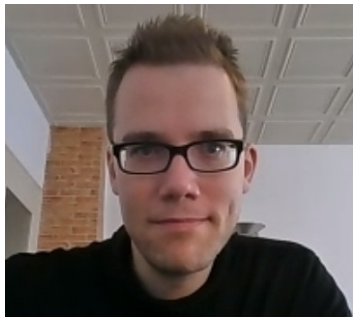
Intrinsically Correct Algorithms & Recursive Coalgebras

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



[1]RPTU Kaiserslautern-Landau & Radboud University Nijmegen

[2]FAU Erlangen-Nürnberg

PLDI 2026

Overview

- Structured Recursion: Modular way, using Category Theory, to express Divide-and-Conquer algorithms s.t. we obtain various theorems "for free", allowing e.g. for program optimization.
- Problem: Lack of criteria for proving termination of algorithms defined like this, in particular amenable to mechanization
- Contribution: Novel notion of well-founded functor which describes division strategies which lead to always-terminating algorithms
  - Useful for both traditional & mechanized proofs
  - Dovetails with intrinsically correct algorithm construction
  - Mechanization of proofs & Agda library
  - Example application to intrinsically correct Quicksort, GCD, CYK algorithms.

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 Study: QuickSort

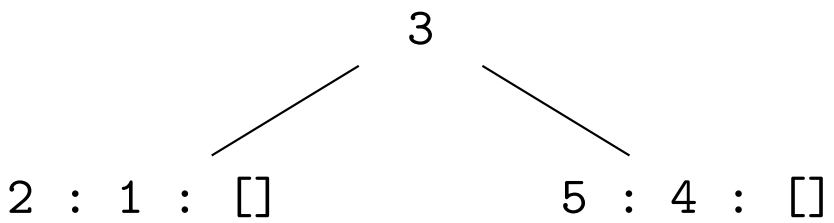
- Quicksort: Main logic in the divide step: partition elements around the pivot. Combine step: concatenation.
- We focus on QuickSort's divide step, `partition`.
- Note: The expression of quicksort as a coalgebra-to-algebra morphism is due to Capretta, Uustalu, and Vene (2006).

QuickSort's divide Step

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

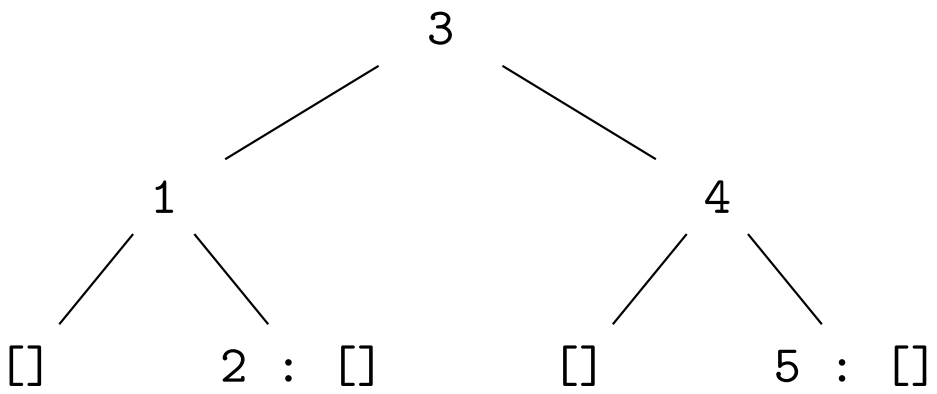
$\mathsf{partition} : \mathsf{List}A \to \bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

QuickSort's divide Step



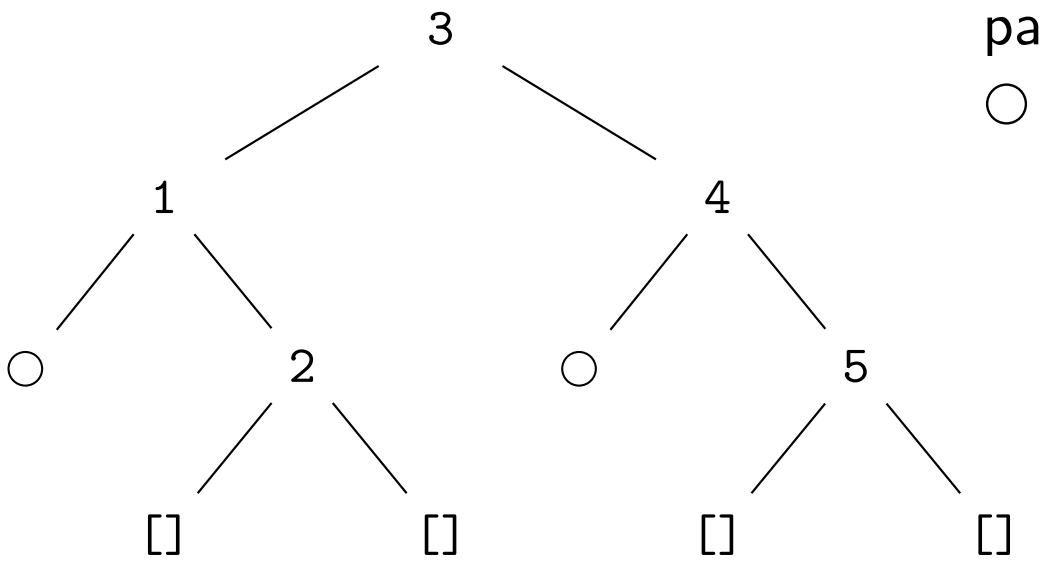
$\mathsf{partition} : \mathsf{List}A \to \bigcirc + \mathsf{List}A \times A \times \mathsf{List}A$

QuickSort's divide Step



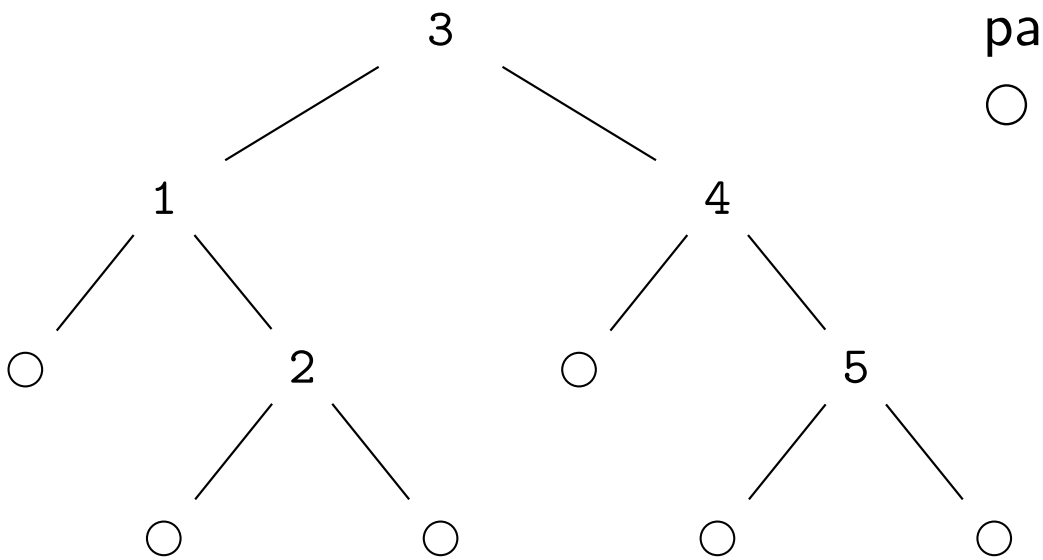
partition : List$A \to$
$\bigcirc + \text{List}A \times A \times \text{List}A$

QuickSort's divide Step



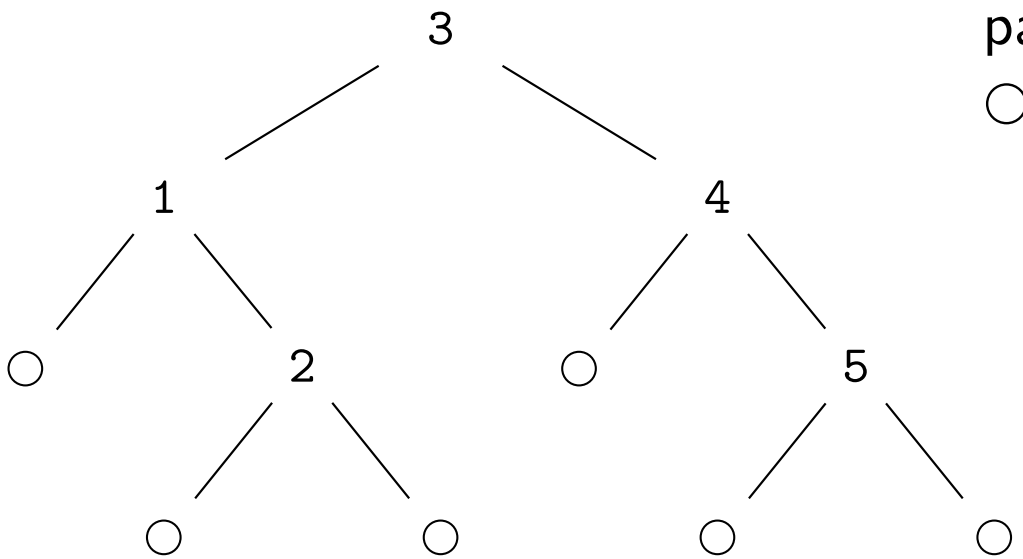
partition : List$A \to$
$\bigcirc + \text{List}A \times A \times \text{List}A$

QuickSort's divide Step



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

QuickSort's divide Step

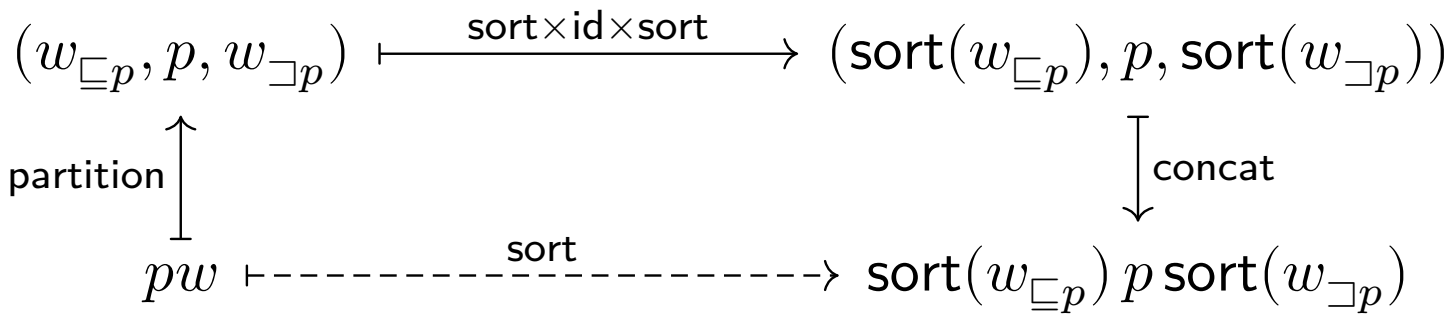


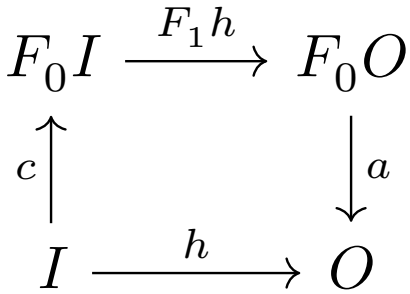
partition : List$A \to$
$\bigcirc + \text{List}A \times A \times \text{List}A$

Call Tree of QuickSort is a Binary Search Tree!

D&CAs as Coalgebra-to-Algebra Morphisms

$\mathsf{partition} : \mathsf{List}A \to 1 + \mathsf{List}A \times A \times \mathsf{List}A$

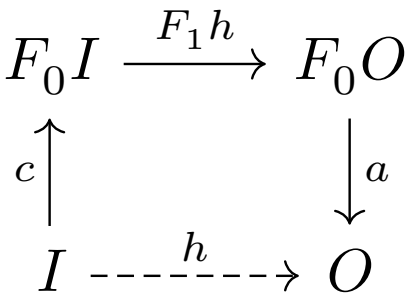




- Coalgebra c : Divide input up into smaller inputs, the distribution of which is given by a functor F;
- Fh: Apply h recursively under F;
- Algebra a: Combine an F-structure of the results of recursive calls to obtain the output.

For Quicksort, $FX = 1 + X \times A \times X$.

Recursive Coalgebras



- A coalgebra c is called recursive if, for every algebra a, it induces a unique h with

$$h = c; Fh; a.$$

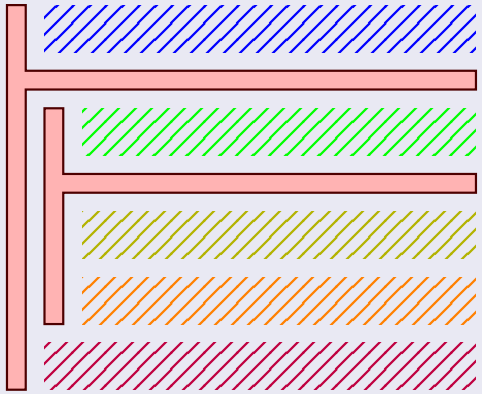
- If a coalgebra is recursive, we can use it to define divide & conquer algorithms.
- How to prove recursivity?

Motivating Our Approach to Proving Recursivity of Coalgebras

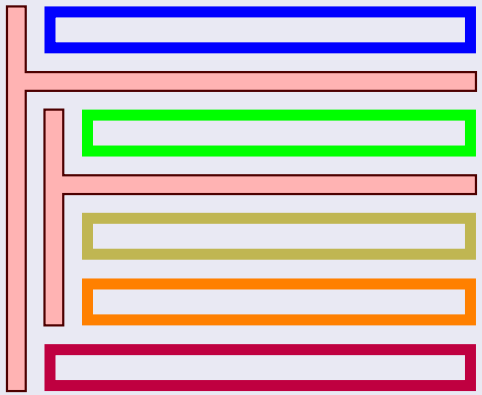
- Begin with intrinsic proofs of partial correctness for coalgebra-to-algebra-morphisms: Index by data relevant for functional correctness properties.
- Observe that this indexing often suffices for also proving termination.
- We capture this with our notion of well-founded functor for which all coalgebras are recursive.

We explain this motivation using the divide step of Quicksort (partition) as an example.

Algorithm

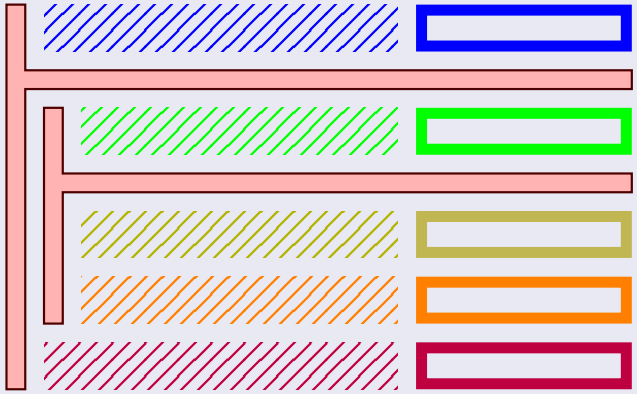


Correctness Proof



$\Rightarrow$

Intrinsically Correct Algorithm



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: xs and partition(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 in the sequel, as orderedness is irrelevant for termination.

Sliced Partition

- Redefine `partition` in the slice category $\mathrm{Set}/\mathcal{B}A$
- We can lift $FX = 1 + X \times A \times X$ to $\bar{F}$ as:

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

- The multiset indices of the recursive positions are smaller than the outer index: $|f(l)|, |f(r)| < |f(l) \uplus \{p\} \uplus f(r)|$. We will use this to prove termination!
- Note: The indexing for partial correctness gives us the data we need for total correctness!
- Next: Formalize "the indices of the recursive positions are smaller than the outer index"

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:

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

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 $j < i$."
- "$F\colon \mathcal{C}^{j \in W} \to \mathcal{C}^{i \in W}$ is morally equivalent to a family $(F_{<i}\colon \mathcal{C}^{j<i} \to \mathcal{C})_{i \in W}$"

Introducing: Well Founded Functors

Intuition:

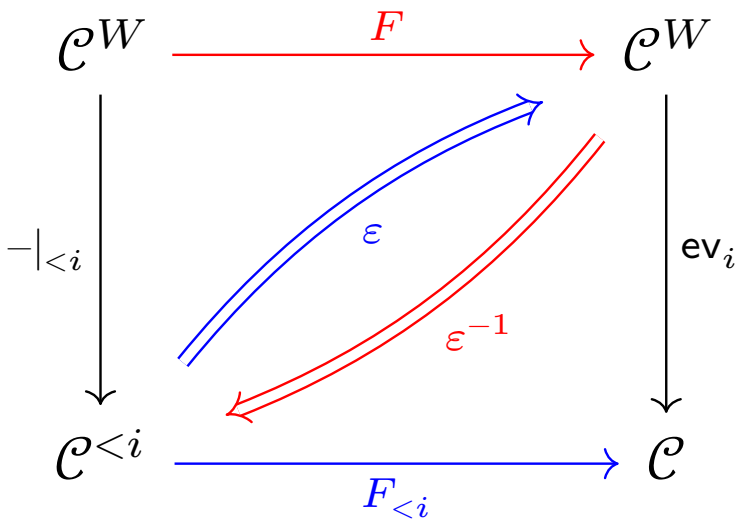
- "The ith output of the functor $F\colon \mathcal{C}^W \to \mathcal{C}^W$ is fully determined by its inputs with indices $j < i$."
- "$F\colon \mathcal{C}^{j \in W} \to \mathcal{C}^{i \in W}$ is morally equivalent to a family $(F_{<i}\colon \mathcal{C}^{j<i} \to \mathcal{C})_{i \in W}$"

Definition (Well-Founded Functor)

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

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

Proving WFness of a functor



client code
library code
specification

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

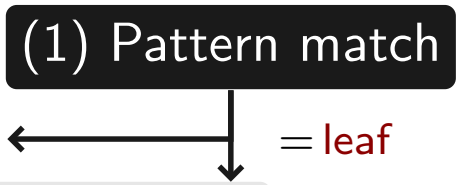
Proving Wellfoundedness of the QuickSort Base Functor

```
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)
```

Proving Wellfoundedness of the QuickSort Base Functor

```
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)
```

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

Proving Wellfoundedness of the QuickSort Base Functor

```
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, 1994), this will refine the original index (seen here as dot patterns);

Proving Wellfoundedness of the QuickSort Base Functor

```
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 (3) Subproofs that bags i_l, i_r are smaller

```
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, 1994), 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.

Future Work

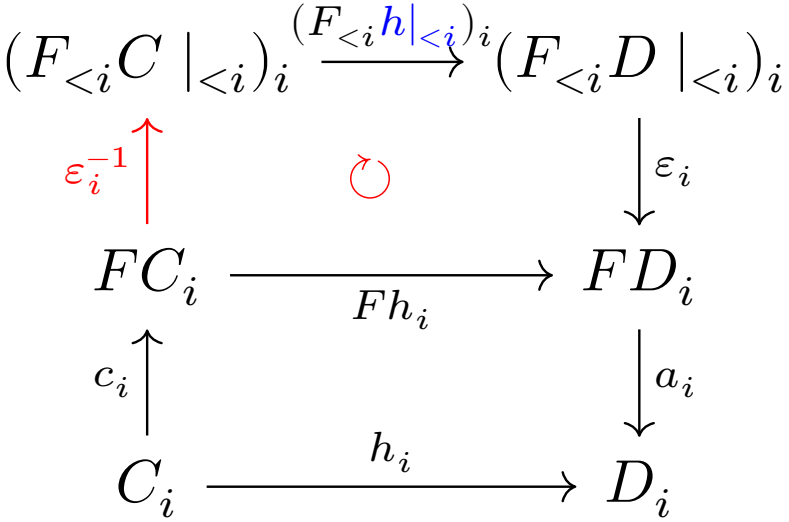
- Express more algorithms intrinsically correctly as coalgebra-to-algebra-morphisms for intrinsically recursive coalgebras
- Develop a variant of our technique for an entirely non-indexed (extrinsic) setting
- Explore connections to similar notions (in particular contractive functions in the topos of trees (Birkedal et al., 2012))

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 (mechanizable) proofs of 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 paper (formalized: correct GCD, CYK)

Recovering the Definition Scheme

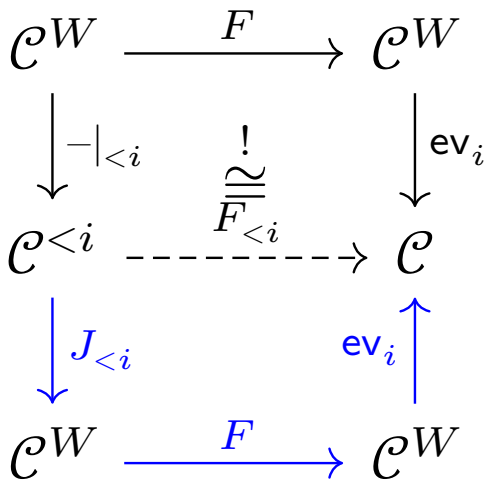
$\forall i \in W$:



In red: what a user of our library provides.

- To define h_i, we need only $h\mid_{<i}$, the restriction of h to $X^{<i}$ $\Rightarrow$
- We can define $(h_i)_{i\in W}$ by well founded induction!
- Next: How we canonically define $F_{<i}$

Wellfoundification



Definition (Inclusion Functor $J_<$)

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

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

- 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:

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