Implementation of Type Theory based on Dependent Inductive and Coinductive Types

Florian Engel

November 29, 2020

Inductive Types in Coq

- defined over their constructors
- each constructor has to give back the defined type

Examples

- Booleans

```
Inductive bool : Set :=
| True : bool
| False : bool.
```

- Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Destructing Inductive Types in Coq

- also called recursion
- pattern matches on the constructor
- gives back values of same type in each match

Examples

- negation

```
Definition neg (b : bool) :=
  match b with
  | True => False
  | False => True
  end.
```

- isZero

```
Definition isZero (n : nat) :=
  match n with
  | O => True
  | S _ => False
  end.
```

Coinductive Types

- positive coinductive types
- negative coinductive types

Positive Coinductive Types

- treats recursive occurrence like a value
- otherwise like inductive types
- functions which produce such types have to be productive

Example

- Stream in coq

```
CoInductive Stream (A : Set) : Set :=
  Cons : A -> Stream A -> Stream A.
```

- repeat function

```
CoFixpoint repeat (A : Set) (x : A) : Stream A :=
  Cons A x (repeat A x).
```

What is wrong about Positive Coinductive Types

- Symmetry with inductive types not clear
- Breaks subject reduction
  - Subject reduction: types are preserved after reduction

Ourys Counterexample

```
CoInductive U : Set := In : U -> U.

CoFixpoint u : U := In u.

Definition force (x: U) : U :=
  match x with
    In y => In y
  end.

Compute u.
> cofix Fcofix : U := In Fcofix : U

Compute (force u).
> In (cofix Fcofix : U := In Fcofix) : U

Definition eq (x : U) : x = force x :=
  match x with
    In y => eq_refl
  end.

Definition eq_u : u = In u := eq u
```

Negative Coinductive Types

- defined over their destructors
- functions use copattern matching

Examples

- Stream

```
CoInductive Stream (A : Set) : Set :=
  Seq { hd : A; tl : Stream A }.
```

- repeat function

```
CoFixpoint repeat (A : Set) (x : A) : Stream A :=
  {| hd := x; tl := repeat A x|}.
```

Type Theory Based on Dependent Inductive and Coinductive Types

- inductive types: $\mu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A})$
- coinductive types $\nu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A})$
- constructors: α_i^{μ}
- destructors: ξ_i^{μ}
- recursion: rec $\overrightarrow{(\Gamma_k, y_k).g_k}$
- corecursion: corec $\overrightarrow{(\Gamma_k, y_k).g_k}$

Symmetry betweeen Inductive and Coinductive Types

$$\text{Product A B} = \mu(X : *; (()); \top)$$

$$\Gamma_1 = (x : A, y : B)$$

```
data Product⟨A : Set,B : Set⟩ : Set where
   MkProduct : (x : A, x : B) → Unit → Product
fst⟨A : Set, B : Set⟩ =
  rec Product<A,B> to A where
     { MkProduct x y u = x }
snd⟨A : Set, B : Set⟩ =
  rec Product<A,B> to B where
     { MkProduct x y u = y }
```

$$\text{Product A B} = \nu(X : *; ((), ()); (A, B))$$

$$\Gamma_1 = \Gamma_2 = \emptyset$$

```
codata Product⟨A : Set, B : Set⟩ : Set where
   Fst : Product → A
   Snd : Product → B
mkProduct⟨A : Set, B : Set⟩ (x:A, y:B) =
  corec Unit to Product<A,B> where
    { Fst u = x
    ; Snd u = y } @ ◊
```

Dependent Coinductive Types

- Partial streams which depend on their definition depth

$$\text{PStr A} = \nu(X : (k : Conat) \to *; ((succ@k), (succ@k)); (A, X@k))$$

$$\Gamma_1 = \Gamma_2 = (k : Conat)$$

```
codata PStr⟨A : Set⟩ : (n : Conat) → Set where
  Hd : (k : Conat) → PStr (succ @ k) → A
  Tl : (k : Conat) → PStr (succ @ k) → PStr @ k
```

- Dependent functions

$$\text{Pi A B} = \nu(X : *; (()); (B@x))$$

$$\Gamma_1 = (x : A)$$

```
codata Pi⟨A : Set, B : (x : A) → Set⟩ : Set where
  Inst : (x : A) → Pi → B @ x
```

Demo

Other Topics in the Thesis

- Comparison with Agda
- Termination and productivity checking with sized types
- Difference between paper and implementation
  - Rules rewritten to syntax directed one
  - Added type "polymorphism"
  - De-Brujin indexes