Internalizing inductive-inductive definitions in Martin-Löf Type Theory

Fredrik Nordvall Forsberg

Swansea University
csfnf@swansea.ac.uk

München 27.11.2012

Joint work with Anton Setzer.

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system.

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).
- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).
- So I try.

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).

- So I try.

```
data SList : Set where
```

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).
- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).
- So I try.
- The empty list is sorted.

```
data SList : Set where
```

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).
- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).
- So I try.
- The empty list is sorted.

```
data SList : Set where
  []   : SList
```

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).

- So I try.

- The empty list is sorted.

- If I have a sorted list $\ell = [\ell_0, \ldots, \ell_m]$, and an element a, and $a \leq$ all ℓ_k in ℓ, then $[a, \ell_0, \ldots, \ell_m]$ is a sorted list.

```
data SList : Set where
  []   : SList
```

A data type of sorted lists?

- Say that I'm working in some functional programming language with an expressive type system (Martin-Löf type theory/Agda).

- I want to declare a data type whose elements are exactly sorted lists (with elements from some ordered set $(A, \leq)$).

- So I try.

- The empty list is sorted.

- If I have a sorted list $\ell = [\ell_0, \ldots, \ell_m]$, and an element a, and $a \leq$ all ℓ_k in ℓ, then $[a, \ell_0, \ldots, \ell_m]$ is a sorted list.

```
data SList : Set where
  []   : SList
  cons : (a : A) -> (ℓ : SList) -> "a ≤_L ℓ" -> SList
```

What is $\leq_\mathsf{L}$?

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> "a ≤_L ℓ" -> SList
```

- "$a \leq_\mathsf{L} \ell$" if $a \leq$ all elements of ℓ.

What is $\leq_{\mathsf{L}}$?

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> "a ≤L ℓ" -> SList
```

- “$a \leq_{\mathsf{L}} \ell$” if $a \leq$ all elements of ℓ.
- Informal? No! We want to express the specification in the types.

What is $\leq_\mathsf{L}$?

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> "a ≤L ℓ" -> SList
```

- "$a \leq_\mathsf{L} \ell$" if $a \leq$ all elements of ℓ.
- Informal? No! We want to express the specification in the types.
- Natural inductive definition:

What is $\leq_{\mathsf{L}}$?

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> "a ≤L ℓ" -> SList
```

- "$a \leq_{\mathsf{L}} \ell$" if $a \leq$ all elements of ℓ.
- Informal? No! We want to express the specification in the types.
- Natural inductive definition:
  - Every a is trivially smaller than all elements of the empty list `[]`.

What is $\leq_\mathsf{L}$?

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> "a ≤L ℓ" -> SList
```

- "$a \leq_\mathsf{L} \ell$" if $a \leq$ all elements of ℓ.
- Informal? No! We want to express the specification in the types.
- Natural inductive definition:
  - Every a is trivially smaller than all elements of the empty list `[]`.
  - If $x \leq a$ and inductively $x \leq_\mathsf{L} \ell$, then $x \leq_\mathsf{L} \mathrm{cons}(a, \ell, p)$.

Sorted lists and $\leq_L$

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> a ≤L ℓ -> SList
```

- "$a \leq_L \ell$" if $a \leq$ all elements of ℓ.
- Informal? No! We want to express the specification in the types.
- Natural inductive definition:
  - Every a is trivially smaller than all elements of the empty list `[]`.
  - If $x \leq a$ and inductively $x \leq_L \ell$, then $x \leq_L \mathrm{cons}(a, \ell, p)$.

```
data _≤L_ : ℕ -> SList -> Set where
 triv     : ∀ a -> a ≤L []
 ≤L-cons : ∀ x  -> x ≤ a -> x ≤L  ℓ -> x ≤L cons(a, ℓ, p)
```

Sorted lists and $\leq_L$

```
data SList : Set where
 []   : SList
 cons : (a : A) -> (ℓ : SList) -> a ≤L ℓ -> SList

data _≤L_ : ℕ -> SList -> Set where
 triv     : ∀ a -> a ≤L []
 ≤L-cons : ∀ x  -> x ≤ a -> x ≤L  ℓ -> x ≤L cons(a, ℓ, p)
```

Sorted lists and $\le_L$

```
mutual
  data SList : Set where
   []   : SList
   cons : (a : A) -> (ℓ : SList) -> a ≤L ℓ -> SList

  data _≤L_ : ℕ -> SList -> Set where
   triv     : ∀ a -> a ≤L []
   ≤L-cons : ∀ x  -> x ≤ a -> x ≤L  ℓ -> x ≤L cons(a, ℓ, p)
```

- Needs to be a mutual definition – cons refers to $\le_L$, which is indexed by SList.

Sorted lists and $\leq_L$

```
mutual
  data SList : Set where
   []   : SList
   cons : (a : A) -> (ℓ : SList) -> a ≤L ℓ -> SList

  data _≤L_ : ℕ -> SList -> Set where
   triv     : ∀ a -> a ≤L  []
   ≤L-cons : ∀ x  -> x ≤ a -> x ≤L  ℓ -> x ≤L  cons(a, ℓ, p)
```

- Needs to be a mutual definition – cons refers to $\leq_L$, which is indexed by `SList`.

- Both `SList` and $\leq_L$ defined inductively – an inductive-inductive definition!

Sorted lists and $\leq_L$

```
mutual
  data SList : Set where
   []   : SList
   cons : (a : A) -> (ℓ : SList) -> a ≤L ℓ -> SList

  data _≤L_ : ℕ -> SList -> Set where
   triv     : ∀ a -> a ≤L []
   ≤L-cons : ∀ x a ℓ p  -> x ≤ a -> x ≤L  ℓ -> x ≤L cons a ℓ p
```

- Needs to be a mutual definition – cons refers to $\leq_L$, which is indexed by SList.
- Both SList and $\leq_L$ defined inductively – an inductive-inductive definition!

Plan

1. Four slides introduction to Martin-Löf type theory

2. A brief history of inductive types in type theory

3. Inductive-inductive definitions

4. A finite axiomatisation

5. Categorical semantics

Martin-Löf type theory

Five kinds of judgements:

$$\Gamma \text{ context}$$

$$\Gamma \vdash A : \mathsf{Set}$$

$$\Gamma \vdash r : A$$

$$\Gamma \vdash A = B : \mathsf{Set}$$

$$\Gamma \vdash r = s : A$$

Some rules

Forming contexts:

$$\frac{}{\varepsilon \text{ context}} \qquad \frac{\Gamma \text{ context} \qquad \Gamma \vdash A : \mathsf{Set}}{\Gamma, x : A \text{ context}}$$

Forming types:

$$\frac{\Gamma \text{ context}}{\Gamma \vdash \mathbf{1} : \mathsf{Set}} \qquad \frac{\Gamma \text{ context} \qquad \Gamma \vdash A : \mathsf{Set} \qquad \Gamma, x : A \vdash B : \mathsf{Set}}{\Gamma \vdash (\Sigma\ x : A.B) : \mathsf{Set}}$$

$$\vdots$$

Introducing terms:

$$\frac{}{\Gamma \vdash \star : \mathbf{1}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b \rangle : \Sigma x : A.B}$$

$$\vdots$$

Types we will be using

- Dependent function space $(x : A) \to B(x)$ (also written $\prod_{x:A} B$).
  - Elements functions f such that $f(a) : B(a)$ whenever $a : A$.
  - Special case: non-dependent function space $A \to B$.
- Dependent pairs $(x : A) \times B(x)$ (also written $\Sigma x : A.B$).
  - Elements pairs $\langle a, b \rangle$ such that $a : A$ and $b : B(a)$.
  - Special case: Cartesian product $A \times B$.
- Disjoint union $A + B$.
  - Elements $\mathrm{inl}(a)$, $\mathrm{inr}(b)$ where $a : A$ and $b : B$.
  - Can be constructed as $\Sigma x : \mathbf{2}.\mathrm{if}\ x\ \mathrm{then}\ A\ \mathrm{else}\ B$ (if large elimination for **2** is available).
- Empty type $\mathbf{0}$, unit type $\mathbf{1}$ (with inhabitant $\star : \mathbf{1}$).
- Logical Framework formulation of type theory.

Propositions as types

Propositions can be seen as types:

- Universal quantification $\forall x \in A.B(x)$ by $(x : A) \to B(x)$.
- Implication $A \to B$ by $A \to B$.
- Existential quantification $\exists x \in A.B(x)$ by $(x : A) \times B(x)$.
- Conjunction $A \wedge B$ by $A \times B$.
- Disjunction $A \vee B$ by $A + B$.
- The false proposition $\perp$ by $\mathbf{0}$ (no proof).
- True propositions by inhabited types.

Will be implicitly used in the rest of the talk.

A brief history of inductive types

In there 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

$$\frac{}{[\,] : \mathsf{List}_{\mathbb{N}}} \qquad \frac{x : \mathbb{N} \qquad xs : \mathsf{List}_{\mathbb{N}}}{(x :: xs) : \mathsf{List}_{\mathbb{N}}}$$

```
data Listℕ : Set where
    []  :  Listℕ
    _::_  :  ℕ → Listℕ → Listℕ
```

$$\frac{}{0 : \mathtt{KleenesO}} \qquad \frac{n : \mathtt{KleenesO}}{\mathsf{suc}(n) : \mathtt{KleenesO}}$$

$$\frac{f : \mathbb{N} \to \mathtt{KleenesO}}{\lim(f) : \mathtt{KleenesO}}$$

```
data KleenesO : Set where
    O  :  KleenesO
    S  :  KleenesO → KleenesO
    lim  :  (ℕ → KleenesO)
                  → KleenesO
```

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

```
data W A B : Set where
    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.

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

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

- 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 $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = U $\gamma_{\mathsf{List}_{\mathbb{N}}}$

[] : $\mathsf{List}_{\mathbb{N}}$
[] = $\{?_0 : \mathsf{List}_{\mathbb{N}}\}$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$$\begin{array}{l}
\mathsf{List}_{\mathbb{N}}\ \ \mathtt{:}\ \ \mathsf{Set} \\
\mathsf{List}_{\mathbb{N}}\ \ \mathtt{=}\ \ \mathtt{U}\ \ \gamma_{\mathsf{List}_{\mathbb{N}}}
\end{array}$$

$$\begin{array}{l}
\mathtt{[]:}\ \ \mathsf{List}_{\mathbb{N}} \\
\mathtt{[]=\ intro}\ \ \{?_2 : \mathtt{Arg}(\gamma_{\mathsf{List}_{\mathbb{N}}}, \mathsf{List}_{\mathbb{N}})\}
\end{array}$$

$$\begin{array}{l}
\mathtt{\_::\_}\ \ \mathtt{:}\ \ \mathbb{N}\ \to\ \mathsf{List}_{\mathbb{N}}\ \to\ \mathsf{List}_{\mathbb{N}} \\
\mathtt{x\ ::\ xs\ =}\ \ \{?_1 : \mathsf{List}_{\mathbb{N}}\}
\end{array}$$

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = `U` $\gamma_{\mathsf{List}_{\mathbb{N}}}$

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

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = `U` $\gamma_{\mathsf{List}_{\mathbb{N}}}$

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

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = `U` $\gamma_{\mathsf{List}_{\mathbb{N}}}$

`[]` : $\mathsf{List}_{\mathbb{N}}$
`[]` = `intro` $\langle$`tt`, $\{?_4 : \mathbf{1}\}$ $\rangle$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = U $\gamma_{\mathsf{List}_{\mathbb{N}}}$

[]: $\mathsf{List}_{\mathbb{N}}$
[]= intro $\langle$tt, $\star\rangle$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = U $\gamma_{\mathsf{List}_{\mathbb{N}}}$

[]: $\mathsf{List}_{\mathbb{N}}$
[]= intro $\langle$tt, $\star\rangle$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = `U` $\gamma_{\mathsf{List}_{\mathbb{N}}}$

`[] :` $\mathsf{List}_{\mathbb{N}}$
`[]= intro` $\langle$`tt, `$\star\rangle$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = `U` $\gamma_{\mathsf{List}_{\mathbb{N}}}$

`[]` : $\mathsf{List}_{\mathbb{N}}$
`[]` = `intro` $\langle$`tt`, $\star\rangle$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

with

$\mathsf{List}_{\mathbb{N}}$: Set
$\mathsf{List}_{\mathbb{N}}$ = U $\gamma_{\mathsf{List}_{\mathbb{N}}}$

[] : $\mathsf{List}_{\mathbb{N}}$
[] = intro $\langle$tt, $\star\rangle$

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

Example: the code for $\mathsf{List}_{\mathbb{N}}$

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

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

We have

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

with

$\mathsf{List}_{\mathbb{N}}\ \ :\ \ \mathsf{Set}$
$\mathsf{List}_{\mathbb{N}}\ \ \texttt{=}\ \ \texttt{U}\ \ \gamma_{\mathsf{List}_{\mathbb{N}}}$

$\texttt{[]:}\ \ \mathsf{List}_{\mathbb{N}}$
$\texttt{[]= intro}\ \langle\texttt{tt, }\star\rangle$

$\texttt{\_::\_}\ :\ \mathbb{N}\ \to\ \mathsf{List}_{\mathbb{N}}\ \to\ \mathsf{List}_{\mathbb{N}}$
$\texttt{x :: xs = intro}\ \langle\texttt{ff, }\langle\texttt{x, (}\lambda\_.\texttt{xs) , }\star\rangle\rangle$

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 $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$.
- 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 $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$.
- 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.”
- Important to see the difference between induction-recursion and induction-induction.

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.”
- Important to see the difference between induction-recursion and induction-induction.
- Proof by induction is just dependent recursion.

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.

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.
2. An ordinary mutual inductive definition (example: even and odd numbers)
    - Because $B : A \to \mathrm{Set}$ 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, conjecture that it 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, conjecture that it 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, conjecture that it 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.

❶ is a special case of ❷, which is a special case of ❸, which is a special case of induction-induction. However ❹ is not.

Examples of inductive-inductive definitions

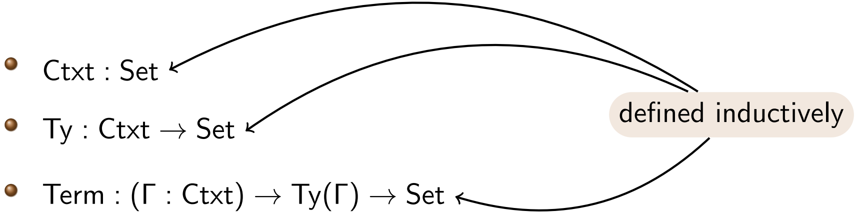
Modelling dependent type theory

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



- Ctxt : Set
- Ty : Ctxt → Set
- Term : (Γ : Ctxt) → Ty(Γ) → Set

defined inductively

- . . .
- Substitutions, . . .
- . . .

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.

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

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

Constructor for Ty referring to constructor for Ctxt

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

Constructor for Ty referring to constructor for Ctxt

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

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

Constructor for Ty referring to constructor for Ctxt

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

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

Constructor for Ty referring to constructor for Ctxt

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

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

Constructor for Ty referring to constructor for Ctxt

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

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

Constructor for Ty referring to constructor for Ctxt

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

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

Constructor for Ty referring to constructor for Ctxt

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

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

(Also have base type ι in any context:

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

Conway's surreal numbers

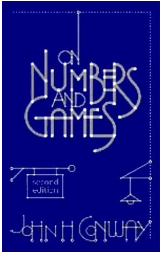
- Totally ordered Field containing the reals and the ordinals (at least classically).

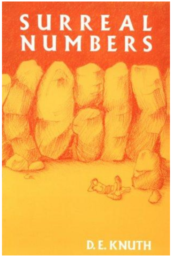
- "Fills the holes" between them as well (think infinitesimals).

- 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!

An inductive-inductive definition

Define simultaneously

$$\begin{aligned} \mathsf{Surreal} &: \mathsf{Set} \\ \leq &: \mathsf{Surreal} \to \mathsf{Surreal} \to \mathsf{Set} \\ \not\leq &: \mathsf{Surreal} \to \mathsf{Surreal} \to \mathsf{Set} \end{aligned}$$

Need to encode some set theory such as $\mathcal{P}(\mathsf{Surreal})$ and $x \in X_{\mathrm{L}}$ in type theory – we deal with this informally.

(Use $\mathcal{P}(X) :\equiv \Sigma a : U . T(a) \to X$ for some universe (U, T). See e.g. Aczel's interpretation of CZF in type theory (Aczel 1978).)

Constructor for Surreal

Definition

A surreal number $\{X_L|X_R\}$ consists of two sets of surreal numbers X_L, X_R such that

- $(\forall x^L \in X_L)(\forall x^R \in X_R)\,\neg(x^L \geq x^R)$.

All surreal numbers are constructed this way.

$$\begin{aligned}
&\textbf{data}\ \text{Surreal} : \text{Set where}\\
&\qquad \{\_|\_\}_{\_} : (X_L : \mathcal{P}(\text{Surreal})) \to (X_R : \mathcal{P}(\text{Surreal}))\\
&\qquad\qquad \to (\forall x^L \in X_L)(\forall x^R \in X_R)((x^L \geq x^R) \to \bot)\\
&\qquad\qquad \to \text{Surreal}
\end{aligned}$$

Constructor for Surreal

Definition

A surreal number $\{X_L|X_R\}$ consists of two sets of surreal numbers X_L, X_R such that

- $(\forall x^L \in X_L)(\forall x^R \in X_R)\,\neg(x^L \geq x^R)$.

All surreal numbers are constructed this way.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Surreal} : \mathsf{Set}\ \mathsf{where}\\
&\qquad \{\_|\_\}\_ : (X_L : \mathcal{P}(\mathsf{Surreal})) \to (X_R : \mathcal{P}(\mathsf{Surreal}))\\
&\qquad\qquad \to (\forall x^L \in X_L)(\forall x^R \in X_R)((x^L \geq x^R) \to \bot)\\
&\qquad\qquad \to \mathsf{Surreal}
\end{aligned}
$$

Constructor for Surreal

Definition

A surreal number $\{X_L | X_R\}$ consists of two sets of surreal numbers X_L, X_R such that

- $(\forall x^L \in X_L)(\forall x^R \in X_R)\,\neg(x^L \geq x^R)$.

All surreal numbers are constructed this way.

We cannot have negative occurrences of the set we are defining!

data Surrea

$$\{\_|\_\}\_ : (X_L : \mathcal{P}(\text{Surreal})) \to (X_R : \mathcal{P}(\text{Surreal}))$$
$$\to (\forall x^L \in X_L)(\forall x^R \in X_R)((x^L \geq x^R) \to \bot)$$
$$\to \text{Surreal}$$

Constructor for Surreal

Definition

A surreal number $\{X_L | X_R\}$ consists of two sets of surreal numbers X_L, X_R such that

- $(\forall x^L \in X_L)(\forall x^R \in X_R)\, \neg(x^L \geq x^R)$.

All surreal numbers are constructed this way.

$$
\begin{aligned}
&\textbf{data}\ \text{Surreal} : \text{Set where} \\
&\quad \{\_|\_\}\_ : (X_L : \mathcal{P}(\text{Surreal})) \to (X_R : \mathcal{P}(\text{Surreal})) \\
&\qquad \to (\forall x^L \in X_L)(\forall x^R \in X_R)(x^L \not\geq x^R) \\
&\qquad \to \text{Surreal}
\end{aligned}
$$

Negative occurrences of $\geq$

Definition

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

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

- Define $x \geq y$ and $x \not\geq y$ simultaneously.
- $\neg(x \geq y)$ iff

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

  if

  $$(\exists x^R \in X_R)\,(y \geq x^R) \text{ or } (\exists y^L \in Y_L)\,(y^L \geq x)$$

  (also "only if" with classical logic).
- So we define $x \not\geq y$ iff

  $$(\exists x^R \in X_R)\,(y \geq x^R) \text{ or } (\exists y^L \in Y_L)\,(y^L \geq x)$$

Mutual definition of $\geq$ and $\not\geq$

Definition

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

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

$$\begin{aligned}
&\textbf{data} \;\; \geq : \text{Surreal} \to \text{Surreal} \to \text{Set where} \\
&\quad geq : \ldots X_L, X_R, p \ldots \\
&\qquad\quad \to \ldots Y_L, Y_R, q \ldots \\
&\qquad\quad \to (\forall x^R \in X_R)(\{Y_L | Y_R\}_q \not\geq x^R) \\
&\qquad\quad \to (\forall y^L \in Y_L)(y^L \not\geq \{X_L | X_R\}_p) \\
&\qquad\quad \to \{X_L | X_R\}_p \geq \{Y_L | Y_R\}_q
\end{aligned}$$

Mutual definition of $\geq$ and $\ngeq$ (cont.)

$\neg(x \geq y)$ if

$$(\exists x^R \in X_R)\,(y \geq x^R) \text{ or } (\exists y^L \in Y_L)\,(y^L \geq x)$$

$$\begin{aligned}
&\textbf{data} \ngeq : \text{Surreal} \to \text{Surreal} \to \text{Set where}\\
&\quad \mathit{ngeql} : \ldots X_L, X_R, p \ldots\\
&\qquad \to \ldots Y_L, Y_R, q \ldots\\
&\qquad \to (\exists x^R \in X_R)(\{Y_L | Y_R\}_q \geq x^R)\\
&\qquad \to \{X_L | X_R\}_p \ngeq \{Y_L | Y_R\}_q\\
&\quad \mathit{ngeqr} : \ldots X_L, X_R, p \ldots\\
&\qquad \to \ldots Y_L, Y_R, q \ldots\\
&\qquad \to (\exists y^L \in Y_L)(y^L \geq \{X_L | X_R\}_p)\\
&\qquad \to \{X_L | X_R\}_p \ngeq \{Y_L | Y_R\}_q
\end{aligned}$$

Constructing the Field structure

- Can then use the elimination rules for inductive-inductive definitions to define negation, addition, multiplication . . .
- Typical pattern: need to define the operation and prove that it preserves the order structure etc simultaneously.
- Work in progress.
- Related work: Mamane: Surreal Numbers in Coq (2006)
    - Encoding of surreal numbers, since Coq does not support induction-induction.

A finite axiomatisation

An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?

An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?

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

An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?
- 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

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

An axiomatisation

- How to axiomatise a type theory with inductive-inductive definitions?

- 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:

$$A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : \mathsf{Set} \qquad B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}} \to \mathsf{Set}$$

Introduction rule for $A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{a : \mathsf{Arg}^{0}_{\mathrm{A}}(\gamma_{\mathrm{A}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}})}{\mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(a) : A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}$$

Introduction rule for $B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}$:

$$\frac{a : \mathsf{Arg}^{0}_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}})}{\mathsf{intro}_{B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}(a) : B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}(\mathsf{Index}^{0}_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}, A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, B_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}, \mathsf{intro}_{A_{\gamma_{\mathrm{A}},\gamma_{\mathrm{B}}}}, a))}$$

Elimination rules: no problem in extensional type theory, not so easy intentionally.

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

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

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

The codes in SP_{A}

non-ind

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

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

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

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

The codes in SP_{A}

A-ind

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

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

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

The codes in SP_{A}

A-ind

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

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

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

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

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

The codes in SP_{A}

A-ind

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

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

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

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

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

In later arguments, we can refer to

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

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

The codes in SP_{A}

B-ind

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

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

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

The codes in SP_{A}

B-ind

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

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

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

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 $\mathrm{SP}^0_{\mathrm{B}} : \mathrm{SP}^0_{\mathrm{A}} \to \mathrm{Set}$ is similar to $\mathrm{SP}^0_{\mathrm{A}}$.
- Need argument $\mathrm{SP}^0_{\mathrm{A}}$ to know the shape of constructor for the first set, which can appear in indices.
- We omit the definition here.

Categorical semantics

Initial-algebra like semantics

Joint work with Thorsten Altenkirch and Peter Morris (CALCO 2011)

- Thorsten was not happy with the axiomatisation presented.
- He wanted something cleaner, like initial-algebra semantics.
- However, seem to need to use dialgebras $f : F(A) \rightarrow G(A)$ instead of ordinary algebras $f : F(A) \rightarrow A$.

Dialgebras

Definition

Let $F, G : \mathbb{C} \to \mathbb{D}$ be functors. An (F, G)-dialgebra (X, f) consists of $X \in \mathbb{C}$ and $f : F(X) \to G(X)$. A morphism between dialgebras (X, f) and (Y, g) is a morphism $\alpha : X \to Y$ in $\mathbb{C}$ such that

$$\begin{tikzcd}
F(X) \arrow[r, "f"] \arrow[d, "F(\alpha)"'] & G(X) \arrow[d, "G(\alpha)"] \\
F(Y) \arrow[r, "g"'] & G(Y)
\end{tikzcd}$$

Write Dialg(F, G) for the category of (F, G)-dialgebras.

Of course, $G = \mathrm{id} : \mathbb{C} \to \mathbb{C}$ gives ordinary F-algebras as a special case.

$\mathrm{Arg}_{\mathrm{A}}$ and $\mathrm{Arg}_{\mathrm{B}}$ as functors

Theorem (extensional type theory)

For all γ_{A}*,* γ_{B}*,* $\mathrm{Arg}_{\mathrm{A}}(\gamma_{\mathrm{A}})$ *and* $\mathrm{Arg}_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}})$ *extends to functors*

$$\mathrm{Arg}_{\mathrm{A}}(\gamma_{\mathrm{A}}) : \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Set}$$
$$\mathrm{Arg}_{\mathrm{B}}(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}) : \mathrm{Dialg}(\mathrm{Arg}_{\mathrm{A}}(\gamma_{\mathrm{A}}), \pi_0) \to \mathrm{Fam}(\mathrm{Set})$$

where $\pi_0 : \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Set}$ *is defined by* $\pi_0(A, B) = A$*.*

Definition of $\mathbb{E}_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}$

Using a pullback of categories, one can define a subcategory $\mathbb{E}_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}$ of the category $\mathrm{Dialg}(\mathrm{Arg}_{\mathrm{B}}, V)$ playing the role of the category of algebras.

$V : \mathrm{Dialg}(\mathrm{Arg}_{\mathrm{A}}, U) \to \mathrm{Fam}(\mathrm{Set})$ is the forgetful functor $V(X, f) = X$.

Elimination rules from initiality

One can then show:

Theorem (extensional type theory)

For an inductive-inductive definition given by a code $(\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}})$, the elimination rules hold if and only if $\mathbb{E}_{\gamma_{\mathrm{A}}, \gamma_{\mathrm{B}}}$ has an initial object.

Main obstacle: Initiality gives non-dependent functions, elimination rules dependent. Solution: Use Σ-types.

Concluding remarks

Status in proof assistants

- Not supported in Coq or Epigram.
- Is supported in Agda!
- Now we know it is sound as well. . .

Conjecture: reducible to indexed inductive definitions

- It seems as if the theory of inductive-inductive definitions can be reduced to the (extensional) theory of indexed inductive definitions.

- 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 (indexed inductive definitions).

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

Summary

Take away message 1

When programming with dependent types, one naturally wants more advanced data structures such as inductive-inductive definitions.

Take away message 2

By using a universe of data types, they can be internalised into the type theory, useful e.g. for generic programming.

- Axiomatisation à la induction-recursion (N. F., Setzer 2010, 2012).
- Alternative categorical characterisation (N. F., Altenkirch, Morris, Setzer 2011).

- Will hopefully turn into a thesis in the spring.

Summary

Take away m

When program ore
advanced data

Take away m

By using a un e type
theory, useful

- Axiomatis 2012).
- Alternativ rris,
Setzer 20
- Will hope

Thanks!