Intrinsically Recursive Coalgebras

Cass Alexandru[1] Henning Urbat[2] Thorsten Wißmann[2]

[1]RPTU Kaiserslautern-Landau & Radboud University Nijmegen

[2]FAU Erlangen-Nürnberg

TYPES 2026

Motivation

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 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:

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”

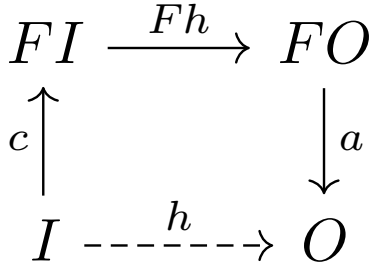
Case Studies: Quick- and Mergesort

- 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.

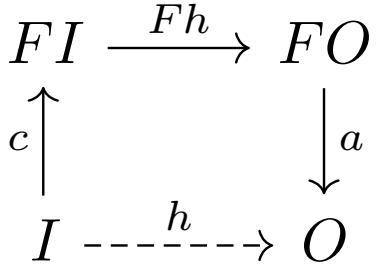
Case Studies: Quick- and Mergesort

- 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

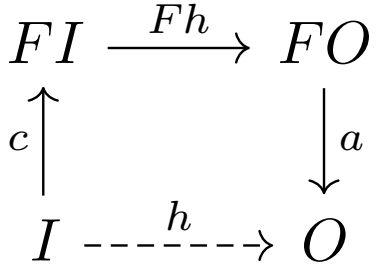


D&CAs as Coalgebra-to-Algebra Morphisms



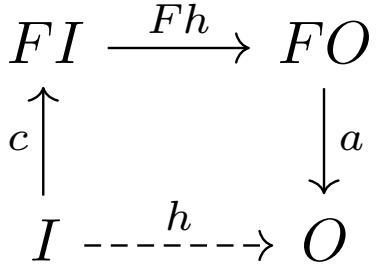
- *Coalgebra* c : Divide input up into smaller inputs, the distribution of which is given by a functor F;

D&CAs as Coalgebra-to-Algebra Morphisms



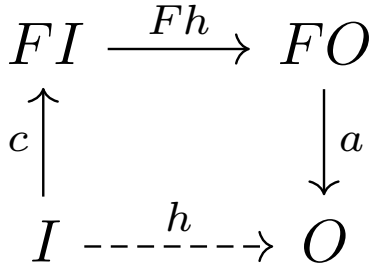
- *Coalgebra* c : Divide input up into smaller inputs, the distribution of which is given by a functor F;
- Fh: Apply h recursively under F;

D&CAs as Coalgebra-to-Algebra Morphisms



- *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

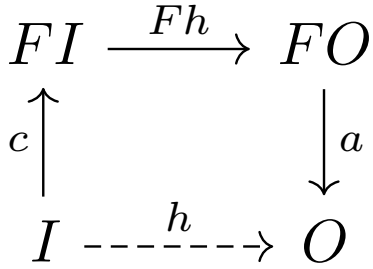


- A coalgebra c is called *recursive* if, for every algebra a, it admits a unique solution to the equation[2]

$$h = c; Fh; a$$

[2] sometimes called the "hylo" or *hylomorphism* equation

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]

$$h = c; Fh; a$$

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

[2] sometimes called the "hylo" or *hylomorphism* 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 : $\mathsf{List}A \to 1 + \mathsf{List}A \times A \times \mathsf{List}A$ ...

Narrowing our focus

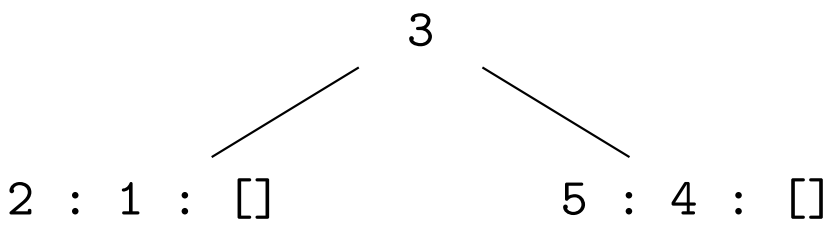
- As mentionend, the *interesting* step of quicksort is the *divide* step, i.e. partitioning a list around a pivot.
- We therefore focus on that step from now.
- partition : $\mathsf{List}A \to 1 + \mathsf{List}A \times A \times \mathsf{List}A$ ...
- ... $\Rightarrow$ Functor F is: $FX = 1 + X \times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

Example: Growing a BST with partition

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

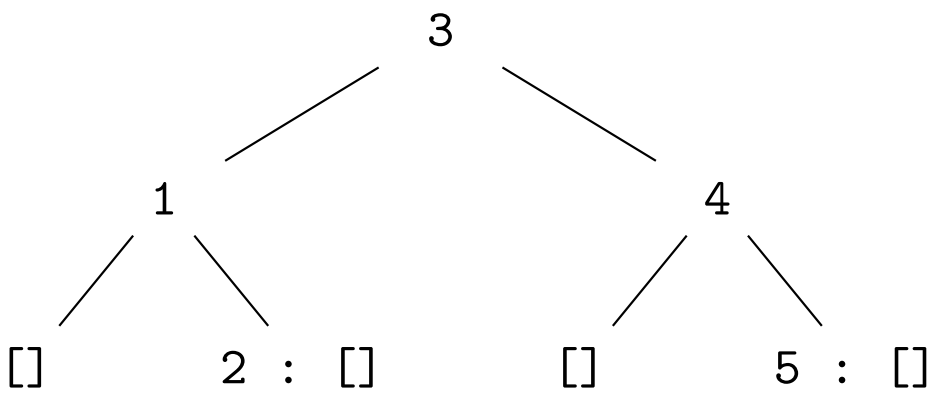
partition : List$A \to$
$\bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

Example: Growing a BST with partition



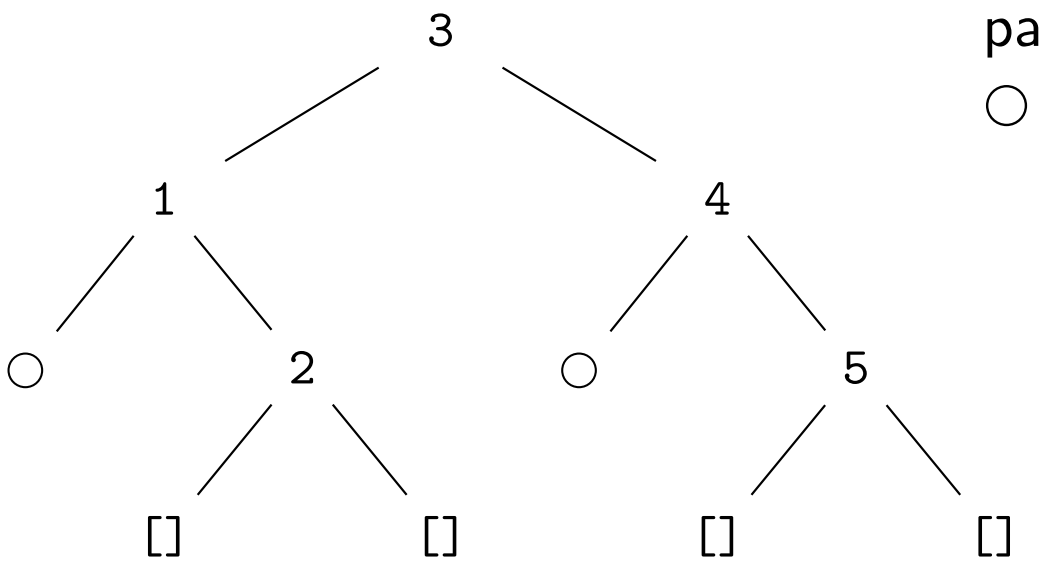
$\mathsf{partition} : \mathsf{List}A \to \bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

Example: Growing a BST with partition



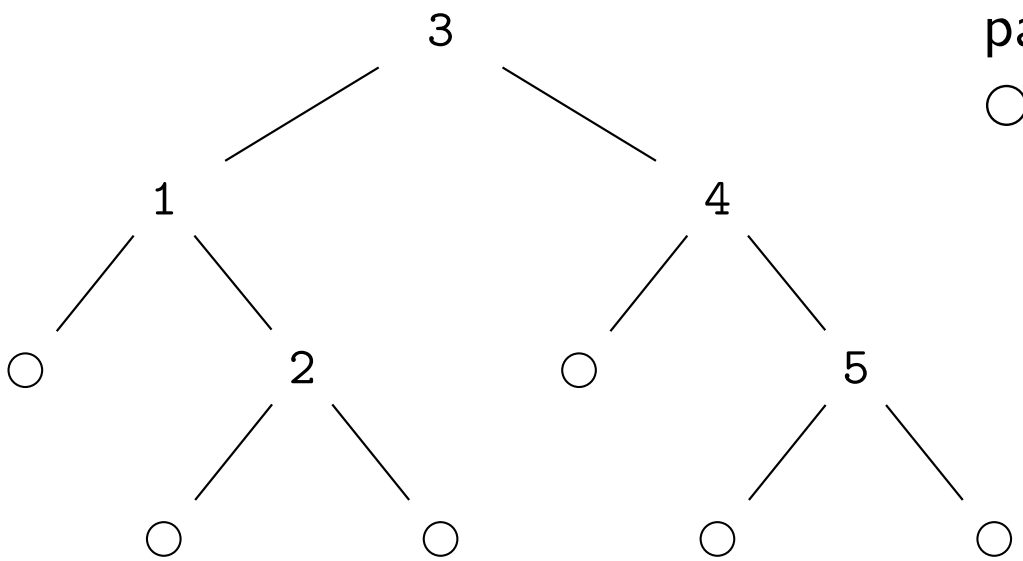
partition : List$A \to$
$\bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

Example: Growing a BST with partition



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

Example: Growing a BST with partition



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

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 ($\mathcal{B}A$) of their elements allows us to express both these properties!
- We focus on the element-preservation property here, as that is relevant for termination.

Sliced Partition

- Redefine partition in the slice category $\mathrm{Set}/\mathcal{B}A$
- We can define lift F to $\bar{F}$ as:

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

- 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

Formalizing "the indices of the recursive positions are smaller than the outer index"

Notation

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

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

Introducing: Well Founded Functors

- Intuition: "The ith output of the functor $F\colon \mathcal{C}^W \to \mathcal{C}^W$ is fully determined by its inputs with indices $j < i$."

Introducing: Well Founded Functors

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

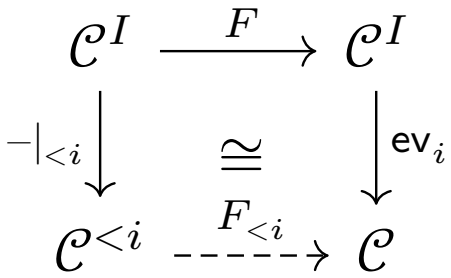
Introducing: Well Founded Functors

Definition (Well-Founded Functor)

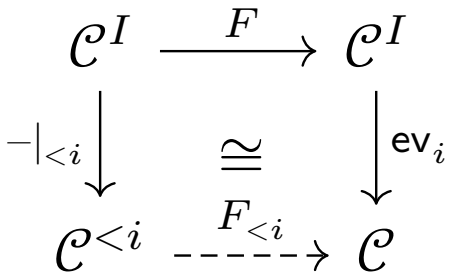
A functor $F\colon \mathcal{C}^W \to \mathcal{C}^W$ is *well-founded* if for every $i \in I$, the composition $\mathsf{ev}_i \cdot F\colon \mathcal{C}^W \to \mathcal{C}$ factors through the restriction $|_{<i}\colon \mathcal{C}^W \to \mathcal{C}^{<i}$, that is, there exists a functor $F_{<i}\colon$ such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

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

Proving WFness of a functor

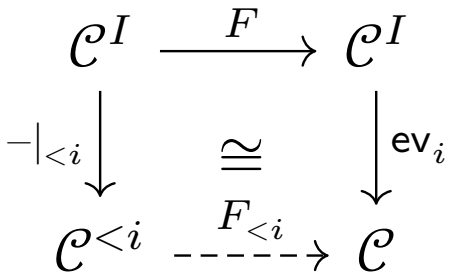


Proving WFness of a functor



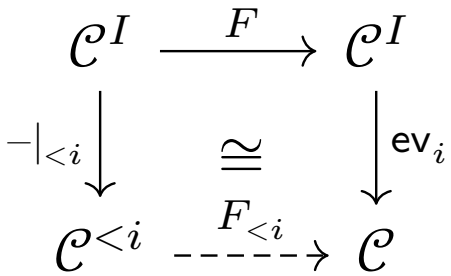
- We define a canonical way to turn any functor F into a family $F_{<i} : (\mathcal{C}^{<i} \to \mathcal{C})_{i \in I}$, for which we obtain a projection $\varepsilon_F X i : F_{<i}(X|_{<i})i \to FXi$.

Proving WFness of a functor



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

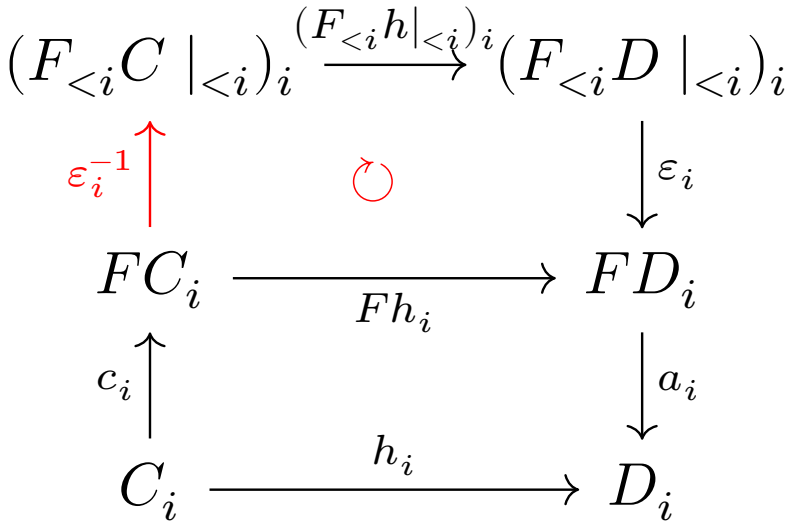
Proving WFness of a functor



- We define a canonical way to turn any functor F into a family $F_{<i} : (\mathcal{C}^{<i} \to \mathcal{C})_{i \in I}$, for which we obtain a projection $\varepsilon_F X i : F_{<i}(X|_{<i})i \to FXi$.
- Client code of the library then consists of defininining an inclusion $\varepsilon_F^{-1} X i : FXi \to F{<}\, i(X|_{<i})i$ which is an inverse to this.
- Next: How we use these parts to solve our original goal of proving recursivity for coalgebras for F

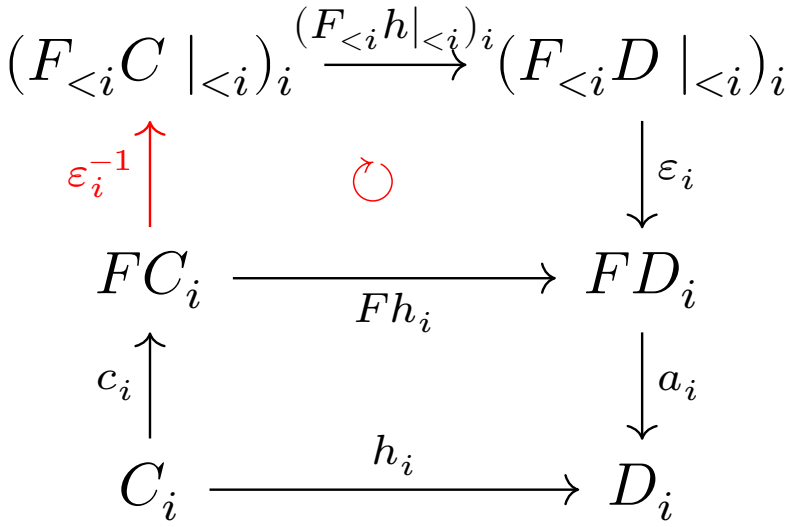
Diagrammatically

$\forall i \in I$:



Diagrammatically

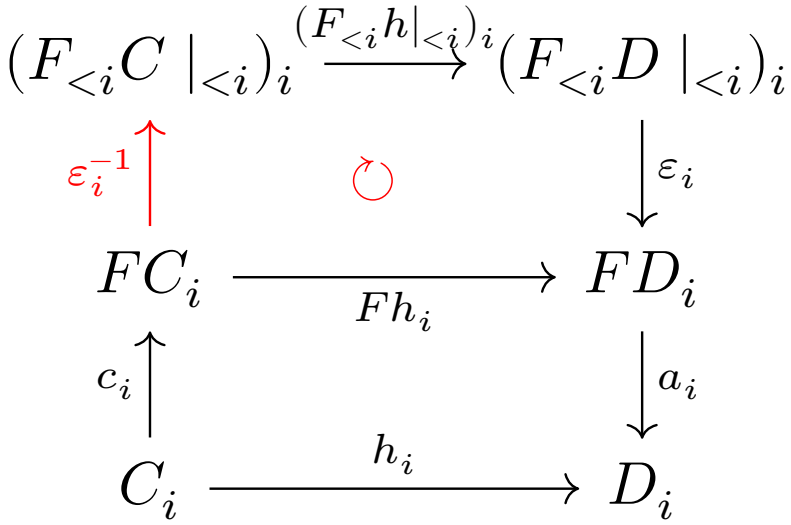
$\forall i \in I$:



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

Diagrammatically

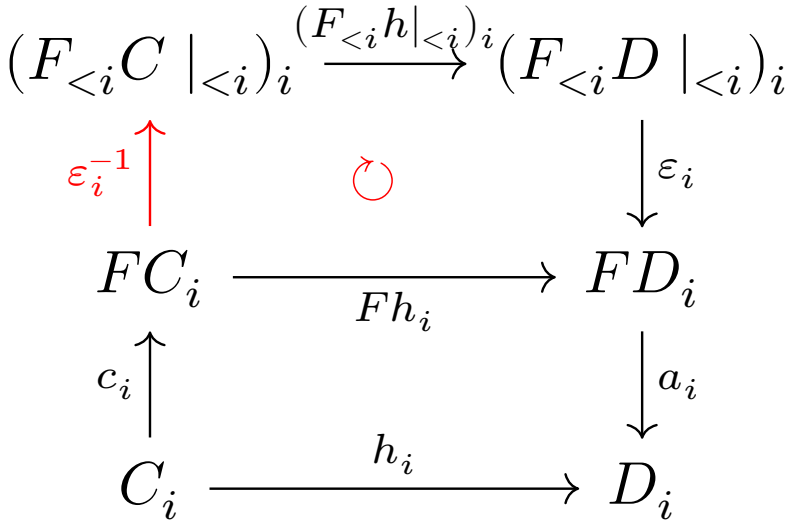
$\forall i \in I$:



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

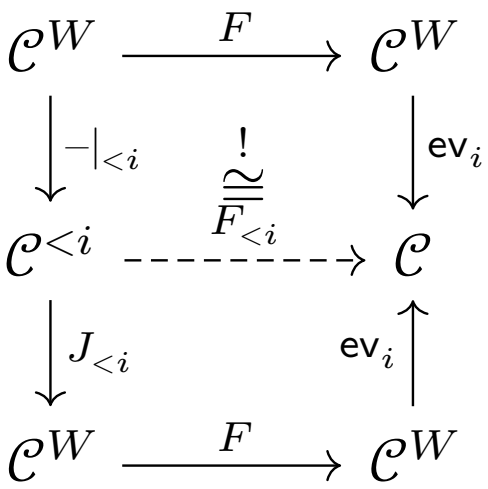
Diagrammatically

$\forall i \in I$:



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

Wellfoundification

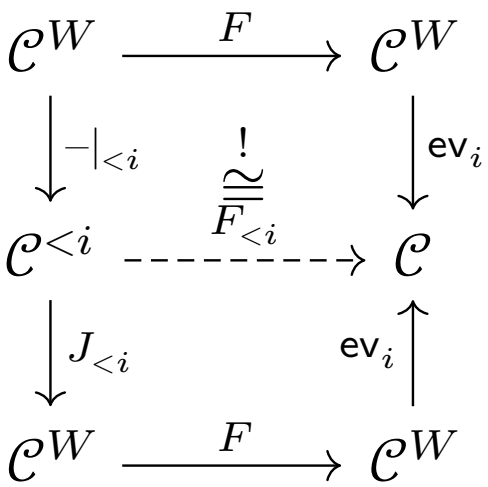


Definition (Inclusion Functor $J_<$)

$$(J_{<i}X)_j \colon \mathcal{C}^{<i} \to \mathcal{C}^W$$

$$(J_{<i}X)_j := \begin{cases} X_j & \text{if } j < i, \\ 0 & \text{otherwise} \end{cases}$$

Wellfoundification



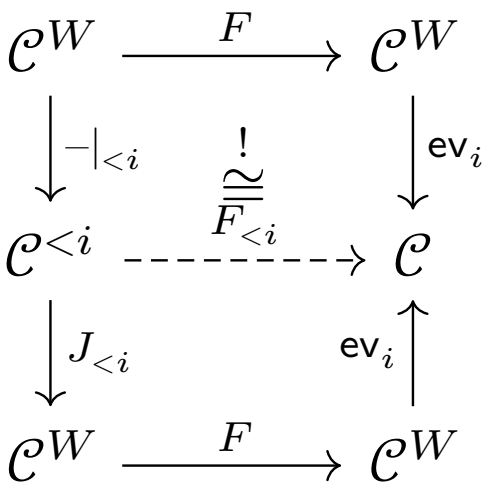
Definition (Inclusion Functor $J_<$)

$$(J_{<i}X)_j \colon \mathcal{C}^{<i} \to \mathcal{C}^W$$

$$(J_{<i}X)_j := \begin{cases} X_j & \text{if } j < i, \\ 0 & \text{otherwise} \end{cases}$$

- N.B.: Need decidability of $<?$ to define $J_{<i}$. Then, to prove inhabitation of $(J_{<i}X)_j$, one need prove $j <? i$ evaluate to $\top$.

Wellfoundification



Definition (Inclusion Functor $J_<$)

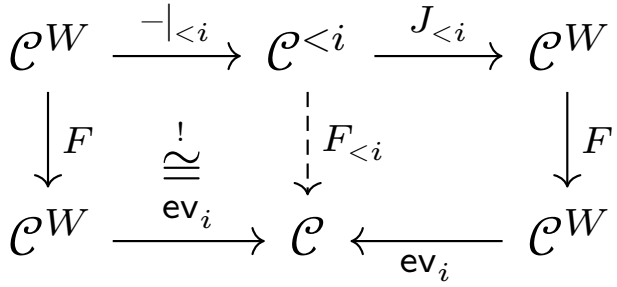
$$(J_{<i}X)_j \colon \mathcal{C}^{<i} \to \mathcal{C}^W$$

$$(J_{<i}X)_j := \begin{cases} X_j & \text{if } j < i, \\ 0 & \text{otherwise} \end{cases}$$

- N.B.: Need decidability of $<?$ to define $J_{<i}$. Then, to prove inhabitation of $(J_{<i}X)_j$, one need prove $j <? i$ evaluate to $\top$.
- We avoid this by specializing $\mathcal{C}$ to Set, and using *copartial elements*:

$$(J_{<i}X)_j \colon \mathsf{Set}^{<i} \to \mathsf{Set}^W \qquad (J_{<i}X)_j := \{x \mid j < i, x \in X_j\}$$

Mechanization




```
_<_ : A → Type -- downset
< i = Σ[ j ∈ A ] (j < i)
-- restriction
_|<_ : (A → Type) → (i : A) → ((< i) → Type)
(X |< i) (j , _pf) = X j

-- inclusion: copartial element formulation
J< : (i : A) → ((< i) → Type) → (A → Type)
J< i X j = Σ[ pf ∈ j < i ] X (j , pf)

-- truncation: restriction, then inclusion: _|<;J< ≈ T
T : (i : A) → (A → Type) → (A → Type)
T i X j = (j < i) × X j -- "annotate with pfs j < i"
```

Structure

Going Back to Definition-Time with Inversion

```
data T (X : ℬ A → Type) : ℬ A → Type where
  leaf : T X []
  _|⌈_⌉|_ : {i_l i_r : ℬ A} → (u : X i_l) → (p : A) →
    (v : X i_r) → T X (p :: i_l ++ i_r)
pattern _^_|⌈_⌉|_^_ u i_l p v i_r = _|⌈_⌉|_ {i_l} {i_r} u p v

T-ε⁻¹ : {X : ℬ A → Type} → (i : ℬ A) → T X i → T (J< i (X |< i)) i
```

```
T-ε⁻¹ .[]                leaf                  = leaf
T-ε⁻¹ .(p :: i_l ++ i_r) (u ^ i_l |⌈ p ⌉| v ^ i_r) =
  ((i_l<i , u) |⌈ p ⌉| (i_r<i , v)) where i_l<i : (i_l <# p :: i_l ++ i_r) ; i_r<i : (i_r <# p :: i_l ++ i_r)
```

Going Back to Definition-Time with Inversion

```
data T (X : ℬ A → Type) : ℬ A → Type where
  leaf : T X []
  _|⌈_⌉|_ : {i_l i_r : ℬ A} → (u : X i_l) → (p : A) →
    (v : X i_r) → T X (p :: i_l ++ i_r)
pattern _^_|⌈_⌉|_^_ u i_l p v i_r = _|⌈_⌉|_ {i_l} {i_r} u p v

T-ε⁻¹ : {X : ℬ A → Type} → (i : ℬ A) → T X i → T (J< i (X |< i)) i
```

(1) Pattern match

```
T-ε⁻¹ .[]                leaf ⟵ = leaf
T-ε⁻¹ .(p :: i_l ++ i_r) (u ^ i_l |⌈ p ⌉| v ^ i_r) =
  ((i_l<i , u) |⌈ p ⌉| (i_r<i , v)) where i_l<i : (i_l <# p :: i_l ++ i_r) ; i_r<i : (i_r <# p :: i_l ++ i_r)
```

1. Pattern match on the value of type T X i;

Going Back to Definition-Time with Inversion

```
data T (X : 𝓑 A → Type ) : 𝓑 A → Type  where
  leaf : T X []
  _|⌈_⌉|_ : {i_l  i_r : 𝓑 A} → (u : X i_l ) → (p : A) →
    (v : X i_r) → T X (p :: i_l ++ i_r)
pattern _^_|⌈_⌉|_^_ u i_l  p v i_r = _|⌈_⌉|_ {i_l} {i_r} u p v

T-ε⁻¹ : {X : 𝓑 A → Type} → (i : 𝓑 A) → T X i → T (J< i (X |< i)) i
```

(2) Inversion (1) Pattern match

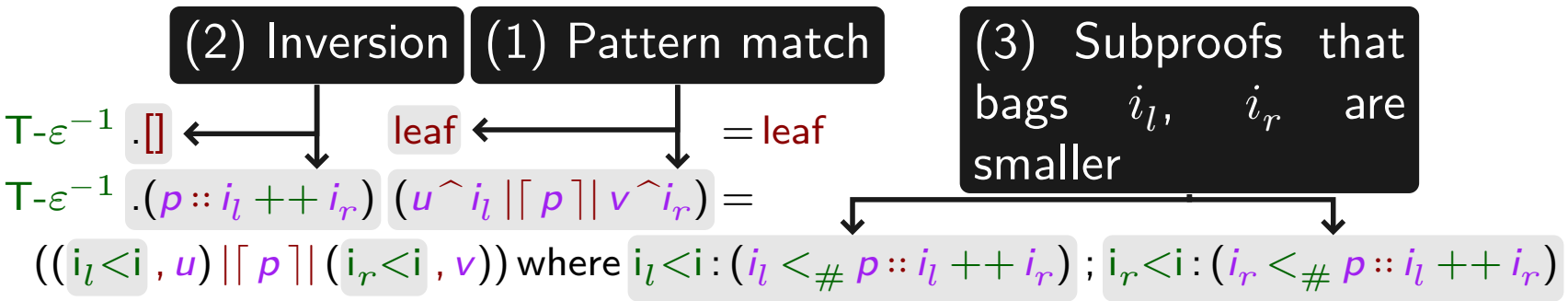
```
T-ε⁻¹ .[]          leaf              = leaf
T-ε⁻¹ .(p :: i_l ++ i_r) (u ^ i_l |⌈ p ⌉| v ^ i_r) =
  ((i_l<i , u) |⌈ p ⌉| (i_r<i , v)) where i_l<i : (i_l <# p :: i_l ++ i_r) ; i_r<i : (i_r <# p :: i_l ++ i_r)
```

1. Pattern match on the value of type T X i;
2. by *inversion* (Dybjer '94), this will refine the original index (seen here as *dot patterns*);

Going Back to Definition-Time with Inversion

```
data T (X : ℬ A → Type) : ℬ A → Type where
  leaf : T X []
  _|⌈_⌉|_ : {i_l i_r : ℬ A} → (u : X i_l) → (p : A) →
    (v : X i_r) → T X (p :: i_l ++ i_r)
pattern _^_|⌈_⌉|_^_ u i_l p v i_r = _|⌈_⌉|_ {i_l} {i_r} u p v

T-ε⁻¹ : {X : ℬ A → Type} → (i : ℬ A) → T X i → T (J< i (X |< i)) i
```



1. Pattern match on the value of type T X i;
2. by *inversion* (Dybjer '94), this will refine the original index (seen here as *dot patterns*);
3. prove that the indices in the functorial positions are smaller than the original, now refined, outer index.

Structure

Inclusion-restriction adjunction

For our original $(J_{<i}X)_j : \mathcal{C}^{<i} \to \mathcal{C}^W$ we have:

$$\mathcal{C}^{<i} \underset{-|_{<i}}{\overset{J_{<i}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathcal{C}^W$$

However, for J<, only the counit can be defined. It turns out that our formalization dualizes to a *partial element* formulation.

```
J< J'< : (i : A) → ((< i) → Type ) → (A → Type )
J'< i X j = ∀ (pf : j < i) → X (j , pf )
J<  i X j = Σ[ pf ∈ j < i ] X (j , pf )
```

Dually for J'< only the unit can be defined. Intuition: partial element defer checking totality to *elimination*; copartial elements must be shown total at introduction.

Future Work

- Explore the (co)partial element connection further to un-specialize from Set
- Develop a variant of our technique for an entirely non-indexed (extrinsic) setting

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.
- General equational definitions, with the possibility to use facilities for generic programming for the remaining boilerplate

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

Conclusion

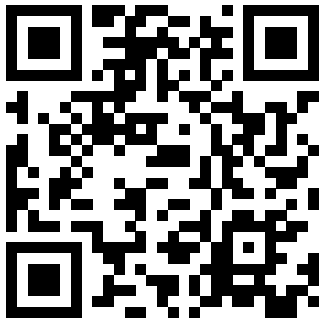
- We have a new way to express a broad class of algorithms describable as coalgebra-to-algebra morphisms in a total functional programming language. More specifically, a novel sufficient criterion for proving & formalizing recursivity of coalgebras.
- Notable use case: Indices already used for proving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

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

- More applications & corollaries in our PLDI paper (formalized: correct GCD, CYK)

Contact

Our PLDI '26 paper:



- Website: `www.cxandru.ee` (these slides are already there!)
- Mastodon: `@cxandru@types.pl`
- Mail: `c.alexandru@cs.rptu.de`
- Discord: `@cxandru`