Inductive Inductive definitions without UIP

HUGUNIN Jasper

東京工業大学

TPP 2017

Outline

1. Inductive types
   - What are Inductive types?
   - Simulating Indexed Inductive types by Inductive types.
2. Inductive Inductive Types
   - What are Inductive Inductive types?
   - Simulating Inductive Inductive types with UIP.
   - Simulating Inductive Inductive types without UIP (In progress).
3. Coinductive types
   - What are Coinductive types?
   - Simulating Coinductive types.

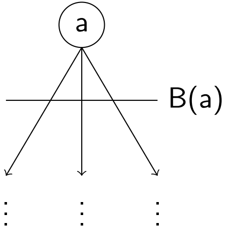
Outline

1. Inductive types
   - What are Inductive types?
   - Simulating Indexed Inductive types by Inductive types.
2. Inductive Inductive Types
   - What are Inductive Inductive types?
   - Simulating Inductive Inductive types with UIP.
   - Simulating Inductive Inductive types without UIP (In progress).
3. Coinductive types
   - What are Coinductive types?
   - Simulating Coinductive types.

What are Inductive types?

- Natural-numbers, binary trees, syntax with operations, etc.
- Well-founded trees
- Least fixed point of a polynomial functor $F(X) = \sum_{a:A} X^{B(a)}$

```
Inductive term :=
  | atom (i : nat) : term
  | and (s : term) (t : term) : term
  | not (t : term) : term
.
```



What are Indexed Inductive types

Indexed Inductive types

- We associate a label $i : I$ to each node,
  with the label calculated from the data ($A \to I$).
- Each child expects a specific label ($\forall a. Ba \to I$)
- We only allow trees where the expected and actual labels agree.

- We can simulate mutual inductive definitions
  by labeling nodes as type 1 or type 2.
- The above definition suggests a way to simulate
  Indexed Inductive types using the equality type
  (which in Coq is a simple example of an Indexed Inductive type).

Example of an Indexed Inductive type

```
Inductive type : Type :=
  | N : type
  | function_type (A : type) (B : type) : type
.
Notation "( A --> B )" := (function_type A B).
Inductive term : type -> Type :=
  | literal (n : nat) : term N
  | sum : term (N --> (N --> N))
  | app A B : term (A --> B) -> term A -> term B
.
```

Simulating Indexed Inductive types

Definition

1. Start with an unlabeled tree.
2. Define a tree to be well-labeled (for label i) if
   - the computed label is equal to i and
   - all the children are well-labeled (for their expected label)
3. Define your Indexed Inductive type to consist of well-labeled trees.

This definition suffices to define the corresponding eliminator, and as long as the equality type eliminator computes definitionally on reflexivity, the resulting type has the expected definitional behavior.

Outline

What are Inductive Inductive types?

We define the type of labels at the same time as the type they label.
An example: types and contexts in a simple dependent type theory

```
InductiveInductive Ctx : Type :=
  | emp : Ctx
  | app (G : Ctx) (A : Ty G) : Ctx
with Ty : Ctx -> Type :=
  | iota (G : Ctx) : Ty G
  | N : Ty emp
  | pi (G : Ctx) (A : Ty G) (B : Ty (app G A)) : Ty G
.
```

Simulating Inductive Inductive types with UIP

We can proceed the same way we did with Indexed Inductive types.

1. Start by dropping the label information, defining pre-contexts and pre-types.
2. Define `dep : preTy -> preCtx` that calculates the label for a type.
3. Recursively add constraints $G = \text{dep}(A)$ wherever we have $A : \text{Ty}(G)$.
4. Define contexts to be recursively well-labeled pre-contexts.
5. Define types in context G to be recursively well-labeled pre-types A, along with a proof that the pre-context part of G equals $\text{dep}(A)$.

Discussion of this approach to simulating Inductive Inductive types

- This is the approach given by Forsberg in his thesis
  where he proposes Inductive Inductive types.
- It suffices to define the introduction rules
  and (restricted) elimination rules in extensional type theory
  or intensional type theory with UIP (uniqueness of identity proofs).
- What about homotopy type theory (which is incompatible with UIP)?
  What do we need UIP for?
- This encoding has poor definitional behavior
  and only weak eliminators.
  Is that necessary?

Use of UIP in simulating Inductive Inductive types

- In defining the eliminator, Forsberg first proves that there is at most one proof of being well-labeled for each node.
- For the equality proofs, you use UIP to show that they are unique.
- This is different from how you would define the eliminator for the simulated Indexed Inductive definitions above.
- There, you could transport along the equality from the computed label to the actual label.
- But here, we only have a proof that the pre-context parts are equal, without considering the proof that that pre-context is well-labeled. So you need those proofs to be unique.

Example

- Consider appending to a context, given pre-context Γ and pre-type A, $\mathsf{preapp}(\Gamma, A)$.
- We add a constraint $p_1 : \Gamma = \mathsf{dep}(A)$, to say that this is a valid context.
- But Γ is actually $\mathsf{preapp}(\Delta, B)$, and we also add $p_2 : \Delta = \mathsf{dep}(B)$.
- On the other side, we have that A is recursively well-labeled, from which we should be able to extract a proof that Γ is well-labeled.
- So we have $\mathsf{dep}_1(A) : \Delta = \mathsf{dep}(B)$.
- But we have no reason to believe that $p_2 = \mathsf{dep}_1(A)$.

$1, 2, 3, \ldots$

So let's just add that in as a constraint as well!

- Define 2nd-well-labeledness, to be where the proofs of 1st-well-labeledness for G and $\text{dep}(A)$ are equal.
- Of course, now we need our proofs of 2nd-well-labeledness to agree, so define 3rd-well-labeledness.
- Of course this continues to infinity, but not past it.

$1, 2, 3, \ldots, \omega$

- Defining ω-well-labeledness to be n-well-labeled for all n, we have a proof that the n-well-labeledness proofs agree for all n, so since (with function extensionality) two functions are equal when they are equal on all inputs,
  the proofs of ω-well-labeledness should also agree.
- Because the definition of $(n+1)$-well-labeledness depends on n-well-labeledness, this is more complicated than it sounds.
  I do not yet have a formal proof of this result.

Recursive Recursive definitions

In Forsberg's thesis, he is prevented from defining stronger versions of the eliminators by a lack of Recursive Recursive definitions. (mutual recursive definitions where the second function's type depends on the first.)
One example of such a Recursive Recursive definition:

```
well-labeled-context : preCtx -> Type
well-labeled-type : forall G, well-labeled-context G -> preTy
well-labeled-context (preapp G A) =
  (Gg : well-labeled-context G) &
  well-labeled-type G Gg A
```

But we claim to have defined one such function,
that computes up to equivalence (propositional equality in HoTT)
Perhaps we can leverage similar techniques to define the strong eliminators.

$1, (1 + 2), (1 + 2 + 3), \ldots$

Definition (ω-well-labeledness)

1. A function f that gives for each n
   a proof of i-well-labeledness for $i \leq n$.
2. For each n, a proof that forgetting the proof
   of $(n + 1)$-well-labeledness in $f(n + 1)$ is equal to $f(n)$

Thus we end up with ω proofs of i-well-labeledness for each i,
but we also have proofs that each is equal to the next.
In the introduction rules, we can take these equalities to be reflexivity.

n times composing reflexivity is not reflexivity

But... In the eliminator, we need to take some properties of one proof of *i*-well-labeledness, and transport it to all the others. The only way to do so is to go step by step up the chain. However,

```
(fix comp n : a = a :=
 match n with
 | O => eq_refl
 | S m => -- eq_trans (comp m) eq_refl --
          comp m
 end) n
```

Is not definitionally equal to reflexivity. It is blocked on recursion on *n*. It is however propositionally equal to reflexivity,
so with function extensionality we can prove comp is a constant function.

Outline

1. Inductive types
   - What are Inductive types?
   - Simulating Indexed Inductive types by Inductive types.
2. Inductive Inductive Types
   - What are Inductive Inductive types?
   - Simulating Inductive Inductive types with UIP.
   - Simulating Inductive Inductive types without UIP (In progress).
3. Coinductive types
   - What are Coinductive types?
   - Simulating Coinductive types.

What are Coinductive types?

- Dual to Inductive types
- Streams, automata, etc.
- Possibly infinite trees
- Greatest fixed point of polynomial functors $FX = \sum_{a:A} X^{B(a)}$
- For $\pi : FA \to A$ with $\pi(a, c) = a$, the limit of the chain

$$A \xleftarrow{\pi} FA \xleftarrow{F\pi} F^2 A \xleftarrow{F^2\pi} \cdots$$

Simulating Coinductive types in Coq

Definition (Coinductive type M)

1. Take a function f with $f(n) : F^n A$.
2. Require $f(n) = F^n \pi(f(n+1))$.

We have a function $M \to FM$ where

$$(f, p) \mapsto (f(0), b \mapsto (n \mapsto f(n+1).2\ b_n, \\ n \mapsto p(n+1).2\ b_n))$$

For b_0 being the transport of b across $p(0)$ from $B(f(0))$ to $B(f(1).1)$
and b_{n+1} being the transport of b_n across $p(n+1).1$
from $B(f(n).1)$ to $B(f(n+1).1)$.
For the introduction rule, we can take $p(n).1$ to be reflexivity,
but we don't get b_n computes to b.