Implementation of Type Theory based on dependent Inductive and Coinductive Types

Florian Engel

November 24, 2020

Inductive types

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

Examples

- Booleans

```
data Bool : Set where
  True : Unit → Bool
  False : Unit → Bool
```

- Natural numbers

```
data Nat : Set where
  Zero : Unit → Nat
  Succ : Nat → Nat
```

Destructing inductive types

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

Examples

- negation

```
rec Bool to Bool where
  True u = False @ ◊
  False u = True @ ◊
```

- isZero

```
rec Nat to Bool where
  Zero u = True @ ◊
  Succ n = False @ ◊
```

postive coinductive types

- treats recursive occurence 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 positve coinductive types

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

Ourys Example

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

CoFixpoint u : U := In u.

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

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

```
codata Stream⟨A : Set⟩ : Set where
  Hd : Stream ⇾ A
  Tl : Stream ⇾ Stream
```

- repeat function

```
repeat⟨A : Set⟩(x : A) =
  corec Unit to Stream⟨A⟩ where
    { Hd s = x
    ; Tl s = ◊ } @ ◊
```

Symmetry with inductive types

```
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 where
    { Fst u → x
    ; Snd u → y } @ ◊

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

Type Theory based on dependent Inductive and Coinductive Types

- kinds: $(x_1 : A_1, \ldots, x_n : A_n) \multimap *$
- types: $(x_1 : A_1, \ldots, x_n : A_n) \multimap B$
- lambda abstraction: $(x).A$
- type application: $A@t$
- term application: $t@s$
- inductive types: $\mu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A})$
- coinductive types $\nu(X : \Gamma \multimap *; \vec{\sigma}; \vec{A})$
- recursion: rec $\overrightarrow{(\Gamma_k, y_k).g_k}$
- corecursion: corec $\overrightarrow{(\Gamma_k, y_k).g_k}$

Dependent coinductive types

- Partial streams whitch depend on their defintion depth

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

- Dependent funcitions

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

Demo

Other topics in the thesis

- Termination and productivity checking with sized types
- Implementation details
  - Rules rewritting
  - De-Brujin indexes