A toy example

Definition (Dense relation)

Recall that a relation $<$ on a set $A$ is dense if

$$ \forall x, y : A. x < y \implies \exists z : A. x < z < y $$

- e.g. $(\mathbb{Q}, <)$ is dense, but $(\mathbb{N}, <)$ is not.
A toy example

Definition (Dense relation)
Recall that a relation $<$ on a set $A$ is dense if

$$\forall x, y : A. x < y \implies \exists z : A. x < z < y$$

- e.g. $(\mathbb{Q}, <)$ is dense, but $(\mathbb{N}, <)$ is not.
- For arbitrary $(A, <)$, consider the dense completion $(A^*, <^*)$: the “smallest” dense relation containing $(A, <)$.
A toy example

Definition (Dense relation)

Recall that a relation $<$ on a set $A$ is dense if

$$\forall x, y : A. \ x < y \implies \exists z : A. \ x < z < y$$

- e.g. $(\mathbb{Q}, <)$ is dense, but $(\mathbb{N}, <)$ is not.

- For arbitrary $(A, <)$, consider the dense completion $(A^*, <^*)$: the “smallest” dense relation containing $(A, <)$.

$$A \xrightarrow{\iota} A^* \quad \text{(dense)}$$
A toy example

Definition (Dense relation)

Recall that a relation $<$ on a set $A$ is dense if

$$\forall x, y : A. x < y \implies \exists z : A. x < z < y$$

e.g. $(\mathbb{Q}, <)$ is dense, but $(\mathbb{N}, <)$ is not.

For arbitrary $(A, <)$, consider the dense completion $(A^*, <^*)$: the “smallest” dense relation containing $(A, <)$.

$$
\begin{align*}
A & \xrightarrow{l} A^* \quad (\text{dense}) \\
\downarrow f \\
\downarrow \\
X & \quad (\text{dense})
\end{align*}
$$
A toy example

Definition (Dense relation)

Recall that a relation \(<\) on a set \(A\) is dense if

\[
\forall x, y : A. x < y \implies \exists z : A. x < z < y
\]

e.g. \((\mathbb{Q}, <)\) is dense, but \((\mathbb{N}, <)\) is not.

For arbitrary \((A, <)\), consider the dense completion \((A^*, <^*)\): the “smallest” dense relation containing \((A, <)\).

\[
\begin{array}{ccc}
A & \xrightarrow{l} & A^* \\
\downarrow f & & \downarrow \bar{f} \\
X & \xleftarrow{\bar{l}} & B^* \\
\end{array}
\]

(dense)
A toy example

Definition (Dense relation)

Recall that a relation $<$ on a set $A$ is dense if

$$\forall x, y : A . x < y \implies \exists z : A . x < z < y$$

- e.g. $(\mathbb{Q}, <)$ is dense, but $(\mathbb{N}, <)$ is not.
- For arbitrary $(A, <)$, consider the dense completion $(A^*, <^*)$: the “smallest” dense relation containing $(A, <)$.

How can we construct this?
Constructing the dense completion

- **Intuitively:**
  1. start with $A$
  2. for each pair $x < y$, add a midpoint $x < \ast \text{mid}(x, y) < \ast y$
  3. now we have new points, so add even more midpoints
  4. etc

- **Formally:** inductive-inductive definition
The dense completion in Fantasy Coq

Parameters $A : \text{Type}, \ < : A \to A \to \text{Type}.$

Inductive $A^* : \text{Type} :=$
The dense completion in Fantasy Coq

Parameters $A : \text{Type, } < : A \rightarrow A \rightarrow \text{Type.}$

Inductive $A^* : \text{Type} :=$

$| \iota : A \rightarrow A^* $
The dense completion in Fantasy Coq

Parameters $A : \text{Type}$, $< : A \to A \to \text{Type}$.

Inductive $A^* : \text{Type} :=
| \iota : A \to A^*
| \text{mid} : \forall x y : A^*, x <^* y \to A^*$
The dense completion in Fantasy Coq

Parameters A : Type, < : A → A → Type.

Inductive A^* : Type :=
  | ι : A → A^*
  | mid : forall x y : A^*, x <^* y → A^*

with <^* : A^* → A^* → Type :=
The dense completion in Fantasy Coq

Parameters $A : \text{Type}$, $\langle : A \to A \to \text{Type}$.  

Inductive $A^* : \text{Type} :=$
  $\mid \iota : A \to A^*$
  $\mid \text{mid} : \forall x y : A^*, x \langle^* y \to A^*$

with $\langle^* : A^* \to A^* \to \text{Type} :=$
  $\mid \iota^< : \forall x y : A, x \langle y \to \iota x \langle^* \iota y$


The dense completion in Fantasy Coq

Parameters A : Type, < : A -> A -> Type.

Inductive A* : Type :=
  | ι : A -> A*
  | mid : forall x y : A*, x <* y -> A*

with <* : A* -> A* -> Type :=
  | ι< : forall x y : A, x < y -> ι x <* ι y
  | mid' : forall x y : A*, forall p : x <* y, x <* mid x y p
The dense completion in Fantasy Coq

Parameters A : Type, < : A -> A -> Type.

Inductive $A^*$ : Type :=
| $\iota$ : A -> $A^*$
| mid : forall x y : $A^*$, x <* y -> $A^*$

with <* : $A^*$ -> $A^*$ -> Type :=
| $\iota^<$ : forall x y : A, x < y -> $\iota$ x <* $\iota$ y
| mid' : forall x y : $A^*$, forall p : x <* y, x <* mid x y p
| mid': forall x y : $A^*$, forall p : x <* y, mid x y p <* y.
The dense completion in Fantasy Coq

Parameters $A : \text{Type}, \prec : A \rightarrow A \rightarrow \text{Type}$.

Inductive $A^* : \text{Type} :=$
  $| \iota : A \rightarrow A^*$
  $| \text{mid} : \forall x y : A^*, x \prec^* y \rightarrow A^*$

with $\prec^* : A^* \rightarrow A^* \rightarrow \text{Type} :=$
  $| \iota^< : \forall x y : A, x < y \rightarrow \iota x \prec^* \iota y$
  $| \text{mid}' : \forall x y : A^*, \forall p : x \prec^* y, x \prec^* \text{mid} x y p$
  $| \text{mid}' : \forall x y : A^*, \forall p : x \prec^* y, \text{mid} x y p \prec^* y$.

Definition dense$_{A^*}$ $(x y : A^*)(p : x \prec^* y)$
  : $\{ z : A^* \& x \prec^* z \& z \prec^* y \}$
  := existT2 (mid x y p) (mid' x y p) (mid' x y p).
Defining $\bar{f}$

Parameters

(X : Type)((<x : X -> X -> Type)
(dense_x : forall x y : X, { z : X & x <x z & z <x y})
(f : A -> X)
(f< : forall x y : A, x < y -> (f x) <x (f y)).
Defining $\bar{f}$

$\begin{align*}
A & \xrightarrow{\bar{f}} A^* \quad \text{(dense)} \\
& \downarrow f \\
X & \xleftarrow{\bar{f}} \quad \text{(dense)}
\end{align*}$

Fixpoint $\bar{f}$ $(z : A^*) : X :=$

match $z$ with
\begin{align*}
| \iota a & \Rightarrow \{?0 : X\} \\
| \text{mid } x \ y \ p & \Rightarrow \{?1 : X\}
\end{align*}
end
Defining $\bar{f}$

```
Fixpoint $\bar{f}$ (z : $A^*$) : X :=
  match z with
    | $\iota$ a => f a
    | mid x y p => {?1 : X}
  end
```
Defining $\overline{f}$

\[
\begin{array}{ccc}
A & \xrightarrow{\lambda} & A^* \quad \text{(dense)} \\
\downarrow f & & \downarrow \overline{f} \\
X & & (\text{dense})
\end{array}
\]

Fixpoint $\overline{f}$ $(z : A^*) : X :=$

match $z$ with
| $\iota \ a$ => $f \ a$
| mid $x \ y \ p$ => proj1 (dense$_X$ $\{?_2 : X\}$ $\{?_3 : X\}$ $\{?_4 : ?_2 <_X ?_3\}$)
end
Defining $\bar{f}$

Fixpoint $\bar{f} (z : A^*) : X :=$
match $z$ with
  | $\iota$ $a$ => $f$ $a$
  | $\mid$ mid $x$ $y$ $p$ => proj1 (dense$_X$ ($\bar{f} x$) \{?3 : $X$\} \{?4 : $\bar{f} x < X ?_3$\})
end
Defining $\bar{f}$

Fixpoint $\bar{f}$ ($z : A^*$) : $X$ :=
match $z$ with
  | $\iota$ $a$ => $f$ $a$
  | mid $x$ $y$ $p$ => proj1 (dense$_X$ ($\bar{f}$ $x$) ($\bar{f}$ $y$) $\{?_4 : \bar{f} x <_X \bar{f} y\}$)
end

\[ A \xrightarrow{f} A^* \quad \text{(dense)} \]
\[ X \xleftarrow{\bar{f}} \]
Defining $\overline{f}$

Fixpoint $\overline{f}$ ($z : A^*$) : $X$ :=
match $z$ with
| $\iota$ $a$ => $f$ $a$
| mid $x$ $y$ $p$ => proj1 (dense$_X$ ($\overline{f}$ $x$) ($\overline{f}$ $y$) {\textcolor{green}{\textit{?4} : $\overline{f}$ $x$ $<$$_X$ $\overline{f}$ $y$}})
end

with $\overline{f}<$ ($x$ $y$ : $A^*$)($p : x$ $<^*$ $y$) : $\overline{f}$ $x$ $<$_$X$ $\overline{f}$ $y$ := ...
Defining $\bar{f}$

\[
\begin{array}{c}
A \\ f \\
\downarrow \bar{f} \\
X
\end{array} \quad \rightarrow \quad \begin{array}{c}
A^* \\
\downarrow \bar{f} \\
X
\end{array} \quad \text{(dense)}
\]

Fixpoint $\bar{f}$ ($z : A^*$) : $X :=$

match $z$ with
| $\iota$ $a$ => $f$ $a$
| mid $x$ $y$ $p$ => proj1 (dense$_X$ ($\bar{f}$ $x$) ($\bar{f}$ $y$) ($\bar{f}<$ $x$ $y$ $p$))
end

with $\bar{f}<$ ($x$ $y$ : $A^*$)($p$ : $x$ $<$* $y$) : $\bar{f}$ $x$ $<_X$ $\bar{f}$ $y$ := $\ldots$
Plan

1. A brief history of inductive types in type theory
2. Inductive-inductive definitions
3. Examples
4. Meta-theoretical results
A brief history of inductive types
In the beginning, there were examples
Martin-Löf (1972, 1979, 1980, …)

First accounts of Martin-Löf type theory includes examples of “inductively generated” types:

- \( \mathbb{N} \), finite sets (1972)
- W-types (1979)
- Kleene’s \( \mathcal{O} \), lists (1980)
- …

The system is considered open; new inductive types should be added as needed.

“We can follow the same pattern used to define natural numbers to introduce other inductively defined sets. We see here the example of lists.” – Martin-Löf 1980
Examples of inductive definitions

\[
\begin{align*}
\text{data } \text{List}_A : \text{Set} \text{ where } \\
[ ] : \text{List}_A \\
( x :: x s ) : \text{List}_A \\
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{O} : \text{Set} \text{ where } \\
0 : \text{O} \\
suc(n) : \text{O} \\
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{W} ( A : \text{Set}) ( B : A \rightarrow \text{Set}) : \text{Set} \\
a : A \\
f : B(a) \rightarrow \text{W}(A, B) \\
\text{sup}(a,f) : \text{W}(A, B) \\
\end{align*}
\]
Induction principles/elimination rules

- Each definition has a corresponding induction principle, stating that it is the least set closed under its constructors.
- E.g.

\[
elim_{\text{List}_A} : (P : \text{List}_A \to \text{Set}) \to \\
(\text{step[]} : P([])) \to \\
(\text{step} :: : (x : \mathbb{N}) \to (xs : \text{List}_A) \to P(xs) \to P(x :: xs)) \to \\
(y : \text{List}_A) \to P(y)
\]

\[
elim_{\text{List}_A}(P, \text{step[]} , \text{step} :: , []) = \text{step}[] \\
elim_{\text{List}_A}(P, \text{step[]} , \text{step} :: , x :: xs) = \text{step} :: (x, xs, \lim_{\text{List}_A}(\ldots , xs))
\]

- How can we talk about all inductive definitions?
Church encodings?
Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.
- E.g.

\[ \mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X \]

\[ \text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b) \]
Church encodings?
Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

  E.g.

  \[
  \mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X
  \]

  \[
  \text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b)
  \]

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).
Church encodings?

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

- E.g.

\[
\mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X : \text{Set}
\]

\[
\text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b) : \text{Set}
\]

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).
Church encodings?
Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

- E.g.

\[
\mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X : \text{Set}
\]

\[
\text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b) : \text{Set}
\]

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).
Church encodings?

Pfenning and Paulin-Mohring (1989)

- First attempt in Calculus of Constructions: use Church encodings of inductive types.

  E.g.

  \[ \mathbb{N} = (X : \text{Set}) \rightarrow X \rightarrow (X \rightarrow X) \rightarrow X : \text{Set} \]

  \[ \text{Id}_A(a, b) = (X : A \rightarrow \text{Set}) \rightarrow X(a) \rightarrow X(b) : \text{Set} \]

- Problems:
  - Uses impredicativity in an essential way.
  - Induction (dependent elimination) is not derivable in CoC for any encoding (Geuvers 2001).

- Solution: Calculus of Inductive Constructions with inductive types builtin (according to schema).
Dybjer (1994) considers constructors of the form

\[
\text{intro}_U : (A :: \sigma) \\
(b :: \beta[A]) \rightarrow \\
(u :: \gamma[A, b]) \rightarrow \\
U
\]

where

- \(\sigma\) is a sequence of types for parameters  
  [‘\(x :: Y\)’ telescope notation]
- \(\beta[A]\) is a sequence of types for non-inductive arguments.
- \(\gamma[A, b]\) is a sequence of types for inductive arguments:
  - Each \(\gamma_i[A, b]\) is of the form \(\xi_i[A, b] \rightarrow U\) (strict positivity).
The elimination and computation rules are determined by an inversion principle.

Infinite axiomatisation.

Inprecise; ‘…’ everywhere.

No way to reason about an arbitrary inductive definition inside the system (generic map etc.).
Setzer wanted to analyse the proof-theoretical strength of Dybjer’s schema version of induction-recursion.

Hard with lots of ‘. . .’ around . . .

So they developed an axiomatisation where the syntax has been internalised into the system.

Basic idea (simplified for inductive definitions) : the type is “given by constructors”, so describe the domain of the constructor

\[ \text{intro}_{U_\gamma} : \text{Arg}(\gamma, U_\gamma) \rightarrow U_\gamma \]

\[ \gamma \text{ is “code” that contains the necessary information to describe } U_\gamma. \]
Basic idea in some more detail

- Universe SP of codes for the domain of constructors of inductively defined sets. [SP stands for Strictly Positive.]

- Decoding function $\text{Arg} : \text{SP} \rightarrow \text{Set} \rightarrow \text{Set}$. [$\text{Arg}(\gamma, X)$ is the domain where $X$ is used for the inductive arguments.]

- For every $\gamma : \text{SP}$, stipulate that there is a set $U_\gamma$ and a constructor $\text{intro}_\gamma : \text{Arg}(\gamma, U_\gamma) \rightarrow U_\gamma$.

- Inversion principle for elimination and computation rules.
SP, Arg and $U_\gamma$

data SP: Set₁ where

  nil : SP
  nonind : (A : Set) → (A → SP) → SP
  ind : (A : Set) → SP → SP

Arg : SP → Set → Set
Arg nil X = 1
Arg (nonind A $\gamma$) X = (y : A) × (Arg ($\gamma$ y) X)
Arg (ind A $\gamma$) X = (A → X) × (Arg $\gamma$ X)

data U ($\gamma$ : SP) : Set where
  intro : Arg $\gamma$ (U $\gamma$) → U $\gamma$
Example: the code for $\text{List}_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_A} = \text{nil} +_{\text{SP}} \text{nonind}\left(\mathbb{N}, \lambda \_\text{ind}(1, \text{nil})\right)$$

with

$\text{List}_A : \text{Set}$

$\text{List}_A = \bigcup \gamma_{\text{List}_A}$

$[] : \text{List}_A$

$[] = \{ ?_0 : \text{List}_A \}$

$\_ : \_ : \mathbb{N} \to \text{List}_A \to \text{List}_A$

$x :: xs = \{ ?_1 : \text{List}_A \}$
Example: the code for $\text{List}_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi ::= \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_A} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda _.\text{ind}(1, \text{nil}))$$

with

$\text{List}_A : \text{Set}$

$\text{List}_A = \text{U} \ \gamma_{\text{List}_A}$

$[] : \text{List}_A$

$[] = \text{intro} \ ?2 : \text{Arg}(\gamma_{\text{List}_A}, \text{List}_A)$

$\_ : \_ : \mathbb{N} \rightarrow \text{List}_A \rightarrow \text{List}_A$

$x :: xs = \ ?1 : \text{List}_A$
Example: the code for $\text{List}_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_A} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda _. \text{ind}(1, \text{nil}))$$

with

$$\text{List}_A : \text{Set}$$

$$\text{List}_A = U \gamma_{\text{List}_A}$$

$$[] : \text{List}_A$$

$$[] = \text{intro} \quad \{?_2 : (x : 2) \times (\text{if } x \text{ then } 1 \text{ else } \mathbb{N} \times (1 \to \text{List}_A) \times 1)\}$$

$$\_ : \_ : \mathbb{N} \to \text{List}_A \to \text{List}_A$$

$$x :: xs = \{?_1 : \text{List}_A\}$$
Example: the code for \( \text{List}_A \)

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_A} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda \_\text{ind}(1, \text{nil}))
\]

with

\[
\text{List}_A : \text{Set}
\]

\[
\text{List}_A = \bigcup \gamma_{\text{List}_A}
\]

\[
[] : \text{List}_A
\]

\[
[] = \text{intro} \langle \{ ?_3 : 2 \} , \{ ?_4 : \text{if } ?_3 \text{ then } 1 \text{ else } \mathbb{N} \times \ldots \} \rangle
\]

\[
_::_ : \mathbb{N} \rightarrow \text{List}_A \rightarrow \text{List}_A
\]

\[
x ::= xs = \{ ?_1 : \text{List}_A \}
\]
Example: the code for $\text{List}_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +_{\text{SP}} \psi := \nonind(2, \lambda x. \text{ if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_A} = \text{nil} +_{\text{SP}} \nonind(\mathbb{N}, \lambda \_. \text{ind}(1, \text{nil}))
\]

with

$\text{List}_A : \text{Set}$

$\text{List}_A = U \gamma_{\text{List}_A}$

$[] : \text{List}_A$

$[] = \text{intro } \langle \text{tt}, \{4:1\} \rangle$

$\_ :: \_ : \mathbb{N} \rightarrow \text{List}_A \rightarrow \text{List}_A$

$x :: xs = \{1 : \text{List}_A\}$
Example: the code for List$_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_A} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda \_\_. \text{ind}(1, \text{nil}))$$

with

List$_A$ : Set
List$_A = U \gamma_{\text{List}_A}$

[] : List$_A$
[] = intro ⟨tt, ⋆⟩

_ :: _ : \mathbb{N} \rightarrow \text{List}_A \rightarrow \text{List}_A
x :: xs = {?1 : \text{List}_A}
Example: the code for List$_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_A} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda_. \text{ind}(1, \text{nil}))$$

with

List$_A$ : Set
List$_A$ = $\bigcup \gamma_{\text{List}_A}$

[] : List$_A$
[] = intro $\langle \text{tt}, * \rangle$

_ :: _ : $\mathbb{N} \rightarrow \text{List}_A \rightarrow \text{List}_A$
x :: xs = intro $\langle \text{ff}, \{?_5 : \mathbb{N} \times (1 \rightarrow \text{List}_A) \times 1\} \rangle$
Example: the code for \( \text{List}_A \)

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_A} = \text{nil} +_{SP} \text{nonind}(N, \lambda . \text{ind}(1, \text{nil}))
\]

with

\( \text{List}_A : \text{Set} \)

\( \text{List}_A = U \gamma_{\text{List}_A} \)

\([] : \text{List}_A\)

\([] = \text{intro} \langle \text{tt}, * \rangle\)

\(\_ :: _ : \mathbb{N} \to \text{List}_A \to \text{List}_A\)

\(x :: xs = \text{intro} \langle \text{ff}, \langle \{?6 : \mathbb{N}\} \, , \, \{?7 : 1 \to \text{List}_A\} \, , \, \{?8 : 1\} \rangle \rangle\)
Example: the code for \( \text{List}_A \)

We can encode two constructors into one using the dependency on non-inductive arguments:

\[
\gamma +_{\text{SP}} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)
\]

We have

\[
\gamma_{\text{List}_A} = \text{nil} +_{\text{SP}} \text{nonind}(\mathbb{N}, \lambda_. \text{ind}(1, \text{nil}))
\]

with

\[
\text{List}_A : \text{Set} \\
\text{List}_A = \cup \gamma_{\text{List}_A}
\]

\[
[] : \text{List}_A \\
[] = \text{intro } \langle \text{tt}, * \rangle
\]

\[
_::_ : \mathbb{N} \to \text{List}_A \to \text{List}_A \\
x :: xs = \text{intro } \langle \text{ff}, \langle x, \{?7 : 1 \to \text{List}_A\} , \{?8 : 1\} \rangle \rangle
\]
Example: the code for $\text{List}_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

$$\gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi)$$

We have

$$\gamma_{\text{List}_A} = \text{nil} +_{SP} \text{nonind}(\mathbb{N}, \lambda_. \text{ind}(1, \text{nil}))$$

with

$\text{List}_A : \text{Set}$

$\text{List}_A = U \gamma_{\text{List}_A}$

$[\cdot] : \text{List}_A$

$[\cdot] = \text{intro} \langle \text{tt}, * \rangle$

$\_ : \_ : \mathbb{N} \to \text{List}_A \to \text{List}_A$

$x :: xs = \text{intro} \langle \text{ff}, \langle x, (\lambda_. \text{xs}) \rangle, \{8 : 1\} \rangle$
Example: the code for List$_A$

We can encode two constructors into one using the dependency on non-inductive arguments:

\[ \gamma +_{SP} \psi := \text{nonind}(2, \lambda x. \text{if } x \text{ then } \gamma \text{ else } \psi) \]

We have

\[ \gamma_{\text{List}_A} = \text{nil} +_{SP} \text{nonind}(N, \lambda_. \text{ind}(1, \text{nil})) \]

with

\[ \text{List}_A : \text{Set} \]
\[ \text{List}_A = U \gamma_{\text{List}_A} \]

[ ] : List$_A$
[ ] = intro ⟨tt, *⟩

_::_: : N → List$_A$ → List$_A$

x :: xs = intro ⟨ff, ⟨x, (λ_.xs) , *⟩⟩
A low-level construction

- The universe described is very much a low-level construction.

- We do not expect the user to deal with the universe directly.

- Rather, high-level constructs (data declarations etc) can be translated to a core type theory with a universe of data types.

- Makes generic operations (decidable equality, map etc) possible.

- Route taken in Epigram 2.
  - Dagand, McBride: Elaborating Inductive Definitions (2012)
The unstoppable march of progress

- So far, we have described “simple” inductive types.

- When programming or proving with dependent types, one quickly feels the need for more advanced data structures.
  
  - Inductive families $U : I \rightarrow \text{Set}$
  
  - Induction-recursion $U : \text{Set}, \; T : U \rightarrow \text{Set}$

  - Inductive-inductive definitions $A : \text{Set}, \; B : A \rightarrow \text{Set}$

- Can we scale the universe just described to handle these data types as well?
The unstoppable march of progress

- So far, we have described “simple” inductive types.

- When programming or proving with dependent types, one quickly feels the need for more advanced data structures.
  
  - Inductive families $U : I \rightarrow \text{Set}$
  
  - Induction-recursion $U : \text{Set}, T : U \rightarrow \text{Set}$
  
  - Inductive-inductive definitions $A : \text{Set}, B : A \rightarrow \text{Set}$

- Can we scale the universe just described to handle these data types as well?

- Anticipated answer: yes! This talk: inductive-inductive definitions.
Inductive-inductive definitions
What is an inductive-inductive definition?

- Induction-induction is a principle for defining data types $A : \text{Set}$, $B : A \rightarrow \text{Set}$.

- Both $A$ and $B$ are defined inductively, “given by constructors”.
What is an inductive-inductive definition?

- Induction-induction is a principle for defining data types $A : \text{Set}$, $B : A \rightarrow \text{Set}$.

- Both $A$ and $B$ are defined inductively, “given by constructors”.

- $A$ and $B$ are defined simultaneously, so the constructors for $A$ can refer to $B$ and vice versa.

- In addition, the constructors for $B$ can even refer to the constructors for $A$. 
Induction versus recursion

- I mean induction as a definitional principle.

- “All natural numbers are generated from zero and successor.”

- By recursion, I mean a structured way to take apart something which is defined by induction.

- “Plus is defined by recursion on its first argument.”

- Amounts to the difference between induction-recursion and induction-induction.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: \(\mathbb{N}\))
   - Because we define \(A : \text{Set}\) and \(B : A \rightarrow \text{Set}\) simultaneously.

However, a weak version of I-I can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because \(B : A \rightarrow \text{Set}\) is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However 4 is not.
But isn’t that...?

An inductive-inductive definition is in general not:

1. **An ordinary inductive definition (example: \( \mathbb{N} \))**
   - Because we define \( A : \text{Set} \) and \( B : A \to \text{Set} \) simultaneously.

2. **An ordinary mutual inductive definition (example: even and odd numbers)**
   - Because \( B : A \to \text{Set} \) is indexed by \( A \).

However, a weak version of I-I can be reduced to IID.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: $\mathbb{N}$)
   - Because we define $A : \text{Set}$ and $B : A \rightarrow \text{Set}$ simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \rightarrow \text{Set}$ is indexed by $A$.

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set $A : \text{Set}$ is defined along with $B : A \rightarrow \text{Set}$, and not fixed beforehand.
   - However, a weak version of I-I can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because $B : A \rightarrow \text{Set}$ is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However 4 is not.
An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: $\mathbb{N}$)
   - Because we define $A : \text{Set}$ and $B : A \rightarrow \text{Set}$ simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \rightarrow \text{Set}$ is indexed by $A$.

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set $A : \text{Set}$ is defined along with $B : A \rightarrow \text{Set}$, and not fixed beforehand.
   - However, a weak version of I-I can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because $B : A \rightarrow \text{Set}$ is defined inductively, not recursively.
But isn’t that...?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: \( \mathbb{N} \))
   - Because we define \( A : \text{Set} \) and \( B : A \to \text{Set} \) simultaneously.

2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because \( B : A \to \text{Set} \) is indexed by \( A \).

3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set \( A : \text{Set} \) is defined along with \( B : A \to \text{Set} \), and not fixed beforehand.
   - However, a weak version of I-I can be reduced to IID.

4. An inductive-recursive definition (example: universes in type theory)
   - Because \( B : A \to \text{Set} \) is defined inductively, not recursively.

1 is a special case of 2, which is a special case of 3, which is a special case of induction-induction. However 4 is not.
Examples of inductive-inductive definitions
Examples of examples

- **Foundations**: Constructive model theory; internal Type Theory.
- **Mathematics**: the Surreal numbers.
- **Computer science**: Sorted lists.
Instances of induction-induction have been used implicitly by

- Dybjer (Internal type theory, 1996),
- Danielsson (A formalisation of a dependently typed language as an inductive-recursive family, 2007), and
- Chapman (Type theory should eat itself, 2009)

to model dependent type theory inside itself.
Type theory inside type theory

- Ctxt : Set
- Ty : Ctxt → Set
- Term : (Γ : Ctxt) → Ty(Γ) → Set
  - defined inductively
- Substitutions, …
- …
The crucial point

- The empty context $\varepsilon$ is a well-formed context.

\[ \varepsilon : \text{Ctxt} \]
The crucial point

- The empty context $\varepsilon$ is a well-formed context.
- If $\tau$ is a well-formed type in context $\Gamma$, then $\Gamma, x : \tau$ is a well-formed context.

\[
\begin{align*}
\varepsilon & : \text{Ctxt} \\
\Gamma : \text{Ctxt} & \quad \tau : \text{Ty}(\Gamma) \\
\implies & \quad \Gamma \triangleright \tau : \text{Ctxt}
\end{align*}
\]
Constructor for Ty referring to constructor for Ctxt

\[ \Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \]

\[ \Gamma \vdash \prod x : \sigma . \tau(x) \text{ type} \]
Constructor for Ty referring to constructor for Ctxt

\[ \Gamma \ 	ext{context} \quad \Gamma \vdash \sigma \ 	ext{type} \quad \Gamma, x : \sigma \vdash \tau(x) \ 	ext{type} \quad \Gamma \vdash \Pi x : \sigma . \tau(x) \ 	ext{type} \]

\[ \Gamma : \text{Ctxt} \]
Constructor for $\text{Ty}$ referring to constructor for $\text{Ctxt}$

\[
\begin{align*}
\Gamma & \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
\Gamma & \vdash \prod x : \sigma . \tau(x) \text{ type}
\end{align*}
\]

\[
\begin{align*}
\Gamma : \text{Ctxt} & \quad \sigma : \text{Ty}(\Gamma)
\end{align*}
\]
Constructor for Ty referring to constructor for Ctxt

\[
\frac{
\text{Γ context} \quad \frac{
\text{Γ \vdash σ type} \quad \frac{\text{Γ, x : σ \vdash τ(x) type}}{
\text{Γ \vdash Π x : σ. τ(x) type}}
}{
\text{Γ \vdash Π x : σ. τ(x) type}}
}{\text{Γ : Ctxt} \quad \frac{\text{σ : Ty(Γ)} \quad \text{τ : Ty(Γ ⊢ σ)}}{
}\text{Γ : Ctxt}}
\]
Constructor for Ty referring to constructor for Ctxt

\[
\begin{align*}
\Gamma \text{ context} & \quad \Gamma \vdash \sigma \text{ type} & \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
\hline
\Gamma \vdash \Pi x : \sigma. \tau(x) \text{ type} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma : \text{Ctxt} & \quad \sigma : \text{Ty}(\Gamma) & \quad \tau : \text{Ty}(\Gamma \triangleright \sigma) \\
\hline
\end{align*}
\]
Constructor for Ty referring to constructor for Ctxt

\[
\begin{align*}
\Gamma \text{ context} & \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
\Gamma & \vdash \Pi x : \sigma. \tau(x) \text{ type}
\end{align*}
\]

\[
\begin{align*}
\Gamma : \text{Ctxt} & \quad \sigma : \text{Ty}(\Gamma) \quad \tau : \text{Ty}(\Gamma \triangleright \sigma) \\
\Pi(\sigma, \tau) & : \text{Ty}(\Gamma)
\end{align*}
\]
Constructor for Ty referring to constructor for Ctxt

\[
\Gamma \text{ context} \quad \Gamma \vdash \sigma \text{ type} \quad \Gamma, x : \sigma \vdash \tau(x) \text{ type} \\
\Gamma \vdash \Pi x : \sigma . \tau(x) \text{ type}
\]

\[
\Gamma : \text{Ctxt} \quad \sigma : \text{Ty}(\Gamma) \quad \tau : \text{Ty}(\Gamma \triangleright \sigma) \\
\Pi(\sigma, \tau) : \text{Ty}(\Gamma)
\]

(Also have base type \( \iota \) in any context:

\[
\Gamma : \text{Ctxt} \\
\iota_\Gamma : \text{Ty}(\Gamma)
\] )
Conway’s surreal numbers

- Totally ordered Field containing the reals and the ordinals (at least classically).

- “Fills the holes” between them as well (think infinitesimals).

- Constructed in one step, instead of $\mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q} \rightarrow \mathbb{R}$.

- John Conway: *On Numbers and Games*.

- Donald Knuth: *Surreal Numbers*. 
From Dedekind cuts to surreal numbers

**Definition (Dedekind cut)**

A Dedekind cut \((L, R)\) consists of two non-empty sets of rational numbers \(L, R \subseteq \mathbb{Q}\) such that

- \(L \cup R = \mathbb{Q}\),
- All elements of \(L\) are less than all elements of \(R\),
- \(L\) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \((L, R)\) consists of two non-empty sets of rational numbers \(L, R \subseteq \mathbb{Q}\) such that

- \(L \cup R = \mathbb{Q}\),
- All elements of \(L\) are less than all elements of \(R\),
- \(L\) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of rational numbers \( L, R \subseteq \mathbb{Q} \) such that

- \( L \cup R = \mathbb{Q} \),
- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

**Definition (Surreal number)**

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \( L \cup R = \mathbb{Q} \),
- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \( L \cup R = \mathbb{Q} \),

- All elements of \( L \) are less than all elements of \( R \),

- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- All elements of \( L \) are less than all elements of \( R \),
- \( L \) contains no greatest element.

All surreal numbers are constructed this way.
Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \((\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R)\),

- \( L \) contains no greatest element.

All surreal numbers are constructed this way.
From Dedekind cuts to surreal numbers

**Definition (Surreal number)**

A surreal number \( \{L|R\} \) consists of two non-empty sets of surreal numbers \( L, R \) such that

- \((\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R),\)
- \(L\) contains no greatest element.
From Dedekind cuts to surreal numbers

**Definition (Surreal number)**

A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

- \((\forall x^L \in L)(\forall x^R \in R)(\neg(x^L \geq x^R))\),
- \( L \) contains no greatest element.
From Dedekind cuts to surreal numbers

Definition (Surreal number)

A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg(x^L \geq x^R).
\]

All surreal numbers are constructed this way.
Definition

A surreal number \( \{ L \mid R \} \) consists of two sets of surreal numbers \( L, R \) such that

\[
(\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R).
\]

All surreal numbers are constructed this way.
From Dedekind cuts to surreal numbers

Definition
A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

- \((\forall x^L \in L)(\forall x^R \in R) \neg(x^L \geq x^R)\).

All surreal numbers are constructed this way.

Definition
Let \( x = \{X_L|X_R\}, y = \{Y_L|Y_R\} \). We say \( x \geq y \) iff

\[
(\forall x^R \in X_R) \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L) \neg(y^L \geq x)
\]
From Dedekind cuts to surreal numbers

Definition
A surreal number \( \{L|R\} \) consists of two sets of surreal numbers \( L, R \) such that

\[(\forall x^L \in L)(\forall x^R \in R) \neg (x^L \geq x^R).\]

All surreal numbers are constructed this way.

Definition
Let \( x = \{X_L|X_R\}, y = \{Y_L|Y_R\} \). We say \( x \geq y \) iff

\[(\forall x^R \in X_R) \neg (y \geq x^R) \) and \( (\forall y^L \in Y_L) \neg (y^L \geq x)\)

An inductive-inductive definition!

  - Encoding of the inductive-inductive definition, since Coq does not support them.
A finite axiomatisation
An axiomatisation

- High-level idea: Add a universe (family) $SP = (SP^0_A, SP^0_B)$ of codes representing the inductive-inductively defined sets.
An axiomatisation

- High-level idea: Add a universe (family) $\text{SP} = (\text{SP}_A^0, \text{SP}_B^0)$ of codes representing the inductive-inductively defined sets.

- Stipulate that for each code $\gamma = (\gamma_A, \gamma_B)$, there are
  
  $$A_\gamma : \text{Set}$$
  
  $$B_\gamma : A_\gamma \to \text{Set}$$

  and constructors

  $$\text{intro}_A : \text{Arg}_A^0(\gamma_A, A_\gamma, B_\gamma) \to A_\gamma$$
  
  $$\text{intro}_B : (x : \text{Arg}_B^0(\gamma_B, A_\gamma, B_\gamma, \text{intro}_A)) \to B_\gamma(i_\gamma(x))$$
An axiomatisation

- High-level idea: Add a universe (family) $SP = (SP^0_A, SP^0_B)$ of codes representing the inductive-inductively defined sets.

- Stipulate that for each code $\gamma = (\gamma_A, \gamma_B)$, there are

  $A_\gamma : \text{Set}$
  $B_\gamma : A_\gamma \to \text{Set}$

  and constructors

  $\text{intro}_A : \text{Arg}_A^0(\gamma_A, A_\gamma, B_\gamma) \to A_\gamma$
  $\text{intro}_B : (x : \text{Arg}_B^0(\gamma_B, A_\gamma, B_\gamma, \text{intro}_A)) \to B_\gamma(i_\gamma(x))$

- The codes describe the “pattern functors” $\text{Arg}_A^0, \text{Arg}_B^0$. 
Main idea

- We define
  - a set
    \[ SP^0_A : \text{Set} \]
    of codes for inductive definitions for \( A \),
  - a set
    \[ SP^0_B : SP^0_A \to \text{Set} \]
    of codes for inductive definitions for \( B \).
  - the set of arguments for the constructor of \( A \):
    \[ \text{Arg}^0_A : SP^0_A \to (X : \text{Set}) \to (Y : X \to \text{Set}) \to \text{Set} \]
Main idea (cont.)

- the set of arguments and indices for the constructor of $B$:

$$\text{Arg}_B^0 : (\gamma_A : \text{SP}_A^0) \rightarrow \text{Set}$$

$$\quad (\gamma_B : \text{SP}_B^0(\gamma_A)) \rightarrow \text{Set}$$

$$\quad (X : \text{Set}) \rightarrow \text{Set}$$

$$\quad (Y : X \rightarrow \text{Set}) \rightarrow \text{Set}$$

$$\quad (\text{intro}_A : \text{Arg}_A^0(\gamma_A, X, Y) \rightarrow X) \rightarrow \text{Set}$$

$$\text{Index}_B^0 : \cdots \text{same arguments as } \text{Arg}_B^0 \cdots$$

$$\quad \text{Arg}_B^0(\gamma_A, \gamma_B, X, Y, \text{intro}_A) \rightarrow X$$
Formation and introduction rules

Formation rules:

\[ A_{\gamma_A, \gamma_B} : \text{Set} \]
\[ B_{\gamma_A, \gamma_B} : A_{\gamma_A, \gamma_B} \rightarrow \text{Set} \]

Introduction rule for \( A_{\gamma_A, \gamma_B} \):

\[ a : \text{Arg}^0_A(\gamma_A, A_{\gamma_A, \gamma_B}, B_{\gamma_A, \gamma_B}) \]
\[ \text{intro}_{A_{\gamma_A, \gamma_B}}(a) : A_{\gamma_A, \gamma_B} \]

Introduction rule for \( B_{\gamma_A, \gamma_B} \):

\[ a : \text{Arg}^0_B(\gamma_A, \gamma_B, A_{\gamma_A, \gamma_B}, B_{\gamma_A, \gamma_B}, \text{intro}_{A_{\gamma_A, \gamma_B}}) \]
\[ \text{intro}_{B_{\gamma_A, \gamma_B}}(a) : B_{\gamma_A, \gamma_B}(\text{Index}^0_B(\gamma_A, \gamma_B, A_{\gamma_A, \gamma_B}, B_{\gamma_A, \gamma_B}, \text{intro}_{A_{\gamma_A, \gamma_B}}, a)) \]

Elimination rules by inversion of introduction rules
Definition of $\text{SP}_A$

- Instead of defining $\text{SP}_A^0$ we define a more general set:

$$\text{SP}_A : (X_{\text{ref}} : \text{Set}) \rightarrow \text{Set}$$

with a set $X_{\text{ref}}$ of elements of the set to be defined which we can refer to.

- In definition of $\text{Arg}_A$, also require function

$$\text{rep}_X : X_{\text{ref}} \rightarrow X$$

mapping elements in $X_{\text{ref}}$ to the element in $X$ they represent.

- Then

$$\text{SP}_A^0 := \text{SP}_A(0)$$

$$\text{rep}_X = !_X : 0 \rightarrow X$$
The codes in $SP_A$

Base case; $\text{intro}_A : 1 \rightarrow A$.

\[
\begin{align*}
\text{nil} : SP_A(X_{\text{ref}}) \\
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1
\end{align*}
\]
The codes in $\text{SP}_A$

Non-inductive argument; $\text{intro}_A : ((x : K) \times \ldots) \rightarrow A.$

\[
\begin{align*}
K : \text{Set} & \quad \gamma : K \rightarrow \text{SP}_A(X_{\text{ref}}) \\
\text{non-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}})
\end{align*}
\]

$\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) = 1$
Noninductive argument; intro$_A : ((x : K) \times \ldots) \rightarrow A.$

\[\begin{align*}
K : \text{Set} & \quad \gamma : K \rightarrow \text{SP}_A(X_{\text{ref}}) \\
\text{non-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}})
\end{align*}\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) &= (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X)
\end{align*}\]
The codes in $\text{SP}_A$

**A-ind**

Inductive argument in $A$; $\text{intro}_A : ((g : K \to A) \times \ldots) \to A$.

$$
\begin{array}{c}
K : \text{Set} \\
\gamma : \text{SP}_A(X_{\text{ref}} + K)
\end{array}
\quad
\begin{array}{c}
\Rightarrow
\end{array}
\quad
\begin{array}{c}
\text{A-ind}(K, \gamma) : \text{SP}_A(X_{\text{ref}})
\end{array}
$$

$$
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) &= \\
&= (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X)
\end{align*}
$$
The codes in $\text{SP}_A$

**A-ind**

Inductive argument in $A$; $\text{intro}_A : \left( (g : K \to A) \times \ldots \right) \to A$.

$$
\begin{align*}
K & : \text{Set} \\
\gamma & : \text{SP}_A(X_{\text{ref}} + K) \\
\text{A-ind}(K, \gamma) & : \text{SP}_A(X_{\text{ref}})
\end{align*}
$$

$$
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) & = 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) & = (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) & = \\
& (g : K \to X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g])
\end{align*}
$$
The codes in $SP_A$

A-ind

Inductive argument in $A$; $\text{intro}_A : ((g : K \to A) \times \ldots) \to A$.

\[
\begin{align*}
K : & \text{Set} \\
\gamma : & SP_A(X_{\text{ref}} + K) \\
\hline
A\text{-ind}(K, \gamma) : & SP_A(X_{\text{ref}})
\end{align*}
\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) & = 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) & = (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, A\text{-ind}(K, \gamma), X, Y, \text{rep}_X) & = (g : K \to X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g])
\end{align*}
\]

In later arguments, we can refer to

\[
X_{\text{ref}} \cup \{g(x) | x \in K\} \subseteq X,
\]

represented by $[\text{rep}_X, g] : X_{\text{ref}} + K \to X$. 
The codes in $\text{SP}_A$

**B-ind**

Inductive argument in $B$; $\text{intro}_A : ((g : (x : K) \to B(i(x))) \times \ldots) \to A.$

\[
\begin{array}{c}
K : \text{Set} \\
h_{\text{index}} : K \to \text{Xref} \\
\gamma : \text{SP}_A
\end{array}
\]

$\text{B-ind}(K, h_{\text{index}}, \gamma) : \text{SP}_A$

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) &= 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) &= (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) &= (g : K \to X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g])
\end{align*}
\]
The codes in $\text{SP}_A$

**B-ind**

Inductive argument in $B$; $\text{intro}_A : ( (g : (x : K) \to B(i(x))) \times \ldots ) \to A$.

\[
\begin{array}{c}
K : \text{Set} \\
h_{\text{index}} : K \to X_{\text{ref}} \\
\gamma : \text{SP}_A \\
\end{array}
\]

\[
\text{B-ind}(K, h_{\text{index}}, \gamma) : \text{SP}_A
\]

\[
\begin{align*}
\text{Arg}_A(X_{\text{ref}}, \text{nil}, X, Y, \text{rep}_X) & = 1 \\
\text{Arg}_A(X_{\text{ref}}, \text{non-ind}(K, \gamma), X, Y, \text{rep}_X) & = (x : K) \times \text{Arg}_A(X_{\text{ref}}, \gamma(x), X, Y, \text{rep}_X) \\
\text{Arg}_A(X_{\text{ref}}, \text{A-ind}(K, \gamma), X, Y, \text{rep}_X) & = (g : K \to X) \times \text{Arg}_A(X_{\text{ref}} + K, \gamma, X, Y, [\text{rep}_X, g]) \\
\text{Arg}_A(X_{\text{ref}}, \text{B-ind}(K, h_{\text{index}}, \gamma), X, Y, \text{rep}_X) & = (g : (x : K) \to Y((\text{rep}_X \circ h_{\text{index}})(x))) \times \text{Arg}_A(X_{\text{ref}}, \gamma, X, Y, \text{rep}_X)
\end{align*}
\]
An example

The constructor

\[ \triangleright : ((\Gamma : \text{Ctxt}) \times \text{Ty}(\Gamma)) \rightarrow \text{Ctxt} \]

is represented by the code

\[ \gamma_\triangleright = \text{A-ind}(1, \text{B-ind}(1, \lambda(\star : 1). \text{inr}(\star), \text{nil})) \]

We have

\[ \text{Arg}_A(0, \gamma_\triangleright, \text{Ctxt}, \text{Ty}, !_{\text{Ctxt}}) = (\Gamma : 1 \rightarrow \text{Ctxt}) \times (1 \rightarrow \text{Ty}(\Gamma(\star))) \times 1 \]

\[ \cong (\Gamma : \text{Ctxt}) \times \text{Ty}(\Gamma) \]
The codes in $\text{SP}_B$:

- The universe $\text{SP}_B^0 : \text{SP}_A^0 \rightarrow \text{Set}$ is similar to $\text{SP}_A^0$.

- Need argument $\text{SP}_A^0$ to know the shape of constructor for the first set, which can appear in indices.

- We omit the definition here.
Meta-theory
Soundness

**Theorem (Soundness)**

*Standard Martin-Löf Type Theory with inductive-inductive definitions is sound.*
Soundness

**Theorem (Soundness)**

*Standard Martin-Löf Type Theory with inductive-inductive definitions is sound.*

**Remark**

Can also include e.g. large elimination, function extensionality, uniqueness of identity proofs, and equality reflection.
Soundness

Theorem (Soundness)

*Standard Martin-Löf Type Theory with inductive-inductive definitions is sound.*

Remark

Can also include e.g. large elimination, function extensionality, uniqueness of identity proofs, and equality reflection.

- This is achieved by constructing a naive set-theoretical model.
- The untyped nature of set-theory is exploited to reduce inductive-inductive definitions to mutual inductive definitions.
- The model is too naive to validate e.g. univalence or parametricity.
Generic programming

- The axiomatisation is given as a universe of codes for inductive-inductive definitions.

- Many **advantages**:  
  - Finite axiomatisation.
  - Small trusted core type theory.
  - Generic programming becomes normal programming.
Concrete advantages

- Deriving e.g. functor instances.
- Proving decidable equality for finitary inductive-inductive definitions.
- Formal embedding of ordinary and indexed inductive definitions into inductive-inductive definitions.
- All available to the user of the theory, inside the theory, and extensible by the user.
Reducing a weak version to IID

- A weak version of the theory of inductive-inductive definitions can be reduced to the (extensional) theory of indexed inductive definitions.

- Implicitly used by Conway (and Mamane) for the surreal numbers (games).

- Weak since restricted elimination rules: Only motives of the form

\[
P : A \rightarrow \text{Set} \\
Q : (x : A) \rightarrow B(x) \rightarrow \text{Set}
\]

instead of the general motive

\[
P : A \rightarrow \text{Set} \\
Q : (x : A) \rightarrow B(x) \rightarrow P(x) \rightarrow \text{Set}
\]

(the “recursive-recursive” eliminator).
The high-level idea

- Define simultaneously

\[ A_{\text{pre}} : \text{Set} \quad B_{\text{pre}} : \text{Set} \]

ignoring dependencies of \( B \) on \( A \).

- Then select \( A \subseteq A_{\text{pre}}, B \subseteq B_{\text{pre}} \) that satisfy the typing by two inductively defined predicates

\[
A_{\text{good}} : A_{\text{pre}} \rightarrow \text{Set} \\
B_{\text{good}} : A_{\text{pre}} \rightarrow B_{\text{pre}} \rightarrow \text{Set}
\]

(indexed inductive definitions).

- Interpretation

\[
\lbrack A \rbrack = \sum x : A_{\text{pre}} . A_{\text{good}}(x) \\
\lbrack B \rbrack(\langle x, x_g \rangle) = \sum y : B_{\text{pre}} . B_{\text{good}}(x, y)
\]
Categorical characterisation

- Ordinary inductive types can be characterised as initial algebras.
- Indexed inductive types can be characterised as initial algebras on slice categories.
- Is there a corresponding result for inductive-inductive types?

Theorem (extensional type theory)
For every inductive-inductive definition \((A, B)\), there is a category \(E_{A, B} \hookrightarrow \text{Dialg}(F_A, B, G)\) such that the elimination rules for \((A, B)\) hold if and only if \(E_{A, B}\) has an initial object.

Main obstacle: Initiality gives non-dependent functions, elimination rules dependent.
Categorical characterisation

- Ordinary inductive types can be characterised as initial algebras.
- Indexed inductive types can be characterised as initial algebras on slice categories.
- Is there a corresponding result for inductive-inductive types?
  - Yes, but need to use dialgebras $f : F(X) \to G(X)$ and not just algebras $g : F(X) \to X$. 

Theorem (extensional type theory)

For every inductive-inductive definition $(A, B)$, there is a category $E_A, B \hookrightarrow \rightarrow \text{Dialg}(F_A, B, G)$ such that the elimination rules for $(A, B)$ hold if and only if $E_A, B$ has an initial object.

Main obstacle: Initiality gives non-dependent functions, elimination rules dependent.
Categorical characterisation

- Ordinary inductive types can be characterised as initial algebras.

- Indexed inductive types can be characterised as initial algebras on slice categories.

- Is there a corresponding result for inductive-inductive types?
  - Yes, but need to use dialgebras $f : F(X) \to G(X)$ and not just algebras $g : F(X) \to X$.

**Theorem (extensional type theory)**

For every inductive-inductive definition $(A, B)$, there is a category $E_{A,B} \hookrightarrow \text{Dialg}(F_{A,B}, G)$ such that the elimination rules for $(A, B)$ hold if and only if $E_{A,B}$ has an initial object.

Main obstacle: Initiality gives non-dependent functions, elimination rules dependent.
Status in proof assistants

- Not supported in Coq or Epigram.
- Is supported in Agda and Idris!
- Now we know it is sound as well.
  - Not obvious; e.g. both Agda and Idris accepts a universe $U$ with a code $\hat{U} : U$ for itself...
Status in proof assistants

- Not supported in Coq or Epigram.

- Is supported in Agda and Idris!

- Now we know it is sound as well.
  - Not obvious; e.g. both Agda and Idris accepts a universe $U$ with a code $\hat{U} : U$ for itself...
  - However, other parts of the system (the strict positivity check) happen to prevent inconsistency.
Status in proof assistants

- Not supported in Coq or Epigram.

- Is supported in Agda and Idris!

- Now we know it is sound as well.
  - Not obvious; e.g. both Agda and Idris accepts a universe $U$ with a code $\hat{U} : U$ for itself...
  - However, other parts of the system (the strict positivity check) happen to prevent inconsistency.
  - Nonetheless, what is the semantic justification?
Summary

- When using Type Theory, one naturally wants more advanced data structures such as inductive-inductive definitions.
- Expressivity rather than strength.
- But still has an interesting meta-theory.
Summary

- When using Type Theory, one naturally wants more advanced data structures.
- Expressivity rather than strength.
- But still has an interesting meta-theory.