Formalising inductive-inductive definitions

Fredrik Nordvall Forsberg

Department of Computer Science
Swansea University
csfnf@swansea.ac.uk

PECP, August 21, 2010

Joint work with Anton Setzer

Outline

This is a talk about an induction principle, called *induction-induction*, in type theory. I will try to:

1. Tell you what induction-induction is.
2. Show you why it is a useful principle.
3. Describe our formalisation.
4. Sketch how we might be able to reduce the principle to indexed inductive definitions.

Notation

- $x : A$ means x is of type A.
  - E.g. $8 : \mathbb{N}$.
- Set is the type of small types.
  - $\mathbf{0}$: Set (the empty type),
  - $\mathbf{1}$: Set (the one element type with $\star : \mathbf{1}$).
- Dependent product: $(x : A) \times B(x)$.
  - Sometimes written $\Sigma x : A.B(x)$.
  - Elements pairs $\langle a, b \rangle$, where $a : A$ and $b : B(a)$.
- Dependent function space: $(x : A) \to B(x)$.
  - Sometimes written $\Pi x : A.B(x)$.
  - Elements functions f that maps $a : A$ to $f(a) : B(a)$.

What is induction-induction?

What is induction-induction?

- Induction-induction is an induction principle in Martin-Löf Type Theory.
- It allows us to define $A : \mathrm{Set}$, together with $B : A \to \mathrm{Set}$, where:
  - Both A and $B(a)$ for $a : A$ are inductively defined.
  - The constructors for A can refer to B and vice versa.

An example

Define

$$A : \mathrm{Set} \qquad B : A \to \mathrm{Set}$$

with constructors

$$\begin{aligned}
&\mathrm{intro}_{\mathrm{A,base}} : A \\
&\mathrm{intro}_{\mathrm{A,step}} : (a_1 : A) \to B(a_1) \to (a_2 : A) \to A \\
&\mathrm{intro}_{\mathrm{B,base}} : \mathbb{N} \to (a : A) \to B(a) \\
&\mathrm{intro}_{\mathrm{B,step}} : (a : A) \to (b : B(a)) \to B(\mathrm{intro}_{\mathrm{A}}(a, b, a))
\end{aligned}$$

An example

Define

$$A : \mathrm{Set} \qquad B : A \to \mathrm{Set}$$

with constructors

$$\begin{aligned}
&\mathrm{intro}_{\mathrm{A,base}} : A \\
&\mathrm{intro}_{\mathrm{A,step}} : (a_1 : A) \to B(a_1) \to (a_2 : A) \to A \\
&\mathrm{intro}_{\mathrm{B,base}} : \mathbb{N} \to (a : A) \to B(a) \\
&\mathrm{intro}_{\mathrm{B,step}} : (a : A) \to (b : B(a)) \to B(\mathrm{intro}_{\mathrm{A}}(a, b, a))
\end{aligned}$$

- This is not an ordinary mutual inductive definition, because B is indexed by A.
- It is not an indexed inductive definition, because the index set A is not fixed beforehand.
- It is not an inductive-recursive definition, because B is defined inductively, not recursively.

An example

Define

$$A : \mathrm{Set} \qquad B : A \to \mathrm{Set}$$

with constructors

$$\mathrm{intro}_{\mathrm{A,base}} : A$$

$$\mathrm{intro}_{\mathrm{A,step}} : (a_1 : A) \to B(a_1) \to (a_2 : A) \to A$$

Now: a meaningful example!

- This is not an ordinary mutual inductive definition, because B is indexed by A.
- It is not an indexed inductive definition, because the index set A is not fixed beforehand.
- It is not an inductive-recursive definition, because B is defined inductively, not recursively.

Martin-Löf Type Theory in Martin-Löf Type Theory

Consider the problem of modelling Martin-Löf Type Theory inside Martin-Löf Type Theory.
Following Danielsson and Chapman, we could define

$$\begin{array}{rl} \mathsf{Ctxt} : \mathsf{Set} & \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context} \\ \mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} & \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma \\ \mathsf{Term} : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Set} & t : \mathsf{Term}(\Gamma, \tau) \Leftrightarrow t \text{ is a term of type } \tau \text{ in context } \Gamma \\ \vdots & \end{array}$$

This way, we can define exactly the well-formed terms.

Martin-Löf Type Theory in Martin-Löf Type Theory

Consider the problem of modelling Martin-Löf Type Theory inside Martin-Löf Type Theory.
Following Danielsson and Chapman, we could define

$$\begin{array}{ll} \mathsf{Ctxt} : \mathsf{Set} & \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context} \\ \mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} & \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma \\ \mathsf{Term} : (\Gamma : \mathsf{Ctxt}) \to \mathsf{Ty}(\Gamma) \to \mathsf{Set} & t : \mathsf{Term}(\Gamma, \tau) \Leftrightarrow t \text{ is a term of type } \tau \text{ in context } \Gamma \\ \vdots & \end{array}$$

This way, we can define exactly the well-formed terms.

Martin-Löf Type Theory in Martin-Löf Type Theory (cont.)

$$\mathsf{Ctxt} : \mathsf{Set} \qquad \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context}$$

$$\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} \qquad \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma$$

The problem is, everything seems to depend on everything else. . .

Martin-Löf Type Theory in Martin-Löf Type Theory (cont.)

$$\mathsf{Ctxt} : \mathsf{Set} \qquad \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context}$$

$$\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} \qquad \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma$$

The problem is, everything seems to depend on everything else...

- In order to extend a context Γ with another variable $x : \tau$, we need to know that τ is a well-formed type in the current context Γ.

Martin-Löf Type Theory in Martin-Löf Type Theory (cont.)

$$\begin{array}{ll} \mathsf{Ctxt} : \mathsf{Set} & \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context} \\ \mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} & \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma \end{array}$$

The problem is, everything seems to depend on everything else...

- In order to extend a context Γ with another variable $x : \tau$, we need to know that τ is a well-formed type in the current context Γ.
- Types obviously depends on contexts, as they are indexed over contexts.

Martin-Löf Type Theory in Martin-Löf Type Theory (cont.)

$$\mathsf{Ctxt} : \mathsf{Set} \qquad \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context}$$

$$\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} \qquad \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma$$

The problem is, everything seems to depend on everything else...

- In order to extend a context Γ with another variable $x : \tau$, we need to know that τ is a well-formed type in the current context Γ.
- Types obviously depends on contexts, as they are indexed over contexts.
- ...

Martin-Löf Type Theory in Martin-Löf Type Theory (cont.)

$$\mathsf{Ctxt} : \mathsf{Set} \qquad \Gamma : \mathsf{Ctxt} \Leftrightarrow \Gamma \text{ is a well-formed context}$$

$$\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set} \qquad \tau : \mathsf{Ty}(\Gamma) \Leftrightarrow \tau \text{ is a type in context } \Gamma$$

The problem is, everything seems to depend on everything else...

- In order to extend a context Γ with another variable $x : \tau$, we need to know that τ is a well-formed type in the current context Γ.
- Types obviously depends on contexts, as they are indexed over contexts.
- ...

Hence we must define $\mathsf{Ctxt} : \mathsf{Set}$, $\mathsf{Ty} : \mathsf{Ctxt} \to \mathsf{Set}$ mutually, and this is an inductive-inductive definition.

Contexts

We use de Bruijn indices, so contexts are represented as lists of the types of the free variables.
The empty context is a context:

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

If τ is a type in the current context, then we can extend it with a new variable of type τ:

$$\frac{\Gamma : \mathsf{Ctxt} \quad \tau : \mathsf{Ty}(\Gamma)}{\mathsf{cons}(\Gamma, \tau) : \mathsf{Ctxt}}$$

Example The context $x_0 : \mathbb{N}, x_1 : \mathbb{N} \to \mathbb{N}$ is represented as $\mathsf{cons}(\mathbb{N}, \mathsf{cons}(\mathbb{N} \to \mathbb{N}, \varepsilon))$.

Types

Base types such as $\mathbb{N}$ and a universe U are types in any context:

$$\frac{\Gamma : \mathsf{Ctxt}}{\mathbb{N} : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt}}{U : \mathsf{Ty}(\Gamma)}$$

The dependent function type is interesting, as the type of the codomain can depend on the variable that is to be bound:

$$\frac{\Gamma : \mathsf{Ctxt} \quad A : \mathsf{Ty}(\Gamma) \quad B : \mathsf{Ty}(\mathsf{cons}(\Gamma, A))}{\Pi(A, B) : \mathsf{Ty}(\Gamma)}$$

Example $\mathbb{N} \to \mathbb{N} = \Pi(\mathbb{N}, \mathbb{N})$ is a type in the empty context, i.e. $\Pi(\mathbb{N}, \mathbb{N}) : \mathsf{Type}(\varepsilon)$, since

- $\varepsilon : \mathsf{Ctxt}$,
- $\mathbb{N} : \mathsf{Ty}(\varepsilon)$,
- $\mathbb{N} : \mathsf{Ty}(\mathsf{cons}(\mathbb{N}, \varepsilon))$.

How can we formalise induction-induction?

Constructor form

First, note that a rule such as

$$\frac{\Gamma : \mathsf{Ctxt} \quad A : \mathsf{Ty}(\Gamma) \quad B : \mathsf{Ty}(\mathsf{cons}(\Gamma, A))}{\Pi(\Gamma, A, B) : \mathsf{Ty}(\Gamma)}$$

can equivalently be presented as a constructor

$$\Pi : \big((\Gamma : \mathsf{Ctxt}) \times (A : \mathsf{Ty}(\Gamma)) \times \mathsf{Ty}(\mathsf{cons}(\langle \Gamma, A \rangle))\big) \to \mathsf{Ty}(\Gamma) \ .$$

It is easier to reason about terms!

An axiomatisation of induction-recursion

- Induction-recursion was given a finite axiomatisation by Dybjer and Setzer.
- The main idea is to extend type theory with:
  - a datatype SP consisting of codes for sets defined by induction-recursion.
  - a function Arg which "decodes" the code from SP, giving the domain of the constructor for the set.
  - for every $\gamma : \mathrm{SP}$, a set U_γ closed under $\mathrm{Arg}(\gamma)$, i.e. there is a constructor

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

(Omitting the recursively defined $T_\gamma : U_\gamma \to D$.)

Extending the idea to induction-induction

Extending the technique used there, we can also give an axiomatisation for induction-induction.

- We will need two datatypes SP_A and SP_B, one for A and one for B.
- In addition, we will need to keep track of the elements we can refer to, so we will need parameters A_{ref} and B_{ref}.
  - For example, in

$$\mathrm{cons} : (\Gamma : \mathrm{Ctxt}) \times \mathrm{Ty}(\Gamma) \to \mathrm{Ctxt} \ ,$$

    $\mathrm{Ty}(\Gamma)$ refers to $\Gamma : \mathrm{Ctxt}$. After the first argument, A_{ref} will be extended to $A_{\mathrm{ref}} + \{\Gamma\}$.
- Also need machinery to allow constructors for B to use constructors for A.

$$\Pi : \big((\Gamma : \mathrm{Ctxt}) \times (\tau : \mathrm{Ty}(\Gamma)) \times \mathrm{Ty}(\mathrm{cons}(\langle \Gamma, \tau \rangle))\big) \to \mathrm{Ty}(\Gamma)$$

SP_{A}

Code	Represents
$\dfrac{}{\mathsf{nil}_{\mathrm{A}} : \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}})}$	$\mathbf{1}$
$\dfrac{K : \mathrm{Set} \quad \gamma : K \to \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}})}{\mathsf{nonind}(K, \gamma) : \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}})}$	$\left(k : K\right) \times \ldots$
$\dfrac{\gamma : \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}} + \mathbf{1}, B_{\mathsf{ref}})}{\text{A-ind}(\gamma) : \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}})}$	$\left(a : A\right) \times \ldots$
$\dfrac{h_{\mathsf{index}} : A_{\mathsf{ref}} \quad \gamma : \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}} + \mathbf{1})}{\text{B-ind}(h_{\mathsf{index}}, \gamma) : \mathrm{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}})}$	$\left(b : B(h_{\mathsf{index}})\right) \times \ldots$

Note Could also extend to generalised induction $\left(j : K \to A\right) \times \ldots$

$\mathrm{Arg}_{\mathrm{A}}$

More formally, "Represents" in the table will be given meaning by the decoding function $\mathrm{Arg}_{\mathrm{A}}$. The types of its arguments:

$$A_{\mathsf{ref}} : \mathsf{Set} \qquad B_{\mathsf{ref}} : \mathsf{Set} \qquad \gamma_A : \mathsf{SP}_{\mathrm{A}}(A_{\mathsf{ref}}, B_{\mathsf{ref}})$$

$$A : \mathsf{Set} \qquad B : A \to \mathsf{Set}$$

$$\mathsf{rep}_{\mathrm{A}} : A_{\mathsf{ref}} \to A \qquad \mathsf{rep}_{\mathrm{index}} : B_{\mathsf{ref}} \to A$$
$$\mathsf{rep}_{\mathrm{B}} : (x : B_{\mathsf{ref}}) \to B(\mathsf{rep}_{\mathrm{index}}(x))$$

$\mathrm{Arg_A}$; $\mathrm{nil_A}$

$$A_{\mathrm{ref}} : \mathrm{Set} \qquad B_{\mathrm{ref}} : \mathrm{Set} \qquad A : \mathrm{Set} \qquad B : A \to \mathrm{Set}$$

$$\mathrm{rep_A} : A_{\mathrm{ref}} \to A \quad \mathrm{rep_{index}} : B_{\mathrm{ref}} \to A \quad \mathrm{rep_B} : (x : B_{\mathrm{ref}}) \to B(\mathrm{rep_{index}}(x))$$

Code	Represents
$\dfrac{}{\mathrm{nil_A} : \mathrm{SP_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}})}$	$\mathbf{1}$

$$\mathrm{Arg_A}(A_{\mathrm{ref}}, B_{\mathrm{ref}}, \mathrm{nil_A}, A, B, \mathrm{rep_A}, \mathrm{rep_{index}}, \mathrm{rep_B}) = \mathbf{1}$$

Arg_A; nonind

$$A_\mathsf{ref} : \mathsf{Set} \qquad B_\mathsf{ref} : \mathsf{Set} \qquad A : \mathsf{Set} \qquad B : A \to \mathsf{Set}$$

$$\mathrm{rep}_\mathrm{A} : A_\mathsf{ref} \to A \quad \mathrm{rep}_\mathrm{index} : B_\mathsf{ref} \to A \quad \mathrm{rep}_\mathrm{B} : (x : B_\mathsf{ref}) \to B(\mathrm{rep}_\mathrm{index}(x))$$

Code	Represents
$\dfrac{K : \mathsf{Set} \quad \gamma : K \to \mathrm{SP}_\mathrm{A}(A_\mathsf{ref}, B_\mathsf{ref})}{\mathsf{nonind}(K, \gamma) : \mathrm{SP}_\mathrm{A}(A_\mathsf{ref}, B_\mathsf{ref})}$	$(k : K) \times \ldots$

$$\mathrm{Arg}_\mathrm{A}(A_\mathsf{ref}, B_\mathsf{ref}, \mathsf{nonind}(K, \gamma), A, B, \mathrm{rep}_\mathrm{A}, \mathrm{rep}_\mathrm{index}, \mathrm{rep}_\mathrm{B}) = (k : K) \times \mathrm{Arg}_\mathrm{A}(\_, \_, \gamma(k), \_, \_, \_, \_, \_)$$

Arg_A; A-ind

$$A_\mathrm{ref} : \mathrm{Set} \qquad B_\mathrm{ref} : \mathrm{Set} \qquad A : \mathrm{Set} \qquad B : A \to \mathrm{Set}$$

$$\mathrm{rep}_\mathrm{A} : A_\mathrm{ref} \to A \quad \mathrm{rep}_\mathrm{index} : B_\mathrm{ref} \to A \quad \mathrm{rep}_\mathrm{B} : (x : B_\mathrm{ref}) \to B(\mathrm{rep}_\mathrm{index}(x))$$

Code	Represents
$\dfrac{\gamma : \mathrm{SP}_\mathrm{A}(A_\mathrm{ref} + \mathbf{1}, B_\mathrm{ref})}{\text{A-ind}(\gamma) : \mathrm{SP}_\mathrm{A}(A_\mathrm{ref}, B_\mathrm{ref})}$	$(a : A) \times \ldots$

$$\mathrm{Arg}_\mathrm{A}(A_\mathrm{ref}, B_\mathrm{ref}, \text{A-ind}(\gamma), A, B, \mathrm{rep}_\mathrm{A}, \mathrm{rep}_\mathrm{index}, \mathrm{rep}_\mathrm{B}) =$$
$$(a : A) \times \mathrm{Arg}_\mathrm{A}(A_\mathrm{ref} + \mathbf{1}, \_, \gamma, \_, \_, \mathrm{rep}_\mathrm{A} \sqcup (\lambda \star .a), \_, \_)$$

Arg_A; B-ind

$$A_\mathsf{ref} : \mathsf{Set} \qquad B_\mathsf{ref} : \mathsf{Set} \qquad A : \mathsf{Set} \qquad B : A \to \mathsf{Set}$$

$$\mathrm{rep}_\mathrm{A} : A_\mathsf{ref} \to A \quad \mathrm{rep}_\mathrm{index} : B_\mathsf{ref} \to A \quad \mathrm{rep}_\mathrm{B} : (x : B_\mathsf{ref}) \to B(\mathrm{rep}_\mathrm{index}(x))$$

Code	Represents
$\dfrac{h_\mathsf{index} : A_\mathsf{ref} \quad \gamma : \mathrm{SP}_\mathrm{A}(A_\mathsf{ref}, B_\mathsf{ref} + \mathbf{1})}{\text{B-ind}(h_\mathsf{index}, \gamma) : \mathrm{SP}_\mathrm{A}(A_\mathsf{ref}, B_\mathsf{ref})}$	$\big(b : B(h_\mathsf{index})\big) \times \ldots$

$$\mathrm{Arg}_\mathrm{A}(A_\mathsf{ref}, B_\mathsf{ref}, \text{B-ind}(h_\mathsf{index}, \gamma), A, B, \mathrm{rep}_\mathrm{A}, \mathrm{rep}_\mathrm{index}, \mathrm{rep}_\mathrm{B}) =$$
$$\big(b : B(\mathrm{rep}_\mathrm{A}(h_\mathsf{index}))\big) \times$$
$$\mathrm{Arg}_\mathrm{A}(\_, B_\mathsf{ref} + \mathbf{1}, \gamma, \_, \_, \_, \mathrm{rep}_\mathrm{index} \sqcup (\lambda \star .\mathrm{rep}_\mathrm{A}(h_\mathsf{index})), \mathrm{rep}_\mathrm{B} \sqcup (\lambda \star .b))$$

An example

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

has code

$$\gamma_{\mathsf{cons}} = \text{A-ind}(\text{B-ind}(\hat{\Gamma}, \mathsf{nil}_{\mathrm{A}})) : \mathsf{SP}_{\mathrm{A}}(\mathbf{0}, \mathbf{0}) \ ,$$

where $\hat{\Gamma} = \mathsf{inr}(\star)$. We have

$$\mathsf{Arg}'_{\mathrm{A}}(\gamma_{\mathsf{cons}}, \mathsf{Ctxt}, \mathsf{Ty}) = (\Gamma : \mathsf{Ctxt}) \times \mathsf{Ty}(\Gamma) \times \mathbf{1} \ .$$

$$(\mathsf{Arg}'_{\mathrm{A}}(\gamma, A, B) \coloneqq \mathsf{Arg}_{\mathrm{A}}(\mathbf{0}, \mathbf{0}, \gamma, A, B, !_A, !_A, !_B))$$

SP_{B}, $\mathrm{Arg}_{\mathrm{B}}$ and $\mathrm{index}_{\mathrm{B}}$

SP_{B} and $\mathrm{Arg}_{\mathrm{B}}$ are similar to SP_{A} and $\mathrm{Arg}_{\mathrm{A}}$, but with some additional complexity to handle using constructors for A as indices for B, e.g.

$$\Pi : \big((\Gamma : \mathsf{Ctxt}) \times (\tau : \mathsf{Ty}(\Gamma)) \times \mathsf{Ty}(\mathsf{cons}(\langle \Gamma, \tau \rangle))\big) \to \mathsf{Ty}(\Gamma) \ .$$

We also need to keep track of the index of the type of the constructor, e.g.

$$\Pi(\Gamma, \tau, \sigma) : \mathsf{Ty}(\Gamma) \ .$$

Formation, introduction (and elimination) rules

Let $\gamma_A : \mathrm{SP}'_{\mathrm{A}}$, $\gamma_B : \mathrm{SP}'_{\mathrm{B}}(\gamma_A)$.
Formation rules:

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

Introduction rule for A_{γ_A,γ_B}:

$$\mathrm{intro}_{\mathrm{A}} : \mathrm{Arg}'_{\mathrm{A}}(\gamma_A,) \to A_{\gamma_A,\gamma_B}$$

Introduction rule for B_{γ_A,γ_B}:

$$\mathrm{intro}_{\mathrm{B}} : (b : \mathrm{Arg}'_{\mathrm{B}}(\gamma_A, \gamma_B, A_{\gamma_A,\gamma_B}, \vec{B}_{\gamma_A,\gamma_B})) \to B_{\gamma_A,\gamma_B}(\mathrm{index}_{\mathrm{B}}(\gamma_A, \gamma_B, A_{\gamma_A,\gamma_B}, \vec{B}_{\gamma_A,\gamma_B}, b))$$

Elimination rules (the ones you would expect) can also be formulated, but we omit them here.

Can we reduce induction-induction to well-known principles?

The example again

Recall our example of contexts and types again:

$$\frac{}{\varepsilon : \mathsf{Ctxt}} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \tau : \mathsf{Ty}(\Gamma)}{\mathsf{cons}(\Gamma, \tau) : \mathsf{Ctxt}}$$

$$\frac{\Gamma : \mathsf{Ctxt}}{U(\Gamma) : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad A : \mathsf{Ty}(\Gamma) \quad B : \mathsf{Ty}(\mathsf{cons}(\Gamma, A))}{\Pi(\Gamma, A, B) : \mathsf{Ty}(\Gamma)}$$

We want to replace this with an indexed inductive definition, so that we still have terms witnessing the introduction and elimination rules.

The idea

The idea is to mutually define preCtxt : Set, preTy : Set and then "goodness predicates"

$$\begin{aligned} \mathsf{goodCtxt} &: \mathsf{preCtxt} \to \mathsf{Set} \\ \mathsf{goodTy} &: \mathsf{preCtxt} \to \mathsf{preTy} \to \mathsf{Set} \end{aligned}$$

where $\mathsf{goodTy}(\Gamma, \tau)$ is meant to be the set of proofs that the pre-type τ is well-formed in the well-formed pre-context Γ.

This goes back to an (unpublished) idea by Dybjer and Setzer.

Pre-contexts and pre-types

For the pre-contexts and pre-types, we simply drop all index information:

$$\frac{}{\varepsilon : \mathsf{Ctxt}} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \tau : \mathsf{Ty}(\Gamma)}{\mathsf{cons}(\Gamma, \tau) : \mathsf{Ctxt}}$$

$$\frac{\Gamma : \mathsf{Ctxt}}{U(\Gamma) : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad A : \mathsf{Ty}(\Gamma) \quad B : \mathsf{Ty}(\mathsf{cons}(\Gamma, A))}{\Pi(\Gamma, A, B) : \mathsf{Ty}(\Gamma)}$$

becomes

$$\frac{}{\mathsf{pre}\varepsilon : \mathsf{preCtxt}} \qquad \frac{\Gamma : \mathsf{preCtxt} \quad \tau : \mathsf{preTy}}{\mathsf{precons}(\Gamma, \tau) : \mathsf{preCtxt}}$$

$$\frac{\Gamma : \mathsf{preCtxt}}{\mathsf{pre}U(\Gamma) : \mathsf{preTy}} \qquad \frac{\Gamma : \mathsf{preCtxt} \quad A : \mathsf{preTy} \quad B : \mathsf{preTy}}{\mathsf{pre}\Pi(\Gamma, A, B) : \mathsf{preTy}}$$

Good contexts

Instead, the indices come back in the goodness predicates.

$$\frac{}{\varepsilon : \mathsf{Ctxt}} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad \tau : \mathsf{Ty}(\Gamma)}{\mathsf{cons}(\Gamma, \tau) : \mathsf{Ctxt}}$$

becomes

$$\frac{}{\mathsf{good}\varepsilon : \mathsf{goodCtxt}(\mathsf{pre}\varepsilon)}$$

$$\frac{\Gamma : \mathsf{preCtxt} \quad \Gamma g : \mathsf{goodCtxt}(\Gamma) \quad \tau : \mathsf{preTy} \quad \tau g : \mathsf{goodTy}(\Gamma, \tau)}{\mathsf{goodcons}(\Gamma, \Gamma g, \tau, \tau g) : \mathsf{goodCtxt}(\mathsf{precons}(\Gamma, \tau))}$$

Good types

$$\frac{\Gamma : \mathsf{Ctxt}}{U(\Gamma) : \mathsf{Ty}(\Gamma)} \qquad \frac{\Gamma : \mathsf{Ctxt} \quad A : \mathsf{Ty}(\Gamma) \quad B : \mathsf{Ty}(\mathsf{cons}(\Gamma, A))}{\Pi(\Gamma, A, B) : \mathsf{Ty}(\Gamma)}$$

becomes

$$\frac{\Gamma : \mathsf{preCtxt} \quad \Gamma g : \mathsf{goodCtxt}(\Gamma)}{\mathsf{good}U(\Gamma, \Gamma g) : \mathsf{goodTy}(\Gamma, \mathsf{pre}U(\Gamma))}$$

$$\frac{\begin{array}{ccc}\Gamma : \mathsf{preCtxt} & A : \mathsf{preTy} & B : \mathsf{preTy} \\ \Gamma g : \mathsf{goodCtxt}(\Gamma) & Ag : \mathsf{goodTy}(\Gamma, A) & Bg : \mathsf{goodTy}(\mathsf{precons}(\Gamma, A), B)\end{array}}{\mathsf{good}\Pi(\Gamma, \Gamma g, A, Ag, B, Bg) : \mathsf{goodTy}(\Gamma, \mathsf{pre}\Pi(\Gamma, A, B))}$$

Formation rules

We define

$$[\![\mathsf{Ctxt}]\!] \coloneqq (\Gamma : \mathsf{preCtxt}) \times \mathsf{goodCtxt}(\Gamma)$$

$$[\![\mathsf{Ty}]\!](x) \coloneqq (\tau : \mathsf{preTy}) \times \mathsf{goodTy}(\mathsf{fst}(x), \tau)$$

This obviously satisfies the formation rules

$$[\![\mathsf{Ctxt}]\!] : \mathsf{Set} \qquad [\![\mathsf{Ty}]\!] : [\![\mathsf{Ctxt}]\!] \to \mathsf{Set} \ .$$

Introduction rules

The defined constructors will simply assemble goodness proofs from its parts:

$$
\begin{aligned}
[\![\varepsilon]\!] &: [\![\mathsf{Ctxt}]\!] \\
[\![\varepsilon]\!] &= \langle \mathsf{pre}\varepsilon, \mathsf{good}\varepsilon \rangle \\
[\![\mathsf{cons}]\!] &: (\Gamma : [\![\mathsf{Ctxt}]\!]) \to [\![\mathsf{Ty}]\!](\Gamma) \to [\![\mathsf{Ctxt}]\!] \\
[\![\mathsf{cons}]\!](\Gamma, \tau) &= \langle \mathsf{precons}(\mathsf{fst}(\Gamma), \mathsf{fst}(\tau)), \\
&\qquad \mathsf{goodcons}(\mathsf{fst}(\Gamma), \mathsf{snd}(\Gamma), \mathsf{fst}(\tau), \mathsf{snd}(\tau)) \rangle \\
[\![U]\!] &: (\Gamma : [\![\mathsf{Ctxt}]\!]) \to [\![\mathsf{Ty}]\!](\Gamma) \\
[\![U]\!](\Gamma) &= \langle \mathsf{pre}U(\mathsf{fst}(\Gamma)), \mathsf{good}U(\mathsf{fst}(\Gamma), \mathsf{snd}(\Gamma)) \rangle \\
[\![\Pi]\!] &: (\Gamma : [\![\mathsf{Ctxt}]\!]) \to (A : [\![\mathsf{Ty}]\!](\Gamma)) \to [\![\mathsf{Ty}]\!](\mathsf{cons}(\Gamma, A)) \to [\![\mathsf{Ty}]\!](\Gamma) \\
[\![\Pi]\!](\Gamma, A, B) &= \langle \mathsf{pre}\Pi(\mathsf{fst}(\Gamma), \mathsf{fst}(A), \mathsf{fst}(B)), \\
&\qquad \mathsf{good}\Pi(\mathsf{fst}(\Gamma), \mathsf{snd}(\Gamma), \mathsf{fst}(A), \mathsf{snd}(A), \mathsf{fst}(B), \mathsf{snd}(B)) \rangle
\end{aligned}
$$

Work in progress: elimination and computation rules

Problem In order to use the base case and step cases the elimination principle gives us, we need that all goodness proofs are equal.

Solution This can be proven with the elimination rules for indexed inductive definitions.

Problem This lets us define the eliminators. However, now the computation rules only hold up to propositional equality.

Possible solution If we work in a type theory with a propositional universe, this could be avoided.

Problem ???

Summary

Summary

- Induction-induction is an induction principle which allows us to define $A : \text{Set}$, together with $B : A \to \text{Set}$, where:
  - Both A and $B(a)$ for $a : A$ are inductively defined.
  - The constructors for A can refer to B and vice versa.
- Natural to use when modelling type theory inside type theory.
- Axiomatisation by defining a universe of codes.
- Work in progress: Reduction to indexed inductive definitions.