Wander Types

Realizing Data Structures Through Coinduction-Recursion

Venanzio Capretta
University of Nottingham

Shonan Meeting on Dependently Typed Programming, 2011
Shonan Viallage Center, Japan

IR definitions

Inductive-Recursive (IR) definitions

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

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

mutually dependent

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

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

mutually dependent

- The constructors of T can use f

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

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

mutually dependent

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

IR definitions

Inductive-Recursive (IR) definitions

Simultaneously define

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

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 $A, \preccurlyeq$

Advanced Data Structures

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

$$\text{Heap} : \text{Set}$$

Advanced Data Structures

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

$$\mathsf{Heap} : \mathsf{Set}$$

$$\mathsf{empty} : \mathsf{Heap}$$
$$\mathsf{isEmpty} : \mathsf{Heap} \to \mathbb{B}$$

Advanced Data Structures

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

$$\mathsf{Heap} : \mathsf{Set}$$

$$\mathsf{empty} : \mathsf{Heap}$$
$$\mathsf{isEmpty} : \mathsf{Heap} \to \mathbb{B}$$
$$\mathsf{insert} : A \to \mathsf{Heap} \to \mathsf{Heap}$$

Advanced Data Structures

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

$$\begin{aligned}
&\mathsf{Heap} : \mathsf{Set} \\
&\\
&\mathsf{empty} : \mathsf{Heap} \\
&\mathsf{isEmpty} : \mathsf{Heap} \to \mathbb{B} \\
&\mathsf{insert} : A \to \mathsf{Heap} \to \mathsf{Heap} \\
&\mathsf{findMin} : \mathsf{Heap} \to A
\end{aligned}$$

Advanced Data Structures

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

$$\begin{aligned}
&\mathsf{Heap} : \mathsf{Set} \\
&\\
&\mathsf{empty} : \mathsf{Heap} \\
&\mathsf{isEmpty} : \mathsf{Heap} \to \mathbb{B} \\
&\mathsf{insert} : A \to \mathsf{Heap} \to \mathsf{Heap} \\
&\mathsf{findMin} : \mathsf{Heap} \to A \\
&\mathsf{deleteMin} : \mathsf{Heap} \to \mathsf{Heap}
\end{aligned}$$

Advanced Data Structures

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

$$\mathsf{Heap} : \mathsf{Set}$$

$$\begin{aligned}
&\mathsf{empty} : \mathsf{Heap} \\
&\mathsf{isEmpty} : \mathsf{Heap} \to \mathbb{B} \\
&\mathsf{insert} : A \to \mathsf{Heap} \to \mathsf{Heap} \\
&\mathsf{findMin} : \mathsf{Heap} \to A \\
&\mathsf{deleteMin} : \mathsf{Heap} \to \mathsf{Heap}
\end{aligned}$$

$$\mathsf{merge} : \mathsf{Heap} \to \mathsf{Heap} \to \mathsf{Heap}$$

Advanced Data Structures

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

$$\begin{aligned}
&\mathsf{Heap} : \mathsf{Set} \\
&\\
&\mathsf{empty} : \mathsf{Heap} \\
&\mathsf{isEmpty} : \mathsf{Heap} \to \mathbb{B} \\
&\mathsf{insert} : A \to \mathsf{Heap} \to \mathsf{Heap} \\
&\mathsf{findMin} : \mathsf{Heap} \to A \\
&\mathsf{deleteMin} : \mathsf{Heap} \to \mathsf{Heap}
\end{aligned}$$

$$\mathsf{merge} : \mathsf{Heap} \to \mathsf{Heap} \to \mathsf{Heap}$$

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

- $\mathsf{IHeap} : \mathsf{Set}$
- $\mathsf{rootor} : \mathsf{IHeap} \to A \to A$

Inductive-Recursive Leftist Heaps

- $\mathsf{IHeap} : \mathsf{Set}$
- $\mathsf{rootor} : \mathsf{IHeap} \to A \to A$
- $\mathsf{rank} : \mathsf{IHeap} \to \mathbb{N}$

Inductive-Recursive Leftist Heaps

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

  $\mathsf{leaf} : \mathsf{IHeap}$

- $\mathsf{rootor} : \mathsf{IHeap} \to A \to A$

  $\mathsf{rootor}\,\mathsf{leaf} = \mathsf{id}$

- $\mathsf{rank} : \mathsf{IHeap} \to \mathbb{N}$

  $\mathsf{rank}\,\mathsf{leaf} = 0$

Inductive-Recursive Leftist Heaps

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

$$\begin{aligned}
&\mathsf{leaf} : \mathsf{IHeap} \\
&\mathsf{node} : \ (a : A; t_1, t_2 : \mathsf{IHeap}) \\
&\qquad \to a \preccurlyeq (\mathsf{rootor}\ t_1\ a) \to a \preccurlyeq (\mathsf{rootor}\ t_2\ a) \\
&\qquad \to (\mathsf{rank}\ t_2) \leq (\mathsf{rank}\ t_1) \to \mathsf{IHeap}
\end{aligned}$$

- $\mathsf{rootor} : \mathsf{IHeap} \to A \to A$

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

- $\mathsf{rank} : \mathsf{IHeap} \to \mathbb{N}$

$$\mathsf{rank}\ \mathsf{leaf} = 0$$

Inductive-Recursive Leftist Heaps

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

$$\begin{aligned}
&\mathsf{leaf} : \mathsf{IHeap} \\
&\mathsf{node} : \ (a : A; t_1, t_2 : \mathsf{IHeap}) \\
&\qquad \to a \preccurlyeq (\mathsf{rootor}\ t_1\ a) \to a \preccurlyeq (\mathsf{rootor}\ t_2\ a) \\
&\qquad \to (\mathsf{rank}\ t_2) \leq (\mathsf{rank}\ t_1) \to \mathsf{IHeap}
\end{aligned}$$

- $\mathsf{rootor} : \mathsf{IHeap} \to A \to A$

$$\begin{aligned}
\mathsf{rootor}\ \mathsf{leaf} &= \mathsf{id} \\
\mathsf{rootor}\ (\mathsf{node}\ a\ t_1\ t_2\ \ldots) &= \lambda x.a
\end{aligned}$$

- $\mathsf{rank} : \mathsf{IHeap} \to \mathbb{N}$

$$\mathsf{rank}\ \mathsf{leaf} = 0$$

Inductive-Recursive Leftist Heaps

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

$$\begin{aligned}
&\mathsf{leaf} : \mathsf{IHeap} \\
&\mathsf{node} : \ (a : A; t_1, t_2 : \mathsf{IHeap}) \\
&\qquad \to a \preccurlyeq (\mathsf{rootor}\ t_1\ a) \to a \preccurlyeq (\mathsf{rootor}\ t_2\ a) \\
&\qquad \to (\mathsf{rank}\ t_2) \leq (\mathsf{rank}\ t_1) \to \mathsf{IHeap}
\end{aligned}$$

- $\mathsf{rootor} : \mathsf{IHeap} \to A \to A$

$$\begin{aligned}
\mathsf{rootor}\ \mathsf{leaf} &= \mathsf{id} \\
\mathsf{rootor}\ (\mathsf{node}\ a\ t_1\ t_2\ \ldots) &= \lambda x.a
\end{aligned}$$

- $\mathsf{rank} : \mathsf{IHeap} \to \mathbb{N}$

$$\begin{aligned}
\mathsf{rank}\ \mathsf{leaf} &= 0 \\
\mathsf{rank}\ (\mathsf{node}\ a\ t_1\ t_2\ \ldots) &= 1 + \mathsf{rank}\ t_2
\end{aligned}$$

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:

$$\mathsf{Elem}_c : \mathsf{Set}$$
$$\mathsf{fun}_c : \mathsf{Elem}_c \to D$$

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:

$$\mathrm{Elem}_c : \mathrm{Set}$$
$$\mathrm{fun}_c : \mathrm{Elem}_c \to D$$

- 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

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

Constructors of IR codes

- A single element with a given value

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

- A constructor with a non-recursive argument

$$\begin{array}{l} c = \text{choose } A\ u \\ \quad A : \text{Set} \\ \quad u : A \to \text{IR} D \end{array}$$

Constructors of IR codes

- A single element with a given value

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

- A constructor with a non-recursive argument

$$\begin{array}{lll} c = \mathsf{choose}\ A\ u & \rightsquigarrow & \mathsf{Elem}_c \simeq (a : A) \times \mathsf{Elem}_{(u\ a)} \\ A : \mathsf{Set} & & \mathsf{fun}_c\langle a, x\rangle = \mathsf{fun}_{(u\ a)}(x) \\ u : A \to \mathsf{IR}D & & \end{array}$$

Constructors of IR codes

- A single element with a given value

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

- A constructor with a non-recursive argument

$$\begin{array}{rcl} c = \mathsf{choose}\ A\ u & \rightsquigarrow & \mathsf{Elem}_c \simeq (a : A) \times \mathsf{Elem}_{(u\,a)} \\ A : \mathsf{Set} & & \mathsf{fun}_c\langle a, x\rangle = \mathsf{fun}_{(u\,a)}(x) \\ u : A \to \mathsf{IR}D & & \end{array}$$

- A constructor with recursive arguments

$$\begin{array}{l} c = \mathsf{recurse}\ I\ v \\ I : \mathsf{Set} \\ v : (I \to D) \to \mathsf{IR}D \end{array}$$

Constructors of IR codes

- A single element with a given value

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

- A constructor with a non-recursive argument

$$\begin{array}{l} c = \mathsf{choose}\ A\ u \\ \quad A : \mathsf{Set} \\ \quad u : A \to \mathsf{IR}D \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \mathsf{Elem}_c \simeq (a : A) \times \mathsf{Elem}_{(u\,a)} \\ \mathsf{fun}_c\langle a, x\rangle = \mathsf{fun}_{(u\,a)}(x) \end{array}$$

- A constructor with recursive arguments

$$\begin{array}{l} c = \mathsf{recurse}\ I\ v \\ \quad I : \mathsf{Set} \\ \quad v : (I \to D) \to \mathsf{IR}D \end{array} \quad \rightsquigarrow \quad \begin{array}{l} \mathsf{Elem}_c \simeq (t : I \to \mathsf{Elem}_c) \times \mathsf{Elem}_{v(\mathsf{fun}_c \circ t)} \\ \mathsf{fun}_c\langle t, y\rangle = \mathsf{fun}_{v(\mathsf{fun}_c \circ t)}(y) \end{array}$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N})$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$\begin{aligned} &\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N}) \\ &= \quad \mathsf{element}\,\langle \mathsf{id}, 0 \rangle + \end{aligned}$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$\begin{aligned} &\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N}) \\ &= \quad \mathsf{element}\,\langle \mathsf{id}, 0\rangle + \\ &\qquad \mathsf{choose}\,A\,\lambda a. \end{aligned}$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$
\begin{aligned}
&\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N}) \\
&= \quad \mathsf{element}\,\langle \mathsf{id}, 0\rangle + \\
&\qquad \mathsf{choose}\,A\,\lambda a. \\
&\qquad\quad \mathsf{recurse}\,\mathbb{B}\,\lambda\langle \mathrm{root}_1, \mathrm{rank}_1, \mathrm{root}_2, \mathrm{rank}_2\rangle.
\end{aligned}
$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$
\begin{aligned}
&\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N}) \\
&= \quad \mathsf{element}\,\langle \mathsf{id}, 0\rangle + \\
&\qquad \mathsf{choose}\,A\,\lambda a. \\
&\qquad\quad \mathsf{recurse}\,\mathbb{B}\,\lambda\langle \mathrm{root}_1, \mathrm{rank}_1, \mathrm{root}_2, \mathrm{rank}_2\rangle. \\
&\qquad\qquad \mathsf{choose}\,(a \preccurlyeq (\mathrm{root}_1\ a) \wedge a \preccurlyeq (\mathrm{root}_2\ a) \wedge (\mathrm{rank}_2 \leq \mathrm{rank}_1))
\end{aligned}
$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$
\begin{aligned}
&\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N}) \\
&= \quad \mathsf{element}\,\langle \mathsf{id}, 0\rangle + \\
&\qquad \mathsf{choose}\,A\,\lambda a. \\
&\qquad\quad \mathsf{recurse}\,\mathbb{B}\,\lambda\langle \mathrm{root}_1, \mathrm{rank}_1, \mathrm{root}_2, \mathrm{rank}_2\rangle. \\
&\qquad\qquad \mathsf{choose}\,(a \preccurlyeq (\mathrm{root}_1\ a) \wedge a \preccurlyeq (\mathrm{root}_2\ a) \wedge (\mathrm{rank}_2 \leq \mathrm{rank}_1)) \\
&\qquad\qquad\quad \lambda\_.\,\mathsf{element}\,\langle \lambda x.a, 1 + \mathrm{rank}_2\rangle
\end{aligned}
$$

Code for Leftist Heaps

The Leftist Heap type has the following IR code:

$$
\begin{aligned}
&\mathsf{lhcode} : \mathsf{IR}\,((A \to A) \times \mathbb{N}) \\
&= \quad \mathsf{element}\,\langle \mathsf{id}, 0\rangle + \\
&\qquad \mathsf{choose}\,A\,\lambda a. \\
&\qquad\quad \mathsf{recurse}\,\mathbb{B}\,\lambda\langle \mathrm{root}_1, \mathrm{rank}_1, \mathrm{root}_2, \mathrm{rank}_2\rangle. \\
&\qquad\qquad \mathsf{choose}\,(a \preccurlyeq (\mathrm{root}_1\ a) \wedge a \preccurlyeq (\mathrm{root}_2\ a) \wedge (\mathrm{rank}_2 \leq \mathrm{rank}_1)) \\
&\qquad\qquad\quad \lambda\_.\,\mathsf{element}\,\langle \lambda x.a, 1 + \mathrm{rank}_2\rangle
\end{aligned}
$$

$+$: separate different constructors, non-dependent choose
Elements of $\mathbb{B} \to (A \to A) \times \mathbb{N}$:
quadruples $\langle \mathrm{root}_1, \mathrm{rank}_1, \mathrm{root}_2, \mathrm{rank}_2\rangle$

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 $\mathsf{Set}\downarrow D$ 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 $\mathsf{Set}\downarrow D$ has:

Objects: pairs $\langle X, f \rangle$
where $X : Set$ $\quad$ $f : X \to D$

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 $\mathsf{Set}\downarrow D$ has:

Objects: pairs $\langle X, f\rangle$
where $X : Set \qquad f : X \to D$

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

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 $\mathsf{Set} \downarrow D$ has:

Objects: pairs $\langle X, f \rangle$
where $X : Set \qquad f : X \to D$

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

Functor $\Theta : Set \downarrow D \to Set \downarrow D$
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

$$\mathsf{zero} : \mathsf{ZeroOne} \to \mathsf{ZeroOne}$$
$$\mathsf{one} : \mathsf{ZeroOne} \to \mathsf{ZeroOne}$$

Mixed Induction-Coinduction

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

- CoInductive ZeroOne : Set

$$\mathsf{zero} : \mathsf{ZeroOne} \to \mathsf{ZeroOne}$$
$$\mathsf{one} : \mathsf{ZeroOne} \to \mathsf{ZeroOne}$$

- Simultaneously $\mathsf{count1} : \mathsf{ZeroOne} \to \mathbb{N}$

$$\mathsf{count1}\,(\mathsf{zero}\,s) = 0$$
$$\mathsf{count1}\,(\mathsf{one}\,s) = 1 + \mathsf{count1}\,s$$

Mixed Induction-Coinduction

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

- CoInductive ZeroOne : Set

$$\mathsf{zero} : \mathsf{ZeroOne} \to \mathsf{ZeroOne}$$
$$\mathsf{one} : \mathsf{ZeroOne} \to \mathsf{ZeroOne}$$

- Simultaneously $\mathsf{count1} : \mathsf{ZeroOne} \to \mathbb{N}$

$$\mathsf{count1}\,(\mathsf{zero}\,s) = 0$$
$$\mathsf{count1}\,(\mathsf{one}\,s) = 1 + \mathsf{count1}\,s$$

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

$$\mathsf{count0} : \mathsf{ZeroOne} \to \mathbb{N}$$
$$\mathsf{count0}\,(\mathsf{zero}\,s) = 1 + \mathsf{count0}\,s$$
$$\mathsf{count0}\,(\mathsf{one}\,s) = 0$$

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 $\quad \mathrm{ZigZag} : \mathrm{Set}$
Simultaneously $\quad \mathrm{zigs} : \mathrm{ZigZag} \to \mathbb{N}$
$\quad \mathrm{zags} : \mathrm{ZigZag} \to \mathbb{N}$

No-zigzag type

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

CoInductive $\quad ZigZag : Set$
Simultaneously $\quad zigs : ZigZag \to \mathbb{N}$
$\quad zags : ZigZag \to \mathbb{N}$

Constructors
$\quad zzlf : ZigZag$
$\quad zznd : ZigZag \to ZigZag \to ZigZag$

No-zigzag type

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

CoInductive $\mathsf{ZigZag} : \mathsf{Set}$
Simultaneously $\mathsf{zigs} : \mathsf{ZigZag} \to \mathbb{N}$
$\mathsf{zags} : \mathsf{ZigZag} \to \mathbb{N}$
Constructors
$\mathsf{zzlf} : \mathsf{ZigZag}$
$\mathsf{zznd} : \mathsf{ZigZag} \to \mathsf{ZigZag} \to \mathsf{ZigZag}$
Equations
$\mathsf{zigs}\,\mathsf{zzlf} = 0$ $\quad$ $\mathsf{zigs}\,(\mathsf{zznd}\,t_1\,t_2) = 1 + \mathsf{zags}\,t_1$
$\mathsf{zags}\,\mathsf{zzlf} = 0$ $\quad$ $\mathsf{zags}\,(\mathsf{zznd}\,t_1\,t_2) = 1 + \mathsf{zigs}\,t_2$

Stream Processors

Other example [Ghani/Hancock/Pattinson 2009]
Continuous Stream Processors $(\text{Stream}\, A) \to (\text{Stream}\, B)$ represented by nested fixed points.

Stream Processors

Other example [Ghani/Hancock/Pattinson 2009]
Continuous Stream Processors $(\text{Stream}\, A) \to (\text{Stream}\, B)$ represented by nested fixed points.

$$\begin{aligned} &\text{StrProc}\,(A, B) : \text{Set} \\ &\quad \text{write} : B \to \text{StrProc}\,(A, B) \to \text{StrProc}\,(A, B) \\ &\quad \text{read} : (A \to \text{StrProc}\,(A, B)) \to \text{StrProc}\,(A, B) \end{aligned}$$

write coinductive, read inductive.

Stream Processors

Other example [Ghani/Hancock/Pattinson 2009]
Continuous Stream Processors $(\text{Stream}\, A) \to (\text{Stream}\, B)$
represented by nested fixed points.

$$\begin{aligned}&\text{StrProc}\,(A, B) : \text{Set}\\ &\quad \text{write} : B \to \text{StrProc}\,(A, B) \to \text{StrProc}\,(A, B)\\ &\quad \text{read} : (A \to \text{StrProc}\,(A, B)) \to \text{StrProc}\,(A, B)\end{aligned}$$

write coinductive, read inductive.

$$\begin{aligned}&\text{eval} : \text{StrProc}\,(A, B) \to \text{Stream}\, A \to \text{Stream}\, B\\ &\text{eval}\,(\text{write}\, b\, p)\, s = b :: \text{eval}\, p\, s\\ &\text{eval}\,(\text{read}\, f)\,(a :: s) = \text{eval}\,(f\ a)\, s\end{aligned}$$

Simultaneous Coinductive-Recursive Definition

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

$$\mathsf{write} : B \to \mathsf{StrProc}\,(A, B) \to \mathsf{StrProc}\,(A, B)$$
$$\mathsf{read} : (A \to \mathsf{StrProc}\,(A, B)) \to \mathsf{StrProc}\,(A, B)$$

Simultaneous Coinductive-Recursive Definition

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

$$\mathsf{write} : B \to \mathsf{StrProc}\,(A, B) \to \mathsf{StrProc}\,(A, B)$$
$$\mathsf{read} : (A \to \mathsf{StrProc}\,(A, B)) \to \mathsf{StrProc}\,(A, B)$$

- Simultaneous $\mathsf{readWf} : \mathsf{StrProc}\,(A, B) \to \mathsf{Prop}$

$$\mathsf{readWf}\,(\mathsf{write}\,b\,s) = \top$$
$$\mathsf{readWf}\,(\mathsf{read}\,f) = \forall x : A.\mathsf{readWf}\,(f\,x)$$

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

Simultaneous Coinductive-Recursive Definition

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

$$\mathsf{write} : B \to \mathsf{StrProc}\,(A, B) \to \mathsf{StrProc}\,(A, B)$$
$$\mathsf{read} : (A \to \mathsf{StrProc}\,(A, B)) \to \mathsf{StrProc}\,(A, B)$$

- Simultaneous $\mathsf{readTree} : \mathsf{StrProc}\,(A, B) \to \mathsf{Tree}\,A$

$$\mathsf{readTree}\,(\mathsf{write}\,b\,s) = \mathsf{leaf}$$
$$\mathsf{readTree}\,(\mathsf{read}\,f) = \mathsf{node}\,(\lambda x.\mathsf{readTree}\,(f\,x))$$

(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