Mixed induction/coinduction from suspension types

Thorsten Altenkirch
(joint work with Nils Anders Danielsson)

School of Computer Science
University of Nottingham

October 15, 2010

Do we need codata?

- Conventional approach: data and codata.

Do we need codata?

- Conventional approach: data and codata.
- Reflects the duality of initial algebras and terminal coalgebras.

Do we need codata?

- Conventional approach: data and codata.
- Reflects the duality of initial algebras and terminal coalgebras.
- Designing ΠΣ: a core language for dependently typed programming.

Do we need codata?

- Conventional approach: data and codata.
- Reflects the duality of initial algebras and terminal coalgebras.
- Designing ΠΣ: a core language for dependently typed programming.
- Inductive types are defined using recursion, e.g.

$$
\begin{array}{l}
\mathit{Nat} \;=\; (\mathit{l} \;:\; \{\mathit{zero}\ \mathit{succ}\})\ * \\
\quad \mathit{case}\ \mathit{l}\ \mathit{of}\ \{ \\
\qquad \mathit{zero} \to \{\mathit{unit}\} \\
\quad \mid\ \mathit{succ} \to \mathit{Nat}\}
\end{array}
$$

Do we need codata?

- Conventional approach: data and codata.
- Reflects the duality of initial algebras and terminal coalgebras.
- Designing ΠΣ: a core language for dependently typed programming.
- Inductive types are defined using recursion, e.g.

$$
\begin{array}{l}
\mathit{Nat} \;=\; (\mathit{l} \;:\; \{\mathit{zero}\ \mathit{succ}\})\ * \\
\quad \mathit{case}\ \mathit{l}\ \mathit{of}\ \{ \\
\qquad \mathit{zero} \to \{\mathit{unit}\} \\
\quad \mid\ \mathit{succ} \to \mathit{Nat}\}
\end{array}
$$

- How to distinguish inductive and coinductive types?

Do we need codata?

- Conventional approach: data and codata.
- Reflects the duality of initial algebras and terminal coalgebras.
- Designing ΠΣ: a core language for dependently typed programming.
- Inductive types are defined using recursion, e.g.

$$
\begin{array}{l}
\mathit{Nat} \;=\; (\mathit{l} \;:\; \{\mathit{zero}\ \mathit{succ}\})\ * \\
\quad \mathit{case}\ \mathit{l}\ \mathit{of}\ \{ \\
\qquad \mathit{zero} \to \{\mathit{unit}\} \\
\quad \mid\ \mathit{succ} \to \mathit{Nat}\}
\end{array}
$$

- How to distinguish inductive and coinductive types?
- Do we need to?

Do we need codata?

- Conventional approach: data and codata.
- Reflects the duality of initial algebras and terminal coalgebras.
- Designing ΠΣ: a core language for dependently typed programming.
- Inductive types are defined using recursion, e.g.

$$
\begin{array}{l}
\mathit{Nat} \;=\; (\mathit{l} \;:\; \{\mathit{zero}\ \mathit{succ}\})\ * \\
\quad \mathit{case}\ \mathit{l}\ \mathit{of}\ \{ \\
\qquad \mathit{zero} \to \{\mathit{unit}\} \\
\quad \mid\ \mathit{succ} \to \mathit{Nat}\}
\end{array}
$$

- How to distinguish inductive and coinductive types?
- Do we need to?
- Yes, symbolic evaluation behaves different for inductive and coinductive types.

Agda

- Implemented by Ulf Norell and others,
  Standard library by Nils Anders Danielsson
- See `http://wiki.portal.chalmers.se/agda/`
  (or google Agda wiki).
- Agda first used **codata**.

Agda

- Implemented by Ulf Norell and others,
  Standard library by Nils Anders Danielsson
- See `http://wiki.portal.chalmers.se/agda/`
  (or google Agda wiki).
- Agda first used **codata**.
- But recently we changed to *suspension types*.

Agda

- Implemented by Ulf Norell and others,
  Standard library by Nils Anders Danielsson
- See `http://wiki.portal.chalmers.se/agda/`
  (or google Agda wiki).
- Agda first used **codata**.
- But recently we changed to *suspension types*.
- which are convenient for mixed inductive/coinductive definitions.

Agda

- Implemented by Ulf Norell and others,
  Standard library by Nils Anders Danielsson
- See `http://wiki.portal.chalmers.se/agda/`
  (or google Agda wiki).
- Agda first used **codata**.
- But recently we changed to *suspension types*.
- which are convenient for mixed inductive/coinductive definitions.
- This is the topic of today's talk. . .

Suspension types

- We introduce a new basic type former:

$$\infty \ : \ \mathit{Set} \to \mathit{Set}$$

where $\infty\, A$ means suspended computations of type A.

Suspension types

- We introduce a new basic type former:

$$\infty \ : \ \mathit{Set} \to \mathit{Set}$$

  where $\infty\ A$ means suspended computations of type A.
- with a constructor

$$\sharp \ : \ A \to \infty\ A$$

  where $\sharp\ a$ turns a computation a into a value.

Suspension types

- We introduce a new basic type former:

$$\infty \; : \; \mathit{Set} \to \mathit{Set}$$

  where $\infty\, A$ means suspended computations of type A.
- with a constructor

$$\sharp \; : \; A \to \infty\, A$$

  where $\sharp\, a$ turns a computation a into a value.
- and a destructor

$$\flat \; : \; \infty\, A \to A$$

  which *forces* a delayed computation:

$$\flat\, (\sharp\, x) \; = \; x$$

Codata from ∞

- We can now turn an inductive type

$$
\begin{array}{l}
\textbf{data}\ \mathit{List}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\
\quad [\,] \quad\quad :\ \mathit{List}\ A \\
\quad \_ :: \_\ :\ (x\ :\ A)\ (\mathit{xs}\ :\ \mathit{List}\ A) \to \mathit{List}\ A
\end{array}
$$

Codata from ∞

- We can now turn an inductive type

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad [\,]\ \ \ \ :\ \mathit{List}\ A\\ \quad \_::\_\ :\ (x\ :\ A)\ (\mathit{xs}\ :\ \mathit{List}\ A) \to \mathit{List}\ A\end{array}$$

- into a coinductive type by inserting ∞

$$\begin{array}{l}\textbf{data}\ \mathit{Colist}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad [\,]\ \ \ \ :\ \mathit{Colist}\ A\\ \quad \_::\_\ :\ (x\ :\ A)\ (\mathit{xs}\ :\ \infty\ (\mathit{Colist}\ A)) \to \mathit{Colist}\ A\end{array}$$

Corecursive programs using $\sharp$ and $\flat$

- Using $\sharp$ we can now define infinite objects:

$$\begin{aligned} &\mathit{from} \;:\; \mathbb{N} \to \mathit{Colist}\;\mathbb{N} \\ &\mathit{from}\; n \;=\; n :: \sharp\,(\mathit{from}\;(n + 1)) \end{aligned}$$

Corecursive programs using $\sharp$ and $\flat$

- Using $\sharp$ we can now define infinite objects:

$$\begin{array}{l} \mathit{from} \;:\; \mathbb{N} \to \mathit{Colist}\;\mathbb{N} \\ \mathit{from}\;n \;=\; n :: \sharp\,(\mathit{from}\;(n+1)) \end{array}$$

- we use $\flat$ in the definition of *map*:

$$\begin{array}{l} \mathit{map} \;:\; (A \to B) \to \mathit{Colist}\;A \to \mathit{Colist}\;B \\ \mathit{map}\;f\;[\,] \;=\; [\,] \\ \mathit{map}\;f\;(x :: xs) \;=\; f\;x :: \sharp\;\mathit{map}\;f\;(\flat\;xs) \end{array}$$

Corecursive proofs

- We define extensional equality of Colists (usually called bisimilarity):

$$
\begin{array}{l}
\textbf{data}\ \_\sim\_ : \mathit{Colist}\ A \to \mathit{Colist}\ A \to \mathit{Set}\ \textbf{where} \\
\quad [] \quad\quad : [] \sim [] \\
\quad \_::\_ : x \equiv y \to \infty\,(\flat\ \mathit{xs} \sim \flat\ \mathit{ys}) \to (x :: \mathit{xs}) \sim (y :: \mathit{ys})
\end{array}
$$

Corecursive proofs

- We define extensional equality of Colists (usually called bisimilarity):

$$\begin{array}{l}\textbf{data}\ \_\sim\_ : \mathit{Colist}\ A \to \mathit{Colist}\ A \to \mathit{Set}\ \textbf{where} \\ \quad [] \quad\quad : [] \sim [] \\ \quad \_::\_ : x \equiv y \to \infty\,(\flat\ \mathit{xs} \sim \flat\ \mathit{ys}) \to (x :: \mathit{xs}) \sim (y :: \mathit{ys})\end{array}$$

- Compare with the usual impredicative definition of bisimilarity.

Corecursive proofs

- We define extensional equality of Colists (usually called bisimilarity):

$$\begin{array}{l}\textbf{data}\ \_\sim\_ : \mathit{Colist}\ A \to \mathit{Colist}\ A \to \mathit{Set}\ \textbf{where}\\ \quad [\,] \quad\ : [\,] \sim [\,]\\ \quad \_::\_ : x \equiv y \to \infty\,(\flat\ \mathit{xs} \sim \flat\ \mathit{ys}) \to (x :: \mathit{xs}) \sim (y :: \mathit{ys})\end{array}$$

- Compare with the usual impredicative definition of bisimilarity.
- A simple corecursive proof:

$$\begin{array}{l}\mathit{lem} : \forall\,\{n\} \to \mathit{from}\,(\mathit{suc}\ n) \sim \mathit{map}\ \mathit{suc}\,(\mathit{from}\ n)\\ \mathit{lem} = \mathit{refl} :: \sharp\ \mathit{lem}\end{array}$$

Termination analysis

- We use a variant of size change termination.

Termination analysis

- We use a variant of size change termination.
- To deal with infinite objects we add a virtual argument to all functions.

Termination analysis

- We use a variant of size change termination.
- To deal with infinite objects we add a virtual argument to all functions.
- Any use of $\sharp$ reduces this argument (and preserves all others).

Termination analysis

- We use a variant of size change termination.
- To deal with infinite objects we add a virtual argument to all functions.
- Any use of $\sharp$ reduces this argument (and preserves all others).
- We allow lexical combinations of the order for several arguments.

Mixed induction/coinduction: Stream processors

- A stream is a colist without [ ]:

$$\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}$$
$$\_ :: \_\ :\ A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A$$

Mixed induction/coinduction: Stream processors

- A stream is a colist without [ ]:

$$\begin{array}{l}\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \_::\_\ :\ A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A\end{array}$$

- We define a type of stream processors representing functions on streams:

$$\begin{array}{l}\textbf{data}\ \mathit{SP}\ (A\ B\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{get}\ :\ (A \to \mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B\\ \quad \mathit{put}\ :\ B \to \infty\ (\mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B\end{array}$$

Mixed induction/coinduction: Stream processors

- A stream is a colist without [ ]:

$$\begin{array}{l}\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \_::\_\ :\ A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A\end{array}$$

- We define a type of stream processors representing functions on streams:

$$\begin{array}{l}\textbf{data}\ \mathit{SP}\ (A\ B\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{get}\ :\ (A \to \mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B\\ \quad \mathit{put}\ :\ B \to \infty\ (\mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B\end{array}$$

- Example of a stream processor:

$$\begin{array}{l}\mathit{pure}\ :\ \forall\ \{A\ B\} \to (A \to B) \to \mathit{SP}\ A\ B\\ \mathit{pure}\ f\ =\ \mathit{get}\ (\lambda\ a \to \mathit{put}\ (f\ a)\ (\sharp\ (\mathit{pure}\ f)))\end{array}$$

Mixed induction/coinduction: Stream processors

- A stream is a colist without []:

$$\begin{array}{l}\textbf{data}\ \mathit{Stream}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \_::\_\ :\ A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A\end{array}$$

- We define a type of stream processors representing functions on streams:

$$\begin{array}{l}\textbf{data}\ \mathit{SP}\ (A\ B\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{get}\ :\ (A \to \mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B\\ \quad \mathit{put}\ :\ B \to \infty\ (\mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B\end{array}$$

- Example of a stream processor:

$$\begin{array}{l}\mathit{pure}\ :\ \forall\ \{A\ B\} \to (A \to B) \to \mathit{SP}\ A\ B\\ \mathit{pure}\ f\ =\ \mathit{get}\ (\lambda\ a \to \mathit{put}\ (f\ a)\ (\sharp\ (\mathit{pure}\ f)))\end{array}$$

- SP is a mixed inductive-coinductive definition. It corresponds to $\nu X.\mu Y.(A \to Y) + (B \times X)$.

Semantics of stream processors

- We define the semantics of a stream processor:

$$
\begin{array}{lcl}
[\![\_]\!]\_ & : & \forall\,\{A\ B\} \to SP\ A\ B \to Stream\ A \to Stream\ B \\
[\![\ get\ f \quad\ ]\!]\ (a :: as) & = & [\![\ f\ a\ ]\!]\ (\flat\ as) \\
[\![\ put\ b\ sp\ ]\!]\ as & = & b :: (\sharp\ ([\![\ \flat\ sp\ ]\!]\ as))
\end{array}
$$

Semantics of stream processors

- We define the semantics of a stream processor:

$$
\begin{array}{lcl}
[\![\_]\!]\_ & : & \forall\,\{A\ B\} \to SP\ A\ B \to Stream\ A \to Stream\ B \\
[\![\ get\ f \quad\ ]\!]\ (a :: as) & = & [\![\ f\ a\ ]\!]\ (\flat\ as) \\
[\![\ put\ b\ sp\ ]\!]\ as & = & b :: (\sharp\ ([\![\ \flat\ sp\ ]\!]\ as))
\end{array}
$$

- This definition uses a lexical combination of structural recursion and corecursion.

Semantics of stream processors

- We define the semantics of a stream processor:

$$
\begin{array}{lcl}
[\![\_]\!]\_ & : & \forall\,\{A\ B\} \to SP\ A\ B \to Stream\ A \to Stream\ B \\
[\![\ get\ f \quad]\!]\ (a :: as) & = & [\![\ f\ a\ ]\!]\ (\flat\ as) \\
[\![\ put\ b\ sp\ ]\!]\ as & = & b :: (\sharp\ ([\![\ \flat\ sp\ ]\!]\ as))
\end{array}
$$

- This definition uses a lexical combination of structural recursion and corecursion.
- We can also define composition operators, e.g.

$$
\begin{array}{lcl}
\_ >\!\!>\!\!> \_ & : & \forall\,\{A\ B\ C\} \to SP\ A\ B \to SP\ B\ C \to SP\ A\ C \\
get\ f \quad >\!\!>\!\!>\ tq & = & get\ (\lambda\ a \to f\ a\ >\!\!>\!\!>\ tq) \\
put\ a\ sp\ >\!\!>\!\!>\ get\ f & = & \flat\ sp\ >\!\!>\!\!>\ f\ a \\
put\ a\ sp\ >\!\!>\!\!>\ put\ b\ tq & = & put\ b\ (\sharp\ put\ a\ sp\ >\!\!>\!\!>\ \flat\ tq)
\end{array}
$$

Semantics of stream processors

- We define the semantics of a stream processor:

$$
\begin{array}{lcl}
[\![\_]\!]\_ & : & \forall\,\{A\ B\} \to SP\ A\ B \to Stream\ A \to Stream\ B \\
[\![\ get\ f \quad\ ]\!]\ (a :: as) & = & [\![\ f\ a\ ]\!]\ (\flat\ as) \\
[\![\ put\ b\ sp\ ]\!]\ as & = & b :: (\sharp\ ([\![\ \flat\ sp\ ]\!]\ as))
\end{array}
$$

- This definition uses a lexical combination of structural recursion and corecursion.
- We can also define composition operators, e.g.

$$
\begin{array}{lcl}
\_ >\!\!>\!\!> \_ & : & \forall\,\{A\ B\ C\} \to SP\ A\ B \to SP\ B\ C \to SP\ A\ C \\
get\ f \quad >\!\!>\!\!>\ tq & = & get\ (\lambda\ a \to f\ a\ >\!\!>\!\!>\ tq) \\
put\ a\ sp\ >\!\!>\!\!>\ get\ f & = & \flat\ sp\ >\!\!>\!\!>\ f\ a \\
put\ a\ sp\ >\!\!>\!\!>\ put\ b\ tq & = & put\ b\ (\sharp\ put\ a\ sp\ >\!\!>\!\!>\ \flat\ tq)
\end{array}
$$

- Exercise: Derive this using categorical combinators (fold/unfold).

Mixed induction/coinduction: Weak bisimilarity

- The *partiality monad* allows us to represent potentially non-terminating computations:

$$\begin{array}{l}\textbf{data}\ \_\bot\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{now}\ :\ A \to A\ \bot\\ \quad \mathit{later}\ :\ \infty\ (A\ \bot) \to A\ \bot\end{array}$$

Mixed induction/coinduction: Weak bisimilarity

- The *partiality monad* allows us to represent potentially non-terminating computations:

$$\begin{aligned}&\textbf{data}\ \_\bot\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ &\quad \mathit{now}\ :\ A \to A\ \bot\\ &\quad \mathit{later}\ :\ \infty\ (A\ \bot) \to A\ \bot\end{aligned}$$

- We want to identify computations which only differ by a finite number of *later*:

$$\begin{aligned}&\textbf{data}\ \_ \approx \_\ \{A\ :\ \mathit{Set}\}\ :\ A\ \bot \to A\ \bot \to \mathit{Set}\ \textbf{where}\\ &\quad \mathit{now}\ :\ \forall\ \{a\} \to \mathit{now}\ a \approx \mathit{now}\ a\\ &\quad \mathit{later}\ :\ \forall\ \{a\ b\} \to \infty\ (\flat\ a \approx \flat\ b) \to \mathit{later}\ a \approx \mathit{later}\ b\\ &\quad \mathit{later}^{l}\ :\ \forall\ \{a\ b\} \to \flat\ a \approx b \to \mathit{later}\ a \approx b\\ &\quad \mathit{later}^{r}\ :\ \forall\ \{a\ b\} \to a \approx \flat\ b \to a \approx \mathit{later}\ b\end{aligned}$$

Mixed induction/coinduction: Weak bisimilarity

- The *partiality monad* allows us to represent potentially non-terminating computations:

$$\begin{aligned}&\textbf{data}\ \_\bot\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ &\quad \mathit{now}\ :\ A \to A\ \bot\\ &\quad \mathit{later}\ :\ \infty\ (A\ \bot) \to A\ \bot\end{aligned}$$

- We want to identify computations which only differ by a finite number of *later*:

$$\begin{aligned}&\textbf{data}\ \_ \approx \_\ \{A\ :\ \mathit{Set}\}\ :\ A\ \bot \to A\ \bot \to \mathit{Set}\ \textbf{where}\\ &\quad \mathit{now}\ :\ \forall\ \{a\} \to \mathit{now}\ a \approx \mathit{now}\ a\\ &\quad \mathit{later}\ :\ \forall\ \{a\ b\} \to \infty\ (\flat\ a \approx \flat\ b) \to \mathit{later}\ a \approx \mathit{later}\ b\\ &\quad \mathit{later}^l\ :\ \forall\ \{a\ b\} \to \flat\ a \approx b \to \mathit{later}\ a \approx b\\ &\quad \mathit{later}^r\ :\ \forall\ \{a\ b\} \to a \approx \flat\ b \to a \approx \mathit{later}\ b\end{aligned}$$

- Another example of a mixed inductive/coinductive definition.

Mixed induction/coinduction: Weak bisimilarity

- The *partiality monad* allows us to represent potentially non-terminating computations:

$$\begin{aligned}&\textbf{data}\ \_\bot\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ &\quad \mathit{now}\ :\ A \to A\ \bot\\ &\quad \mathit{later}\ :\ \infty\ (A\ \bot) \to A\ \bot\end{aligned}$$

- We want to identify computations which only differ by a finite number of *later*:

$$\begin{aligned}&\textbf{data}\ \_ \approx \_\ \{A\ :\ \mathit{Set}\}\ :\ A\ \bot \to A\ \bot \to \mathit{Set}\ \textbf{where}\\ &\quad \mathit{now}\ :\ \forall\ \{a\} \to \mathit{now}\ a \approx \mathit{now}\ a\\ &\quad \mathit{later}\ :\ \forall\ \{a\ b\} \to \infty\ (\flat\ a \approx \flat\ b) \to \mathit{later}\ a \approx \mathit{later}\ b\\ &\quad \mathit{later}^l\ :\ \forall\ \{a\ b\} \to \flat\ a \approx b \to \mathit{later}\ a \approx b\\ &\quad \mathit{later}^r\ :\ \forall\ \{a\ b\} \to a \approx \flat\ b \to a \approx \mathit{later}\ b\end{aligned}$$

- Another example of a mixed inductive/coinductive definition.
- Exercise: Prove transitivity.

Infinite types

- Using ∞ we can represent recursive types as infinite type expressions.

Infinite types

- Using ∞ we can represent recursive types as infinite type expressions.
- We define a set constructor which allows to turn suspended sets into sets:

$$\begin{array}{l}\textbf{data}\ \mathit{Rec}\ (A\ :\ \infty\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{fold}\ :\ \flat\ A \to \mathit{Rec}\ A\end{array}$$

Infinite types

- Using ∞ we can represent recursive types as infinite type expressions.
- We define a set constructor which allows to turn suspended sets into sets:

$$\begin{array}{l}\textbf{data}\ \mathit{Rec}\ (A\ :\ \infty\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{fold}\ :\ \flat\ A \to \mathit{Rec}\ A\end{array}$$

- We can now define the natural numbers recursively:

$$\begin{array}{l}\mathbb{N}\ :\ \mathit{Set}\\ \mathbb{N}\ =\ \top\ \uplus\ \mathit{Rec}\ (\sharp\ \mathbb{N})\end{array}$$

Infinite types

- Using ∞ we can represent recursive types as infinite type expressions.
- We define a set constructor which allows to turn suspended sets into sets:

$$\begin{array}{l}\textbf{data}\ \mathit{Rec}\ (A\ :\ \infty\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{fold}\ :\ \flat\ A \to \mathit{Rec}\ A\end{array}$$

- We can now define the natural numbers recursively:

$$\begin{array}{l}\mathbb{N}\ :\ \mathit{Set}\\ \mathbb{N}\ =\ \top\ \uplus\ \mathit{Rec}\ (\sharp\ \mathbb{N})\end{array}$$

- This requires an experimental modification of Agda's termination checker (*guardeness-preserving-typeconstructors*).

Infinite types

- Using ∞ we can represent recursive types as infinite type expressions.
- We define a set constructor which allows to turn suspended sets into sets:

$$\begin{array}{l}\textbf{data}\ \mathit{Rec}\ (A\ :\ \infty\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{fold}\ :\ \flat\ A \to \mathit{Rec}\ A\end{array}$$

- We can now define the natural numbers recursively:

$$\begin{array}{l}\mathbb{N}\ :\ \mathit{Set}\\ \mathbb{N}\ =\ \top\ \uplus\ \mathit{Rec}\ (\sharp\ \mathbb{N})\end{array}$$

- This requires an experimental modification of Agda's termination checker (*guardeness-preserving-typeconstructors*).
- We can only define strictly positive recursive types because Π-types do not preserve guardedness in the first argument.

Universes from recursion

- A universe is a set U of names for sets together with an interpretation $El : U \to Set$.

Universes from recursion

- A universe is a set U of names for sets together with an interpretation $El : U \to Set$.
- Defining universes usually requires induction-recursion, e.g.

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \textbf{data}\ U : Set\ \textbf{where} \\
\qquad nat : U \\
\qquad \pi : (A : U) \to (El\ A \to U) \to U \\
\quad El : U \to Set \\
\quad El\ nat = \mathbb{N} \\
\quad El\ (\pi\ A\ B) = (a : El\ A) \to El\ (B\ a)
\end{array}
$$

- However, using *Rec* we can define universes simply by mutual recursion:

mutual

$$U : Set$$
$$U = \top \uplus Rec\ (\sharp\ \Sigma\ U\ (\lambda\ A \to (El\ A \to U)))$$
$$El : U \to Set$$
$$El\ (inj_1\ \_) = \mathbb{N}$$
$$El\ (inj_2\ (fold\ (A, B))) = (x : El\ A) \to El\ (B\ x)$$

$\mu\nu$?

- Using ∞ we can represent types corresponding to $\nu X.\mu Y.FXY$.

$\mu\nu$?

- Using ∞ we can represent types corresponding to $\nu X.\mu Y.FXY$.
- E.g. $\nu X.\mu Y.(0 : Y) + (1 : X)$ can be represented as

data ZO : Set **where**
$0, : ZO \to ZO$
$1, : \infty\, ZO \to ZO$

$\mu\nu$?

- Using ∞ we can represent types corresponding to $\nu X.\mu Y.FXY$.
- E.g. $\nu X.\mu Y.(0 : Y) + (1 : X)$ can be represented as

$$\begin{array}{l}\textbf{data}\ ZO\ :\ Set\ \textbf{where}\\ \quad 0, : ZO \to ZO\\ \quad 1, : \infty\ ZO \to ZO\end{array}$$

- The alternating infinite sequence of 0s and 1s is in this type:

$$\begin{array}{l}01^{\omega}\ :\ ZO\\ 01^{\omega}\ =\ 0, (1, (\sharp\ 01^{\omega}\ \omega))\end{array}$$

$\mu\nu$?

- Using ∞ we can represent types corresponding to $\nu X.\mu Y.FXY$.
- E.g. $\nu X.\mu Y.(0 : Y) + (1 : X)$ can be represented as

$$\begin{array}{l}\textbf{data}\ ZO\ :\ Set\ \textbf{where}\\ \quad 0, : ZO \to ZO\\ \quad 1, : \infty\ ZO \to ZO\end{array}$$

- The alternating infinite sequence of 0s and 1s is in this type:

$$\begin{array}{l}01^{\omega}\ :\ ZO\\ 01^{\omega}\ =\ 0, (1, (\sharp\ 01^{\omega}\ \omega))\end{array}$$

- But what about $\mu Y.\nu X.(0 : Y) + (1 : X)$?

- $\mu Y.\nu X.(0 : Y) + (1 : X) = \mu Y.FY$
  with $FY = \nu X.(0 : Y) + (1 : X)$

- $\mu Y.\nu X.(0 : Y) + (1 : X) = \mu Y.FY$
  with $FY = \nu X.(0 : Y) + (1 : X)$
- In Agda:

```
data F (Y : Set) : Set where
  0, : Y → F Y
  1, : ∞ (F Y) → F Y
data OZ : Set where
  emb : F OZ → OZ
```

- $\mu Y.\nu X.(0 : Y) + (1 : X) = \mu Y.FY$
  with $FY = \nu X.(0 : Y) + (1 : X)$
- In Agda:

```
data F (Y : Set) : Set where
  0, : Y → F Y
  1, : ∞ (F Y) → F Y
data OZ : Set where
  emb : F OZ → OZ
```

- We expect that the infinite alternating sequence of 1s and 0s doesn't live in this type.

- $\mu Y.\nu X.(0 : Y) + (1 : X) = \mu Y.FY$
  with $FY = \nu X.(0 : Y) + (1 : X)$
- In Agda:

$$
\begin{array}{l}
\textbf{data}\ F\ (Y\ :\ Set)\ :\ Set\ \textbf{where} \\
\quad 0, : Y \to F\ Y \\
\quad 1, : \infty\ (F\ Y) \to F\ Y \\
\textbf{data}\ OZ\ :\ Set\ \textbf{where} \\
\quad emb\ :\ F\ OZ \to OZ
\end{array}
$$

- We expect that the infinite alternating sequence of 1s and 0s doesn't live in this type.
- But:

$$
\begin{array}{l}
10^{\omega}\ :\ OZ \\
10^{\omega}\ =\ emb\ (1, (\sharp\ 0, 10^{\omega}))
\end{array}
$$

Inconsistent ?

- Assume we can construct

$$OZfold \; : \; \forall \, \{A\} \to (F \; A \to A) \to OZ \to A$$

Inconsistent ?

- Assume we can construct

$$OZfold \;:\; \forall\,\{A\} \to (F\ A \to A) \to OZ \to A$$

- Then we can derive a non-terminating term:

$$\begin{aligned} &foo \;:\; F\ OZ \to OZ \\ &foo\ (0, x) \;=\; x \\ &foo\ (1, x) \;=\; emb\ (\flat\ x) \\ &bar \;:\; OZ \\ &bar \;=\; OZfold\ foo\ 10^{\omega}\ \omega \end{aligned}$$

- However, the termination checker rejects the following definition of *OZfold*:

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \mathit{OZfold} \;:\; \forall\,\{A\} \to (F\ A \to A) \to \mathit{OZ} \to A \\
\quad \mathit{OZfold}\ f\ (\mathit{emb}\ x) \;=\; f\ (\mathit{FmapOZfold}\ f\ x) \\
\quad \mathit{FmapOZfold} \;:\; \forall\,\{A\} \to (F\ A \to A) \to F\ \mathit{OZ} \to F\ A \\
\quad \mathit{FmapOZfold}\ f\ (0, x) \;=\; 0, (\mathit{OZfold}\ f\ x) \\
\quad \mathit{FmapOZfold}\ f\ (1, x) \;=\; 1, (\sharp\ \mathit{FmapOZfold}\ f\ (\flat\ x))
\end{array}
$$

- However, the termination checker rejects the following definition of *OZfold*:

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \mathit{OZfold} \;:\; \forall\,\{A\} \to (F\ A \to A) \to \mathit{OZ} \to A \\
\quad \mathit{OZfold}\ f\ (\mathit{emb}\ x) \;=\; f\ (\mathit{FmapOZfold}\ f\ x) \\
\quad \mathit{FmapOZfold} \;:\; \forall\,\{A\} \to (F\ A \to A) \to F\ \mathit{OZ} \to F\ A \\
\quad \mathit{FmapOZfold}\ f\ (0, x) \;=\; 0, (\mathit{OZfold}\ f\ x) \\
\quad \mathit{FmapOZfold}\ f\ (1, x) \;=\; 1, (\sharp\ \mathit{FmapOZfold}\ f\ (\flat\ x))
\end{array}
$$

- The reason is that it is not known that $\flat\ x$ is structurally smaller than $(1, x)$.

Summary

- Use suspension types ∞ to construct infinite data types instead of **codata**.

Summary

- Use suspension types ∞ to construct infinite data types instead of **codata**.
- Easy to define and use mixed coinductive/inductive types.
  More examples:
  - *Subtyping, Declaratively* (MPC 2010)
    TA & Nils Anders Danielsson
  - *Total Parser Combinators* (ICFP 2010)
    Nils Anders Danielsson

Summary

- Use suspension types ∞ to construct infinite data types instead of **codata**.
- Easy to define and use mixed coinductive/inductive types.
  More examples:
  - *Subtyping, Declaratively* (MPC 2010)
    TA & Nils Anders Danielsson
  - *Total Parser Combinators* (ICFP 2010)
    Nils Anders Danielsson
- Recursive types can be represented as infinite type expressions using ∞.

Summary

- Use suspension types ∞ to construct infinite data types instead of **codata**.
- Easy to define and use mixed coinductive/inductive types.
  More examples:
  - *Subtyping, Declaratively* (MPC 2010)
    TA & Nils Anders Danielsson
  - *Total Parser Combinators* (ICFP 2010)
    Nils Anders Danielsson
- Recursive types can be represented as infinite type expressions using ∞.
- Simple extension of the termination checker.

Summary

- Use suspension types ∞ to construct infinite data types instead of **codata**.
- Easy to define and use mixed coinductive/inductive types.
  More examples:
  - *Subtyping, Declaratively* (MPC 2010)
    TA & Nils Anders Danielsson
  - *Total Parser Combinators* (ICFP 2010)
    Nils Anders Danielsson
- Recursive types can be represented as infinite type expressions using ∞.
- Simple extension of the termination checker.
- Can only define $\nu\mu$, nesting not handled properly.
  Can we fix this?
  - Termination Checking Nested Inductive and Coinductive Types (PAR 2010)
    TA & Nils Anders Danielsson