

Wander Types

Realizing Data Structures Through Coinduction-Recursion

Venanzio Capretta
University of Nottingham

Shonan Meeting on Dependently Typed Programming, 2011

IR definitions

Inductive-Recursive (IR) definitions

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

- an inductive type $T : \text{Set}$
- a recursive function on it $f : T \rightarrow D$

mutually dependent

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

- an inductive type $T : \text{Set}$
- a recursive function on it $f : T \rightarrow D$

mutually dependent

- The constructors of T can use f

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

- an inductive type $T : \text{Set}$
- a recursive function on it $f : T \rightarrow D$

mutually dependent

- The constructors of T can use f
- f recursive over T

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

- an inductive type $T : \text{Set}$
- a recursive function on it $f : T \rightarrow D$

mutually dependent

- The constructors of T can use f
- f recursive over T

Definition of Type Universes [Martin-Löf 1984, Palmgren 1998]

General Definition [Dybjer 2001, Dybjer/Setzer 1999]

Advanced Data Structures

Heap (priority queue) on ordered set A, \preceq

Advanced Data Structures

Heap (priority queue) on ordered set A, \preceq

Heap : Set

Advanced Data Structures

Heap (priority queue) on ordered set A, \preccurlyeq

Heap : Set

empty : Heap

isEmpty : Heap $\rightarrow \mathbb{B}$

Advanced Data Structures

Heap (priority queue) on ordered set A, \preccurlyeq

Heap : Set

empty : Heap

isEmpty : Heap $\rightarrow \mathbb{B}$

insert : $A \rightarrow \text{Heap} \rightarrow \text{Heap}$

Advanced Data Structures

Heap (priority queue) on ordered set A, \preceq

Heap : Set

empty : Heap

isEmpty : Heap $\rightarrow \mathbb{B}$

insert : $A \rightarrow \text{Heap} \rightarrow \text{Heap}$

findMin : Heap $\rightarrow A$

Advanced Data Structures

Heap (priority queue) on ordered set A, \preceq

Heap : Set

empty : Heap

isEmpty : Heap $\rightarrow \mathbb{B}$

insert : $A \rightarrow \text{Heap} \rightarrow \text{Heap}$

findMin : Heap $\rightarrow A$

deleteMin : Heap $\rightarrow \text{Heap}$

Advanced Data Structures

Heap (priority queue) on ordered set A, \preceq

Heap : Set

empty : Heap

isEmpty : Heap $\rightarrow \mathbb{B}$

insert : $A \rightarrow \text{Heap} \rightarrow \text{Heap}$

findMin : Heap $\rightarrow A$

deleteMin : Heap $\rightarrow \text{Heap}$

merge : Heap $\rightarrow \text{Heap} \rightarrow \text{Heap}$

Advanced Data Structures

Heap (priority queue) on ordered set A, \preceq

Heap : Set

empty : Heap

isEmpty : Heap $\rightarrow \mathbb{B}$

insert : $A \rightarrow \text{Heap} \rightarrow \text{Heap}$

findMin : Heap $\rightarrow A$

deleteMin : Heap $\rightarrow \text{Heap}$

merge : Heap $\rightarrow \text{Heap} \rightarrow \text{Heap}$

With lists: linear complexity

With leftist heaps: logarithmic complexity



Leftist Heaps

Definition of [Leftist Heaps](#) [Crane 1972, Knuth 1973]

Leftist Heaps

Definition of **Leftist Heaps** [Crane 1972, Knuth 1973]

Heaps are often implemented as heap-ordered trees, in which the element at each node is no larger than the elements at its children. Under this ordering, the minimum element in a tree is always at the root. Leftist heaps are heap-ordered binary trees that satisfy the leftist property: the rank of any left child is at least as large as the rank of its right sibling. The rank of a node is defined to be the length of its right spine (i.e., the rightmost path from the node in question to an empty node).

[Okasaki 1998]

Inductive-Recursive Leftist Heaps

- IHeap : Set

Inductive-Recursive Leftist Heaps

- $\text{IHeap} : \text{Set}$

- $\text{rootor} : \text{IHeap} \rightarrow A \rightarrow A$

Inductive-Recursive Leftist Heaps

- $\text{IHeap} : \text{Set}$

- $\text{rootor} : \text{IHeap} \rightarrow A \rightarrow A$

- $\text{rank} : \text{IHeap} \rightarrow \mathbb{N}$

Inductive-Recursive Leftist Heaps

- $\text{IHeap} : \text{Set}$

$\text{leaf} : \text{IHeap}$

- $\text{rootor} : \text{IHeap} \rightarrow A \rightarrow A$

$\text{rootor leaf} = \text{id}$

- $\text{rank} : \text{IHeap} \rightarrow \mathbb{N}$

$\text{rank leaf} = 0$

Inductive-Recursive Leftist Heaps

- $\text{IHeap} : \text{Set}$

leaf : IHeap

node : $(a : A; t_1, t_2 : \text{IHeap})$

$\rightarrow a \preceq (\text{rootor } t_1 a) \rightarrow a \preceq (\text{rootor } t_2 a)$

$\rightarrow (\text{rank } t_2) \leq (\text{rank } t_1) \rightarrow \text{IHeap}$

- $\text{rootor} : \text{IHeap} \rightarrow A \rightarrow A$

rootor leaf = id

- $\text{rank} : \text{IHeap} \rightarrow \mathbb{N}$

rank leaf = 0

Inductive-Recursive Leftist Heaps

■ IHeap : Set

leaf : IHeap

node : (a : A; t₁, t₂ : IHeap)

→ a ≼ (rootor t₁ a) → a ≼ (rootor t₂ a)

→ (rank t₂) ≤ (rank t₁) → IHeap

■ rootor : IHeap → A → A

rootor leaf = id

rootor (node a t₁ t₂ ...) = λx.a

■ rank : IHeap → ℕ

rank leaf = 0

Inductive-Recursive Leftist Heaps

- $\text{IHeap} : \text{Set}$

leaf : IHeap

node : $(a : A; t_1, t_2 : \text{IHeap})$

$\rightarrow a \preceq (\text{rootor } t_1 a) \rightarrow a \preceq (\text{rootor } t_2 a)$

$\rightarrow (\text{rank } t_2) \leq (\text{rank } t_1) \rightarrow \text{IHeap}$

- $\text{rootor} : \text{IHeap} \rightarrow A \rightarrow A$

rootor leaf = id

rootor (node a t_1 t_2 ...) = $\lambda x. a$

- $\text{rank} : \text{IHeap} \rightarrow \mathbb{N}$

rank leaf = 0

rank (node a t_1 t_2 ...) = $1 + \text{rank } t_2$

Codes for IR Definitions

Dybjer/Setzer: codes for Inductive Recursive definitions

Codes for IR Definitions

Dybjer/Setzer: codes for Inductive Recursive definitions

- Given D result type: IRD type of codes for ind/rec defs

Codes for IR Definitions

Dybjer/Setzer: codes for Inductive Recursive definitions

- Given D result type: IRD type of codes for ind/rec defs
- $c : IRD$ decodes to:

Codes for IR Definitions

Dybjer/Setzer: codes for Inductive Recursive definitions

- Given D result type: IRD type of codes for ind/rec defs
- $c : IRD$ decodes to:

$$\begin{aligned} \text{Elem}_c &: \text{Set} \\ \text{fun}_c &: \text{Elem}_c \rightarrow D \end{aligned}$$

Codes for IR Definitions

Dybjer/Setzer: codes for Inductive Recursive definitions

- Given D result type: IRD type of codes for ind/rec defs
- $c : IRD$ decodes to:

$$\begin{aligned} \text{Elem}_c &: \text{Set} \\ \text{fun}_c &: \text{Elem}_c \rightarrow D \end{aligned}$$

- IRD is itself a simple inductive type

Constructors of IR codes

Constructors of IR codes

- A single element with a given value

$$c = \text{element } d$$
$$d : D$$

Constructors of IR codes

- A single element with a given value

$$\begin{array}{l}
 c = \text{element } d \\
 d : D
 \end{array}
 \rightsquigarrow
 \begin{array}{l}
 \text{Elem}_c \simeq \{\bullet\} \\
 \text{fun}_c(\bullet) = d
 \end{array}$$

Constructors of IR codes

- A single element with a given value

$$\begin{array}{l}
 c = \text{element } d \\
 d : D
 \end{array}
 \rightsquigarrow
 \begin{array}{l}
 \text{Elem}_c \simeq \{\bullet\} \\
 \text{fun}_c(\bullet) = d
 \end{array}$$

- A constructor with a non-recursive argument

$$\begin{array}{l}
 c = \text{choose } A \ u \\
 A : \text{Set} \\
 u : A \rightarrow \text{IRD}
 \end{array}$$

Constructors of IR codes

- A single element with a given value

$$\begin{array}{l}
 c = \text{element } d \\
 d : D
 \end{array}
 \rightsquigarrow
 \begin{array}{l}
 \text{Elem}_c \simeq \{\bullet\} \\
 \text{fun}_c(\bullet) = d
 \end{array}$$

- A constructor with a non-recursive argument

$$\begin{array}{l}
 c = \text{choose } A \ u \\
 A : \text{Set} \\
 u : A \rightarrow \text{IRD}
 \end{array}
 \rightsquigarrow
 \begin{array}{l}
 \text{Elem}_c \simeq (a : A) \times \text{Elem}_{(u\ a)} \\
 \text{fun}_c\langle a, x \rangle = \text{fun}_{(u\ a)}(x)
 \end{array}$$

Constructors of IR codes

- A single element with a given value

$$c = \text{element } d \quad \rightsquigarrow \quad \text{Elem}_c \simeq \{\bullet\}$$

$$d : D \quad \text{fun}_c(\bullet) = d$$

- A constructor with a non-recursive argument

$$c = \text{choose } A \ u \quad \rightsquigarrow \quad \text{Elem}_c \simeq (a : A) \times \text{Elem}_{(u a)}$$

$$A : \text{Set} \quad \text{fun}_c \langle a, x \rangle = \text{fun}_{(u a)}(x)$$

$$u : A \rightarrow \text{IRD}$$

- A constructor with recursive arguments

$$c = \text{recurse } l \ v$$

$$l : \text{Set}$$

$$v : (l \rightarrow D) \rightarrow \text{IRD}$$

Constructors of IR codes

- A single element with a given value

$$c = \text{element } d \quad \rightsquigarrow \quad \text{Elem}_c \simeq \{\bullet\}$$

$$d : D \quad \text{fun}_c(\bullet) = d$$

- A constructor with a non-recursive argument

$$c = \text{choose } A \ u \quad \rightsquigarrow \quad \text{Elem}_c \simeq (a : A) \times \text{Elem}_{(u\ a)}$$

$$A : \text{Set} \quad \text{fun}_c\langle a, x \rangle = \text{fun}_{(u\ a)}(x)$$

$$u : A \rightarrow \text{IRD}$$

- A constructor with recursive arguments

$$c = \text{recurse } I \ v \quad \rightsquigarrow \quad \text{Elem}_c \simeq (t : I \rightarrow \text{Elem}_c) \times \text{Elem}_{v(\text{fun}_c \circ t)}$$

$$I : \text{Set} \quad \text{fun}_c\langle t, y \rangle = \text{fun}_{v(\text{fun}_c \circ t)}(y)$$

$$v : (I \rightarrow D) \rightarrow \text{IRD}$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)
```

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)  
= element ⟨id, 0⟩ +
```

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)
= element ⟨id, 0⟩ +
  choose A λa.
```

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)
= element ⟨id, 0⟩ +
  choose A λa.
    recurse ℬ λ⟨root1, rank1, root2, rank2⟩.
```

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)
= element ⟨id, 0⟩ +
  choose A λa.
    recurse ℬ λ⟨root1, rank1, root2, rank2⟩.
      choose (a ≲ (root1 a) ∧ a ≲ (root2 a) ∧ (rank2 ≤ rank1))
```


Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)
= element ⟨id, 0⟩ +
  choose A λa.
    recurse ℬ λ⟨root1, rank1, root2, rank2⟩.
      choose (a ≲ (root1 a) ∧ a ≲ (root2 a) ∧ (rank2 ≤ rank1))
        λ_. element ⟨λx.a, 1 + rank2⟩
```

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

```
lhcode : IR ((A → A) × ℕ)
= element ⟨id, 0⟩ +
  choose A λa.
    recurse ℬ λ⟨root1, rank1, root2, rank2⟩.
      choose (a ≲ (root1 a) ∧ a ≲ (root2 a) ∧ (rank2 ≤ rank1))
        λ.. element ⟨λx.a, 1 + rank2⟩
```

+: separate different constructors, non-dependent choose

Elements of $\mathbb{B} \rightarrow (A \rightarrow A) \times \mathbb{N}$:

quadruples $\langle \text{root}_1, \text{rank}_1, \text{root}_2, \text{rank}_2 \rangle$

Slice Category

Inductive Types: initial algebras in the base category.

Slice Category

Inductive Types: initial algebras in the base category.

IR Types: initial algebras in a slice category.

[Dybjer/Setzer 1999, Hancock/Ghani 2011]

Slice Category

Inductive Types: initial algebras in the base category.

IR Types: initial algebras in a slice category.

[Dybjer/Setzer 1999, Hancock/Ghani 2011]

For every type D the slice category $\mathbf{Set} \downarrow D$ has:

Slice Category

Inductive Types: initial algebras in the base category.

IR Types: initial algebras in a slice category.

[Dybjer/Setzer 1999, Hancock/Ghani 2011]

For every type D the slice category $\mathbf{Set} \downarrow D$ has:

Objects: pairs $\langle X, f \rangle$
where $X : \mathbf{Set}$ $f : X \rightarrow D$

Slice Category

Inductive Types: initial algebras in the base category.

IR Types: initial algebras in a slice category.

[Dybjer/Setzer 1999, Hancock/Ghani 2011]

For every type D the slice category $\mathbf{Set} \downarrow D$ has:

Objects: pairs $\langle X, f \rangle$
 where $X : \mathbf{Set}$ $f : X \rightarrow D$

Morphisms: functions $g : \langle X_1, f_1 \rangle \rightarrow \langle X_2, f_2 \rangle$
 where $g : X_1 \rightarrow X_2$ $f_2 \circ g = f_1$

Slice Category

Inductive Types: initial algebras in the base category.

IR Types: initial algebras in a slice category.

[Dybjer/Setzer 1999, Hancock/Ghani 2011]

For every type D the slice category $\mathbf{Set} \downarrow D$ has:

Objects: pairs $\langle X, f \rangle$

where $X : \mathbf{Set}$ $f : X \rightarrow D$

Morphisms: functions $g : \langle X_1, f_1 \rangle \rightarrow \langle X_2, f_2 \rangle$

where $g : X_1 \rightarrow X_2$ $f_2 \circ g = f_1$

Functor $\Theta : \mathbf{Set} \downarrow D \rightarrow \mathbf{Set} \downarrow D$

IR definition: Initial Θ -algebra

What are the final coalgebras?

Call them **Wander Types**

Wander Types: Final Slice Coalgebra

Coinductive version of leftist heaps?

Wander Types: Final Slice Coalgebra

Coinductive version of leftist heaps?

Potentially infinite binary trees.

But the rank function must still be defined.

Right spine must be finite.

Wander Types: Final Slice Coalgebra

Coinductive version of leftist heaps?

Potentially infinite binary trees.

But the rank function must still be defined.

Right spine must be finite.

Wander Types: Define a coinductive type and a recursive function simultaneously.

Final coalgebras in slice categories

Wander Types: Final Slice Coalgebra

Coinductive version of leftist heaps?

Potentially infinite binary trees.

But the rank function must still be defined.

Right spine must be finite.

Wander Types: Define a coinductive type and a recursive function simultaneously.

Final coalgebras in slice categories

Mixed Inductive-Coinductive Definitions [Danielsson/Altenkirch 2009]

Mixed Induction-Coinduction

Streams 0s and 1s, with no infinite consecutive 1s.

Mixed Induction-Coinduction

Streams 0s and 1s, with no infinite consecutive 1s.

- CoInductive `ZeroOne : Set`

`zero : ZeroOne → ZeroOne`

`one : ZeroOne → ZeroOne`

Mixed Induction-Coinduction

Streams 0s and 1s, with no infinite consecutive 1s.

- CoInductive `ZeroOne : Set`

`zero : ZeroOne → ZeroOne`

`one : ZeroOne → ZeroOne`

- Simultaneously `count1 : ZeroOne → ℕ`

`count1 (zero s) = 0`

`count1 (one s) = 1 + count1 s`

Mixed Induction-Coinduction

Streams 0s and 1s, with no infinite consecutive 1s.

- CoInductive `ZeroOne` : `Set`

`zero` : `ZeroOne` \rightarrow `ZeroOne`

`one` : `ZeroOne` \rightarrow `ZeroOne`

- Simultaneously `count1` : `ZeroOne` \rightarrow \mathbb{N}

`count1` (`zero s`) = 0

`count1` (`one s`) = 1 + `count1 s`

- We can even make the zeros finite (still infinite sequences)

`count0` : `ZeroOne` \rightarrow \mathbb{N}

`count0` (`zero s`) = 1 + `count0 s`

`count0` (`one s`) = 0

No-zigzag type

Define a type of potentially infinite binary trees, but with the restriction that there can't be infinite zigzags.

No-zigzag type

Define a type of potentially infinite binary trees, but with the restriction that there can't be infinite zigzags.

CoInductive ZigZag : Set

No-zigzag type

Define a type of potentially infinite binary trees, but with the restriction that there can't be infinite zigzags.

CoInductive `ZigZag` : Set
Simultaneously `zigs` : `ZigZag` \rightarrow \mathbb{N}
`zags` : `ZigZag` \rightarrow \mathbb{N}

No-zigzag type

Define a type of potentially infinite binary trees, but with the restriction that there can't be infinite zigzags.

CoInductive `ZigZag` : Set
 Simultaneously `zigs` : `ZigZag` \rightarrow \mathbb{N}
 `zags` : `ZigZag` \rightarrow \mathbb{N}

Constructors
`zzlf` : `ZigZag`
`zznd` : `ZigZag` \rightarrow `ZigZag` \rightarrow `ZigZag`

No-zigzag type

Define a type of potentially infinite binary trees, but with the restriction that there can't be infinite zigzags.

CoInductive $\text{ZigZag} : \text{Set}$

Simultaneously $\text{zigs} : \text{ZigZag} \rightarrow \mathbb{N}$

$\text{zags} : \text{ZigZag} \rightarrow \mathbb{N}$

Constructors

$\text{zzlf} : \text{ZigZag}$

$\text{zznd} : \text{ZigZag} \rightarrow \text{ZigZag} \rightarrow \text{ZigZag}$

Equations

$\text{zigs } \text{zzlf} = 0$ $\text{zigs } (\text{zznd } t_1 t_2) = 1 + \text{zags } t_1$

$\text{zags } \text{zzlf} = 0$ $\text{zags } (\text{zznd } t_1 t_2) = 1 + \text{zigs } t_2$

Stream Processors

Other example [Ghani/Hancock/Pattinson 2009]

Continuous Stream Processors (Stream A) \rightarrow (Stream B)
represented by nested fixed points.

Stream Processors

Other example [Ghani/Hancock/Pattinson 2009]

Continuous Stream Processors (Stream A) \rightarrow (Stream B)
represented by nested fixed points.

$\text{StrProc}(A, B) : \text{Set}$

$\text{write} : B \rightarrow \text{StrProc}(A, B) \rightarrow \text{StrProc}(A, B)$

$\text{read} : (A \rightarrow \text{StrProc}(A, B)) \rightarrow \text{StrProc}(A, B)$

write coinductive, read inductive.

Stream Processors

Other example [Ghani/Hancock/Pattinson 2009]

Continuous Stream Processors (Stream A) \rightarrow (Stream B)
represented by nested fixed points.

$\text{StrProc}(A, B) : \text{Set}$

$\text{write} : B \rightarrow \text{StrProc}(A, B) \rightarrow \text{StrProc}(A, B)$

$\text{read} : (A \rightarrow \text{StrProc}(A, B)) \rightarrow \text{StrProc}(A, B)$

write coinductive, read inductive.

$\text{eval} : \text{StrProc}(A, B) \rightarrow \text{Stream } A \rightarrow \text{Stream } B$

$\text{eval}(\text{write } b p) s = b :: \text{eval } p s$

$\text{eval}(\text{read } f)(a :: s) = \text{eval}(f a) s$

Simultaneous Coinductive-Recursive Definition

- Coinductive $\text{StrProc}(A, B) : \text{Type}$

$\text{write} : B \rightarrow \text{StrProc}(A, B) \rightarrow \text{StrProc}(A, B)$

$\text{read} : (A \rightarrow \text{StrProc}(A, B)) \rightarrow \text{StrProc}(A, B)$

Simultaneous Coinductive-Recursive Definition

- Coinductive $\text{StrProc}(A, B) : \text{Type}$

$$\text{write} : B \rightarrow \text{StrProc}(A, B) \rightarrow \text{StrProc}(A, B)$$

$$\text{read} : (A \rightarrow \text{StrProc}(A, B)) \rightarrow \text{StrProc}(A, B)$$

- Simultaneous $\text{readWf} : \text{StrProc}(A, B) \rightarrow \text{Prop}$

$$\text{readWf}(\text{write } b s) = \top$$

$$\text{readWf}(\text{read } f) = \forall x : A. \text{readWf}(f x)$$

That's a large type. But alternatively ...

Simultaneous Coinductive-Recursive Definition

- Coinductive $\text{StrProc}(A, B) : \text{Set}$

$$\text{write} : B \rightarrow \text{StrProc}(A, B) \rightarrow \text{StrProc}(A, B)$$

$$\text{read} : (A \rightarrow \text{StrProc}(A, B)) \rightarrow \text{StrProc}(A, B)$$

- Simultaneous $\text{readTree} : \text{StrProc}(A, B) \rightarrow \text{Tree } A$

$$\text{readTree}(\text{write } b s) = \text{leaf}$$

$$\text{readTree}(\text{read } f) = \text{node}(\lambda x. \text{readTree}(f x))$$

(Tree A : well-founded A -branching trees.)

Perspective

- Induction-recursion:
Direct implementation of advanced data types

Perspective

- Induction-recursion:
Direct implementation of advanced data types
- Leftist heaps

Perspective

- Induction-recursion:
Direct implementation of advanced data types
- Leftist heaps
- Dybjer/Setzer codes

Perspective

- Induction-recursion:
Direct implementation of advanced data types
- Leftist heaps
- Dybjer/Setzer codes
- Coinductive version (Wander types) leads to a realization of mixed induction-coinduction

Perspective

- Induction-recursion:
Direct implementation of advanced data types
- Leftist heaps
- Dybjer/Setzer codes
- Coinductive version (Wander types) leads to a realization of mixed induction-coinduction
- Data with fine control on structural properties