Coalgebras in Dependent Type Theory

Anton Setzer (Swansea), Peter Hancock (Edinburgh)
1. Interactive Programs and why we need Coalgebras.
Coalgebras in Dependent Type Theory

Anton Setzer (Swansea), Peter Hancock (Edinburgh)

1. Interactive Programs and why we need Coalgebras.

2. Rules for Coalgebras.
Coalgebras in Dependent Type Theory

Anton Setzer (Swansea), Peter Hancock (Edinburgh)

1. Interactive Programs and why we need Coalgebras.

2. Rules for Coalgebras.

Coalgebras in Dependent Type Theory

Anton Setzer (Swansea), Peter Hancock (Edinburgh)

1. Interactive Programs and why we need Coalgebras.

2. Rules for Coalgebras.


4. The $\mu$-Operator and Coalgebras.
1. Interactive Programs and why we need Coalgebras

\( c : C \quad c' : C \)

\( r : R(c) \quad r' : R(c') \)
1. Interactive Programs and why we need Coalgebras

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

c: C
r': R(c')
r: R(c)
c': C

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

\[ c : C \quad r : R(c) \quad c' : C \quad r' : R(c') \]
1. Interactive Programs and why we need Coalgebras

r:R(c)  
c:C  
r':R(c')  
c':C

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

\[ c : C \quad c' : C \quad r : R(c) \quad r' : R(c') \]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
1. Interactive Programs and why we need Coalgebras

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
• Assume $C$: Set (set of commands)
  $R(c)$: Set for $c \in C$ (set of responses for command $c$).

IO $\rightarrow$ set of non-well-founded trees with nodes labeled by $c \in C$, node with label $c$ has branching degree $R(c)$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
• Assume

- \( C' : \text{Set} \) (set of commands)
• Assume

- \( C : \text{Set} \) (set of commands)
- \( R(c) : \text{Set} \) for \( c : C \) (set of responses for command \( c \)).
**Assume**

- $C : \text{Set}$ (set of commands)
- $R(c) : \text{Set}$ for $c : C$ (set of responses for command $c$).

**IO** = set of non-well-founded trees with
• **Assume**

- \( C : \text{Set} \) (set of commands)
- \( R(c) : \text{Set} \) for \( c : C \) (set of responses for command \( c \)).

• \( \text{IO} \) = set of non-well-founded trees with

- nodes *labeled* by \( c : C \),
• **Assume**
  - $C : \text{Set}$ (set of commands)
  - $R(c) : \text{Set}$ for $c : C$ (set of responses for command $c$).

• $\text{IO}$ = set of non-well-founded trees with
  - nodes labeled by $c : C$,
  - node with label $c$ has **branching degree** $R(c)$

![Diagram of IO-Trees]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Representation of Interactive Programs: IO-Trees

- **Assume**
  - \( C : \text{Set} \) (set of commands)
  - \( R(c) : \text{Set} \) for \( c : C \) (set of responses for command \( c \)).

- **IO** = set of non-well-founded trees with
  - nodes labeled by \( c : C \),
  - node with label \( c \) has **branching degree** \( R(c) \)

\[
\begin{array}{c}
c_3 \\
r_2 \\
c_1 \\
r_0 \\
c_0 \\
c_4 \\
r_3 \\
c_2 \\
r_1 \\
c_5 \\
r_4 \\
r_5 \\
\end{array}
\]
• Assume
  - \( C : \text{Set} \) (set of commands)
  - \( R(c) : \text{Set} \) for \( c : C \) (set of responses for command \( c \)).

• \( \text{IO} \) = set of non-well-founded trees with
  - nodes \textit{labeled} by \( c : C \),
  - node with label \( c \) has \textbf{branching degree} \( R(c) \)
• Assume
  - $C : \text{Set}$ (set of commands)
  - $R(c) : \text{Set}$ for $c : C$ (set of responses for command $c$).

• $\text{IO}$ = set of non-well-founded trees with
  - nodes labeled by $c : C$,
  - node with label $c$ has branching degree $R(c)$
Representation of Interactive Programs: IO-Trees

- **Assume**
  - $C : \text{Set}$ (set of commands)
  - $R(c) : \text{Set}$ for $c : C$ (set of responses for command $c$).

- **IO** = set of non-well-founded trees with
  - nodes labeled by $c : C$,
  - node with label $c$ has **branching degree** $R(c)$

![Diagram of IO-Trees](image)
Representation of Interactive Programs: IO-Trees

• **Assume**
  - \( C : \text{Set} \) (set of commands)
  - \( R(c) : \text{Set} \) for \( c : C \) (set of responses for command \( c \)).

• **IO** = set of non-well-founded trees with
  - nodes labeled by \( c : C \),
  - node with label \( c \) has **branching degree** \( R(c) \)

\[
\begin{array}{ccccc}
  & c3 & & c4 & & c6 & & c5 \\
  r2 & & r3 & & r4 & & r5 \\
  c1 & & & & c2 & & \\
  & r0 & & & & r1 & \text{: } R(c0) \\
  & & c0 & & & \\
\end{array}
\]
• **Assume**
  - $C : \text{Set}$ (set of commands)
  - $R(c) : \text{Set}$ for $c : C$ (set of responses for command $c$).

• $\text{IO} = \text{set of non-well-founded trees with}$
  - nodes labeled by $c : C$,
  - node with label $c$ has **branching degree** $R(c)$
Problem

What do we mean by the set of non-well-founded trees?

In predicative dependent type theory, only inductive data types available. Only well-founded trees directly definable.

⇒ Need for representation of coinductive data types.

If IO is defined, we will have a function

\[
\text{elim} :: :: :: IO \to (\Sigma c : \mathbb{C.R}(c) \to IO) \\
\text{F} :: :: :: \lambda X. \Sigma c : \mathbb{C.R}(c) \to X
\]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Problem

- What do we mean by the set of non-well-founded trees?
Problem

- What do we mean by the set of non-well-founded trees?
- In predicative dependent type theory, only inductive data types available.
  - Only well-founded trees directly definable.
Problem

• What do we mean by the set of **non-well-founded trees**?

• In predicative dependent type theory, only **inductive** data types available.
  - Only well-founded trees directly definable.

⇒ Need for representation of **coinductive** data types.
Problem

- What do we mean by the set of non-well-founded trees?
- In predicative dependent type theory, only inductive data types available.
  - Only well-founded trees directly definable.
  
  \[ \Rightarrow \] Need for representation of coinductive data types.

- If IO is defined, we will have a function

\[ \text{elim} : \text{IO} \rightarrow (\Sigma c : C.R(c) \rightarrow \text{IO}) \]
What do we mean by the set of non-well-founded trees?

In predicative dependent type theory, only inductive data types available.
- Only well-founded trees directly definable.

Need for representation of coinductive data types.

If IO is defined, we will have a function

\[
\text{elim} : IO \to (\Sigma c : C.R(c) \to IO)
\]
Problem

- What do we mean by the set of non-well-founded trees?

- In predicative dependent type theory, only inductive data types available.
  - Only well-founded trees directly definable.

⇒ Need for representation of coinductive data types.

- If IO is defined, we will have a function

\[
\text{elim} : \text{IO} \rightarrow (\Sigma c : C.R(c) \rightarrow \text{IO}) ,
\]

\[
F(\text{IO})
\]
Problem

• What do we mean by the set of non-well-founded trees?

• In predicative dependent type theory, only inductive data types available.
  - Only well-founded trees directly definable.

⇒ Need for representation of coinductive data types.

• If IO is defined, we will have a function

\[
\text{elim} : \text{IO} \rightarrow \left( \sum c : C.R(c) \rightarrow \text{IO} \right) ,
\]

\[
F := \lambda X. \sum c : C.R(c) \rightarrow X
\]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Generalization

• Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X. \Sigma c : C.R(c) \to X$ for some $C, R$.

- $\lambda X.X$.
- $\lambda X.C$.
- $\lambda X.R$.

- If $F, G$ are isomorphic to it, so is $\lambda X.F(X) + G(X)$.

- If $A : \text{Set}$, $F_a$ is isomorphic to it ($a : A$), so are $\ast \lambda X. \Sigma a : A.F_a(X)$.

$\ast \lambda X. \Pi a : A.F_a(X)$ (use of axiom of choice).

• Call such operations strictly positive functors.

• Notion could be extended to include $F^\ast$ (initial algebra functor) and $F^\infty$ (final coalgebra functor; see below) for $F$ strictly positive.
Generalization

- Many functions $F : \text{Set} \rightarrow \text{Set}$ isomorphic to $\lambda X. \Sigma c : C.R(c) \rightarrow X$ for some $C, R$.

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Generalization

- Many functions $F : \text{Set} \rightarrow \text{Set}$ isomorphic to $\lambda X.\Sigma c : C.R(c) \rightarrow X$ for some $C, R$.
  - $\lambda X.X$.

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X. \Sigma c : C.R(c) \to X$ for some $C, R$.

- $\lambda X.X$.
- $\lambda X.C$.
Generalization

- Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X. \Sigma c : C.R(c) \to X$ for some $C, R$.
  - $\lambda X.X$.
  - $\lambda X.C$.
  - If $F$, $G$ are isomorphic to it, so is $\lambda X.F(X) + G(X)$.
Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X.\Sigma c : C.R(c) \to X$ for some $C$, $R$.

- $\lambda X.X$.
- $\lambda X.C$.
- If $F$, $G$ are isomorphic to it, so is $\lambda X.F(X) + G(X)$.
- If $A : \text{Set}$, $F_a$ is isomorphic to it ($a : A$), so are
  * $\lambda X.\Sigma a : A.F_a(X)$.
Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X. \Sigma c : C.R(c) \to X$ for some $C, R$.

- $\lambda X.X$.
- $\lambda X.C$.
- If $F, G$ are isomorphic to it, so is $\lambda X.F(X) + G(X)$.
- If $A : \text{Set}$, $F_a$ is isomorphic to it ($a : A$), so are
  * $\lambda X.\Sigma a : A.F_a(X)$.
  * $\lambda X.\Pi a : A.F_a(X)$ (use of axiom of choice).
Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X.\Sigma c : C.R(c) \to X$ for some $C, R$.

- $\lambda X.X$.
- $\lambda X.C$.
- If $F, G$ are isomorphic to it, so is $\lambda X.F(X) + G(X)$.
- If $A : \text{Set}$, $F_a$ is isomorphic to it ($a : A$), so are
  * $\lambda X.\Sigma a : A.F_a(X)$.
  * $\lambda X.\Pi a : A.F_a(X)$ (use of axiom of choice).

Call such operations **strictly positive functors**.
Generalization

- Many functions $F : \text{Set} \to \text{Set}$ isomorphic to $\lambda X. \Sigma c : C.R(c) \to X$ for some $C, R$.
  - $\lambda X.X$.
  - $\lambda X.C$.
  - If $F, G$ are isomorphic to it, so is $\lambda X.F(X) + G(X)$.
  - If $A : \text{Set}$, $F_a$ is isomorphic to it ($a : A$), so are
    * $\lambda X.\Sigma a : A.F_a(X)$.
    * $\lambda X.\Pi a : A.F_a(X)$ (use of axiom of choice).

- Call such operations strictly positive functors.

- Notion could be extended to include $F^*$ (initial algebra functor) and $F^\infty$ (final coalgebra functor; see below) for $F$ strictly positive.
Operation on Morphisms

- Operation on morphisms for $F = \lambda X. \Sigma c : C.R(c) \to X$: 

$$F(f)(\langle c, n \rangle) = \langle c, f \circ n \rangle.$$
Operation on Morphisms

- Operation on morphisms for \( F = \lambda X. \Sigma c : C.R(c) \to X \):

  - If \( f : X \to Y \), \( F(f) : F(X) \to F(Y) \),

\[
F(f)(\langle c, n \rangle) = \langle c, f \circ n \rangle .
\]
Notation

\[ C_0(A) + C_1(B) := \text{data}\{C_0(a : A) \mid C_1(b : B)\} \]
2. Rules for Coalgebras.

Let $F$ be strictly positive.
We need rules expressing
2. Rules for Coalgebras.

Let $F$ be strictly positive. We need rules expressing

- $F^\infty_0$
2. Rules for Coalgebras.

Let $F$ be strictly positive.
We need rules expressing

- $F_0^\infty$ is (semi-) largest set s.t. there exists

$$\text{elim} : F_0^\infty \to F(F_0^\infty) .$$

($F$ strictly positive).
2. Rules for Coalgebras.

Let $F$ be strictly positive. We need rules expressing

- $F_0^\infty$ is (semi-) largest set s.t. there exists

$$\text{elim} : F_0^\infty \rightarrow F(F_0^\infty).$$

($F$ strictly positive).

- Idea from Peter Aczel, non-well-founded set theory: Elements introduced as graphs.
Examples of Non-Wf Sets

\{\{\cdots}\} \text{ given by}

\[
\begin{array}{c}
\emptyset \\
\emptyset \\
\emptyset \\
\emptyset \\
\emptyset \\
\emptyset \\
\end{array}
\]
or

\[
\begin{array}{c}
\emptyset \\
\emptyset \\
\emptyset \\
\emptyset \\
\emptyset \\
\end{array}
\]
Examples of Non-Wf Sets

$\{\{\ldots\}\}$ given by

\[\begin{array}{c}
\text{or}\\
\end{array}\]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Examples of Non-Wf Sets

\{\{\cdots\}\}\text{ given by}

\{\{\\}\{\\}\{\\} \cdots\}\text{ given by}

\begin{tikzpicture}
\draw (0,0) -- (1,1);
\draw (1,1) -- (2,2);
\draw (2,2) -- (3,3);
\draw (3,3) -- (4,4);
\end{tikzpicture}

or

\begin{tikzpicture}
\draw (0,0) -- (1,1);
\draw (1,1) -- (2,2);
\draw (2,2) -- (3,3);
\draw (3,3) -- (4,4);
\end{tikzpicture}
Examples of Non-Wf Sets

\{\{\cdots\}\}\ given\ by

\{\}\{\}\{\}\{\} \cdots \} \given\ by

\{\}\{\}\{\}\{\} \cdots \} \given\ by

or

\{\}\{\}\{\}\{\} \cdots \} \given\ by

or

\{\}\{\}\{\}\{\} \cdots \} \given\ by

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Examples of Non-Wf Sets

\{\{\cdots}\}\ given \ by

\{\}{\}{\}{\}{\cdots}\} \ given \ by

or

\begin{center}
\begin{tikzpicture}
  \node (x) at (0,0) {\times};
  \node (a) at (-1,-1) {\{};
  \node (b) at (0,-1) {\{};
  \node (c) at (1,-1) {\}};
  \draw[->] (x) -- (a);
  \draw[->] (x) -- (b);
  \draw[->] (x) -- (c);
  \node (d) at (-2,-2) {\{};
  \node (e) at (-1,-2) {\}};
  \node (f) at (0,-2) {\{};
  \node (g) at (1,-2) {\}};
  \draw[->] (a) -- (d);
  \draw[->] (a) -- (e);
  \draw[->] (b) -- (f);
  \draw[->] (b) -- (g);
  \node (h) at (-3,-3) {\{};
  \node (i) at (-2,-3) {\}};
  \node (j) at (-1,-3) {\}};
  \node (k) at (0,-3) {\}};
  \draw[->] (d) -- (h);
  \draw[->] (d) -- (i);
  \draw[->] (e) -- (j);
  \draw[->] (e) -- (k);
  \node (l) at (-4,-4) {\}.
  \end{tikzpicture}
\end{center}

or

\begin{center}
\begin{tikzpicture}
  \node (x) at (0,0) {\times};
  \node (a) at (-1,-1) {\{};
  \node (b) at (0,-1) {\}};
  \draw[->] (x) -- (a);
  \draw[->] (x) -- (b);
  \node (c) at (-2,-2) {\{};
  \node (d) at (-1,-2) {\}};
  \node (e) at (0,-2) {\}};
  \draw[->] (a) -- (c);
  \draw[->] (a) -- (d);
  \draw[->] (b) -- (e);
  \node (f) at (-3,-3) {\}.
  \end{tikzpicture}
\end{center}
Graphs for $F_0^\infty$

Assume $F(X) = \Sigma c : C.R(c) \to X$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Graphs for $F_0^\infty$

Assume $F(X) = \Sigma c : C.R(c) \to X$.

- A **graph** for $F$ consists of
Assume $F(X) = \Sigma c : C \cdot R(c) \to X$.

- A **graph** for $F$ consists of
  - a set $A$,
Assume $F(X) = \Sigma c : C.R(c) \rightarrow X$.

- A graph for $F$ consists of
  - a set $A$,
  - a labelling function $c : A \rightarrow C$, 
Assume $F(X) = \Sigma c : C. R(c) \to X$.

- A graph for $F$ consists of
  - a set $A$,
  - a labelling function $c : A \to C$,
  - a next function $n : (a : A, R(c(a))) \to A$. 
Graphs for $F_0^\infty$

Assume $F(X) = \sum c : C.R(c) \rightarrow X$.

- A **graph** for $F$ consists of
  - a set $A$,
  - a labelling function $c : A \rightarrow C$,
  - a next function $n : (a : A, R(c(a))) \rightarrow A$.
  - a starting node $a : A$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory 

10f
More Abstractly

A graph for \( F \) consists of:
- a set \( A \),
- an \( f: A \to (\Sigma c: C.R(c) \to A) \),
- an \( a: A \).

Introduction rule for \( F_0^\infty \): every graph introduces an element of \( F_0^\infty \).

However: no full elimination – Only:
\( \text{elim} : F_0^\infty \to F(F_0^\infty) \).
• A graph for $F$ consists of

More Abstractly

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory 12b
More Abstractly

- A graph for $F$ consists of
  - a set $A$,
More Abstractly

- A graph for $F$ consists of
  - a set $A$,
  - an $f : A \rightarrow (\Sigma c : C.R(c) \rightarrow A)$,
More Abstractly

• A graph for $F$ consists of
  - a set $A$,
  - an $f : A \rightarrow (\sum c : C.R(c) \rightarrow A)$.

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
More Abstractly

- A graph for $F$ consists of
  - a set $A$,
  - an $f : A \rightarrow (\Sigma c : C.R(c) \rightarrow A)$,
  - an $a : A$. 
More Abstractly

- A graph for $F$ consists of
  - a set $A$,
  - an $f : A \rightarrow (\sum c : C.R(c) \rightarrow A)_\downarrow F(A)$,
  - an $a : A$.

- Introduction rule for $F_0^\infty$:
More Abstractly

- A graph for $F$ consists of
  - a set $A$,
  - an $f : A \to (\Sigma c : C.R(c) \to A)_F(A)$,
  - an $a : A$.
- Introduction rule for $F_0^\infty$:
every graph introduces an element of $F_0^\infty$. 
• A graph for $F$ consists of
  - a set $A$,
  - an $f : A \to (\Sigma c : C.R(c) \to A)$,
  - an $a : A$.

• Introduction rule for $F^\infty_0$:
  every graph introduces an element of $F^\infty_0$.

• However: no full elimination – Only: elim : $F^\infty_0 \to F(F^\infty_0)$. 
Rules for Coiteration

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory

13a
Rules for Coiteration

Formation

\[ \text{Introduction} \]

\[ A : \text{Set} \]

\[ \gamma : A \rightarrow F(A) \]

\[ a : A \]

\[ \text{intro}'(A, \gamma, a) : F(\infty)_0 \]

\[ \text{Elimination} \]

\[ p : F(\infty)_0 \]

\[ \text{elim}(p) : F(F(\infty)_0) \]

\[ \text{Equality} \]

\[ \text{elim}(\text{intro}'(A, \gamma, a)) = F(\lambda x. \text{intro}'(A, \gamma, x))(\gamma(a)) \]

\[ F(\infty)_0 \]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Rules for Coiteration

Formation

\[ F^\infty_0 : \text{Set} \]
Rules for Coiteration

Formation

$F^\infty_0 : \text{Set}$

Introduction
Rules for Coiteration

Formation

$$F_0^{\infty} : \text{Set}$$

Introduction

$$\frac{A : \text{Set}}{\gamma : A \rightarrow F(A)} \quad a : A \quad \frac{\text{intro}'(A, \gamma, a)}{\text{intro}'(A, \gamma, a) : F_0^{\infty}}$$
Rules for Coiteration

Formation

$F_0^\infty : \text{Set}$

Introduction

\[
\frac{A : \text{Set} \quad \gamma : A \rightarrow F(A) \quad a : A}{\text{intro}'(A, \gamma, a) : F_0^\infty}
\]

Elimination
Rules for Coiteration

Formation

\[ F_0^\infty : \text{Set} \]

Introduction

\[
\begin{array}{c}
A : \text{Set} \quad \gamma : A \to F(A) \quad a : A \\
\hline
\text{intro}'(A, \gamma, a) : F_0^\infty
\end{array}
\]

Elimination

\[
\begin{array}{c}
p : F_0^\infty \\
\hline
\text{elim}(p) : F(F_0^\infty)
\end{array}
\]
Rules for Coiteration

Formation

\[ F_0^\infty : \text{Set} \]

Introduction

\[ A : \text{Set} \quad \gamma : A \rightarrow F(A) \quad a : A \]
\[ \text{intro}'(A, \gamma, a) : F_0^\infty \]

Elimination

\[ p : F_0^\infty \]
\[ \text{elim}(p) : F(F_0^\infty) \]

Equality

\[ \text{elim}(\text{intro}'(A, \gamma, a)) = F(F_0^\infty) \]
Rules for Coiteration

Formation

\[ F_0^\infty : \text{Set} \]

Introduction

\[
\begin{align*}
A : \text{Set} & \quad \gamma : A \to F(A) & a : A \\
\text{intro}'(A, \gamma, a) : F_0^\infty
\end{align*}
\]

Elimination

\[
\begin{align*}
p : F_0^\infty \\
elim(p) : F(F_0^\infty)
\end{align*}
\]

Equality

\[
\begin{align*}
elim(\text{intro}'(A, \gamma, a)) &= F(\lambda x. \text{intro}'(A, \gamma, x))(\gamma(a)) \\
&: F(F_0^\infty)
\end{align*}
\]
Anton Setzer, Peter Hancock: Coalgebras in dependent type theory

Easier to define successor —— Definable using Coiteration
Easier to define successor —— Definable using Coiteration

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Easier to define successor —— Definable using Coiteration

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Rules for Corecursion

Introduction

\[ A : \text{Set} \]
\[ \gamma : A \rightarrow F(\text{cont}(A) + \text{fin}(F_0)) \]
\[ a : A \]
\[ \text{intro}(A, \gamma, a) : F_0 \]

Elimination

\[ p : F_0 \]
\[ \text{elim}(p) : F(F_0) \]

Equality

\[ \text{elim}(\text{intro}(A, \gamma, a)) = F(f) \]
\[ \text{where} \]
\[ f(\text{cont}(a)) = \text{intro}(A, \gamma, a) \]
\[ f(\text{fin}(p)) = p \]
Rules for Corecursion

Formation

\begin{align*}
\text{Formation} & \quad \text{Set} \\
A & \quad \text{Set} \\
\gamma & \colon A \to F(\text{cont}(A) + \text{fin}(F)) \\
a & \colon A \\
\text{intro}(A, \gamma, a) & \colon F \\
\end{align*}

\begin{align*}
\text{Elimination} & \quad F(\text{cont}(a)) = \text{intro}(A, \gamma, a) \\
& \quad F(\text{fin}(p)) = p \\
\text{Equality} & \quad \text{elim}(\text{intro}(A, \gamma, a)) = F(f(\gamma(a))) \\
& \quad \text{where} \quad f(\text{cont}(a)) = \text{intro}(A, \gamma, a) \\
& \quad f(\text{fin}(p)) = p
\end{align*}
Rules for Corecursion

Formation

\[ F_0^\infty : \text{Set} \]
Rules for Corecursion

Formation

\( F^\infty_0 : \text{Set} \)

Introduction

\( \text{intro}(A, \gamma, a) : F^\infty_0 \)

Elimination

\( \text{elim}(p) : F^\infty_0 \)

Equality

\[ \text{elim}(\text{intro}(A, \gamma, a)) = F(f)(\gamma(a)) : F(F^\infty_0) \]

where

\[ f(\text{cont}(a)) = \text{intro}(A, \gamma, a) \]

\[ f(\text{fin}(p)) = p \]
Rules for Corecursion

Formation

\( F_0^\infty : \text{Set} \)

Introduction

\[
\frac{A : \text{Set} \quad \gamma : A \to F(\text{cont}(A) + \text{fin}(F_0^\infty)) \quad a : A}{\text{intro}(A, \gamma, a) : F_0^\infty}
\]
Rules for Corecursion

Formation

$F_0^\infty : \text{Set}$

Introduction

$A : \text{Set} \quad \gamma : A \to F(\text{cont}(A) + \text{fin}(F_0^\infty)) \quad a : A$

$\text{intro}(A, \gamma, a) : F_0^\infty$

Elimination

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Rules for Corecursion

Formation

\[ F_0^\infty : \text{Set} \]

Introduction

\[
\begin{align*}
A : \text{Set} & \quad \gamma : A \to F(\text{cont}(A) + \text{fin}(F_0^\infty)) \quad a : A \\
\text{intro}(A, \gamma, a) : F_0^\infty
\end{align*}
\]

Elimination

\[
\begin{align*}
p : F_0^\infty \\
\text{elim}(p) : F(F_0^\infty)
\end{align*}
\]
Rules for Corecursion

Formation

\[ F_0^{\infty} : \text{Set} \]

Introduction

\[
\begin{align*}
A : \text{Set} & \quad \gamma : A \rightarrow F(\text{cont}(A) + \text{fin}(F_0^{\infty})) & a : A \\
& \quad \text{intro}(A, \gamma, a) : F_0^{\infty}
\end{align*}
\]

Elimination

\[
\begin{align*}
p : F_0^{\infty} \\
& \quad \text{elim}(p) : F(F_0^{\infty})
\end{align*}
\]

Equality

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Rules for Corecursion

Formation

\[ F^\infty_0 : \text{Set} \]

Introduction

\[
A : \text{Set} \quad \gamma : A \rightarrow F(\text{cont}(A) + \text{fin}(F^\infty_0)) \quad a : A
\]

\[ \text{intro}(A, \gamma, a) : F^\infty_0 \]

Elimination

\[ p : F^\infty_0 \]

\[ \text{elim}(p) : F(F^\infty_0) \]

Equality

\[
\text{elim}(\text{intro}(A, \gamma, a)) = F(f)(\gamma(a)) : F(F^\infty_0)
\]
Rules for Corecursion

**Formation**

\[ F_0^\infty : \text{Set} \]

**Introduction**

\[
\begin{align*}
A &: \text{Set} \\
\gamma &: A \rightarrow F(\text{cont}(A) + \text{fin}(F_0^\infty)) \\
a &: A \\
\text{intro}(A, \gamma, a) &: F_0^\infty
\end{align*}
\]

**Elimination**

\[
\begin{align*}
p &: F_0^\infty \\
\text{elim}(p) &: F(F_0^\infty)
\end{align*}
\]

**Equality**

\[
\text{elim}(\text{intro}(A, \gamma, a)) = F(f)(\gamma(a)) : F(F_0^\infty)
\]

where \( f(\text{cont}(a)) = \text{intro}(A, \gamma, a) \)

\( f(\text{fin}(p)) = p \)
Want to construct a functor based on $\mathcal{F}_\infty$.  

Idea: start from atomic elements ($a : A$) and "build possibly non-well-founded many constructors of $\mathcal{F}$ on top of it".

More precisely: Let $\mathcal{F}_A ::= \lambda X. \text{at}(A) + \text{do}(\mathcal{F}(X))$.

$\mathcal{F}_\infty(A) ::= (\mathcal{F}_A)_\infty^0$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
• Want to construct a functor based on $F_0^\infty$. 

$F^\infty(A)$
Want to construct a functor based on $F_0^\infty$.

Idea: start from atomic elements $(a : A)$ and “build possibly non-well-founded many constructors of $F$ on top of it”.

$F^\infty(A)$
• Want to construct a functor based on $F_0^\infty$.

• Idea: start from atomic elements ($a : A$) and "build possibly non-well-founded many constructors of $F$ on top of it".

• More precisely: Let $F_A^\infty := \lambda X.\text{at}(A) + \text{do}(F(X))$. 

$F^\infty(A)$
Want to construct a functor based on $F_0^\infty$.

Idea: start from atomic elements ($a : A$) and “build possibly non-well-founded many constructors of $F$ on top of it”.

More precisely: Let $F_A := \lambda X. \text{at}(A) + \text{do}(F(X))$.

$F^\infty(A) := (F_A)_0^\infty$. 
Graphs for $F^\infty(A)$
3. Some Operations on Coalgebras

Let

\[ \text{Atoms} \]

\[ \text{At}(a) := \text{intro}(\{\star\}, \lambda x. \text{cont}(\text{at}(a)), \star) : F_\infty(A) \]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
3. Some Operations on Coalgebras

Atoms

\[ \text{Atoms} \]

\[ \text{At}(a) \leq \text{intro}(\{\star\}, \lambda x. \text{cont}(\text{at}(a)), \star) : \text{F}_\infty(A) \]

\[ \text{start} \quad \text{leaf}(a) \]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
3. Some Operations on Coalgebras

Atoms

start \rightarrow leaf(a)

\text{Atoms}

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
3. Some Operations on Coalgebras

Atoms

For $a : A$ let $\text{At}(a) := \text{intro}(\{\star\}, \lambda x.\text{cont}(\text{at}(a)), \star) : F^\infty(A)$.

\[
\begin{array}{ccc}
\text{start} & \rightarrow & \text{leaf}(a) \\
& \text{leaf}(a) & \\
\end{array}
\]

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory

18d
Repeat

B b0 b1 b2
cont(b0) g(b0) g(b1) g(b2)
cont(b1) cont(b0) cont(b2)
fin(p)

B: Set
\(g: B \to F(\infty(\text{cont}(B)) + \text{fin}(F_\infty(0))))\)
$B : \text{Set}$
\[ B : \text{Set} \quad g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))) \]
\( B : \text{Set} \quad g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) \)
\[ B : \text{Set} \quad g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))) \]
\[ B : \text{Set} \quad g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) \]
$B : \text{Set} \quad g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))$
\[ B : \text{Set} \quad g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) \]
\[ B : \text{Set} \quad g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) \]
\[ B : \text{Set} \quad g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))) \]
$B : \text{Set}$ \quad $g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))))$
\( B : \text{Set} \quad g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))) \)
• Assume $B$; Set $g: B \rightarrow F(\infty(\text{cont}(B) + \text{fin}(F^\infty 0)))$,
  $b: B$.

• Define $\text{repeat}(B, g, b)$ :::::::::::::::::::::::::::: := intro($F^\infty(\text{cont}(B) + \text{fin}(F^\infty 0)), f, \text{at}(\text{cont}(b))$).

  where we define $f: F^\infty(\text{cont}(B) + \text{fin}(F^\infty 0)) \rightarrow (\text{cont}(F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty 0)))) + \text{fin}(F^\infty 0))$,
  - if $\text{elim}(a) = \text{at}(\text{cont}(b))$, then $f(a) = \text{cont}(g(b))$.
  - if $\text{elim}(a) = \text{at}(\text{fin}(p))$, then $f(a) = \text{fin}(p)$.
  - if $\text{elim}(a) = \text{do}(p)$, then $f(a) = \text{cont}(p)$. 
repeat(\(B, g, b\))

- Assume \(B : \text{Set}, g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))), b : B.\)
Repeat $B, g, b$

- Assume $B : \text{Set}$, $g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))$, $b : B$.

- Define $\text{repeat}(B, g, b) := \text{intro}(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)), f, \text{At}(\text{cont}(b))$)
• Assume $B : \text{Set}$, $g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))$, $b : B$.

• Define $\text{repeat}(B, g, b) := \text{intro}(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)), f, \text{At}(\text{cont}(b)) )$

• where we define

$$f : F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)) \rightarrow (\text{cont}(F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) + \text{fin}(F_0^\infty))$$
repeat(B, g, b)

- Assume $B : \text{Set}$, $g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty)))$, $b : B$.

- Define $\text{repeat}(B, g, b) := \text{intro}(F^\infty(\text{cont}(B) + \text{fin}(F^\infty)), f, \text{At}(\text{cont}(b)) )$

- where we define

  $f : F^\infty(\text{cont}(B) + \text{fin}(F^\infty)) \rightarrow (\text{cont}(F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty)))) + \text{fin}(F^\infty))$

  - if $\text{elim}(a) = \text{at}(\text{cont}(b))$, 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Assume $B : \text{Set}$, $g : B \to F(F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})))$, $b : B$.

Define $\text{repeat}(B, g, b) := \text{intro}(F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})), f, \text{At}(\text{cont}(b)))$

where we define

$$f : F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})) \to (\text{cont}(F(F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})))) + \text{fin}(F_0^{\infty}))$$

- if $\text{elim}(a) = \text{at}(\text{cont}(b))$, then $f(a) = \text{cont}(\underbrace{g(b)}_{: F(F^{\infty}(\text{fin}(F_0^{\infty}) + \text{cont}(B)))})$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
repeat(B, g, b)

- Assume \( B : \text{Set}, \ g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty_0))), \ b : B. \)

- Define \( \text{repeat}(B, g, b) := \text{intro}(F^\infty(\text{cont}(B) + \text{fin}(F^\infty_0)), f, \text{At}(\text{cont}(b)) \ ) \)

- where we define

\[
f : F^\infty(\text{cont}(B) + \text{fin}(F^\infty_0)) \rightarrow (\text{cont}(F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty_0)))) + \text{fin}(F^\infty_0))
\]

  - if \( \text{elim}(a) = \text{at}(\text{cont}(b)), \) then \( f(a) = \text{cont}(g(b)) \).

  - if \( \text{elim}(a) = \text{at}(\text{fin}(p)), \)

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
repeat(B, g, b)

- Assume \( B : \text{Set}, \ g : B \rightarrow F(F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty}))), \ b : B \).

- Define \( \text{repeat}(B, g, b) := \text{intro}(F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})), f, \text{At}(\text{cont}(b))) \)

- where we define

\[
f : F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})) \rightarrow (\text{cont}(F(F^{\infty}(\text{cont}(B) + \text{fin}(F_0^{\infty})))) + \text{fin}(F_0^{\infty}))
\]

  - if \( \text{elim}(a) = \text{at}(\text{cont}(b)) \), then \( f(a) = \text{cont}(\underbrace{g(b)}_{:F(F^{\infty}(\text{fin}(F_0^{\infty})+\text{cont}(B))}) \).

  - if \( \text{elim}(a) = \text{at}(\text{fin}(p)) \), then \( f(a) = \text{fin}(p) \).
repeat(B, g, b)

- Assume $B : \text{Set}$, $g : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))), b : B$.

- Define $\text{repeat}(B, g, b) := \text{intro}(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)), f, \text{At}(\text{cont}(b)) )$

- where we define

  \[
  f : F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)) \to (\text{cont}(F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) + \text{fin}(F_0^\infty))
  \]

  - if $\text{elim}(a) = \text{at}(\text{cont}(b))$, then $f(a) = \text{cont}(\underbrace{g(b)}_{:F(F^\infty(\text{fin}(F_0^\infty)+\text{cont}(B)))})$.
  - if $\text{elim}(a) = \text{at}(\text{fin}(p))$, then $f(a) = \text{fin}(p)$.
  - if $\text{elim}(a) = \text{do}(\underbrace{p}_{:F(F^\infty(\text{fin}(F_0^\infty)+\text{cont}(B)a))})$.
\textbf{repeat}(B, g, b)

- Assume $B : \text{Set}$, $g : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty))), b : B$.

- Define $\text{repeat}(B, g, b) := \text{intro}(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)), f, \text{At}(\text{cont}(b)) )$

- where we define

  \[
  f : F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)) \rightarrow (\text{cont}(F(F^\infty(\text{cont}(B) + \text{fin}(F_0^\infty)))) + \text{fin}(F_0^\infty))
  \]

  - if $\text{elim}(a) = \text{at}(\text{cont}(b))$, then $f(a) = \text{cont}(g(b))$.
  - if $\text{elim}(a) = \text{at}(\text{fin}(p))$, then $f(a) = \text{fin}(p)$.
  - if $\text{elim}(a) = \text{do}(p)$, then $f(a) = \text{cont}(p)$.
4. The $\mu$-Operator and Coalgebras

- $\text{NStream} = F_\infty^0$, where $F(X) = \text{data}\{\text{cons}(n, x)\} \approx N \times X$.
- Elements of $\text{NStream}$ have the form $\text{cons}(n_1, \text{cons}(n_2, \text{cons}(n_3, \cdots )))$.
- Would like to define elements of $\text{NStream}$ recursively.
  - E.g. $f : N \to \text{NStream}$, $f(n) = \text{cons}(n, f(n+1))$.
- In this form non-normalizing.
4. The $\mu$-Operator and Coalgebras

- $\text{NStream} = F_0^\infty$, where
  
  $F(X) = \text{data}\{\text{cons}(n : \mathbb{N}, x : X)\}$  
  $(\approx \mathbb{N} \times X)$
4. The $\mu$-Operator and Coalgebras

- $\text{NStream} = F_0^\infty$, where
  
  $F(X) = \text{data}\{\text{cons}(n : \mathbb{N}, x : X)\}$  
  \( \approx \mathbb{N} \times X \)

- Elements of $\text{NStream}$ have the form
  
  $\text{cons}(n_1, \text{cons}(n_2, \text{cons}(n_3, \cdots)))$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
4. The $\mu$-Operator and Coalgebras

- $\text{NStream} = F_0^\infty$, where
  
  $F(X) = \text{data}\{\text{cons}(n : N, x : X)\}$  
  
  ($\approx N \times X$)

- Elements of $\text{NStream}$ have the form
  
  $\text{cons}(n_1, \text{cons}(n_2, \text{cons}(n_3, \cdots)))$.

- Would like to define elements of $\text{NStream}$ recursively.
4. The $\mu$-Operator and Coalgebras

- $\text{NStream} = F_0^\infty$, where
  \[ F(X) = \text{data}\{\text{cons}(n : \mathbb{N}, x : X)\} \quad (\approx \mathbb{N} \times X) \]

- Elements of $\text{NStream}$ have the form
  \[ \text{cons}(n_1, \text{cons}(n_2, \text{cons}(n_3, \cdots))) \].

- Would like to define elements of $\text{NStream}$ recursively. E.g.
  \[ f : \mathbb{N} \to \text{NStream}, \quad f(n) = \text{cons}(n, f(n + 1)). \]
4. The $\mu$-Operator and Coalgebras

- $\text{NStream} = F_0^\infty$, where
  $F(X) = \text{data}\{\text{cons}(n : N, x : X)\} \quad (\simeq N \times X)$

- Elements of $\text{NStream}$ have the form
  $\text{cons}(n_1, \text{cons}(n_2, \text{cons}(n_3, \cdots)))$.

- Would like to define elements of $\text{NStream}$ recursively. E.g.
  - $f : N \to \text{NStream}$, $f(n) = \text{cons}(n, f(n + 1))$.

- In this form non-normalizing.
The $\mu$-Operator

- Instead try a constructor $\mu$. (Idea from T. Coquand).

Assume $A : \text{Set}$, $g : (A \to \text{NStream}) \to A \to \text{NStream}$.

Then $\mu A (g) : A \to \text{NStream}$.

$f$ as above can be defined as $\mu N (\lambda g, n. \text{cons}(n, g(n) + 1))$.

$\mu$ is a constructor $\Rightarrow$ recursion evaluated only when applying $\text{elim}$.

In order to define $\text{elim}$, we need to apply $\text{elim}$ to the body of $\mu$.

Better: replace the type of $g$ above by: $g : (A \to \text{NStream}) \to A \to F(\text{NStream})$.

Now define $\text{elim}(\mu A (g, a)) = g(\mu A (g), a)$.
The $\mu$-Operator

• Instead try a constructor $\mu$. (Idea from T. Coquand).
  Assume $A : \text{Set}$, $g : (A \to \text{NStream}) \to A \to \text{NStream}$.
  Then $\mu_A(g) : A \to \text{NStream}$.
The $\mu$-Operator

- Instead try a constructor $\mu$. (Idea from T. Coquand).
  Assume $A : \text{Set}$, $g : (A \to \text{NStream}) \to A \to \text{NStream}$.
  Then $\mu_A(g) : A \to \text{NStream}$.

- $f$ as above can be defined as $\mu_N(\lambda g, n. \text{cons}(n, g(n + 1)))$. 
The $\mu$-Operator

- Instead try a constructor $\mu$. (Idea from T. Coquand).
  Assume $A : \text{Set}$, $g : (A \to \text{NStream}) \to A \to \text{NStream}$.
  Then $\mu_A(g) : A \to \text{NStream}$.

- $f$ as above can be defined as $\mu_N(\lambda g, n. \text{cons}(n, g(n + 1)))$.

- $\mu$ is a constructor $\Rightarrow$ recursion evaluated only when applying $\text{elim}$.
The $\mu$-Operator

- Instead try a constructor $\mu$. (Idea from T. Coquand).
  Assume $A : \text{Set}$, $g : (A \to \text{NStream}) \to A \to \text{NStream}$.
  Then $\mu_A(g) : A \to \text{NStream}$.

- $f$ as above can be defined as $\mu_N(\lambda g, n.\text{cons}(n, g(n + 1)))$.

- $\mu$ is a constructor $\Rightarrow$ recursion evaluated only when applying $\text{elim}$.

- In order to define $\text{elim}$, we need to apply $\text{elim}$ to the body of $\mu$. 
The \( \mu \)-Operator

- Instead try a constructor \( \mu \). (Idea from T. Coquand).
  Assume \( A : \text{Set} \), \( g : (A \to \text{NStream}) \to A \to \text{NStream} \).
  Then \( \mu_A(g) : A \to \text{NStream} \).

- \( f \) as above can be defined as \( \mu_N(\lambda g, n. \text{cons}(n, g(n + 1))) \).

- \( \mu \) is a constructor \( \Rightarrow \) recursion evaluated only when applying \( \text{elim} \).

- In order to define \( \text{elim} \), we need to apply \( \text{elim} \) to the body of \( \mu \).
  Better: replace the type of \( g \) above by:

\[
g : (A \to \text{NStream}) \to A \to F(\text{NStream})
\]
The $\mu$-Operator

- Instead try a constructor $\mu$. (Idea from T. Coquand).
  Assume $A : \text{Set}$, $g : (A \to \text{NStream}) \to A \to \text{NStream}$.
  Then $\mu_A(g) : A \to \text{NStream}$.

- $f$ as above can be defined as $\mu_N(\lambda g, n. \text{cons}(n, g(n+1)))$.

- $\mu$ is a constructor $\Rightarrow$ recursion evaluated only when applying $\text{elim}$.

- In order to define $\text{elim}$, we need to apply $\text{elim}$ to the body of $\mu$.
  Better: replace the type of $g$ above by:

  $$g : (A \to \text{NStream}) \to A \to F(\text{NStream})$$

- Now define $\text{elim}(\mu_A(g, a)) = g(\mu_A(g), a)$.
Problems with the $\mu$-Operator

Example:

$$f = \mu \{ \ast \} (\lambda g, x. \text{elim}(g(x))).$$

$$\text{elim}(f(x)) = (\lambda g, x. \text{elim}(g(x)))(f, x) = \text{elim}(f(x)).$$

Not normalizing. Instead demand $\ast$ in $\mu A (\lambda g, x.t)$, $\text{elim}$ should not be applied to a term depending on $g$.

Thierry Coquand calls such $t$ guarded. (He demands as well one constructor to the outside. Automatically fulfilled because of the type of $t$).

Principle of guarded induction:

Elements of $F_{\infty}(A)$ are introduced by $\mu$ applied to guarded $\mu$-terms.

$\mu$ generalizes to arbitrary $F_{\infty}$.
Problems with the $\mu$-Operator

- Example: $f = \mu\{\ast\}(\lambda g, x.\text{elim}(g(x)))$. 
Problems with the $\mu$-Operator

- Example: $f = \mu\{\ast\}(\lambda g, x. \text{elim}(g(x)))$.
  
  - $\text{elim}(f(x)) = (\lambda g, x. \text{elim}(g(x)))(f, x) = \text{elim}(f(x))$. 
  
Thierry Coquand calls such $t$-terms guarded. (He demands as well one constructor to the outside. Automatically fulfilled because of the type of $t$.)

$\mu$ generalizes to arbitrary $F_{\infty}$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Problems with the $\mu$-Operator

- Example: $f = \mu\{\ast\}(\lambda g, x.\text{elim}(g(x)))$.
  
  - $\text{elim}(f(x)) = (\lambda g, x.\text{elim}(g(x)))(f, x) = \text{elim}(f(x))$.

  Not normalizing.
Problems with the $\mu$-Operator

- Example: $f = \mu\{\ast\}(\lambda g, x.\text{elim}(g(x)))$.
  - $\text{elim}(f(x)) = (\lambda g, x.\text{elim}(g(x)))(f, x) = \text{elim}(f(x))$.
    Not normalizing. Instead demand
    - In $\mu_A(\lambda g, x.t)$, elim should not be applied to a term depending on $g$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Problems with the \( \mu \)-Operator

- Example: \( f = \mu \{ \ast \}(\lambda g, x. \text{elim}(g(x))) \).
  
  \[ \begin{align*}
  \text{elim}(f(x)) &= (\lambda g, x. \text{elim}(g(x)))(f, x) = \text{elim}(f(x)). \\
  \text{Not normalizing. Instead demand}
  \end{align*} \]

- Thierry Coquand calls such \( t \) guarded.
  (He demands as well one constructor to the outside. Automatically fulfilled because of the type of \( t \).)
Problems with the $\mu$-Operator

- Example: $f = \mu\{\ast\}(\lambda g, x.\text{elim}(g(x)))$.
  - $\text{elim}(f(x)) = (\lambda g, x.\text{elim}(g(x)))(f, x) = \text{elim}(f(x))$.
    Not normalizing. Instead demand
    $\ast$ In $\mu_A(\lambda g, x.t)$, elim should not be applied to a term depending on $g$.

- Thierry Coquand calls such $t$ guarded.
  (He demands as well one constructor to the outside. Automatically fulfilled because of the type of $t$).

- Principle of guarded induction:
  Elements of $F^\infty(A)$ are introduced by $\mu$ applied to guarded $\mu$-terms.
Problems with the $\mu$-Operator

- **Example:** $f = \mu\{\ast\}(\lambda g, x.\text{elim}(g(x)))$.
  
  - $\text{elim}(f(x)) = (\lambda g, x.\text{elim}(g(x)))(f, x) = \text{elim}(f(x))$.
  
  Not normalizing. Instead demand
  
  * In $\mu_A(\lambda g, x.t)$, elim should not be applied to a term depending on $g$.

- Thierry Coquand calls such $t$ **guarded**.
  (He demands as well one constructor to the outside. Automatically fulfilled because of the type of $t$).

- **Principle of guarded induction:**
  Elements of $F^\infty(A)$ are introduced by $\mu$ applied to guarded $\mu$-terms.

- $\mu$ generalizes to arbitrary $F^\infty$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Generating $\mu$-Terms

We had:

- If $B$: Set, $f: B \to F(\infty(cont(B) + fin(F(\infty(A)))))$, then $repeat(B,f,b): F(\infty(A))$.

Define for $f$ above $\bar{f}: (B \to F(\infty(A))) \to B \to F(F(\infty(A)))$,

$\bar{f}(g,b) = F(h_0)(F(F(\infty(h)))(f(b)))$,

where $h: (cont(B) + fin(F(\infty(A)))) \to F(\infty(A))$,

$h(cont(b)) = g(b)$,

$h(fin(p)) = p$.

And $h_0: F(\infty(F(\infty(A)))) \to F(\infty(A))$.

Then $repeat(B,f,b) = \mu B(\bar{f},b)$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Generating $\mu$-Terms

- We had:

Define for $f$ above $\tilde{f} : (B \to F^\infty(A)) \to B \to F(F^\infty(A))$,

$$\tilde{f}(g,b) = F(h_0)(F(F^\infty(h))(f(b)))$$

where $h : (\text{cont}(B) + \text{fin}(F^\infty(A))) \to F^\infty(A)$,

$$h(\text{cont}(b)) = g(b)$$

and $h_0 : F^\infty(F^\infty(A)) \to F^\infty(A)$.

Then $\text{repeat}(B,f,b) = \mu B(\tilde{f},b)$.
Generating $\mu$-Terms

- We had:

  If $B : \text{Set}$, $f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$, $b : B$, then
  \[ \text{repeat}(B, f, b) : F^\infty(A). \]
Generating μ-Terms

- We had:
  - If \( B : \text{Set} \), \( f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A)))) \), \( b : B \), then \( \text{repeat}(B, f, b) : F^\infty(A) \).

- Define for \( f \) above

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
We had:
- If $B : \text{Set}$, $f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$, $b : B$, then $\text{repeat}(B, f, b) : F^\infty(A)$.

Define for $f$ above

$\tilde{f} : (B \to F^\infty(A)) \to B \to F(F^\infty(A))$.
Generating $\mu$-Terms

- We had:
  
  If $B : \text{Set}$, $f : B \rightarrow F(F^\omega(\text{cont}(B) + \text{fin}(F^\omega(A))))$, $b : B$, then
  
  \[ \text{repeat}(B, f, b) : F^\omega(A). \]

- Define for $f$ above

\[
\tilde{f} : (B \rightarrow F^\omega(A)) \rightarrow B \rightarrow F(F^\omega(A)), \\
\tilde{f}(g, b) = F(h_0)(F(F^\omega(h))(f(b)))
\]
Generating $\mu$-Terms

- We had:
  
  If $B : \text{Set}$, $f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$, $b : B$, then
  
  \[
  \text{repeat}(B, f, b) : F^\infty(A).
  \]

- Define for $f$ above

\[
\tilde{f} : (B \to F^\infty(A)) \to B \to F(F^\infty(A)),
\]

\[
\tilde{f}(g, b) = F(h_0)(F(F^\infty(h))(f(b))),
\]

where

\[
h : (\text{cont}(B) + \text{fin}(F^\infty(A))) \to F^\infty(A),
\]

\[
h(\text{cont}(b)) = g(b),
\]

\[
h(\text{fin}(p)) = p.
\]
Generating $\mu$-Terms

- We had:

  - If $B : \text{Set}$, $f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$, $b : B$, then
    \[
    \text{repeat}(B, f, b) : F^\infty(A).
    \]

- Define for $f$ above

  \[
  \tilde{f} : (B \to F^\infty(A)) \to B \to F(F^\infty(A)),
  \]

  \[
  \tilde{f}(g, b) = F(h_0)(F(F^\infty(h))(f(b))),
  \]

  where

  \[
  h : (\text{cont}(B) + \text{fin}(F^\infty(A))) \to F^\infty(A),
  \]

  \[
  h(\text{cont}(b)) = g(b),
  \]

  \[
  h(\text{fin}(p)) = p.
  \]

  and

  \[
  h_0 : F^\infty(F^\infty(A)) \to F^\infty(A).
  \]
Generating $\mu$-Terms

- We had:
  
  If $B : \text{Set}$, $f : B \to F(F^{\infty}(\text{cont}(B) + \text{fin}(F^{\infty}(A))))$, $b : B$, then
  
  \[
  \text{repeat}(B, f, b) : F^{\infty}(A).
  \]

- Define for $f$ above

  \[
  \tilde{f} : (B \to F^{\infty}(A)) \to B \to F(F^{\infty}(A)),
  
  \tilde{f}(g, b) = F(h_0)(F(F^{\infty}(h))(f(b))),
  
  \text{where}
  
  h : (\text{cont}(B) + \text{fin}(F^{\infty}(A))) \to F^{\infty}(A),
  
  h(\text{cont}(b)) = g(b),
  
  h(\text{fin}(p)) = p.
  
  \text{and}
  
  h_0 : F^{\infty}(F^{\infty}(A)) \to F^{\infty}(A).
  
- Then
  
  \[\text{repeat}(B, f, b) = \mu_B(\tilde{f}, b).\]
Comparison of $\mu$ and repeat

Consider for $f$:

$$B \rightarrow F \left( F^{\infty}(\text{cont}(B)) + \text{fin}(F^{\infty}(A)) \right)$$

$\tilde{f} := \lambda g,b. F (h_0)(F (F^{\infty}(h))(f(b)))$.

Now $\tilde{f}$ is "extended guarded": no elim applied to a term containing $g$.

- But now infinitely many constructors of $F$ (even unbounded chains) can be applied to it.
- No longer syntactic condition.
- Subsumes all cases of functions definable by guarded induction principle, but extends this notion.

If we replace type of $f$ by $F(\text{cont}(B) + \text{fin}(F^{\infty}(A)))$ then $\tilde{f}$ can be defined by the guarded induction principle.

Suffices (together with $\text{At}$) to define intro.
Comparison of $\mu$ and repeat

- Consider for $f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$

  $\tilde{f} := \lambda g, b. F(h_0)(F(F^\infty(h))(f(b)))$.

- Now $\tilde{f}$ is "extended guarded": no elim applied to a term containing $g$.
- But now infinitely many constructors of $F$ (even unbounded chains) can be applied to it.
- No longer syntactic condition.
- Subsumes all cases of functions definable by guarded induction principle, but extends this notion.

- If we replace type of $f$ by $F(\text{cont}(B) + \text{fin}(F^\infty(A)))$ then $\tilde{f}$ can be defined by the guarded induction principle.
- Suffices (together with $\text{At}$) to define $\text{intro}$.
Comparison of $\mu$ and repeat

- Consider for $f : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$
  \[ \tilde{f} := \lambda g, b. F(h_0)(F(F^\infty(h))(f(b))). \]

- Now $\tilde{f}$ is “extended guarded”: no elim applied to a term containing $g$. 

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Comparison of $\mu$ and repeat

- Consider for $f : B \rightarrow F(F^{\infty}(\text{cont}(B) + \text{fin}(F^{\infty}(A))))$
  $$\tilde{f} := \lambda g, b. F(h_0)(F(F^{\infty}(h))(f(b))).$$

- Now $\tilde{f}$ is "extended guarded": no elim applied to a term containing $g$.
  - But now infinitely many constructors of $F$ (even unbounded chains) can be applied to it.
Comparison of $\mu$ and repeat

- Consider for $f : B \to F(F^{\infty}(\text{cont}(B) + \text{fin}(F^{\infty}(A))))$
  $\tilde{f} := \lambda g, b. F(h_0)(F(F^{\infty}(h))(f(b))).$

- Now $\tilde{f}$ is "extended guarded": no elim applied to a term containing $g$.
  - But now infinitely many constructors of $F$ (even unbounded chains) can be applied to it.
  - No longer syntactic condition.

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Comparison of \( \mu \) and repeat

Consider for \( f : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A)))) \)
\[ \tilde{f} := \lambda g, b. F(h_0)(F(F^\infty(h))(f(b))). \]

Now \( \tilde{f} \) is “extended guarded”: no elim applied to a term containing \( g \).
- But now infinitely many constructors of \( F \) (even unbounded chains) can be applied to it.
- No longer syntactic condition.
- Subsumes all cases of functions definable by guarded induction principle, but extends this notion.
Comparison of $\mu$ and repeat

- Consider for $f : B \to F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$

  \[ \tilde{f} := \lambda g, b. F(h_0)(F(F^\infty(h))(f(b))). \]

- Now $\tilde{f}$ is “extended guarded”: no elim applied to a term containing $g$.
  - But now infinitely many constructors of $F$ (even unbounded chains) can be applied to it.
  - No longer syntactic condition.
  - Subsumes all cases of functions definable by guarded induction principle, but extends this notion.

- If we replace type of $f$ by $F(\text{cont}(B) + \text{fin}(F^\infty(A)))$
  then $\tilde{f}$ can be defined by the guarded induction principle.
Comparison of $\mu$ and repeat

- Consider for $f : B \rightarrow F(F^\infty(\text{cont}(B) + \text{fin}(F^\infty(A))))$
  \[ \tilde{f} := \lambda g, b. F(h_0)(F(F^\infty(h))(f(b))). \]

- Now $\tilde{f}$ is “extended guarded”: no elim applied to a term containing $g$.
  - But now infinitely many constructors of $F$ (even unbounded chains) can be applied to it.
  - No longer syntactic condition.
  - Subsumes all cases of functions definable by guarded induction principle, but extends this notion.

- If we replace type of $f$ by $F(\text{cont}(B) + \text{fin}(F^\infty(A)))$ then $\tilde{f}$ can be defined by the guarded induction principle.
  - Suffices (together with At) to define intro.
Therefore functions definable by guarded induction principle and by our rules are the same.
Therefore functions definable by guarded induction principle and by our rules are the same.

**Formal Calculus and Implementations**

- Formal rules = recursion operator.
- Implemented ones = µ + guardedness check + termination check.
- Syntactic condition.

**Anton Setzer, Peter Hancock: Coalgebras in dependent type theory**
Therefore functions definable by guarded induction principle and by our rules are the same.

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Formal Calculus and Implementations

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Therefore functions definable by guarded induction principle and by our rules are the same.

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td></td>
</tr>
<tr>
<td>Formal rules = implemented ones. (Constructor).</td>
<td></td>
</tr>
</tbody>
</table>
- Therefore functions definable by guarded induction principle and by our rules are the same.

**Formal Calculus and Implementations**

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td>Formal rules = implemented ones. (Constructor).</td>
<td>Formal rules = implemented ones. (elim or case distinction)</td>
</tr>
</tbody>
</table>
Therefore functions definable by guarded induction principle and by our rules are the same.

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td></td>
</tr>
<tr>
<td>Formal rules = implemented ones. (Constructor).</td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td></td>
<td>Formal rules = implemented ones. (elim or case distinction)</td>
</tr>
<tr>
<td><strong>Elimination Rules</strong></td>
<td></td>
</tr>
<tr>
<td>Formal rules = recursion operator.</td>
<td></td>
</tr>
</tbody>
</table>
Therefore functions definable by guarded induction principle and by our rules are the same.

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td>Formal rules =</td>
<td>Formal rules =</td>
</tr>
<tr>
<td>implemented ones.</td>
<td>implemented ones.</td>
</tr>
<tr>
<td>(Constructor).</td>
<td>(elim or case distinction)</td>
</tr>
<tr>
<td><strong>Elimination Rules</strong></td>
<td></td>
</tr>
<tr>
<td>Formal rules =</td>
<td></td>
</tr>
<tr>
<td>recursion operator.</td>
<td></td>
</tr>
<tr>
<td>Implemented ones=</td>
<td></td>
</tr>
<tr>
<td>pattern matching</td>
<td></td>
</tr>
<tr>
<td>+termination check</td>
<td></td>
</tr>
</tbody>
</table>

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Therefore functions definable by guarded induction principle and by our rules are the same.

## Formal Calculus and Implementations

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td>Formal rules = implemented ones. (Constructor).</td>
<td>Formal rules = implemented ones. (elim or case distinction)</td>
</tr>
<tr>
<td><strong>Elimination Rules</strong></td>
<td></td>
</tr>
<tr>
<td>Formal rules = recursion operator. Implemented ones= pattern matching + termination check Syntactic condition.</td>
<td></td>
</tr>
</tbody>
</table>
Therefore functions definable by guarded induction principle and by our rules are the same.

### Formal Calculus and Implementations

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td>Formal rules = implemented ones. (Constructor).</td>
<td>Formal rules = implemented ones. (elim or case distinction)</td>
</tr>
<tr>
<td><strong>Elimination Rules</strong></td>
<td><strong>Introduction Rules</strong></td>
</tr>
<tr>
<td>Formal rules = recursion operator. Implemented ones = pattern matching + termination check</td>
<td>Formal rules = intro.</td>
</tr>
<tr>
<td>Syntactic condition.</td>
<td></td>
</tr>
</tbody>
</table>
Therefore functions definable by guarded induction principle and by our rules are the same.

Formal Calculus and Implementations

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td>Formal rules = implemented ones.</td>
<td>Formal rules = implemented ones. (elim or case distinction)</td>
</tr>
<tr>
<td>(Constructor).</td>
<td></td>
</tr>
<tr>
<td><strong>Elimination Rules</strong></td>
<td><strong>Introduction Rules</strong></td>
</tr>
<tr>
<td>Implemented ones= pattern matching</td>
<td>Implemented ones= ( \mu ) + guardedness check</td>
</tr>
<tr>
<td>+termination check</td>
<td>+At (Atom)</td>
</tr>
<tr>
<td>Syntactic condition.</td>
<td></td>
</tr>
</tbody>
</table>
• Therefore functions definable by guarded induction principle and by our rules are the same.

## Formal Calculus and Implementations

<table>
<thead>
<tr>
<th>Algebras</th>
<th>Coalgebras</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Rules</strong></td>
<td><strong>Elimination Rules</strong></td>
</tr>
<tr>
<td>Formal rules =</td>
<td>Formal rules =</td>
</tr>
<tr>
<td>implemented ones.</td>
<td>implemented ones.</td>
</tr>
<tr>
<td>(Constructor)</td>
<td>(elim or case distinction)</td>
</tr>
<tr>
<td><strong>Elimination Rules</strong></td>
<td><strong>Introduction Rules</strong></td>
</tr>
<tr>
<td>Formal rules =</td>
<td>Formal rules =</td>
</tr>
<tr>
<td>recursion operator.</td>
<td>intro.</td>
</tr>
<tr>
<td>Implemented ones=</td>
<td>Implemented ones=</td>
</tr>
<tr>
<td>pattern matching</td>
<td>µ + guardedness check</td>
</tr>
<tr>
<td>+termination check</td>
<td>+At (Atom)</td>
</tr>
<tr>
<td>Syntactic condition.</td>
<td>Syntactic condition.</td>
</tr>
</tbody>
</table>

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
• Rules for coiteration seem to be the appropriate ones.
• $\mu$-operator = correct principle for implementations.
• Both allow to define non-terminating programs in a hopefully normalizing type theory.
• Model can be defined.
• Normalization still to be shown.
• Extension to dependent coalgebras exists.

Anton Setzer, Peter Hancock: Coalgebras in dependent type theory
Conclusion

- Rules for coiteration seem to be the appropriate ones.
Conclusion

- Rules for coiteration seem to be the appropriate ones.
- $\mu$-operator = correct principle for implementations.
Conclusion

- Rules for coiteration seem to be the appropriate ones.

- $\mu$-operator = correct principle for implementations.

- Both allow to define non-terminating programs in a hopefully normalizing type theory.
Conclusion

- Rules for coiteration seem to be the appropriate ones.

- $\mu$-operator $= \text{correct principle for implementations.}$

- Both allow to define non-terminating programs in a hopefully normalizing type theory.

- Model can be defined.
Conclusion

- Rules for coiteration seem to be the appropriate ones.

- $\mu$-operator = correct principle for implementations.

- Both allow to define non-terminating programs in a hopefully normalizing type theory.

- Model can be defined.

- Normalization still to be shown.
Conclusion

- Rules for coiteration seem to be the appropriate ones.

- $\mu$-operator = correct principle for implementations.

- Both allow to define non-terminating programs in a hopefully normalizing type theory.

- Model can be defined.

- Normalization still to be shown.

- Extension to dependent coalgebras exists.
  Dependent introduction rule for (dependent) coalgebras = analogue of dependent elimination rule for algebras.