Coinductive Reasoning in Dependent Type Theory - Copatterns, Objects, Processes

Anton Setzer
http://www.cs.swan.ac.uk/~csetzer/index.html

Swansea University
http://www.swansea.ac.uk/compsci/

(Part on Processes presented by Bashar Igried on separate slides, Remaining parts with contributions by Peter Hancock, Andreas Abel, Brigitte Pientka, David Thibodeau)

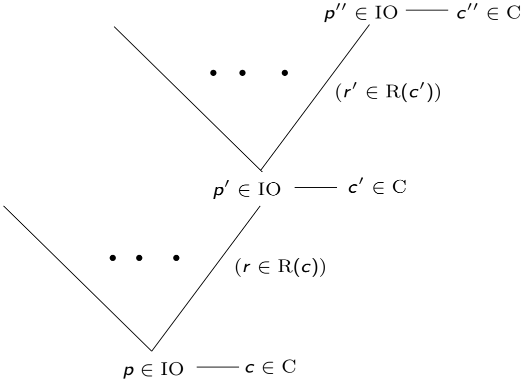
Talk given at JAIST, Japan

6 September 2016

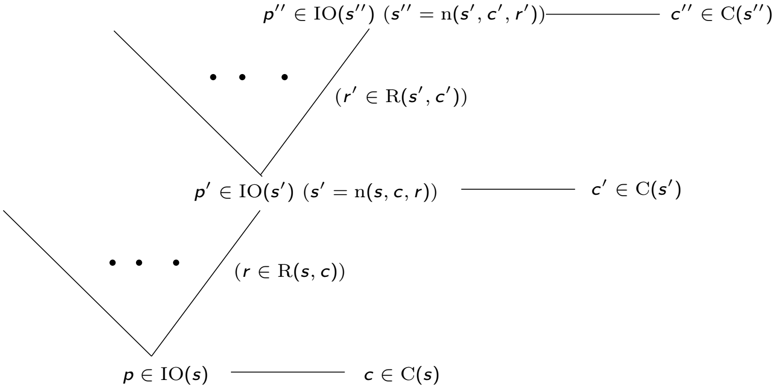
Need for Coinductive Proofs

- In the beginning of computing, computer programs were batch programs.
  - One input one output
  - Correct programs correspond to **well-founded** structures (termination).
- Nowadays most programs are interactive;
  - A possibly infinite sequence of interactions, often concurrently.
  - Correspond to **non-well-founded** structures.
  - For instance non-concurrent computations can be represented as **IO-trees**.
  - A simple form of objects in object-oriented programs can be represented as non-well-founded trees.

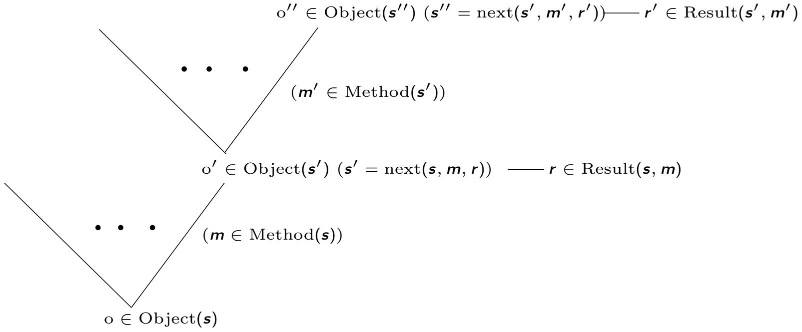
IO-Trees (Non-State Dependent)



IO-Trees State Dependent



Objects (State Dependent)

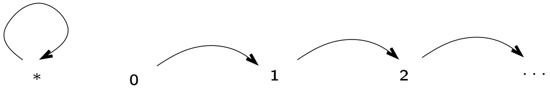


Need for Good Framework for Coinductive Structures

- Non-well-founded trees are defined coinductively.
- Relations between coinductive structures are coinductively defined
- Need suitable notion of reasoning coinductively.

Coinductive Proofs

- Reasoning about bisimulation is often very formalist. Consider an unlabelled Transition system:

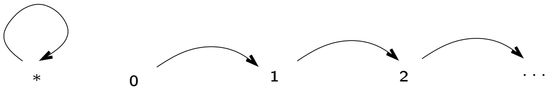


- For showing $* \sim n$ one defines
  - $R := \{(*, n) \mid n \in \mathbb{N}\}$
  - Shows that R is a bisimulation relation:
    - Let $(a, b) \in R$. Then $a = *$, $b = n \in \mathbb{N}$ for some n.
    - Assume $a = * \longrightarrow a'$.
      Then $a' = *$. We have $b = n \longrightarrow n + 1$ and $(*, n + 1) \in R$.
    - Assume $b = n \longrightarrow b'$.
      Then $b' = n + 1$. We have $a = * \longrightarrow *$ and $(*, n + 1) \in R$.
  - Therefore $x \sim y$ for $(x, y) \in R$.

Comparison

- Above is similar when carrying an inductive proof, e.g. of $\varphi := \forall n, m, k.(n+m)+k = n+(m+k)$ to defining
$$A := \{k \mid (n+m)+k = n+(m+k)\}$$
and showing that A is closed under 0 and successor.
- Instead we prove φ by induction on k using in the successor case the IH.
- Both proofs amount the same, but the second one would be far more difficult to teach and cumbersome to use.

Desired Coinductive Proof



- We show $\forall n \in \mathbb{N}. * \sim n$ by coinduction on $\sim$.
  - Assume $* \longrightarrow x$. We need to find y s.t. $n \longrightarrow y$ and $x \sim y$. Choose $y = n + 1$. By **co-IH** $* \sim n + 1$.
  - Assume $n \longrightarrow y$. We need to find x s.t. $* \longrightarrow x$ and $x \sim y$. Choose $x = *$. By **co-IH** $* \sim n + 1$.
- In essence same proof, but hopefully easier to teach and use.

Desired Coinductive Proof for Streams

- Consider $\mathrm{Stream} : \mathrm{Set}$ given coinductively by

$$\begin{array}{lcll} \mathrm{head} & : & \mathrm{Stream} \to \mathbb{N} & , \\ \mathrm{tail} & : & \mathrm{Stream} \to \mathrm{Stream} & . \end{array}$$

- Consider 3 versions of the stream $n, n+1, n+2, \ldots$

$$\begin{array}{lclclcl} \multicolumn{7}{l}{\mathrm{inc}, \mathrm{inc}', \mathrm{inc}'' : \mathbb{N} \to \mathrm{Stream}} \\ \mathrm{head}(\mathrm{inc}(n)) & = & \mathrm{head}(\mathrm{inc}'(n)) & = & \mathrm{head}(\mathrm{inc}''(n)) & = & n \\ \mathrm{tail}(\mathrm{inc}(n)) & = & \mathrm{inc}(n+1) \\ \mathrm{tail}(\mathrm{inc}'(n)) & = & \mathrm{inc}''(n+1) \\ \mathrm{tail}(\mathrm{inc}''(n)) & = & \mathrm{inc}'(n+1) \end{array}$$

Desired Coinductive Proof for Streams

- We show

$$\forall n \in \mathbb{N}.\mathrm{inc}(n) = \mathrm{inc}'(n) \land \mathrm{inc}(n) = \mathrm{inc}''(n)$$

by coinduction on Stream.

  - $\mathrm{head}(\mathrm{inc}(n)) = n = \mathrm{head}(\mathrm{inc}'(n)) = \mathrm{head}(\mathrm{inc}''(n))$
  - $\mathrm{tail}(\mathrm{inc}(n)) = \mathrm{inc}(n+1) \stackrel{\mathrm{co-IH}}{=} \mathrm{inc}''(n+1) = \mathrm{tail}(\mathrm{inc}'(n))$
  - $\mathrm{tail}(\mathrm{inc}(n)) = \mathrm{inc}(n+1) \stackrel{\mathrm{co-IH}}{=} \mathrm{inc}'(n+1) = \mathrm{tail}(\mathrm{inc}''(n))$

Goal

- Identify the precised dual of iteration, primitive recursion, induction.
- Identify the correct use of co-IH.
- Use of coalgebras as defined by their elimination rules.
- Generalise to indexed coinductively defined sets.

Introduction/Elimination of Inductive/Coinductive Sets

- Introduction rules for Natural numbers means that we have

$$\begin{array}{l} 0 \in \mathbb{N} \\ \mathrm{S} : \mathbb{N} \to \mathbb{N} \end{array}$$

  so we have an $\mathbb{N}$-algebra

$$(\mathbb{N}, 0, \mathrm{S}) \in (X \in \mathrm{Set}) \times X \times (X \to X)$$

- Dually, coinductive sets are given by their elimination rules i.e. by **observations** or **eliminators**.
  As an example we consider Stream:

$$\begin{array}{lcl} \mathrm{head} & : & \mathrm{Stream} \to \mathbb{N} \\ \mathrm{tail} & : & \mathrm{Stream} \to \mathrm{Stream} \end{array}$$

  We obtain a Stream-coalgebra

$$(\mathrm{Stream}, \mathrm{head}, \mathrm{tail}) \in (X \in \mathrm{Set}) \times (X \to \mathbb{N}) \times (X \to X)$$

Problem of Defining Coalgebras by their Introduction Rules

- Commonly one defines coalgebras by their introduction rules:
  Stream is the largest set closed under

$$\mathrm{cons} : \mathrm{Stream} \times \mathbb{N} \to \mathrm{Stream}$$

- Problem:
  - In **set theory** cons cannot be defined as a constructor such as

$$\mathrm{cons}(n, s) := \langle \lceil \mathrm{cons} \rceil, n, s \rangle$$

    as for inductively defined sets, since we would need
    non-well-founded sets.
    We can define a set Stream closed under a function cons, but that's no longer the same operation one would use for defining a corresponding inductively defined set.
  - In a **term model** we obtain **non-normalisation**:
    We get elements such as

$$\mathrm{zerostream} := \mathrm{cons}(0, \mathrm{cons}(0, \mathrm{cons}(0, \cdots))) \in \mathrm{Stream}$$

Problem of Defining Coalgebras by their Introduction Rules

- If we define Stream by its elimination rules, problems vanish:
  - In set theory Stream is a set which allows operations $\mathrm{head} : \mathrm{Stream} \to \mathbb{N}$, $\mathrm{tail} : \mathrm{Stream} \to \mathrm{Set}$.
    For instance we can take
    $$\begin{array}{lcl} \mathrm{Stream} & := & \mathbb{N} \to \mathbb{N} \\ \mathrm{head}(f) & := & f(0) \\ \mathrm{tail}(f) & := & f \circ \mathrm{S} \end{array}$$
    and obtain a largest set in the sense given below.
  - In a term model we can define the streams as the largest set which allows to define head and tail.
    $\mathrm{zerostream}$ can be a term such that $\mathrm{head}(\mathrm{zerostream}) \longrightarrow 0$, $\mathrm{tail}(\mathrm{zerostream}) \longrightarrow \mathrm{zerostream}$.
    $\mathrm{zerostream}$ itself is in normal form.
- In both cases cons can now be **defined** by the principle of coiteration.

Unique Iteration

- That $(\mathbb{N}, 0, \mathrm{S})$ are minimal can be given by:
  - Assume another $\mathbb{N}$-algebra (X, z, s), i.e.

$$\begin{array}{l} z \in X \\ s : X \to X \end{array}$$

  - Then there exist a **unique homomorphism** $g : (\mathbb{N}, 0, \mathrm{S}) \to (X, z, s)$, i.e.

$$\begin{array}{lcl} g : \mathbb{N} \to X & & \\ g(0) & = & z \\ g(\mathrm{S}(n)) & = & s(g(n)) \end{array}$$

  - This is the same as saying $\mathbb{N}$ is an initial $\mathrm{F}_{\mathbb{N}}$-algebra.
  - This means we can define uniquely

$$\begin{array}{lcll} g : \mathbb{N} \to X & & & \\ g(0) & = & x & \text{for some } x \in X \\ g(\mathrm{S}(n)) & = & x' & \text{for some } x' \in X \text{ depending on } g(n) \end{array}$$

  - This is the principle of **unique iteration**.
  - Definition by **pattern matching**.

Unique Coiteration

- Dually, that $(\mathrm{Stream}, \mathrm{head}, \mathrm{tail})$ is maximal can be given by:
  - Assume another Stream-coalgebra (X, h, t):

$$\begin{array}{rcl} h & : & X \to \mathbb{N} \\ t & : & X \to X \end{array}$$

  - Then there exist a **unique homomorphism** $g : (X, h, t) \to (\mathrm{Stream}, \mathrm{head}, \mathrm{tail})$, i.e.:

$$\begin{array}{lcl} g : X \to \mathrm{Stream} & & \\ \mathrm{head}(g(x)) & = & h(x) \\ \mathrm{tail}(g(x)) & = & g(t(x)) \end{array}$$

- Means we can define uniquely

$$\begin{array}{lcll} g : X \to \mathrm{Stream} & & & \\ \mathrm{head}(g(x)) & = & n & \text{for some } n \in \mathbb{N} \text{ depending on } x \\ \mathrm{tail}(g(x)) & = & g(x') & \text{for some } x' \in X \text{ depending on } x \end{array}$$

  This is the principle of **unique coiteration**.
- Definition by **copattern matching**.

Comparison

- When using iteration the instance of g we can use is restricted, but we can apply an arbitrary function to it.
- When using coiteration we can choose any instance a of g, but cannot apply any function to $g(a)$.

Duality

[1]

Inductive Definition	Coinductive Definition
Determined by Introduction	Determined by Observation/Elimination
Iteration	Coiteration
Pattern matching	Copattern matching
Primitive Recursion	?
Induction	?
Induction-Hypothesis	?

[1] Part of this table is due to Peter Hancock, see acknowledgements at the end.

Unique Primitive Recursion

- From unique iteration for $\mathbb{N}$ we can derive principle of **unique primitive recursion**
  - We can define uniquely

$$
\begin{array}{lcll}
g : \mathbb{N} \to X & & & \\
g(0) & = & x & \text{for some } x \in X \\
g(\mathrm{S}(n)) & = & x' & \text{for some } x' \in X \text{ depending on } n,\ g(n)
\end{array}
$$

Unique Primitive Corecursion

- From unique coiteration we can derive principle of **unique primitive corecursion**
  - We can define uniquely

$$
\begin{array}{lcl}
g : X \to \mathrm{Stream} & & \\
\mathrm{head}(g(x)) & = & n \text{ for some } n \in \mathbb{N} \text{ depending on } x \\
\mathrm{tail}(g(x))) & = & g(x') \text{ for some } x' \in X \text{ depending on } x \\
 & & \text{or} \\
 & = & s \text{ for some } s \in \mathrm{Stream} \text{ depending on } x
\end{array}
$$

Duality

- For primitive recursion we could make use of the pair $(n, g(n))$ consisting of n and the IH, i.e. an element of

$$\mathbb{N} \times X$$

- For primitive corecursion we can make use of either $s \in \mathrm{Stream}$ or $g(x')$, i.e. of an element of

$$\mathrm{Stream} + X$$

- $+$ is the dual of $\times$.

Duality

Inductive Definition	Coinductive Definition
Determined by Introduction	Determined by Observation/Elimination
Iteration	Coiteration
Pattern matching	Copattern matching
Primitive Recursion	Primitive Corecursion
Induction	?
Induction-Hypothesis	?

Example

$$s \in \mathrm{Stream}$$
$$\begin{array}{lcl} \mathrm{head}(s) & = & 0 \\ \mathrm{tail}(s) & = & s \end{array}$$

$$s' : \mathbb{N} \to \mathrm{Stream}$$
$$\begin{array}{lcl} \mathrm{head}(s'(n)) & = & 0 \\ \mathrm{tail}(s'(n)) & = & s'(n+1) \end{array}$$

$$\mathrm{cons} : (\mathbb{N} \times \mathrm{Stream}) \to \mathrm{Stream}$$
$$\begin{array}{lcl} \mathrm{head}(\mathrm{cons}(n, s)) & = & n \\ \mathrm{tail}(\mathrm{cons}(n, s)) & = & s \end{array}$$

Induction

- From unique iteration one can derive principle of **induction**:

  We can prove $\forall n \in \mathbb{N}.\varphi(n)$ by proving
  $\varphi(0)$
  $\forall n \in \mathbb{N}.\varphi(n) \rightarrow \varphi(\mathrm{S}(n))$

- Using induction we can prove (assuming extensionality of functions) uniqueness of iteration and primitive recursion.

Equivalence

Theorem

Let $(\mathbb{N}, 0, \mathrm{S})$ *be an* $\mathbb{N}$*-algebra. The following is equivalent*

1. *The principle of unique iteration.*
2. *The principle of unique primitive recursion.*
3. *The principle of iteration + induction.*
4. *The principle of primitive recursion + induction.*

Coinduction

- Uniqueness in coiteration is equivalent to the principle:
  Bisimulation implies equality
- Bisimulation on Stream is the largest relation $\sim$ on Stream s.t.

$$s \sim s' \to \mathrm{head}(s) = \mathrm{head}(s') \land \mathrm{tail}(s) \sim \mathrm{tail}(s')$$

- Largest can be expressed as $\sim$ being an indexed coinductively defined set.
- Primitive corecursion over $\sim$ means:
  We can prove

$$\forall s, s'.X(s, s') \to s \sim s'$$

  by showing

$$\begin{array}{rcl} X(s, s') & \to & \mathrm{head}(s) = \mathrm{head}(s') \\ X(s, s') & \to & X(\mathrm{tail}(s), \mathrm{tail}(s')) \lor \mathrm{tail}(s) \sim \mathrm{tail}(s') \end{array}$$

Coinduction

- Combining
  - bisimulation implies equality
  - bisimulation can be shown corecursively

  we obtain the following principle of **coinduction**

Schema of Coinduction

- We can prove
$$\forall s, s'.X(s,s') \to s = s'$$
by showing
$$\begin{array}{lcl} \forall s, s'.X(s,s') & \to & \mathrm{head}(s) = \mathrm{head}(s') \\ \forall s, s'.X(s,s') & \to & \mathrm{tail}(s) = \mathrm{tail}(s') \end{array}$$
where $\mathrm{tail}(s) = \mathrm{tail}(s')$ can be derived
  - directly or
  - from a proof of
  $$X(\mathrm{tail}(s), \mathrm{tail}(s'))$$
  invoking the **co-induction-hypothesis**
  $$X(\mathrm{tail}(s), \mathrm{tail}(s')) \to \mathrm{tail}(s) = \mathrm{tail}(s')$$
- **Note:** Only direct use of co-IH allowed.

Equivalence

Theorem

Let $(\mathrm{Stream}, \mathrm{head}, \mathrm{tail})$ *be a* Stream*-coalgebra. The following is equivalent*

1. *The principle of unique coiteration.*
2. *The principle of unique primitive corecursion.*
3. *The principle of coiteration + coinduction*
4. *The principle of primitive corecursion + coinduction*

Duality

Inductive Definition	Coinductive Definition
Determined by Introduction	Determined by Observation/Elimination
Iteration	Coiteration
Pattern matching	Copattern matching
Primitive Recursion	Primitive Corecursion
Induction	Coinduction
Induction-Hypothesis	Coinduction-Hypothesis

General Strictly Positive Indexed Inductive Definitions

- Strictly positive indexed inductively defined sets over index set I are collection of sets $\mathrm{D} : \mathrm{I} \to \mathrm{Set}$ closed under constructors

$$\begin{aligned}\mathrm{C}_j : & (x_1 \in A_1) \times (x_2 \in A_2(x_1)) \times \cdots \times (x_n \in A_n(x_1, \ldots, x_{n-1})) \\ & \to \mathrm{D}(\mathrm{i}(x_1, \ldots, x_n))\end{aligned}$$

- Here $A_k(\vec{x})$ is
  - either a non-inductive argument, i.e. a set independent of A,
  - or it is an inductive argument, i.e.

$$A_k(\vec{x}) = (b \in B(\vec{x})) \to \mathrm{D}(\mathrm{i}'_k(\vec{x}, b))$$

- Later arguments **cannot depend** on inductive arguments, only on non-inductive arguments.

Simplification

- Therefore we can move the inductive arguments to the end $(\vec{x} := x_1, \ldots, x_k)$

$$
\begin{array}{l}
\mathrm{C}_j : \underbrace{(x_1 \in A_1) \times (x_2 \in A_2(x_1)) \times \cdots \times x_k \in A_k(x_1, \ldots, x_{k-1})}_{\text{non-inductive arguments}} \times \\
\quad\underbrace{(b \in B_1(\vec{x})) \to \mathrm{D}(\mathrm{i}'_1(\vec{x}, b)) \times \cdots \times (b \in B_l(\vec{x})) \to \mathrm{D}(\mathrm{i}'_l(\vec{x}, b)))}_{\text{inductive arguments}} \\
\quad\to \mathrm{D}(\mathrm{i}_j(\vec{x}))
\end{array}
$$

Simplification

$$\mathrm{C}_j : \underbrace{(x_1 \in A_1) \times (x_2 \in A_2(x_1)) \times \cdots \times x_k \in A_k(x_1, \ldots, x_{k-1})}_{\text{non-inductive arguments}} \times$$
$$\underbrace{(b \in B_1(\vec{x})) \to \mathrm{D}(\mathrm{i}'_1(\vec{x}, b)) \times \cdots \times (b \in B_l(\vec{x})) \to \mathrm{D}(\mathrm{i}'_l(\vec{x}, b))}_{\text{inductive arguments}})$$
$$\to A(\mathrm{i}_j(\vec{x}))$$

- We can form now the product of the non-inductive arguments and obtain a single non-inductive argument.
- We can replace the inductive arguments by one non-inductive argument

$$(b \in (B_1(\vec{x}) + \cdots + B_l(\vec{x}))) \to \mathrm{D}(\mathrm{i}''(\vec{x}, b))$$

for some i''.

Simplification

- We obtain for some new sets $A_j, B_j(x)$ and function j, i

$$\mathrm{C}_j : ((a \in A_j) \times ((b \in B_j(a)) \to \mathrm{D}(\mathrm{j}(a,b)))) \to \mathrm{D}(\mathrm{i}(a))$$

- We can replace all constructors $\mathrm{C}_1, \ldots, C_n$ by one constructor C by adding an additional argument $j \in \{1, \ldots, n\}$ selecting the constructor, and then combine it with the non-inductive argument.
- So we have one constructor

$$\mathrm{C} : ((a \in A) \times ((b \in B(a)) \to \mathrm{D}(\mathrm{j}(a,b)))) \to \mathrm{D}(\mathrm{i}(a))$$

Restricted Indexed (Co)Inductively Defined Sets

$$\mathrm{C} : ((a \in A) \times ((b \in B(a)) \to \mathrm{D}(\mathrm{j}(a, b)))) \to \mathrm{D}(\mathrm{i}(a))$$

- In order to obtain the corresponding observations/eliminators for the corresponding co-inductive definitions, we need to invert the arrows.
- The more natural dual is obtained if we use restricted indexed inductive definitions:

$$\mathrm{C} : (i \in \mathrm{I}) \to ((a \in \mathrm{A}(i)) \times ((b \in \mathrm{B}(i, a)) \to \mathrm{D}(\mathrm{j}(i, a, b)))) \to \mathrm{D}(i)$$

- The corresponding observations/eliminators are

$$\mathrm{E} : (i \in \mathrm{I}) \to \mathrm{D}(i) \to ((a \in \mathrm{A}(i)) \times ((b \in \mathrm{B}(i, a)) \to \mathrm{D}(\mathrm{j}(i, a, b))))$$

or

$$\mathrm{E} : ((i \in \mathrm{I}) \times \mathrm{D}(i)) \to ((a \in \mathrm{A}(i)) \times ((b \in \mathrm{B}(i, a)) \to \mathrm{D}(\mathrm{j}(i, a, b))))$$

Petersson-Synek Trees

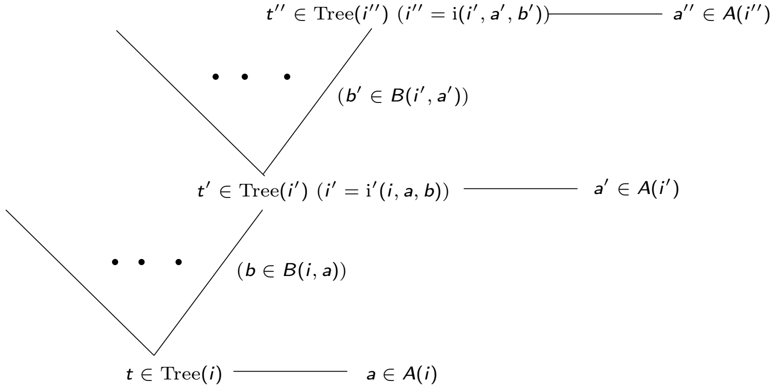
- $\mathrm{D}(i)$ form the Petersson-Synek trees (observation by Hancock), which correspond as well to the containers by Abbott, Altenkirch and Ghani.
- Replacing D by the more meaningful name Tree we obtain

$$
\begin{array}{l}
\text{data } \mathrm{Tree} : \mathrm{I} \to \mathrm{Set} \text{ where} \\
\quad \mathrm{C} : ((i \in \mathrm{I}) \times \\
\qquad (a \in \mathrm{A}(i)) \times ((b \in \mathrm{B}(i,a)) \to \mathrm{Tree}(\mathrm{j}(i,a,b)))) \\
\qquad \to \mathrm{Tree}(i)
\end{array}
$$

- For the corresponding coinductive defined set Tree^∞ we divide E into its two components and obtain

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{coalg } \mathrm{Tree}^\infty : \mathrm{I} \to \mathrm{Set} \text{ where}} \\
\mathrm{E}_1 & : & ((i \in \mathrm{I}) \times \mathrm{Tree}^\infty(i)) \to \mathrm{A}(i) \\
\mathrm{E}_2 & : & ((i \in \mathrm{I}) \times (t \in \mathrm{Tree}^\infty(i)) \times (b \in \mathrm{B}(i, \mathrm{E}_1(i,t)))) \\
 & & \quad \to \mathrm{Tree}^\infty(\mathrm{j}(i, \mathrm{E}_1(i,t), b))
\end{array}
$$

Petersson-Synek Trees



Equivalence of unique (Co)induction, (Co)recursion, (Co)induction

- The notions of (co)iteration, primitive (co)recursion, (co)induction can be generalised in a straightforward way to Petersson-Synek Trees and Co-Trees.
- One can show the equivalence of
  - unique iteration, unique primitive recursion, iteration + induction, primitive recursion + induction
  - unique coiteration, unique primitive corecursion, coiteration + coinduction, primitive corecursion + coinduction
- We call Petersson-Synek algebras fulfilling unique iteration initial Petersson-Synek algebras.
- We call Petersson-Synek coalgebras fulfilling unique coiteration final Petersson-Synek coalgebras.

Concrete Model of Tree^∞

- Tree can be modelled in a straightforward way set theoretically.
- A very concrete model of Tree^∞ can be defined by following the principle that a coalgebra is given by its observations.
  - The result of E_1 can be observed directly.
  - The result of E_2 is an element of $\mathrm{Tree}^\infty(i')$ for some i' which can be observed by carrying out more observations.

Concrete Model of Tree^∞

- Let for $i \in \mathrm{I}$

$$
\begin{aligned}
\mathrm{Path}_{[\![\,\mathrm{Tree}^\infty\,]\!]}(i) := \{\langle i_0, a_0, b_0, i_1, a_1, b_1, \ldots, i_n, a_n\rangle \mid\ & \\
& n \geq 0, i_0 = i, \\
& (\forall k \in \{0, \ldots, n-1\}.b_k \in \mathrm{B}(i_k, a_k) \wedge \\
& \qquad\qquad i_{k+1} = \mathrm{j}(i_k, a_k, b_k)), \\
& \forall k \in \{0, \ldots, n\}.a_k \in \mathrm{A}(i_k)\}
\end{aligned}
$$

- Let $[\![\,\mathrm{Tree}^\infty\,]\!](i)$ be the set of $t \subseteq \mathrm{Path}_{[\![\,\mathrm{Tree}^\infty\,]\!]}(i)$ which form the set of paths of a tree:
  - $\langle i_0, a_0, b_0, \ldots, i_{n+1}, a_{n+1}\rangle \in t \to \langle i_0, a_0, b_0, \ldots, i_n, a_n\rangle \in t$
  - $\exists! a.\langle i, a\rangle \in t$,
  - $\langle i_0, a_0, b_0, \ldots, i_n, a_n\rangle \in t \wedge b_n \in \mathrm{B}(i_n, a_n) \wedge i_{n+1} = \mathrm{j}(i_n, a_n, b_n)$
    $\to \exists! a_{n+1}.\langle i_0, a_0, b_0, \ldots, i_n, a_n, b_n, i_{n+1}, a_{n+1}\rangle \in t$

Concrete Model of Tree^∞

- Define

$$\begin{array}{l} \mathrm{E}_1 : (i \in \mathrm{I}) \to [\![\, \mathrm{Tree}^\infty \,]\!](i) \to \mathrm{A}(i) \\ \mathrm{E}_1(i, t) := a \qquad \text{if } \langle i, a \rangle \in t \end{array}$$

- Define

$$\begin{array}{l} \mathrm{E}_2 : ((i \in \mathrm{I}) \to (t \in [\![\, \mathrm{Tree}^\infty \,]\!](i)) \to (b \in \mathrm{B}(i, \mathrm{E}_1(i, t))) \\ \qquad \to [\![\, \mathrm{Tree}^\infty \,]\!](\mathrm{j}(i, \mathrm{E}_1(i, t), b)) \\ \mathrm{E}_2(i, t, b) := \{\langle i_1, a_1, b_1, \ldots, i_{n+1}, a_{n+1} \rangle \\ \qquad\qquad\qquad \mid \langle i, \mathrm{E}_1(i, t), b, i_1, a_1, b_1, \ldots, i_{n+1}, a_{n+1} \rangle \in t\} \end{array}$$

Concrete Model of Tree^∞

Theorem

$(\llbracket \mathrm{Tree}^\infty \rrbracket, \mathrm{E}_1, \mathrm{E}_2)$ *is a final* Tree^∞*-coalgebra.*

Schema for Primitive Corecursion

- Assume $A : \mathrm{I} \to \mathrm{Set}$, $[\![\, \mathrm{Tree}^\infty \,]\!], \mathrm{E}_1, \mathrm{E}_2$ as before. We can define a function
$$f : (i \in \mathrm{I}) \to X(i) \to [\![\, \mathrm{Tree}^\infty \,]\!](i)$$
corecursively by defining for $i \in \mathrm{I}$, $x \in X(i)$
  - a value $a' := \mathrm{E}_1(i, f(i, x)) \in \mathrm{A}(i)$
  - and for $b \in \mathrm{B}(i, a)$ a value $\mathrm{E}_2(i, f(i, x), b) \in [\![\, \mathrm{Tree}^\infty \,]\!](i', b)$
    where $i' := \mathrm{j}(i, a', b)$
    and we can define $\mathrm{E}_2(i, f(i, x), b)$
    - as an element of $[\![\, \mathrm{Tree}^\infty \,]\!](i')$ defined before
    - or corecursively define $\mathrm{E}_2(i, f(i, x), b) = f(i', x')$
      for some $x' \in X(i')$.
      Here $f(i', x')$ will be called the corecursion hypothesis.

Example

- Define the set of increasing streams $\mathrm{IncStream} : \mathbb{N} \to \mathrm{Set}$ starting with at least n coinductively by

$$\begin{array}{lcl} \mathrm{head} & : & (n : \mathbb{N}) \to \mathrm{IncStream}(n) \to \mathbb{N}_{\geq n} \\ \mathrm{tail} & : & (n : \mathbb{N}) \to (s : \mathrm{IncStream}(n)) \to \mathrm{IncStream}(\mathrm{head}(n, s) + 1) \end{array}$$

where $\mathbb{N}_{\geq n} := \{m : \mathbb{N} \mid m \geq n\}$.
Define

$$\begin{array}{lcl} \mathrm{inc}, \mathrm{inc}', \mathrm{inc}'' & : & (n : \mathbb{N}) \to \mathrm{IncStream}(n) \\ \mathrm{head}(n, \mathrm{inc}(n)) & = & \mathrm{head}(n, \mathrm{inc}'(n)) = \mathrm{head}(n, \mathrm{inc}''(n)) = n \\ \mathrm{tail}(n, \mathrm{inc}(n)) & = & \mathrm{inc}(n + 1) \\ \mathrm{tail}(n, \mathrm{inc}'(n)) & = & \mathrm{inc}''(n + 1) \\ \mathrm{tail}(n, \mathrm{inc}''(n)) & = & \mathrm{inc}'(n + 1) \end{array}$$

Schema for Indexed Corecursively Defined Functions

- Assume $X \in \mathrm{Set}$, $\widehat{j} : X \to \mathrm{I}$.
  We can define
  $$f : (x \in X) \to [\![\, \mathrm{Tree}^{\infty} \,]\!](\widehat{j}(x))$$
  corecursively by determining for $x \in X$ with $i := \widehat{j}(x)$,
  - $a := \mathrm{E}_1(i, f(x)) \in \mathrm{A}(i)$
  - and for $b \in \mathrm{B}(i, a)$ with $i' := \mathrm{j}(i, a, b)$ the value $\mathrm{E}_2(i, f(x), b) \in [\![\, \mathrm{Tree}^{\infty} \,]\!](i')$
    where we can define $\mathrm{E}_2(i, f(x), b)$ as
    - a previously defined value of $[\![\, \mathrm{Tree}^{\infty} \,]\!](i')$
    - or corecursively define $\mathrm{E}_2(i, f(x), b) = f(x')$ for some x' such that $\widehat{j}(x') = i'$.
      $f(x')$ will be called the corecursion hypothesis.

Example

- Define $\mathrm{Stack} : \mathbb{N} \to \mathrm{Set}$ coinductively with destructors

$$\begin{array}{lcl} \mathrm{top} & : & ((n \in \mathbb{N}) \times (n > 0) \times \mathrm{Stack}(n)) \to \mathbb{N} \\ \mathrm{pop} & : & ((n \in \mathbb{N}) \times (n > 0) \times \mathrm{Stack}(n)) \to \mathrm{Stack}(n-1) \end{array}$$

- We can define $\mathrm{empty} : \mathrm{Stack}(0)$, where we do not need to define anything since $0 > 0 = \emptyset$.
- We can define

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathrm{push} : (n, m \in \mathbb{N}) \to \mathrm{Stack}(n) \to \mathrm{Stack}(n+1) \qquad \text{s.t.}} \\ \mathrm{top}(n+1, *, \mathrm{push}(n, m, s)) & = & m \\ \mathrm{pop}(n+1, *, \mathrm{push}(n, m, s)) & = & s \end{array}$$

Schema for Coinduction

- Assume

$$\begin{array}{lcl} J & : & \mathrm{Set} \\ \widehat{i} & : & J \to \mathrm{I} \\ x_0, x_1 & : & (j \in J) \to [\![\, \mathrm{Tree}^{\infty} \,]\!](\widehat{i}(j)) \end{array}$$

  We can show $\forall j \in J.x_0(j) = x_0(j')$ coinductively by showing
  - $\mathrm{E}_0(\widehat{i}(j), x_0(j))$ and $\mathrm{E}_0(\widehat{i}(j), x_1(j))$ are equal
  - and for all b that
    $\mathrm{E}_1(\widehat{i}(j), x_0(j), b)$ and $\mathrm{E}_1(\widehat{i}(j), x_0(j), b)$ are equal,
    where we can use either the fact that
    - this was shown before,
    - or we can use the coinduction-hypothesis, which means using the fact $\mathrm{E}_1(\widehat{i}(j), x_0(j), b) = x_0(j')$ and $\mathrm{E}_1(\widehat{i}(j), x_1(j), b) = x_1(j')$ for some $j' \in J$.

Example

- Let

$$\begin{array}{rcl} s \in \mathrm{Stream} & & \\ \mathrm{head}(s) & = & 0 \\ \mathrm{tail}(s) & = & s \end{array} \qquad \begin{array}{rcl} s' : \mathbb{N} \to \mathrm{Stream} & & \\ \mathrm{head}(s'(n)) & = & 0 \\ \mathrm{tail}(s'(n)) & = & s'(n+1) \end{array}$$

$$\begin{array}{rcl} \mathrm{cons} : \mathbb{N} \to \mathrm{Stream} \to \mathrm{Stream} & & \\ \mathrm{head}(\mathrm{cons}(n,s)) & = & n \\ \mathrm{tail}(\mathrm{cons}(n,s)) & = & s \end{array}$$

- We show $\forall n \in \mathbb{N}.s = s'(n)$ by coinduction:
  Assume $n \in \mathbb{N}$. $\mathrm{head}(s) = \mathrm{head}(s'(n))$ and $\mathrm{tail}(s) = s = s'(n+1) = \mathrm{tail}(s'(n))$, where $s = s'(n+1)$ follows by the coinduction hypothesis.
- We show $\mathrm{cons}(0,s) = s$ by coinduction:
  $\mathrm{head}(\mathrm{cons}(0,s)) = 0 = \mathrm{head}(s)$ and $\mathrm{tail}(\mathrm{cons}(0,s)) = s = \mathrm{tail}(s)$, where we did not use the coinduction hypothesis.

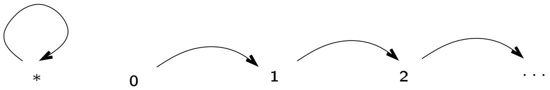
Schema for Bisimulation on Labelled Transition Systems

- Bisimulation is an indexed coinductively defined relation and therefore proofs of bisimulation can be shown by corecursion.
- Assume a labelled transition system with states S, labels L and a relation $\longrightarrow \subseteq \mathrm{S} \times \mathrm{L} \times \mathrm{S}$

Schema for Bisimulation on Labelled Transition Systems

- Let $I \in \mathrm{Set}$, $s, s' : I \to \mathrm{S}$.
- We can prove $\forall i \in \mathrm{I}.\mathrm{Bisim}(s(i), s'(i))$ coinductively by defining for any $i \in \mathrm{I}$
  - for any $l \in \mathrm{L}$, $s_0 \in \mathrm{S}$ s.t. $s(i) \xrightarrow{l} s_0$ an $s_0' \in \mathrm{S}$ s.t.
    - $s'(i) \xrightarrow{l} s_0'$
    - and s.t. $\mathrm{Bisim}(s_0, s_0')$
      where one can for prove the latter by invoking the Coinduction Hypothesis
      $\mathrm{Bisim}(s(i'), s'(i'))$ for some i' such that $s(i') = s_0$, $s'(i') = s_0'$.
  - for any $l \in \mathrm{L}$, $s_0' \in \mathrm{S}$ s.t. $s'(i) \xrightarrow{l} s_0'$ an $s_0 \in \mathrm{S}$ s.t.
    - $s(i) \xrightarrow{l} s_0$
    - and s.t. $\mathrm{Bisim}(s_0, s_0')$
      where one can prove the latter by invoking the Coinduction Hypothesis
      $\mathrm{Bisim}(s(i'), s'(i'))$ for some i' such that $s(i') = s_0$, $s'(i') = s_0'$.

Example from Introduction



- We show $\forall n \in \mathbb{N}. * \sim n$ by coinduction on $\sim$.
  - Assume $* \longrightarrow x$. We need to find y s.t. $n \longrightarrow y$ and $x \sim y$. Choose $y = n + 1$. By **co-IH** $* \sim n + 1$.
  - Assume $n \longrightarrow y$. We need to find x s.t. $* \longrightarrow x$ and $x \sim y$. Choose $x = *$. By **co-IH** $* \sim n + 1$.
- In essence same proof, but hopefully easier to teach and use.

Generalisation

- The previous example can be generalised to arbitrary coinductively defined relations.

Object-Oriented/Based Programming

- Object-oriented (OO) programming is currently main programming paradigm.
- OO programming has lots of components, we will here only look at the notion of an object.
- The component of OO programming dealing with objects only is called **object-based programming**.

Example: cell in Java

```
class cell <A> {

    /* Instance Variable */
    A content;

    /* Constructor */
    cell (A s) { content = s; }

    /* Method put */
    public void put (A s) { content = s; }

    /* Method get */
    public A get () { return content; }
}
```

Modelling Methods as Objects

- The type of objects cell is modelled as
  - a coalgebra Cell
  - with observations being the methods.
- A method m with argument A and return type B is modelled as observation
$$\mathsf{m} : \mathsf{Cell} \to \mathsf{A} \to \mathsf{B} \times \mathsf{Cell}$$
which for a cell and an argument A returns the return type and the updated cell.
- Return type void is modelled as the one element type Unit.
- Access to instant variables is not needed, since we can use get and put for it.
- A constructor with argument A is modelled as a function defined by guarded recursion
$$\mathsf{cell} : \mathsf{A} \to \mathsf{Cell}$$

Object as a Coalgebra

We define the cell using the notation we would like to have:

$$
\begin{array}{l}
\text{coalg Cell } (\mathsf{A} : \mathrm{Set}) \text{ where} \\
\quad \begin{array}{lcllllllll}
\text{put} & : & \text{Cell A} & \rightarrow & \text{A} & \rightarrow & (\text{Unit} & \times & \text{Cell A}) \\
\text{get} & : & \text{Cell A} & \rightarrow & \text{Unit} & \rightarrow & (\text{A} & \times & \text{Cell A})
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\text{cell} : \{\mathsf{A} : \text{Set}\} \rightarrow \mathsf{A} \rightarrow \text{Cell A} \\
\begin{array}{llclcl}
\text{put} & (\text{cell a}) & \text{b} & = & (\text{unit} & , \ \text{cell b}) \\
\text{get} & (\text{cell a}) & \_ & = & (\text{a} & , \ \text{cell a})
\end{array}
\end{array}
$$

- {A : Set} denotes a hidden argument which can be omitted and inferred by the system.

Object as a Coalgebra

- Official Agda code uses record instead of coalg
  - in the fields one needs to add as argument the record being defined.
  - there is a bit more generality which results in more beaurocracy.

```
record Cell (X : Set) : Set where
  coinductive
  field
    put : X    → Unit × Cell X
    get : Unit → X    × Cell X


cell : {X : Set} → X → Cell X
put (cell x) y = (unit , cell y)
get (cell x) _ = (x    , cell x)
```

- More details see Kyoto Talk.

Conclusion

- **Coiteration, primitive corecursion, coinduction** are the **duals** of iteration, primitive recursion, induction.
- In iteration/recursion/induction,
  - the **instances** of the **IH** used are **restricted**,
  - the **result can be used** in **arbitrary** functions and formulas.
- In **coiteration/corecursion/coinduction**,
  - the **instances** of the co-IH are **unrestricted**,
  - but the **result** can be only used **only directly**.
- General case of **indexed coinductive definitions** can be **reduced** to **Petersson-Synek Cotrees**.

Conclusion

- **Schemata** for **primitive corecursion and coinduction**.
- Schemata can be applied to indexed coinductively defined sets and relations.
- **Relations** on coinductively defined sets seem to be often **coinductively defined indexed relations** and can be shown by **indexed corecursion** which can be regarded as **coinduction**.
- **Objects** as in OOprogramming are **naturally occurring coalgebras**.
- **Objects** are **determined** by their **observations** and can be defined in a **natural way** in **Agda**

References

- Most of this talk (on coalgebras) was based on [Set16].
- Article on coalgebras in Martin-Löf Type Theory [Set12].
- Copatterns: [APTS13, SAPT14].
- Objects in Martin-Löf Type Theory:
  - First article [Set07],
  - Implementation in Agda [AAS16].
- Bashar's talk on processes in Agda [IS16].

Bibliography I

Andreas Abel, Stephan Adelsberger, and Anton Setzer.
Interactive programming in Agda – objects and graphical user interfaces.
To appear in Journal of Functional Programming. Preprint available at http://www.cs.swan.ac.uk/~csetzer/articles/ooAgda.pdf, 2016.

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer.
Copatterns: Programming infinite structures by observations.
In Roberto Giacobazzi and Radhia Cousot, editors, Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13, pages 27–38, New York, NY, USA, 2013. ACM.

Bibliography II

Bashar Igried and Anton Setzer.
Programming with monadic CSP-style processes in dependent type theory.
To appear in proceedings of TyDe 2016, Type-driven Development, preprint available from http://www.cs.swan.ac.uk/~csetzer/articles/TyDe2016.pdf, 2016.

Anton Setzer, Andreas Abel, Brigitte Pientka, and David Thibodeau.
Unnesting of copatterns.
In Gilles Dowek, editor, Rewriting and Typed Lambda Calculi. Proceedings RTA-TLCA 2014, volume 8560 of Lecture Notes in Computer Science, pages 31–45. Springer International Publishing, 2014.

Bibliography III

Anton Setzer.
Object-oriented programming in dependent type theory.
In Henrik Nilsson, editor, Trends in Functional Programming Volume 7, pages 91 – 108, Bristol and Chicago, 2007. Intellect.

Anton Setzer.
Coalgebras as types determined by their elimination rules.
In P. Dybjer, Sten Lindström, Erik Palmgren, and G. Sundholm, editors, Epistemology versus Ontology, volume 27 of Logic, Epistemology, and the Unity of Science, pages 351–369. Springer, Dordrecht, Heidelberg, New York, 2012.
10.1007/978-94-007-4435-6_16.

Bibliography IV

Anton Setzer.
How to reason coinductively informally.
In Reinhard Kahle, Thomas Strahm, and Thomas Studer, editors, Advances in Proof Theory, pages 377–408. Springer, 2016.

Switch over to Talk by Bashar