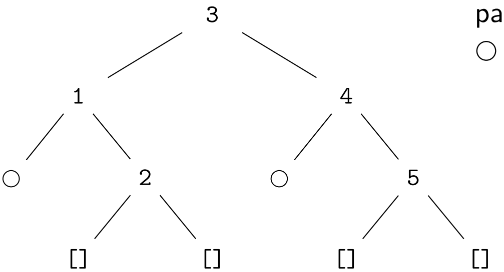
$\mathsf{partition} \colon \mathsf{List} A \to$
$\bigcirc + \mathsf{List} A \times A \times \mathsf{List} A$

Example: Growing a BST with partition



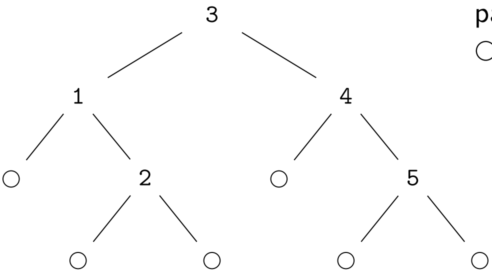
$\mathsf{partition} : \mathsf{List}\, A \to \bigcirc + \mathsf{List}\, A \times A \times \mathsf{List}\, A$

Example: Growing a BST with partition



$\mathsf{partition} : \mathsf{List}A \to \bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

Example: Growing a BST with partition



$\mathsf{partition}\colon \mathsf{List}A \to \bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

Partial Correctness of Quicksort

- Orderedness: The elements to the left/right of the pivot in the List $A \times (p : A) \times$ List A case are smaller/greater than p
- Element-preservation: partition(xs) and xs have the same multiset of elements.
- Working in the setting of data with mappings to the multiset $(\mathcal{M}A)$ of their elements allows us to express both these properties!

Sliced Partition

- Redefine partition for $(X, f\colon X \to \mathcal{M}A)$.
- We define the *Predicate lifting:*
  $-_{\#} : (P : A \to \text{Bool}) \to \mathcal{M}A \to \text{Bool}$
  $P_{\#} xs := \forall x \in xs.\, P(x)$
- We can then lift F to $\mathcal{M}A$-*indexed* sets $(X, f\colon X \to \mathcal{M}A)$ as:

$$\bar{F}\begin{pmatrix} X \\ f \end{pmatrix} := \begin{pmatrix} 1 \\ \emptyset \end{pmatrix} + \begin{pmatrix} \{(l,p,r) \in X \times A \times X \mid f(l) \leq_{\#} p \wedge p >_{\#} f(r)\} \\ f(l) \uplus \{p\} \uplus f(r) \end{pmatrix}$$

- Note: The multiset indices of the recursive positions are smaller than the outer index: $|f(l)|, |f(r)| < |f(l) \uplus \{p\} \uplus f(r)|$.

Structure

How to express "the indices of the recursive positions are smaller than the outer index"

Notation

For $i \in I$, we denote by $<i := \{j \in I \mid j < i\}$ the set of indices strictly smaller than i (the *downset* of i). We have two projection functors: *restriction* and *evaluation*:

$$
\begin{array}{ll}
- \,|_{<i} : \mathcal{C}^I \to \mathcal{C}^{<i} & \mathsf{ev}_i \quad : \mathcal{C}^I \to \mathcal{C} \\
X\,|_{<i} := (X_j)_{j<i} & \mathsf{ev}_i X := X_i \\
f\,\;|_{<i} := (f_j)_{j<i} & \mathsf{ev}_i f \;:= f_i
\end{array}
$$

Introducing: Well Founded Functors

Definition (Well-Founded Functor)

A functor $F\colon \mathcal{C}^I \to \mathcal{C}^I$ is *well-founded* if for every $i \in I$, the functor $\mathsf{ev}_i \cdot F\colon \mathcal{C}^I \to \mathcal{C}$ factors through the projection $|_{<i}\colon \mathcal{C}^I \to \mathcal{C}^{<i}$, that is, there exists a functor $F_{<i}\colon$ such that the diagram below commutes up to natural isomorphism:

$$\forall i \in I\colon \quad \begin{array}{ccc} \mathcal{C}^I & \xrightarrow{\quad F \quad} & \mathcal{C}^I \\ {\scriptstyle -|_{<i}}\big\downarrow & \cong & \big\downarrow{\scriptstyle \mathsf{ev}_i} \\ \mathcal{C}^{<i} & \overset{\exists F_{<i}}{\dashrightarrow} & \mathcal{C} \end{array}$$

Introducing: Well Founded Functors

Definition (Well-Founded Functor)

$$
\forall i \in I\colon \quad
\begin{array}{ccc}
\mathcal{C}^I & \xrightarrow{\;F\;} & \mathcal{C}^I \\
{\scriptstyle -|_{<i}}\big\downarrow & \cong & \big\downarrow{\scriptstyle \mathsf{ev}_i} \\
\mathcal{C}^{<i} & \overset{\exists F_{<i}}{\dashrightarrow} & \mathcal{C}
\end{array}
$$

Introducing: Well Founded Functors

Definition (Well-Founded Functor)

$$\forall i \in I\colon \quad \begin{array}{ccc} \mathcal{C}^I & \xrightarrow{F} & \mathcal{C}^I \\ {\scriptstyle -|_{<i}}\big\downarrow & \cong & \big\downarrow{\scriptstyle \mathsf{ev}_i} \\ \mathcal{C}^{<i} & \xrightarrow{\exists F_{<i}} & \mathcal{C} \end{array}$$

- "The ith output of the functor F is fully determined by its inputs with indices $j < i$."

Introducing: Well Founded Functors

Definition (Well-Founded Functor)

$$\forall i \in I\colon \qquad \begin{array}{ccc} \mathcal{C}^I & \xrightarrow{\;F\;} & \mathcal{C}^I \\ {\scriptstyle -|_{<i}}\big\downarrow & \cong & \big\downarrow{\scriptstyle \mathsf{ev}_i} \\ \mathcal{C}^{<i} & \overset{\exists F_{<i}}{\dashrightarrow} & \mathcal{C} \end{array}$$

- "The ith output of the functor F is fully determined by its inputs with indices $j < i$."
- "$F\colon \mathcal{C}^{I} \to \mathcal{C}^{I}$ is *equivalent* to a family $(F_{<i}\colon \mathcal{C}^{<i} \to \mathcal{C})_{i \in I}$"

Introducing: Well Founded Functors

Definition (Well-Founded Functor)

$$\forall i \in I\colon \quad \begin{array}{ccc} \mathcal{C}^I & \xrightarrow{F} & \mathcal{C}^I \\ {\scriptstyle -|_{<i}}\downarrow & \cong & \downarrow{\scriptstyle \mathsf{ev}_i} \\ \mathcal{C}^{<i} & \overset{\exists F_{<i}}{\dashrightarrow} & \mathcal{C} \end{array}$$

- "The ith output of the functor F is fully determined by its inputs with indices $j < i$."
- "$F\colon \mathcal{C}^{j \in I} \to \mathcal{C}^{i \in I}$ is *equivalent* to a family $(F_{<i}\colon \mathcal{C}^{j<i} \to \mathcal{C})_{i \in I}$"

Minimizing the interface

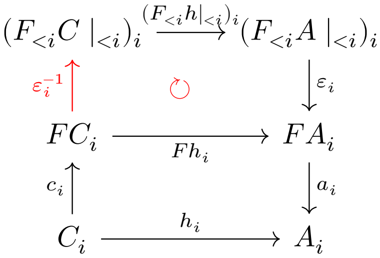
- We define a canonical way to turn any functor F *into* such a family $F_{<i} : (\mathcal{C}^{<i} \to \mathcal{C})_{i \in I}$, for which we obtain a projection $\varepsilon_F X i : F_{<i}(X|_{<i}) i \to F X i$.
- Client code of the library then consists of definining an inclusion $\varepsilon_F^{-1} X i : F X i \to F{<}\ i(X|_{<i}) i$ which is an inverse to this.

Diagrammatically

$$\forall i \in I: \quad \begin{array}{ccc} (F_{<i} C\,|_{<i})_i & \xrightarrow{(F_{<i} h|_{<i})_i} & (F_{<i} A\,|_{<i})_i \\ \varepsilon_i^{-1} \uparrow & \circlearrowright & \downarrow \varepsilon_i \\ FC_i & \xrightarrow[Fh_i]{} & FA_i \\ c_i \uparrow & & \downarrow a_i \\ C_i & \xrightarrow{h_i} & A_i \end{array}$$

Diagrammatically

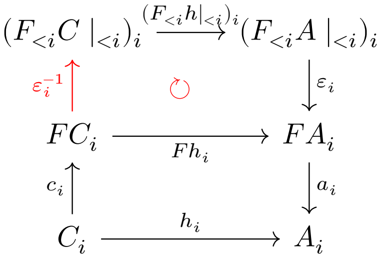
$\forall i \in I$:



- To define h_i, we need only $h\mid_{<i}$ ...

Diagrammatically

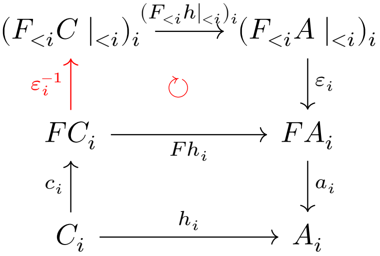
$\forall i \in I$:



- To define h_i, we need only $h \restriction_{<i}$ ...
- We can define $(h_i)_{i \in I}$ by well founded induction!

Diagrammatically

$\forall i \in I$:



- To define h_i, we need only $h \mid_{<i}$ …
- We can define $(h_i)_{i \in I}$ by well founded induction!
- Next: Type-theoretical interface (in Agda).

Wellfoundification

$$\begin{array}{ccc} \mathcal{C}^I & \xrightarrow{F} & \mathcal{C}^I \\ {\scriptstyle -|_{<i}}\downarrow & \cong & \downarrow{\scriptstyle \mathrm{ev}_i} \\ \mathcal{C}^{<i} & \overset{F_{<i}}{\dashrightarrow} & \mathcal{C} \end{array}$$

```
< : A → Type -- : downset
< i = Σ[ j ∈ A ] (j < i)
--restriction
|< : (i : A) → (A → Type) → ((< i) → Type)
|< _i X (j , _pf) = X j
--inclusion (F<i X := F (J< i X) i)
J< : (i : A) → ((< i) → Type) → (A → Type)
J< i X j = Σ[ pf ∈ j < i ] X (j , pf)
--truncation: restriction, then inclusion: |< i;J< i ≈ T
T : (i : A) → (A → Type) → (A → Type)
T i X j = (j < i) × X j -- "annotate with pfs j < i"
--wellfoundification
_↓ : ((A → Type) → (A → Type)) → ((A → Type) → (A → Type))
(F ↓) X i = F (λ j → (j < i) × X j) i -- = F (T i X) i
```

Structure

Going Back to Definition-Time with Inversion

```
data S (X : ℳ A → Type) : ℳ A → Type where
  leaf : S X []
  _|⌈_⌉|_ : {i_l i_r : ℳ A} → (t_l : X i_l) → (x : A) → (t_r : X i_r) →
    x ⊐ i_l → x ⊑ i_r → S X (x :: i_l ++ i_r)
pattern _^_|⌈_⌉|_^_ t_l i_l x t_r i_r p_1 p_2 = _|⌈_⌉|_ {i_l} {i_r} t_l x t_r p_1 

S-ε⁻¹ : {X : ℳ A → Type} → (i : ℳ A) → S X i → (S ↓) X i
S-ε⁻¹ .[] leaf = leaf
S-ε⁻¹ .(x :: i_l ++ i_r) ((t_l ^ i_l |⌈ x ⌉| t_r ^ i_r) p_1 p_2) =
  ((i<x::i++ i_r , t_l) |⌈ x ⌉| (i<x::[ i_l ]++i , t_r)) p_1 p_2
```

Going Back to Definition-Time with Inversion

```
data S (X : 𝓜 A → Type) : 𝓜 A → Type where
  leaf : S X []
  _|⌈_⌉|_ : {i_l i_r : 𝓜 A} → (t_l : X i_l) → (x : A) → (t_r : X i_r) →
    x ⊐ i_l → x ⊑ i_r → S X (x ∷ i_l ++ i_r)
pattern _^_|⌈_⌉|_^_ t_l i_l x t_r i_r p₁ p₂ = _|⌈_⌉|_ {i_l} {i_r} t_l x t_r p₁

S-ε⁻¹ : {X : 𝓜 A → Type} → (i : 𝓜 A) → S X i → (S ↓) X i
S-ε⁻¹ .[] leaf = leaf
S-ε⁻¹ .(x ∷ i_l ++ i_r) ((t_l ^ i_l |⌈ x ⌉| t_r ^ i_r) p₁ p₂) =
  ((i<x::i++ i_r , t_l) |⌈ x ⌉| (i<x::[ i_l ]++i , t_r)) p₁ p₂
```

- Pattern match on the value of type $X\ i$;

Going Back to Definition-Time with Inversion

$\mathsf{data}\ \mathsf{S}\ (X : \mathcal{M}\ A \to \mathsf{Type}) : \mathcal{M}\ A \to \mathsf{Type}\ \ \mathsf{where}$
$\quad \mathsf{leaf} : \mathsf{S}\ X\ []$
$\quad \_|\lceil\_\rceil|\_ : \{i_l\ i_r : \mathcal{M}\ A\} \to (t_l : X\ i_l) \to (x : A) \to (t_r : X\ i_r) \to$
$\quad\quad x \sqsupset i_l \to x \sqsubseteq i_r \to \mathsf{S}\ X\ (x :: i_l +\!\!+\ i_r)$
$\mathsf{pattern}\ \_\hat{}\_|\lceil\_\rceil|\_\hat{}\_\ t_l\ i_l\ x\ t_r\ i_r\ p_1\ p_2 = \_|\lceil\_\rceil|\_\ \{i_l\}\ \{i_r\}\ t_l\ x\ t_r\ p_1$

$\mathsf{S}\text{-}\varepsilon^{-1} : \{X : \mathcal{M}\ A \to \mathsf{Type}\} \to (i : \mathcal{M}\ A) \to \mathsf{S}\ X\ i \to (\mathsf{S} \downarrow)\ X\ i$
$\mathsf{S}\text{-}\varepsilon^{-1}\ .[]\ \mathsf{leaf} = \mathsf{leaf}$
$\mathsf{S}\text{-}\varepsilon^{-1}\ .(x :: i_l +\!\!+\ i_r)\ ((t_l\ \hat{}\ i_l\ |\lceil\ x\ \rceil|\ t_r\ \hat{}\ i_r)\ p_1\ p_2) =$
$\quad ((\text{i<x::i++}\ i_r\ ,\ t_l)\ |\lceil\ x\ \rceil|\ (\text{i<x::[}\ i_l\ \text{]++i}\ ,\ t_r))\ p_1\ p_2$

- Pattern match on the value of type $X\ i$;
- by *inversion* (Dybjer '94), this will refine the original index (seen here as *dot patterns*);

Going Back to Definition-Time with Inversion

$$\textbf{data}\ \mathsf{S}\ (X : \mathcal{M}\,A \to \mathsf{Type}) : \mathcal{M}\,A \to \mathsf{Type}\ \textbf{where}$$
$$\quad \mathsf{leaf} : \mathsf{S}\ X\ []$$
$$\quad \_\|\lceil\_\rceil\|\_ : \{i_l\ i_r : \mathcal{M}\,A\} \to (t_l : X\ i_l) \to (x : A) \to (t_r : X\ i_r) \to$$
$$\quad\quad x \sqsupset i_l \to x \sqsubseteq i_r \to \mathsf{S}\ X\ (x :: i_l \mathbin{++} i_r)$$
$$\textbf{pattern}\ \_\hat{}\_\|\lceil\_\rceil\|\_\hat{}\_\ t_l\ i_l\ x\ t_r\ i_r\ p_1\ p_2 = \_\|\lceil\_\rceil\|\_\ \{i_l\}\ \{i_r\}\ t_l\ x\ t_r\ p_1\ \ldots$$

$$\mathsf{S}\text{-}\varepsilon^{-1} : \{X : \mathcal{M}\,A \to \mathsf{Type}\} \to (i : \mathcal{M}\,A) \to \mathsf{S}\ X\ i \to (\mathsf{S}\downarrow)\ X\ i$$
$$\mathsf{S}\text{-}\varepsilon^{-1}\ .[]\ \mathsf{leaf} = \mathsf{leaf}$$
$$\mathsf{S}\text{-}\varepsilon^{-1}\ .(x :: i_l \mathbin{++} i_r)\ ((t_l\ \hat{}\ i_l\ \|\lceil\ x\ \rceil\|\ t_r\ \hat{}\ i_r)\ p_1\ p_2) =$$
$$\quad ((\mathrm{i{<}x{::}i{+}{+}}\ i_r\ ,\ t_l)\ \|\lceil\ x\ \rceil\|\ (\mathrm{i{<}x{::}[}\ i_l\ \mathrm{]{+}{+}i}\ ,\ t_r))\ p_1\ p_2$$

- Pattern match on the value of type $X\ i$;
- by *inversion* (Dybjer '94), this will refine the original index (seen here as *dot patterns*);
- prove that the indices in the functorial positions are smaller than the original, now refined, outer index.

Structure

Conclusion

- We have a new way to express a broad class of algorithms describable as coalgebra-to-algebra morphisms in a total functional programming language. More specifically, a novel sufficient criterion for proving & formalizing recursivity of coalgebras.

Conclusion

- We have a new way to express a broad class of algorithms describable as coalgebra-to-algebra morphisms in a total functional programming language. More specifically, a novel sufficient criterion for proving & formalizing recursivity of coalgebras.
- Notable use case: Indices already used for proving functional properties intrinsically can also serve as a termination measure.

Conclusion

- We have a new way to express a broad class of algorithms describable as coalgebra-to-algebra morphisms in a total functional programming language. More specifically, a novel sufficient criterion for proving & formalizing recursivity of coalgebras.
- Notable use case: Indices already used for proving functional properties intrinsically can also serve as a termination measure.
- More applications & corollaries in our draft paper (formalized: correct GCD, CYK)

Other Niceties

- Get the recursive coalgebra counterpart of *apomorphisms* for free for the ε^{-1} definition, also of course, *cata* (inverse of the initial algebra is a coalgebra)

Other Niceties

- Get the recursive coalgebra counterpart of *apomorphisms* for free for the ε^{-1} definition, also of course, *cata* (inverse of the initial algebra is a coalgebra)
- Current code development is in an indexed setting but should transfer to applications with slice categories in the object language, if one wants to avoid indexing

Other Niceties

- Get the recursive coalgebra counterpart of *apomorphisms* for free for the ε^{-1} definition, also of course, *cata* (inverse of the initial algebra is a coalgebra)
- Current code development is in an indexed setting but should transfer to applications with slice categories in the object language, if one wants to avoid indexing
- General equational definitions, with the possibility to use facilities for generic programming for the remaining boilerplate

$$a\ i \circ F_1\ (\text{iuncurry}\ i\ IH)\ i \circ Fwf\ \ i \circ c\ i$$

Structure

(Mutual) *Nested* Recursion

```
mutual
  evA : Env → Assgt → Env
  evE : Env → Expr → ℕ

  evA env ( x ↦ expr ) = λ y → case x ≈? y of
    λ{ (yes _) → evE env expr
     ; (no _) → env y }

  evE env (x :+: y) = evE env x + evE env y
  evE env (Var x) = env x
  evE env (Lit n) = n
  evE env (Let assgt In expr) = evE (evA env assgt) expr
```

Contact

- @cxandru@types.pl
- c.alexandru@cs.rptu.de
- @cxandru on Discord
- cxandru.ee