Intrinsically correct sorting using bialgebraic semantics

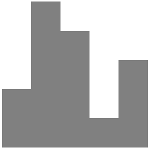
Cass Alexandru

2024-07-28

Background

- "Sorting with Bialgebras and Distributive Laws" (HJHWM, 2012)
- Intrinsically correct version using the same categorical construction
- Brief recap, identify problems, state our solution

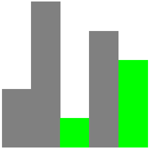
Insertion Sort



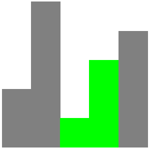
Insertion Sort

Insertion Sort

Insertion Sort

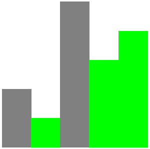


Insertion Sort

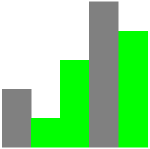


Insertion Sort

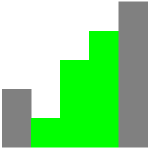
Insertion Sort



Insertion Sort

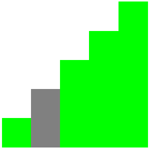


Insertion Sort

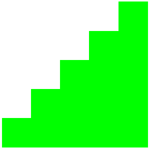


Insertion Sort

Insertion Sort



Insertion Sort



Analyzing its recursion behaviour

- InsertionSort is a *fold* over the input list

Analyzing its recursion behaviour

- InsertionSort is a *fold* over the input list
  - We start with the empty list and build up an ordered list

Analyzing its recursion behaviour

- InsertionSort is a *fold* over the input list
  - We start with the empty list and build up an ordered list
- The argument algebra to this fold is inself an *unfold*

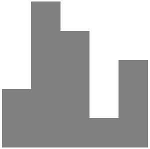
Analyzing its recursion behaviour

- InsertionSort is a *fold* over the input list
  - We start with the empty list and build up an ordered list
- The argument algebra to this fold is inself an *unfold*
  - we output elements of the ordered list at each step

Analyzing its recursion behaviour

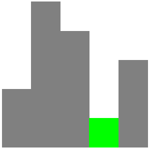
- InsertionSort is a *fold* over the input list
  - We start with the empty list and build up an ordered list
- The argument algebra to this fold is inself an *unfold*
  - we output elements of the ordered list at each step
  - The seed is an an *unordered* pair of element and ordered list

Selection Sort

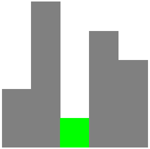


Selection Sort

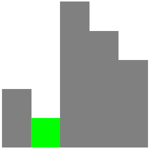
Selection Sort



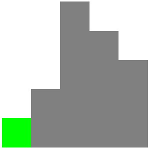
Selection Sort



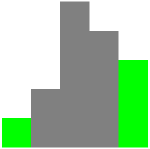
Selection Sort



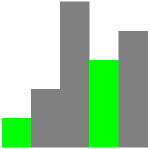
Selection Sort



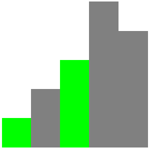
Selection Sort



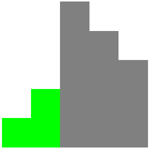
Selection Sort



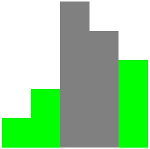
Selection Sort



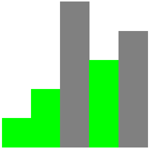
Selection Sort



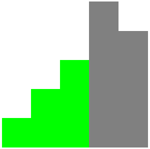
Selection Sort



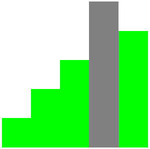
Selection Sort



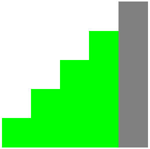
Selection Sort



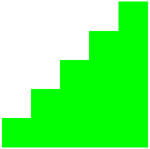
Selection Sort



Selection Sort



Selection Sort



Analyzing its recursion behaviour

- SelectionSort is an *unfold* over the input list

Analyzing its recursion behaviour

- SelectionSort is an *unfold* over the input list
  - We output elements of the ordered list at each step

Analyzing its recursion behaviour

- SelectionSort is an *unfold* over the input list
  - We output elements of the ordered list at each step
  - The seed is an unordered list, initially the whole input list

Analyzing its recursion behaviour

- SelectionSort is an *unfold* over the input list
  - We output elements of the ordered list at each step
  - The seed is an unordered list, initially the whole input list
- The argument coalgebra to this unfold is itself a *fold*

Analyzing its recursion behaviour

- SelectionSort is an *unfold* over the input list
  - We output elements of the ordered list at each step
  - The seed is an unordered list, initially the whole input list
- The argument coalgebra to this unfold is itself a *fold*
  - We start at the bottom of the list and output an *ordered* pair of element and unordered list

Bialgebraic semantics

```
swap : {x : Set ℓ'} → L (x × O x) → O (x ⊎ L x)
swap [] = []ᵀ
swap (a ∷ (r , []ᵀ)) = a ≤∷ inl r
swap (a ∷ (r , b ≤∷ r')) with a ≤?≥ b
...| inl a≤b = a ≤∷ inl r
...| inr b≤a = b ≤∷ inr (a ∷ r')
```

```
insertSort₁ = fold (apo (swap ∘ L₁ ⟨ id , out ⟩))
selectSort₁ = unfold (para (O₁ [ id , ın ] ∘ swap))
```

The Problem

- We need to specify what it means for a sorting algorithm to be correct
- Problematic use of unfold:

```
unfold : {X : Set ℓ'} (grow : X → O X) → X → νO
unfold grow seed with grow seed
...| []ᵀ          = ⌊[]ᵀ⌋
...| (x ≤:: seed') = ⌊ x ≤:: unfold grow seed' ⌋
```

- `repeat a = unfold (λ tt → a ≤:: tt) tt`

Specification of Sorting

- The output list must be *ordered*, i.e. all pairs of consecutive elements are related to each other via the ordering relation $\leq$.
- The second rule of sorting is: The output is a permutation of the input.

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

- Output is a permutation of the input $\Leftrightarrow$ Mapping a list to the multiset of its element is invariant under sorting $\Leftrightarrow$ sorting preserves the `FMSet` index

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

- Output is a permutation of the input $\Leftrightarrow$ Mapping a list to the multiset of its element is invariant under sorting $\Leftrightarrow$ sorting preserves the `FMSet` index
- Use it to encode ordering at the level of the ordered list base functor using `All (x ≤_)` (similar to "Fresh Lists")

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

- Output is a permutation of the input ⇔ Mapping a list to the multiset of its element is invariant under sorting ⇔ sorting preserves the `FMSet` index
- Use it to encode ordering at the level of the ordered list base functor using `All (x ≤_)` (similar to "Fresh Lists")
- Acts as a size parameter: All coalgebras are well founded!

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

- Output is a permutation of the input ⇔ Mapping a list to the multiset of its element is invariant under sorting ⇔ sorting preserves the `FMSet` index
- Use it to encode ordering at the level of the ordered list base functor using `All (x ≤_)` (similar to "Fresh Lists")
- Acts as a size parameter: All coalgebras are well founded!
- This also works for the other sorting algorithms formalized in Hinze et al. 2012

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

- Output is a permutation of the input ⇔ Mapping a list to the multiset of its element is invariant under sorting ⇔ sorting preserves the `FMSet` index
- Use it to encode ordering at the level of the ordered list base functor using `All (x ≤_)` (similar to "Fresh Lists")
- Acts as a size parameter: All coalgebras are well founded!
- This also works for the other sorting algorithms formalized in Hinze et al. 2012
    - More lemmata about equalities (paths) in `FMSet A` to substitute along

The Solution

Our solution: index lists by the finite multiset of their elements (a QIT)

- Output is a permutation of the input ⇔ Mapping a list to the multiset of its element is invariant under sorting ⇔ sorting preserves the `FMSet` index
- Use it to encode ordering at the level of the ordered list base functor using `All (x ≤_)` (similar to "Fresh Lists")
- Acts as a size parameter: All coalgebras are well founded!
- This also works for the other sorting algorithms formalized in Hinze et al. 2012
  - More lemmata about equalities (paths) in `FMSet A` to substitute along
  - Well foundedness of coalgebras proven using WFI on the length of the `FMSet` index

swap, refined

```
swap : {g : FMSet A} {r : FMSet A → Type ℓ} → L ((O x) r) g → O ((L +) r) g
swap [] = []
swap (a ∷ (x , [])) = (a ≤∷ inl x) []-A
swap (a ∷ (x , (b ≤∷ x') b#x')) with a ≤?≥ b
...| inl a≤b = (a ≤∷ inl x) $ a≤b ∷-A ≤-to-# a≤b b#x'
...| inr b≤a = (b ≤∷ inr (a ∷ x')) $ b≤a ∷-A b#x' €
     subst (O ((L +) _)) (comm _ _ _)
```