Inductive-Inductive Definitions

Anton Setzer
(Joint work with Fredrik Forsberg)

Swansea University, Swansea UK

5 June 2012
Introduction

Examples

Closed Formalisation of Inductive-Inductive Definitions

Conclusion
Introduction

Examples

Closed Formalisation of Inductive-Inductive Definitions

Conclusion
Preliminary Remarks

- Type Theory is only the syntactic framework. Induction-induction and induction-recursion not necessarily bound to this framework.
Type Theory

- Judgements:
  \[ \Gamma \Rightarrow \text{Context} \]
  \[ \Gamma \Rightarrow A : \text{Set} \quad \Gamma \Rightarrow A = B : \text{Set} \]
  \[ \Gamma \Rightarrow r : A \quad \Gamma \Rightarrow r = s : A \]

- Some Rules:
  \[ \emptyset : \text{Context} \]
  \[ \frac{\Gamma \Rightarrow A : \text{Set}}{\Gamma, x : A \Rightarrow \text{Context}} \]
  \[ \frac{\Gamma, x : A \Rightarrow B : \text{Set}}{\Gamma \Rightarrow (\Sigma x : A.B) : \text{Set}} \]
Simplifications

- Logical Framework:
  - Allows to form e.g.

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

- With the logical framework, rules for $\Sigma$ becomes

\[
\Sigma : (A : \text{Set}) \rightarrow (B : A \rightarrow \text{Set}) \rightarrow \text{Set}
\]

- That’s how it occurs in theorem provers (Alf, Half, Agda, Coq, NuPrl).
Formulate Semantics of Type Theory inside Type Theory.

So we formulate in type theory a model \((\widehat{\text{Set}}, \llbracket \rrbracket)\) of a weaker type theory.

Done by defining

- A set \(\widehat{\text{Set}}\) of codes for elements of \(\text{Set}\) inductively
- a function \(\llbracket \rrbracket : \widehat{\text{Set}} \to \text{Set}\) recursively.
Define inductive-recursively

\[ \hat{\text{Set}} : \text{Set} \quad [\quad] : \hat{\text{Set}} \to \text{Set} \]

- Rule for \( \Sigma \):

\[ \Sigma : (A : \text{Set}) \to (B : A \to \text{Set}) \to \text{Set} \]

is reflected into

\[ \hat{\Sigma} : (a : \hat{\text{Set}}) \to (b : [a] \to \hat{\text{Set}}) \to \hat{\text{Set}} \]

\[ [\hat{\Sigma} \ a \ b] = \Sigma [a] (\lambda x. [b \ x]) : \text{Set} \]
From Induction-Recursion to Induction-Induction

General induction-recursion:
- Define $A : \text{Set}$ inductively,
- while defining a function $B : A \rightarrow \text{Set}$ recursively.
  ($\text{Set}$ can be generalised to types).

Induction-induction:
Instead of defining $B$ recursively define $B$ inductively.
So we define simultaneously
- $A : \text{Set}$ inductively,
- $B : A \rightarrow \text{Set}$ inductively.
Defining Syntax using Induction-Induction

- Formulate Syntax of Type Theory inside Type Theory (Nils Danielsson)
- Define inductively simultaneously:
  - \( \widehat{\text{Context}} : \text{Set} \)
    - \( \Gamma : \widehat{\text{Context}} \) represents \( \Gamma \Rightarrow \sim \text{Context} \).
  - \( \widehat{\text{Set}} : \text{Context} \rightarrow \text{Set} \)
    - \( A : \widehat{\text{Set}} \Gamma \) represents \( \Gamma \Rightarrow A : \text{Set} \).
  - \( \widehat{\text{Term}} : (\Gamma : \text{Context}) \rightarrow (A : \widehat{\text{Set}} \Gamma) \rightarrow \text{Set} \)
    - \( r : \widehat{\text{Term}} \Gamma A \) represents \( \Gamma \Rightarrow r : A \).
  - \( \widehat{\text{SynSet}} = : (\Gamma : \text{Context}) \rightarrow (A, B : \widehat{\text{Set}} \Gamma) \rightarrow \text{Set} \)
    - \( p : \widehat{\text{SynSet}} = \Gamma A B \) represents a derivation of \( \Gamma \Rightarrow A = B : \text{Set} \).
  - etc.
Representation of Rules

- Rule

  \[ \emptyset : \text{Context} \]

  represented as

  \[ \widehat{\emptyset} : \widehat{\text{Context}} \]

- Rule

  \[
  \Gamma \Rightarrow A : \text{Set} \\
  \Gamma, x : A \Rightarrow \text{Context}
  \]

  represented (variable-free)

  \[ \widehat{\widehat{\widehat{\Gamma} \Rightarrow A}} : (\Gamma : \widehat{\text{Context}}) \rightarrow (A : \widehat{\text{Set}} \Gamma) \rightarrow \widehat{\text{Context}} \]

  where we write \( \Gamma \widehat{\widehat{\widehat{\cdot}}} A \) for \( \widehat{\widehat{\widehat{\cdot}}} \Gamma A \).
Representation of Rules

- Rule

\[ \Gamma, x : A \Rightarrow B : \text{Set} \]
\[ \Gamma \Rightarrow \Sigma x : A.B : \text{Set} \]

which in full reads

\[ \Gamma : \text{Context} \]
\[ \Gamma \Rightarrow A : \text{Set} \]
\[ \Gamma, x : A \Rightarrow B : \text{Set} \]
\[ \Gamma \Rightarrow \Sigma x : A.B : \text{Set} \]

is represented as

\[ \widehat{\Sigma} : (\Gamma : \widehat{\text{Context}}) \]
\[ \rightarrow (A : \widehat{\text{Set}} \Gamma) \]
\[ \rightarrow (B : \widehat{\text{Set}} (\Gamma \triangleright A)) \]
\[ \rightarrow \widehat{\text{Set}} \Gamma \]
Observation

- We define simultaneously
  - Context : Set inductively,
  - Set : Context \rightarrow Set inductively,
  - Term : (\Gamma : \text{Context}) \rightarrow \text{Set} \Gamma \rightarrow \text{Set} inductively.
  - \ldots

- Here restriction to only 2 levels, we define
  - A : \text{Set}
  - B : A \rightarrow \text{Set}

inductive-inductively.
Observation

- **In**
  - $A : \text{Set}$
  - $B : A \to \text{Set}$

  the constructor of $B \times$ might refer to the constructor of $A$. 

- For instance in

  \[
  \Sigma : (\Gamma : \text{Context}) \to (A : \hat{\text{Set}} \Gamma) \to (B : \hat{\text{Set}} (\Gamma \vdash A)) \to \hat{\text{Set}} \Gamma
  \]

  the second argument refers to the constructor $\vdash$ for $\hat{\text{Set}}$. 

---

Anton Setzer  
Inductive-Inductive Definitions
Induction-Induction is not Indexed Induction

- In indexed inductive definitions
  - we have a given \( I : \text{Set} \)
  - and define sets \( A : I \to \text{Set} \) inductively simultaneously.

- In induction-induction
  - the index set \( A : \text{Set} \) is defined simultaneously inductively with \( B : A \to \text{Set} \).
Introduction

Induction-Induction is not Induction-Recursion

- For a constructor
  \[ C \ a \ b : A \]
  we have no recursive equation:
  \[ B (C \ a \ b) = \cdots \]

- In fact constructors for \( A \) and constructors for \( B \) are not necessarily connected.

- However constructors of \( B \) might refer to constructors of \( A \).

- \( B : A \to \text{Set} \) is defined inductively not recursively.

- Constructors of \( A, B \) can refer to \( B \) only strictly positively.
Introduction

Examples

Closed Formalisation of Inductive-Inductive Definitions

Conclusion
Ordinal Notation System

Typical definition:

The set of pre ordinals $T$ is defined inductively by:

- If $a_1, \ldots, a_k \in T$ and $n_1, \ldots, n_k \in \mathbb{N} \setminus \{0\}$ then
  \[ \omega^{a_1} n_1 + \cdots + \omega^{a_k} n_k \in T \]

We define $\prec$ on $T$ recursively by

\[ \omega^{a_1} n_1 + \cdots + \omega^{a_k} n_k \prec \omega^{b_1} m_1 + \cdots + \omega^{b_l} m_l \]

iff

\[ (a_1, n_1, \ldots, a_k, n_k) \prec_{\text{lex}} (b_1, m_1, \ldots, b_l, m_l) \]

We define $OT \subseteq T$ inductively:

- If $a_1, \ldots, a_k \in OT$ and $a_k \prec \cdots \prec a_1$ and $n_1, \ldots, n_k \in \mathbb{N} \setminus \{0\}$ then
  \[ \omega^{a_1} n_1 + \cdots + \omega^{a_k} n_k \in OT \]
Definition of $OT$ Inductive-Inductively

- Define $OT : \text{Set}$ and $\prec : OT \to OT \to \text{Set}$ inductive-inductively:
  - If $a_1, \ldots, a_k \in OT$ and $a_k \prec \cdots \prec a_1$ and $n_1, \ldots, n_k \in \mathbb{N} \setminus \{0\}$ then
    \[
    \omega^{a_1} n_1 + \cdots + \omega^{a_k} n_k \in OT
    \]
  - If
    \[
    \omega^{a_1} n_1 + \cdots + \omega^{a_k} n_k
    \omega^{b_1} m_1 + \cdots + \omega^{b_l} m_l \in OT
    \]
    and
    \[
    (a_1, n_1, \ldots, a_k, n_k) \prec_{\text{lex}} (b_1, m_1, \ldots, b_l, m_l)
    \]
    then
    \[
    \omega^{a_1} n_1 + \cdots + \omega^{a_k} n_k \prec \omega^{b_1} m_1 + \cdots + \omega^{b_l} m_l
    \]
Conway’s Surreal Numbers

- Like Dedekind cuts, but replacing rationals by previously defined surreal numbers.
- Surreal numbers contain all ordered fields.
- Definition in set theory.
- Definition of the class of surreal numbers $\text{Surreal}$ together with an ordering $\leq$:
  - If $X_L, X_R \in \mathcal{P}(\text{Surreal})$ such that
    \[ \neg \exists x_L \in X_L. \exists x_R \in X_R. x_R \leq x_L \]
    then $(X_L, X_R) \in \text{Surreal}$
  - $X = (X_L, X_R) \leq (Y_L, Y_R) = Y$ iff
    - $\neg \exists x_L \in X_L. Y \leq x_L$
    - $\neg \exists y_R \in Y_R. y_R \leq X$
Surreal Numbers as an Inductive-Inductive Definition

- Define simultaneously inductively

\[
\begin{align*}
\text{Surreal} : & \quad \text{Set} \\
\preceq & : \quad \text{Surreal} \to \text{Surreal} \to \text{Set} \\
\nleq & : \quad \text{Surreal} \to \text{Surreal} \to \text{Set}
\end{align*}
\]

- \( P(\text{Surreal}) \) replaced by \( \Sigma a : U.T \ a \to \text{Surreal} \) for some universe \( U \).
- We refer to this and \( x \in X_L \) informally.
Inductive-Inductive Definition of Surreal

If $X_L, X_R \in \mathcal{P}(\text{Surreal})$, and

$p : \forall x_L \in X_L. \forall x_R \in X_R. x_R \not\leq x_L$

then $(X_L, X_R)_p : \text{Surreal}$.

Assume $X = (X_L, X_R)_p$, $Y = (Y_L, Y_R)_q : \text{Surreal}$. Assume

$\forall x_L \in X_L. Y \not\leq x_L$

$\forall y_R \in Y_R. y_R \not\leq X$

then $X \leq Y$. 
Examples

Inductive-Inductive Definition of Surreal

Assume $X = (X_L, X_R)_p$, $Y = (Y_L, Y_R)_q : 	ext{Surreal}$.

- If
  
  $$\exists x_L \in X_L. Y \leq x_L$$

  then $X \not\leq Y$.

- If
  
  $$\exists y_R \in Y_R. y_R \leq X$$

  then $X \not\leq Y$. 

Anton Setzer

Inductive-Inductive Definitions
Inductive-inductive definitions seem to be very frequent in mathematics.

Usually reduced to inductive definitions by

- first defining simultaneously inductively $A_{pre} : \text{Set}$, $B_{pre} : \text{Set}$ by ignoring dependencies of $B$ on $A$.
- then selecting $A \subseteq A_{pre}$, $B \subseteq B_{pre}$ by selecting those elements which fulfil the correct rules.

Seems to be a general method of reducing inductive-inductive definitions to inductive definitions (work in progress).
Introduction

Examples

Closed Formalisation of Inductive-Inductive Definitions

Conclusion
We define as for inductive-inductive definitions a closed formalisation.

Complicated since it will define not just examples but all inductive-inductive definitions in one set of rules.
Main Idea

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

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

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

$$\text{Index}^0_B : \cdots \text{same arguments as for } \text{Arg}^0_B \cdots$$
$$\rightarrow \text{Arg}^0_B \gamma_A X Y \text{intro}_A \gamma_B$$
$$\rightarrow X$$
Assume $\gamma_A : \text{SP}_A^0$, $\gamma_B : \text{SP}_B^0 \gamma_A$.
Let $\gamma : = (\gamma_A, \gamma_B)$.

Formation rules

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

Introduction rule for $A_\gamma$:

$\text{introA}_\gamma : \text{Arg}_A^0 \gamma_A A_\gamma B_\gamma \rightarrow A_\gamma$

Introduction rule for $B_\gamma$:

$\text{introB}_\gamma : (\text{arg} : \text{Arg}_B^0 \gamma_A A_\gamma B_\gamma \text{intro}_\gamma \gamma_B) \\
\rightarrow B_\gamma (\text{Index}_B^0 \gamma_A A_\gamma B_\gamma \text{intro}_\gamma \gamma_B \text{arg})$
Definition of $SP_A$

- Instead of defining $SP_A^0$ we define a more general set

$$SP_A : (A_{ref} : \text{Set}) \rightarrow \text{Type}$$

which refers to elements $A_{ref}$ of the set to be defined already referred to in inductive arguments.

- Then

$$SP_A^0 := SP_A \emptyset$$
Constructors for $SP_A$

- Initial case: constructor with no arguments:
  \[
  \text{nil} : SP_A \, A_{ref}
  \]

- One non-inductive argument of type $K$ followed by other arguments given by $\gamma$:
  \[
  \text{non−ind} : (K : \text{Set}) \to (\gamma : K \to SP_A \, A_{ref}) \to SP_A \, A_{ref}
  \]

- Inductive arguments of type $A$ indexed over a set $K$ followed by arguments (which can refer to these arguments) given by $\gamma$:
  \[
  \text{A−ind} : (K : \text{Set}) \to (\gamma : SP_A \,(A_{ref} + K)) \to SP_A \, A_{ref}
  \]
Constructors for $\text{SP}_A$

- Inductive arguments of type $B$ indexed over a set $K$; we need to have the indices for $B$, for which we use a function $\text{index} : K \to A_{ref}$; later arguments are given by $\gamma$:

$$
\text{B–ind} : (K : \text{Set}) \\
\to (\text{index} : K \to A_{ref}) \\
\to (\gamma : \text{SP}_A A_{ref}) \\
\to \text{SP}_A A_{ref}
$$
Remaining Steps

- Define $\text{Arg}_A$ recursively (straightforward).
- For defining $\text{Arg}_B$ we need to define the set of terms $\text{ATerm}$ of type $A$ we can form from given elements of type $A$ and the later defined constructor $\text{intro}_A$.
- Then define $\text{SP}_B$ and $\text{Arg}_B$, $\text{Index}_B$.
- Requires some functoriality problems.
- Main problems arise due to intensional equality.
Introduction

Examples

Closed Formalisation of Inductive-Inductive Definitions

Conclusion
Summary

▶ Induction-induction is a natural way of defining the syntax of type theory inside type theory.
▶ Induction-induction occur naturally in mathematics.
  ▶ Seem to be more common than induction-recursion.
  ▶ Maybe, because they are more easily reduced to well-understood inductive definitions.
    ▶ Usage of inductive-recursive definitions without having the concept is much more difficult.
▶ Having them as first-class citizens reduces some of the complexity.
Summary

- Examples can be formulated easily.
- Closed formalisation more complicated.
Open Problems

- Elimination Rules (induction over an induction-inductive definitions).
  - Elimination rules for concrete examples can be written down easily.
  - An abstract general elimination rule has been defined.
  - A general concrete elimination rule complicated (due to intensional equality).
- Formulation in ordinary mathematics (first order).
- Generalisations
  - More levels.
  - More complex structures such as \( B : A \rightarrow A \rightarrow \text{Set} \).
  - Combination with induction-recursion.