Induction-Recursion
A polymorphic representation

Venanzio Capretta
University of Nottingham

DTP 2011, Nijmegen, The Netherlands

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 [Dybjier 2001, Dybjier/Setzer 1999]

Type Universes

- U : Type
  (large) type of codes for (small) types

Type Universes

- $\mathsf{U} : \mathsf{Type}$
  (large) type of codes for (small) types

- $\mathsf{El} : \mathsf{U} \to \mathsf{Type}$
  decoding function, giving type of elements of codes

Type Universes

- $U : \mathsf{Type}$

$$\mathsf{nat} : U$$

- $\mathsf{El} : U \to \mathsf{Type}$

$$\mathsf{El}\,\mathsf{nat} = \mathbb{N}$$

Type Universes

- $\mathsf{U} : \mathsf{Type}$

$$\mathsf{nat} : \mathsf{U}$$
$$\mathsf{prod} : \mathsf{U} \to \mathsf{U} \to \mathsf{U}$$

- $\mathsf{El} : \mathsf{U} \to \mathsf{Type}$

$$\mathsf{El}\,\mathsf{nat} = \mathbb{N}$$
$$\mathsf{El}\,(\mathsf{prod}\,a\,b) = (\mathsf{El}\,a) \times (\mathsf{El}\,b)$$

Type Universes

- $\mathsf{U} : \mathsf{Type}$

$$\begin{aligned}\mathsf{nat} &: \mathsf{U}\\ \mathsf{prod} &: \mathsf{U} \to \mathsf{U} \to \mathsf{U}\\ \mathsf{sum} &: \mathsf{U} \to \mathsf{U} \to \mathsf{U}\end{aligned}$$

- $\mathsf{El} : \mathsf{U} \to \mathsf{Type}$

$$\begin{aligned}\mathsf{El}\,\mathsf{nat} &= \mathbb{N}\\ \mathsf{El}\,(\mathsf{prod}\,a\,b) &= (\mathsf{El}\,a) \times (\mathsf{El}\,b)\\ \mathsf{El}\,(\mathsf{sum}\,a\,b) &= (\mathsf{El}\,a) + (\mathsf{El}\,b)\end{aligned}$$

Type Universes

- $\mathsf{U} : \mathsf{Type}$

$$\begin{aligned}
&\mathsf{nat} : \mathsf{U}\\
&\mathsf{prod} : \mathsf{U} \to \mathsf{U} \to \mathsf{U}\\
&\mathsf{sum} : \mathsf{U} \to \mathsf{U} \to \mathsf{U}\\
&\mathsf{arrow} : \mathsf{U} \to \mathsf{U} \to \mathsf{U}
\end{aligned}$$

- $\mathsf{El} : \mathsf{U} \to \mathsf{Type}$

$$\begin{aligned}
\mathsf{El}\,\mathsf{nat} &= \mathbb{N}\\
\mathsf{El}\,(\mathsf{prod}\,a\,b) &= (\mathsf{El}\,a) \times (\mathsf{El}\,b)\\
\mathsf{El}\,(\mathsf{sum}\,a\,b) &= (\mathsf{El}\,a) + (\mathsf{El}\,b)\\
\mathsf{El}\,(\mathsf{arrow}\,a\,b) &= (\mathsf{El}\,a) \to (\mathsf{El}\,b)
\end{aligned}$$

Type Universes

- $\mathsf{U} : \mathsf{Type}$

$$\ldots$$
$$\mathsf{pi} : (a : \mathsf{U}; b : \mathsf{El}\, a \to \mathsf{U}) \to \mathsf{U}$$

- $\mathsf{El} : \mathsf{U} \to \mathsf{Type}$

$$\ldots$$
$$\mathsf{El}\,(\mathsf{pi}\, a\, b) = \Pi x : \mathsf{El}\, a.\mathsf{El}\,(b\, x)$$

Type Universes

- $\mathsf{U} : \mathsf{Type}$

$$\begin{aligned}&\ldots\\ &\mathsf{pi} : (a : \mathsf{U}; b : \mathsf{El}\, a \to \mathsf{U}) \to \mathsf{U}\\ &\mathsf{sig} : (a : \mathsf{U}; b : \mathsf{El}\, a \to \mathsf{U}) \to \mathsf{U}\end{aligned}$$

- $\mathsf{El} : \mathsf{U} \to \mathsf{Type}$

$$\begin{aligned}&\ldots\\ &\mathsf{El}\,(\mathsf{pi}\, a\, b) = \Pi x : \mathsf{El}\, a.\mathsf{El}\,(b\, x)\\ &\mathsf{El}\,(\mathsf{sig}\, a\, b) = \Sigma x : \mathsf{El}\, a.\mathsf{El}\,(b\, x)\end{aligned}$$

Advanced Data Structures

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

Advanced Data Structures

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

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

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 $A, \preccurlyeq$

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

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

Advanced Data Structures

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

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

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

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 $A, \preccurlyeq$

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

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

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

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

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

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

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

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

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

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

Inductive-Recursive Leftist Heaps

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

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

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

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

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

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

Inductive-Recursive Leftist Heaps

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

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

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

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

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

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

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 $g : X_1 \to X_2 \qquad f_2 \circ g = f_1$

In the case of leftist heaps:
$D = (A \to A) \times \mathbb{N}$
$\langle \mathsf{IHeap}, \langle \mathsf{rootor}, \mathsf{rank}\rangle\rangle$ object of $Set\downarrow D$

Slice Functors

A functor $\Theta : Set \downarrow D \to Set \downarrow D$ has three components:

Slice Functors

A functor $\Theta : Set \downarrow D \to Set \downarrow D$ has three components:

- Set part, for every $\langle X, f \rangle$: $F\ X\ f : \mathrm{Set}$

Slice Functors

A functor $\Theta : Set \downarrow D \to Set \downarrow D$ has three components:

- Set part, for every $\langle X, f \rangle$: $F\ X\ f : \mathrm{Set}$
- Arrow part: $e\ X\ f : F\ X\ f \to D$

Slice Functors

A functor $\Theta : Set \downarrow D \to Set \downarrow D$ has three components:

- Set part, for every $\langle X, f \rangle$: $F\ X\ f : \mathrm{Set}$
- Arrow part: $e\ X\ f : F\ X\ f \to D$
- Mapping: for every $g : \langle X_1, f_1 \rangle \to \langle X_2, f_2 \rangle$: $\Theta\ g : F\ X_1\ f_1 \to F\ X_2\ f_2$

satisfying some equalities

Slice Functors

A functor $\Theta : Set \downarrow D \to Set \downarrow D$ has three components:

- Set part, for every $\langle X, f \rangle$: $F\ X\ f : \mathrm{Set}$
- Arrow part: $e\ X\ f : F\ X\ f \to D$
- Mapping: for every $g : \langle X_1, f_1 \rangle \to \langle X_2, f_2 \rangle$: $\Theta\ g : F\ X_1\ f_1 \to F\ X_2\ f_2$

satisfying some equalities

Note:

No underlying functor on Set: F essentially depends on f

Initial Slice Algebras

A Θ-algebra is a triple $\langle X, f, g \rangle$

$$X : \mathrm{Set}$$
$$f : X \to D$$
$$g : \langle F\, X\, f, e\, X\, f \rangle \to \langle X, f \rangle$$

The inductive-recursive pair is the initial Θ-algebra

Initial Slice Algebras

A Θ-algebra is a triple $\langle X, f, g \rangle$

$$X : \mathrm{Set}$$
$$f : X \to D$$
$$g : \langle F\, X\, f, e\, X\, f \rangle \to \langle X, f \rangle$$

The inductive-recursive pair is the initial Θ-algebra

How to represent ind-rec in a system (COQ) not supporting it?

Second Order Encoding

Future Work:

Second-order representation of induction recursion [VC 2004]

Similar to polymorphic definition of (weak) inductive types:

[Böhm/Berarducci 1985, Girard 1989]

$$\mu\Theta = \Pi\langle X, f\rangle : \mathsf{Set}\downarrow D.(\Theta\langle X, f\rangle \to \langle X, f\rangle)$$

The inductive-recursive type is the product of all Θ-algebras

[Also coinductive version, Wraith 1989, Geuvers 1992]

Second Order Encoding

Future Work:

Second-order representation of induction recursion [VC 2004]

Similar to polymorphic definition of (weak) inductive types:

[Böhm/Berarducci 1985, Girard 1989]

$$\mu\Theta = \Pi\langle X, f\rangle : \mathsf{Set}\downarrow D.(\Theta\langle X, f\rangle \to \langle X, f\rangle)$$

The inductive-recursive type is the product of all Θ-algebras

[Also coinductive version, Wraith 1989, Geuvers 1992]

Better Way [Conor McBride]

Inductive family indexed on the result of the function.

Wander Types: Final Θ-algebra

Coinductive version of leftist heaps?

Wander Types: Final Θ-algebra

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 Θ-algebra

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 Θ-algebra

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 $\mathsf{ZeroOne} : \mathsf{Set}$

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

- 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.

$$\begin{array}{ll} \text{CoInductive} & \mathsf{ZigZag} : \mathsf{Set} \\ \text{Simultaneously} & \mathsf{zigs} : \mathsf{ZigZag} \to \mathbb{N} \\ & \mathsf{zags} : \mathsf{ZigZag} \to \mathbb{N} \end{array}$$

No-zigzag type

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

$$\begin{array}{ll}\text{CoInductive} & \mathsf{ZigZag} : \mathsf{Set} \\ \text{Simultaneously} & \mathsf{zigs} : \mathsf{ZigZag} \to \mathbb{N} \\ & \mathsf{zags} : \mathsf{ZigZag} \to \mathbb{N} \\ \text{Constructors} & \\ \quad \mathsf{zzlf} : \mathsf{ZigZag} & \\ \quad \mathsf{zznd} : \mathsf{ZigZag} \to \mathsf{ZigZag} \to \mathsf{ZigZag} & \end{array}$$

No-zigzag type

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

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

Perspective

- Induction-recursion: direct implementation of advanced data types

Perspective

- Induction-recursion: direct implementation of advanced data types
- Realization by polymorphic quantification in slice category

Perspective

- Induction-recursion: direct implementation of advanced data types
- Realization by polymorphic quantification in slice category
- Coinductive version (Wander types) leads to a realization of mixed induction-coinduction

Perspective

- Induction-recursion: direct implementation of advanced data types
- Realization by polymorphic quantification in slice category
- Coinductive version (Wander types) leads to a realization of mixed induction-coinduction
- Data with fine control on structural properties

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

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

($\mathsf{Tree}\,A$: well-founded A-branching trees.)