Inductive-inductive definitions in Intuitionistic Type Theory

Fredrik Nordvall Forsberg

MSP group, Strathclyde, Glasgow
fredrik.nordvall-forsberg@strath.ac.uk

Stockholm, 11 June 2014

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 $(A, <)$.

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

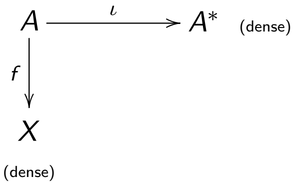
A toy example

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 $(A, <)$.



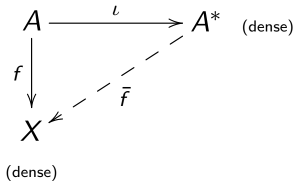
A toy example

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 $(A, <)$.



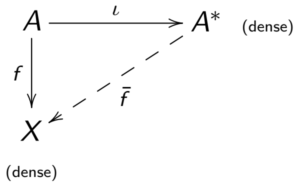
A toy example

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 $(A, <)$.



- How can we construct this?

Constructing the dense completion

- Intuitively:
  1. start with A
  2. for each pair $x < y$, add a midpoint $x <^* mid(x, y) <^* y$
  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 : Type, < : A -> A -> Type.
```

```
Inductive A* : Type :=
```

The dense completion in Fantasy Coq

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

Inductive A* : Type :=
 | ι : A -> 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*
```

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 : 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
```

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
 | midʳ : forall x y : A*, forall p : x <* y, x <* mid x y p
```

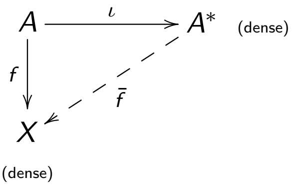
The dense completion in Fantasy Coq

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

Inductive A* : Type :=
 | 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
 | midʳ : forall x y : A*, forall p : x <* y, x <* mid x y p
 | midˡ : forall x y : A*, forall p : x <* y, mid x y p <* y.
```

The dense completion in Fantasy Coq

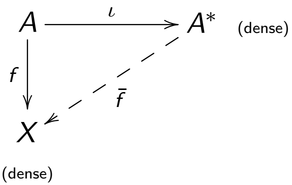
```
Parameters A : Type, < : A -> A -> Type.

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^r : forall x y : A*, forall p : x <* y, x <* mid x y p
 | mid^l : forall x y : A*, forall p : x <* y, mid x y p <* y.

Definition dense_{A*} (x y : A*)(p : x <* y)
                             : { z : A* & x <* z & z <* y }
  := existT2 (mid x y p) (mid^r x y p) (mid^l x y p).
```

Defining $\bar{f}$



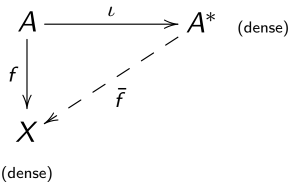

```
Parameters
  (X : Type)(<_X : X -> X -> Type)
  (dense_X : forall x y : X, { z : X & x <_X z & z <_X y})
  (f : A -> X)
  (f^< : forall x y : A, x < y -> (f x) <_X (f y)).
```

Defining $\bar{f}$



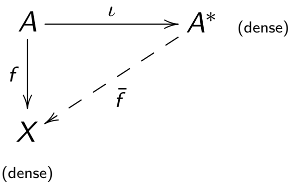

```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => {?₀ : X}
  | mid x y p => {?₁ : X}
 end
```

Defining $\bar{f}$



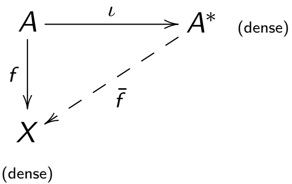

```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => f a
  | mid x y p => {?₁ : X}
 end
```

Defining $\bar{f}$



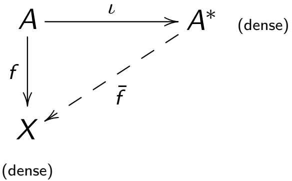

```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => f a
  | mid x y p => proj1 (dense_X {?2 : X} {?3 : X} {?4 :?2 <_X ?3})
 end
```

Defining $\bar{f}$



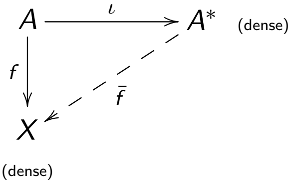

```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => f a
  | mid x y p => proj1 (dense_X (f̄ x) {?3 : X} {?4 : f̄ x <_X ?3})
 end
```

Defining $\bar{f}$



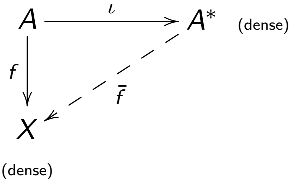

```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => f a
  | mid x y p => proj1 (dense_X (f̄ x) (f̄ y) {?4 : f̄ x <_X f̄ y})
 end
```

Defining $\bar{f}$




```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => f a
  | mid x y p => proj1 (dense_X (f̄ x) (f̄ y) {?4 : f̄ x <_X f̄ y})
 end

with f̄^< (x y : A*)(p : x <* y) : f̄ x <_X f̄ y := ...
```

Defining $\bar{f}$




```
Fixpoint f̄ (z : A*) : X :=
 match z with
  | ι a => f a
  | mid x y p => proj1 (dense_X (f̄ x) (f̄ y) (f̄^< x y p))
 end
```

```
with f̄^< (x y : A*)(p : x <* y) : f̄ x <_X f̄ y := ...
```

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

$$\frac{}{[] : \mathsf{List}_A} \qquad \frac{x : A \qquad xs : \mathsf{List}_A}{(x :: xs) : \mathsf{List}_A}$$

```
data List_A : Set where
    []  : List_A
    _::_ : A → List_A → List_A
```

$$\frac{}{0 : \mathcal{O}} \qquad \frac{n : \mathcal{O}}{\mathsf{suc}(n) : \mathcal{O}}$$

$$\frac{f : \mathbb{N} \to \mathcal{O}}{\mathsf{sup}(f) : \mathcal{O}}$$

```
data 𝒪 : Set where
    0 : 𝒪
    S : 𝒪 → 𝒪
    sup : (ℕ → 𝒪) → 𝒪
```

$$\frac{a : A \qquad f : B(a) \to W(A, B)}{\mathsf{sup}(a, f) : W(A, B)}$$

```
data W (A : Set)(B : A → Set) : Set
  sup : (a : A) →
        (f : B a → W A B) → W A B
```

Induction principles/elimination rules

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

$$
\begin{aligned}
\mathrm{elim}_{\mathrm{List}_A} : {} & (P : \mathrm{List}_A \to \mathrm{Set}) \to \\
& (\mathrm{step}_{[]} : P([])) \to \\
& (\mathrm{step}_{::} : (x : \mathbb{N}) \to (xs : \mathrm{List}_A) \to P(xs) \to P(x :: xs)) \to \\
& (y : \mathrm{List}_A) \to P(y)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{elim}_{\mathrm{List}_A}(P, \mathrm{step}_{[]}, \mathrm{step}_{::}, []) &= \mathrm{step}_{[]} \\
\mathrm{elim}_{\mathrm{List}_A}(P, \mathrm{step}_{[]}, \mathrm{step}_{::}, x :: xs) &= \mathrm{step}_{::}(x, xs, \mathrm{elim}_{\mathrm{List}_A}(\ldots, xs))
\end{aligned}
$$

- 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.

$$\mathbb{N} = (X : \mathsf{Set}) \to X \to (X \to X) \to X$$

$$\mathsf{Id}_A(a, b) = (X : A \to \mathsf{Set}) \to X(a) \to X(b)$$

Church encodings?

Pfenning and Paulin-Mohring (1989)

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

$$\mathbb{N} = (X : \mathsf{Set}) \to X \to (X \to X) \to X$$

$$\mathsf{Id}_A(a, b) = (X : A \to \mathsf{Set}) \to X(a) \to X(b)$$

- 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.

$$\mathbb{N} = (X : \mathsf{Set}) \to X \to (X \to X) \to X : \mathsf{Set}$$

$$\mathsf{Id}_A(a, b) = (X : A \to \mathsf{Set}) \to X(a) \to X(b) : \mathsf{Set}$$

- 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.

$$\mathbb{N} = (X : \mathsf{Set}) \to X \to (X \to X) \to X : \mathsf{Set}$$

$$\mathsf{Id}_A(a, b) = (X : A \to \mathsf{Set}) \to X(a) \to X(b) : \mathsf{Set}$$

- 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.

$$\mathbb{N} = (X : \mathsf{Set}) \to X \to (X \to X) \to X : \mathsf{Set}$$

$$\mathsf{Id}_A(a, b) = (X : A \to \mathsf{Set}) \to X(a) \to X(b) : \mathsf{Set}$$

- 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).

Syntactic schemata

Backhouse (1987), Coquand and Paulin-Mohring (1990), Dybjer (1994), ...

Dybjer (1994) considers constructors of the form

$$\begin{aligned} \mathsf{intro}_U : {} & (A :: \sigma) \\ & (b :: \beta[A]) \to \\ & (u :: \gamma[A, b]) \to \\ & U \end{aligned}$$

where

- σ 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 $\xi_i[A, b] \to U$ (strict positivity).

Syntactic schemata (cont.)

- 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.).

Syntax internalised

Dybjer and Setzer (1999, 2003, 2006) [for IR]

- 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

$$\mathsf{intro}_{U_\gamma} : \mathsf{Arg}(\gamma, U_\gamma) \to U_\gamma$$

[γ is "code" that contains the necessary information to describe U_γ.]

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 $\mathrm{Arg} : \mathrm{SP} \to \mathrm{Set} \to \mathrm{Set}$. [$\mathrm{Arg}(\gamma, X)$ is the domain where X is used for the inductive arguments.]
- For every $\gamma : \mathrm{SP}$, stipulate that there is a set U_γ and a constructor $\mathrm{intro}_\gamma : \mathrm{Arg}(\gamma, U_\gamma) \to U_\gamma$.
- Inversion principle for elimination and computation rules.

SP, Arg and U_γ

```
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 γ) X = (y : A) × (Arg (γ y) X)
Arg (ind A γ) X = (A → X) × (Arg γ X)

data U (γ : SP) : Set where
   intro : Arg γ (U γ) → U γ
```

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = U γ_{List_A}

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

_::_ : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
x :: xs = $\{?_1 : \mathsf{List}_A\}$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = U γ_{List_A}

[] : List_A
[] = intro $\{?_2 : \mathtt{Arg}(\gamma_{\mathsf{List}_A}, \mathsf{List}_A)\}$

_::_ : $\mathbb{N} \rightarrow \mathsf{List}_A \rightarrow \mathsf{List}_A$
x :: xs = $\{?_1 : \mathsf{List}_A\}$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = `U` γ_{List_A}

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

`_::_` : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
`x :: xs` = $\{?_1 : \mathsf{List}_A\}$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = `U` γ_{List_A}

`[]` : List_A
`[]` = `intro` $\langle \{?_3 : \mathbf{2}\}, \{?_4 : \mathsf{if}\ ?_3\ \mathsf{then}\ \mathbf{1}\ \mathsf{else}\ \mathbb{N} \times \ldots\} \rangle$

`_::_` : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
`x :: xs` = $\{?_1 : \mathsf{List}_A\}$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = `U` γ_{List_A}

`[]` : List_A
`[]` = `intro` $\langle$`tt`, $\{?_4 : \mathbf{1}\}$ $\rangle$

`_::_` : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
`x :: xs` = $\{?_1 : \mathsf{List}_A\}$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = U γ_{List_A}

[] : List_A
[] = intro $\langle$tt, $\star\rangle$

_::_ : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
x :: xs = $\{?_1 : \mathsf{List}_A\}$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = `U` γ_{List_A}

`[]`: List_A
`[]= intro` $\langle$`tt,` $\star\rangle$

`_::_` : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
`x :: xs = intro` $\langle$`ff,` $\{?_5 : \mathbb{N} \times (\mathbf{1} \to \mathsf{List}_A) \times \mathbf{1}\}\ \rangle$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = U γ_{List_A}

[] : List_A
[] = intro $\langle$tt, $\star\rangle$

_::_ : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
x :: xs = intro $\langle$ff, $\langle \{?_6 : \mathbb{N}\}$, $\{?_7 : \mathbf{1} \to \mathsf{List}_A\}$, $\{?_8 : \mathbf{1}\}\rangle\rangle$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = U γ_{List_A}

[]: List_A
[]= intro $\langle$tt, $\star\rangle$

_::_ : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
x :: xs = intro $\langle$ff, $\langle$x, $\{?_7 : \mathbf{1} \to \mathsf{List}_A\}$, $\{?_8 : \mathbf{1}\}\rangle\rangle$

Example: the code for List_A

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

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

We have

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

with

List_A : Set
List_A = U γ_{List_A}

[] : List_A
[] = intro $\langle$tt, $\star\rangle$

_::_ : $\mathbb{N} \to \mathsf{List}_A \to \mathsf{List}_A$
x :: xs = intro $\langle$ff, $\langle$x, (λ_. xs) , $\{?_8 : \mathbf{1}\}\rangle\rangle$

Example: the code for List_A

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

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

We have

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

with

```
List_A : Set
List_A = U γ_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.
  - Chapman, Dagand, McBride and Morris: The Gentle Art of Levitation (2010)
  - 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 $U : I \to \mathsf{Set}$
  - Induction-recursion $U : \mathsf{Set},\ T : U \to \mathsf{Set}$
  - Inductive-inductive definitions $A : \mathsf{Set},\ B : A \to \mathsf{Set}$
- 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 $U : I \to \mathsf{Set}$
  - Induction-recursion $U : \mathsf{Set}$, $T : U \to \mathsf{Set}$
  - Inductive-inductive definitions $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$
- 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 $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$.
- 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 $A : \mathsf{Set}$, $B : A \to \mathsf{Set}$.
- 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 $A : \mathrm{Set}$ and $B : A \to \mathrm{Set}$ simultaneously.

But isn't that. . . ?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: $\mathbb{N}$)
   - Because we define A : Set and $B : A \to$ Set simultaneously.
2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \to$ Set is indexed by A.

But isn't that. . . ?

An inductive-inductive definition is in general not:

1. An ordinary inductive definition (example: $\mathbb{N}$)
    - Because we define A : Set and $B : A \to$ Set simultaneously.
2. An ordinary mutual inductive definition (example: even and odd numbers)
    - Because $B : A \to$ Set is indexed by A.
3. An indexed inductive definition (example: lists of a certain length)
    - Because the index set A : Set is defined along with $B : A \to$ Set, and not fixed beforehand.
    - 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 : Set and $B : A \to$ Set simultaneously.
2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \to$ Set is indexed by A.
3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set A : Set is defined along with $B : A \to$ 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 $B : A \to$ 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 : Set and $B : A \to$ Set simultaneously.
2. An ordinary mutual inductive definition (example: even and odd numbers)
   - Because $B : A \to$ Set is indexed by A.
3. An indexed inductive definition (example: lists of a certain length)
   - Because the index set A : Set is defined along with $B : A \to$ 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 $B : A \to$ 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.

Modelling dependent type theory

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

- $\mathsf{Ctxt} : \mathsf{Set}$
- $\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set}$
- $\mathsf{Term} : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Set}$
- ...
- Substitutions, ...
- ...

defined inductively

The crucial point

- The empty context ε is a well-formed context.

$$\frac{}{\varepsilon : \mathsf{Ctxt}}$$

The crucial point

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

$$\frac{}{\varepsilon : \mathsf{Ctxt}}$$

$$\frac{\Gamma : \mathsf{Ctxt} \qquad \tau : \mathsf{Ty}(\Gamma)}{\Gamma \triangleright \tau : \mathsf{Ctxt}}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x{:}\sigma \,.\, \tau(x) \text{ type}}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x{:}\sigma \,.\, \tau(x) \text{ type}}$$

$$\frac{\Gamma : \mathsf{Ctxt}}{}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x{:}\sigma \,.\, \tau(x) \text{ type}}$$

$$\frac{\Gamma : \mathsf{Ctxt} \qquad \sigma : \mathsf{Ty}(\Gamma) \qquad\qquad\qquad}{}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x{:}\sigma\,.\,\tau(x) \text{ type}}$$

$$\frac{\Gamma : \mathsf{Ctxt} \qquad \sigma : \mathsf{Ty}(\Gamma) \qquad \tau : \mathsf{Ty}(\Gamma \triangleright \sigma)}{}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \color{red}{\Gamma, x : \sigma} \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x{:}\sigma \,.\, \tau(x) \text{ type}}$$

$$\frac{\Gamma : \mathsf{Ctxt} \qquad \sigma : \mathsf{Ty}(\Gamma) \qquad \tau : \mathsf{Ty}(\color{red}{\Gamma \triangleright \sigma})}{}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x\!:\!\sigma\, .\, \tau(x) \text{ type}}$$

$$\frac{\Gamma : \mathsf{Ctxt} \qquad \sigma : \mathsf{Ty}(\Gamma) \qquad \tau : \mathsf{Ty}(\Gamma \triangleright \sigma)}{\Pi(\sigma, \tau) : \mathsf{Ty}(\Gamma)}$$

Constructor for Ty referring to constructor for Ctxt

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash \sigma \text{ type} \qquad \Gamma, x : \sigma \vdash \tau(x) \text{ type}}{\Gamma \vdash \Pi\, x{:}\sigma\,.\,\tau(x) \text{ type}}$$

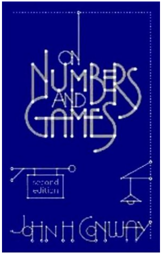
$$\frac{\Gamma : \mathsf{Ctxt} \qquad \sigma : \mathsf{Ty}(\Gamma) \qquad \tau : \mathsf{Ty}(\Gamma \triangleright \sigma)}{\Pi(\sigma, \tau) : \mathsf{Ty}(\Gamma)}$$

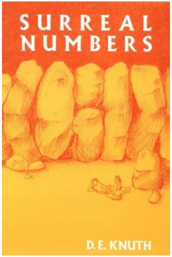
(Also have base type ι in any context:

$$\frac{\Gamma : \mathsf{Ctxt}}{\iota_\Gamma : \mathsf{Ty}(\Gamma)}\ )$$

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 $\mathbb{N} \rightsquigarrow \mathbb{Z} \rightsquigarrow \mathbb{Q} \rightsquigarrow \mathbb{R}$.
- 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.

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 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.

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.

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

$$(\forall x^R \in X_R)\, \neg(y \geq x^R) \text{ and } (\forall y^L \in Y_L)\, \neg(y^L \geq x)$$

An inductive-inductive definition!

- Mamane: Surreal Numbers in Coq (2006)
  - Encoding of the inductive-inductive definition, since Coq does not support them.

A finite axiomatisation

An axiomatisation

- High-level idea: Add a universe (family) $\mathsf{SP} = (\mathsf{SP}^0_{\mathrm{A}}, \mathsf{SP}^0_{\mathrm{B}})$ of codes representing the inductive-inductively defined sets.

An axiomatisation

- High-level idea: Add a universe (family) $\mathsf{SP} = (\mathsf{SP}^0_{\mathrm{A}}, \mathsf{SP}^0_{\mathrm{B}})$ of codes representing the inductive-inductively defined sets.

- Stipulate that for each code $\gamma = (\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}})$, there are

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

and constructors

$$\mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, A_\gamma, B_\gamma) \to A_\gamma$$
$$\mathsf{intro}_{\mathrm{B}} : (x : \mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{B}}, A_\gamma, B_\gamma, \mathsf{intro}_{\mathrm{A}})) \to B_\gamma(i_\gamma(x))$$

An axiomatisation

- High-level idea: Add a universe (family) $\mathsf{SP} = (\mathsf{SP}^0_{\mathrm{A}}, \mathsf{SP}^0_{\mathrm{B}})$ of codes representing the inductive-inductively defined sets.

- Stipulate that for each code $\gamma = (\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}})$, there are

$$\begin{aligned} A_\gamma &: \mathsf{Set} \\ B_\gamma &: A_\gamma \to \mathsf{Set} \end{aligned}$$

  and constructors

$$\begin{aligned} \mathsf{intro}_{\mathrm{A}} &: \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, A_\gamma, B_\gamma) \to A_\gamma \\ \mathsf{intro}_{\mathrm{B}} &: (x : \mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{B}}, A_\gamma, B_\gamma, \mathsf{intro}_{\mathrm{A}})) \to B_\gamma(i_\gamma(x)) \end{aligned}$$

- The codes describe the "pattern functors" $\mathsf{Arg}^0_{\mathrm{A}}$, $\mathsf{Arg}^0_{\mathrm{B}}$.

Main idea

- We define
  - a set
    $$\mathsf{SP}^0_{\mathrm{A}} : \mathsf{Set}$$
    of codes for inductive definitions for A,
  - a set
    $$\mathsf{SP}^0_{\mathrm{B}} : \mathsf{SP}^0_{\mathrm{A}} \to \mathsf{Set}$$
    of codes for inductive definitions for B.
  - the set of arguments for the constructor of A:
    $$\mathsf{Arg}^0_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}} \to (X : \mathsf{Set}) \to (Y : X \to \mathsf{Set}) \to \mathsf{Set}$$

Main idea (cont.)

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

$$
\begin{aligned}
\mathsf{Arg}^0_{\mathrm{B}} :\, & (\gamma_{\mathrm{A}} : \mathsf{SP}^0_{\mathrm{A}}) \to \\
& (\gamma_{\mathrm{B}} : \mathsf{SP}^0_{\mathrm{B}}(\gamma_{\mathrm{A}})) \\
& (X : \mathsf{Set}) \to \\
& (Y : X \to \mathsf{Set}) \to \\
& (\mathsf{intro}_{\mathrm{A}} : \mathsf{Arg}^0_{\mathrm{A}}(\gamma_{\mathrm{A}}, X, Y) \to X) \\
& \quad \to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Index}^0_{\mathrm{B}} :\, & \cdots \text{same arguments as } \mathsf{Arg}^0_{\mathrm{B}} \cdots \\
& \quad \mathsf{Arg}^0_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, X, Y, \mathsf{intro}_{\mathrm{A}}) \to X
\end{aligned}
$$

Formation and introduction rules

Formation rules:

$$A_{\gamma_A,\gamma_B} : \mathsf{Set} \qquad\qquad B_{\gamma_A,\gamma_B} : A_{\gamma_A,\gamma_B} \to \mathsf{Set}$$

Introduction rule for A_{γ_A,γ_B}:

$$\frac{a : \mathsf{Arg}^0_A(\gamma_A, A_{\gamma_A,\gamma_B}, B_{\gamma_A,\gamma_B})}{\mathsf{intro}_{A_{\gamma_A,\gamma_B}}(a) : A_{\gamma_A,\gamma_B}}$$

Introduction rule for B_{γ_A,γ_B}:

$$\frac{a : \mathsf{Arg}^0_B(\gamma_A, \gamma_B, A_{\gamma_A,\gamma_B}, B_{\gamma_A,\gamma_B}, \mathsf{intro}_{A_{\gamma_A,\gamma_B}})}{\mathsf{intro}_{B_{\gamma_A,\gamma_B}}(a) : B_{\gamma_A,\gamma_B}(\mathsf{Index}^0_B(\gamma_A, \gamma_B, A_{\gamma_A,\gamma_B}, B_{\gamma_A,\gamma_B}, \mathsf{intro}_{A_{\gamma_A,\gamma_B}}, a))}$$

Elimination rules by inversion of introduction rules

Definition of SP_{A}

- Instead of defining $\mathsf{SP}^0_{\mathrm{A}}$ we define a more general set

$$\mathsf{SP}_{\mathrm{A}} : (X_{\mathrm{ref}} : \mathsf{Set}) \to \mathsf{Set}$$

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

- In definition of $\mathsf{Arg}_{\mathrm{A}}$, also require function

$$\mathsf{rep}_{\mathrm{X}} : X_{\mathrm{ref}} \to X$$

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

- Then

$$\mathsf{SP}^0_{\mathrm{A}} := \mathsf{SP}_{\mathrm{A}}(\mathbf{0})$$
$$\mathsf{rep}_{\mathrm{X}} = !_X : \mathbf{0} \to X$$

The codes in SP_{A}

nil

Base case; $\mathsf{intro}_{\mathrm{A}} : \mathbf{1} \to A$.

$$\frac{}{\mathsf{nil} : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) \quad = \quad \mathbf{1}$$

The codes in SP_{A}

non-ind

Noninductive argument; $\mathsf{intro}_{\mathrm{A}} : \big((x : K) \times \ldots\big) \rightarrow A$.

$$\frac{K : \mathsf{Set} \qquad \gamma : K \rightarrow \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}{\mathsf{non\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) = \mathbf{1}$

The codes in SP_{A}

non-ind

Noninductive argument; $\mathsf{intro}_{\mathrm{A}} : \big((x : K) \times \ldots\big) \to A$.

$$\frac{K : \mathsf{Set} \qquad \gamma : K \to \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}{\mathsf{non\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) = \mathbf{1}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{non\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) = (x : K) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma(x), X, Y, \mathsf{rep}_{\mathrm{X}})$$

The codes in SP_{A}

A-ind

Inductive argument in A; $\mathrm{intro}_{\mathrm{A}} : \big((g : K \to A) \times \dots\big) \to A$.

$$\frac{K : \mathrm{Set} \qquad \gamma : \mathrm{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K)}{\text{A-ind}(K, \gamma) : \mathrm{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

$$\mathrm{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathrm{nil}, X, Y, \mathrm{rep}_{\mathrm{X}}) = \mathbf{1}$$

$$\mathrm{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \text{non-ind}(K, \gamma), X, Y, \mathrm{rep}_{\mathrm{X}}) = (x : K) \times \mathrm{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma(x), X, Y, \mathrm{rep}_{\mathrm{X}})$$

The codes in SP_{A}

A-ind

Inductive argument in A; $\mathsf{intro}_{\mathrm{A}} : \big((g : K \to A) \times \ldots\big) \to A$.

$$\frac{K : \mathsf{Set} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K)}{\mathsf{A\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

$$\begin{aligned}
\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) &= \mathbf{1} \\
\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{non\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &= \\
&(x : K) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma(x), X, Y, \mathsf{rep}_{\mathrm{X}}) \\
\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &= \\
&(g : K \to X) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}} + K, \gamma, X, Y, [\mathsf{rep}_{\mathrm{X}}, g])
\end{aligned}$$

The codes in SP_{A}

A-ind

Inductive argument in A; $\mathsf{intro}_{\mathrm{A}} : \big((g : K \to A) \times \ldots\big) \to A$.

$$\frac{K : \mathsf{Set} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}} + K)}{\mathsf{A\text{-}ind}(K, \gamma) : \mathsf{SP}_{\mathrm{A}}(X_{\mathrm{ref}})}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) = \mathbf{1}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{non\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) = (x : K) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma(x), X, Y, \mathsf{rep}_{\mathrm{X}})$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) = (g : K \to X) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}} + K, \gamma, X, Y, [\mathsf{rep}_{\mathrm{X}}, g])$$

In later arguments, we can refer to

$$X_{\mathrm{ref}} \cup \{g(x) | x \in K\} \subseteq X,$$

represented by $[\mathsf{rep}_{\mathrm{X}}, g] : X_{\mathrm{ref}} + K \to X$.

The codes in SP_{A}

B-ind

Inductive argument in B; $\mathsf{intro}_{\mathrm{A}} : \big((g : (x : K) \to B(i(x))) \times \ldots\big) \to A$.

$$\frac{K : \mathsf{Set} \qquad h_{\mathrm{index}} : K \to X_{\mathrm{ref}} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}}{\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma) : \mathsf{SP}_{\mathrm{A}}}$$

$$\begin{aligned}
\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) &= \mathbf{1} \\
\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{non\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &= \\
&(x : K) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma(x), X, Y, \mathsf{rep}_{\mathrm{X}}) \\
\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) &= \\
&(g : K \to X) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}} + K, \gamma, X, Y, [\mathsf{rep}_{\mathrm{X}}, g])
\end{aligned}$$

The codes in SP_{A}

B-ind

Inductive argument in B; $\mathsf{intro}_{\mathrm{A}} : \big((g : (x : K) \to B(i(x))) \times \ldots\big) \to A$.

$$\frac{K : \mathsf{Set} \qquad h_{\mathrm{index}} : K \to X_{\mathrm{ref}} \qquad \gamma : \mathsf{SP}_{\mathrm{A}}}{\mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma) : \mathsf{SP}_{\mathrm{A}}}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{nil}, X, Y, \mathsf{rep}_{\mathrm{X}}) = \mathbf{1}$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{non\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) = (x : K) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma(x), X, Y, \mathsf{rep}_{\mathrm{X}})$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{A\text{-}ind}(K, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) = (g : K \to X) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}} + K, \gamma, X, Y, [\mathsf{rep}_{\mathrm{X}}, g])$$

$$\mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \mathsf{B\text{-}ind}(K, h_{\mathrm{index}}, \gamma), X, Y, \mathsf{rep}_{\mathrm{X}}) = (g : (x : K) \to Y((\mathsf{rep}_{\mathrm{X}} \circ h_{\mathrm{index}})(x))) \times \mathsf{Arg}_{\mathrm{A}}(X_{\mathrm{ref}}, \gamma, X, Y, \mathsf{rep}_{\mathrm{X}})$$

An example

The constructor

$$\triangleright : ((\Gamma : \mathsf{Ctxt}) \times \mathsf{Ty}(\Gamma)) \to \mathsf{Ctxt}$$

is represented by the code

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

We have

$$\begin{aligned} \mathrm{Arg}_{\mathrm{A}}(\mathbf{0}, \gamma_{\triangleright}, \mathsf{Ctxt}, \mathsf{Ty}, !_{\mathsf{Ctxt}}) &= (\Gamma : \mathbf{1} \to \mathsf{Ctxt}) \times (\mathbf{1} \to \mathsf{Ty}(\Gamma(\star))) \times \mathbf{1} \\ &\cong (\Gamma : \mathsf{Ctxt}) \times \mathsf{Ty}(\Gamma) \end{aligned}$$

The codes in SP_{B}

- The universe $\mathsf{SP}^0_{\mathrm{B}} : \mathsf{SP}^0_{\mathrm{A}} \to \mathsf{Set}$ is similar to $\mathsf{SP}^0_{\mathrm{A}}$.
- Need argument $\mathsf{SP}^0_{\mathrm{A}}$ 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

$$
\begin{aligned}
P &: A \to \mathsf{Set} \\
Q &: (x : A) \to B(x) \to \mathsf{Set}
\end{aligned}
$$

instead of the general motive

$$
\begin{aligned}
P &: A \to \mathsf{Set} \\
Q &: (x : A) \to B(x) \to P(x) \to \mathsf{Set}
\end{aligned}
$$

(the "recursive-recursive" eliminator).

The high-level idea

- Define simultaneously

$$A_{\mathrm{pre}} : \mathsf{Set} \qquad B_{\mathrm{pre}} : \mathsf{Set}$$

ignoring dependencies of B on A.

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

$$A_{\mathrm{good}} : A_{\mathrm{pre}} \to \mathsf{Set}$$
$$B_{\mathrm{good}} : A_{\mathrm{pre}} \to B_{\mathrm{pre}} \to \mathsf{Set}$$

(indexed inductive definitions).

- Interpretation

$$[\![A]\!] = \Sigma\, x : A_{\mathrm{pre}} \,.\, A_{\mathrm{good}}(x)$$
$$[\![B]\!](\langle x, x_g \rangle) = \Sigma\, y : B_{\mathrm{pre}} \,.\, B_{\mathrm{good}}(x, y)$$

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?

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 $g : F(X) \to X$.

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 $\mathbb{E}_{A,B} \hookrightarrow \mathrm{Dialg}(F_{A,B}, G)$ such that the elimination rules for (A, B) hold if and only if $\mathbb{E}_{A,B}$ 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

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.
- More details in my thesis.

Summary

- When usi… …ata structures…
- Expressiv…
- But still…
- More det…

Thanks!