Verification of Functional Programs
Co-Induction

Andrés Sicard-Ramírez

EAFIT University

Semester 2014-1

Non-Well-Founded Sets

Axiom of foundation (ZFC)

All sets are well-founded.

Non-Well-Founded Sets

Axiom of foundation (ZFC)

All sets are well-founded.

Theorem

A set X is well-founded iff there is no sequence $\langle X_n \mid n \in \mathbb{N} \rangle$ such that $X_0 = X$ and $X_{x+1} \in X_n$ for all $n \in \mathbb{N}$ [Hrbacek and Jech 1999, Theorem 2.4, p. 256].

Definition

A set X is **non-well-founded** iff there is an infinite sequence $X_1, X_2, \dots$ such that X_{n+1} is a member of X_n, for all $n \in \mathbb{N}$ [Milner and Tofte 1991, p. 209].

Co-Inductive Types

Description

'The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, such objects contain only a finite number of constructors. Co-inductive types arise from relaxing this condition, and admitting types whose objects contain an infinity of constructors.' [The Coq Development Team 2016, § 1.3.3].

Co-Inductive Types

Description

'The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, such objects contain only a finite number of constructors. Co-inductive types arise from relaxing this condition, and admitting types whose objects contain an infinity of constructors.' [The Coq Development Team 2016, § 1.3.3].

Remark

Potentially infinity of constructors.

Co-Inductive Types

Example (Haskell)

The canonical example of an co-inductive data type are streams.

```
data Stream a = Cons a (Stream a)
```

Co-Inductive Types

Example (Haskell)

The canonical example of an co-inductive data type are streams.

```
data Stream a = Cons a (Stream a)
data Nat = Z | S Nat

zeros :: Stream Nat
zeros = Cons Z zeros
```

Co-Inductive Types

Example (Haskell)

The canonical example of an co-inductive data type are streams.

```haskell
data Stream a = Cons a (Stream a)
data Nat = Z | S Nat

zeros :: Stream Nat
zeros = Cons Z zeros
```

Remark

Haskell's **data** keyword defines both inductive and co-inductive data types. That is not a good idea!

Co-Inductive Types

Remark

The `Set Implicit Arguments` command can be used in Coq for handling the implicit arguments.

Co-Inductive Types

Remark

The `Set Implicit Arguments` command can be used in Coq for handling the implicit arguments.

Example (Coq)

```
Require Import Unicode.Utf8.

Set Implicit Arguments.

CoInductive Stream (A : Type) : Type :=
  cons : A → Stream A → Stream A.

CoFixpoint zeros : Stream nat := cons 0 zeros.
```

Co-Inductive Types

Example (cont.)

```
Notation "x :: xs" :=
  (cons x xs) (at level 60, right associativity).

CoFixpoint zeros : Stream nat := 0 :: zeros.
```

Co-Inductive Types

Example (cont.)

```
Notation "x :: xs" :=
  (cons x xs) (at level 60, right associativity).
CoFixpoint zeros : Stream nat := 0 :: zeros.
```

Remark

We will continue using Coq for the examples related to co-induction.

Co-Inductive Types

Example (co-inductive natural numbers)

Intuition: $\mathrm{Co}\mathbb{N} = \mathbb{N} \cup \{\infty\}$

```
Require Import Unicode.Utf8.

CoInductive Conat : Set :=
| cozero : Conat
| cosucc : Conat → Conat.

CoFixpoint inf : Conat := cosucc inf.
```

Definition

Let D be a set, let $(D, \sqsubseteq)$ be a poset and let f be a function $f : D \to D$. An element $d \in D$ is a **post-fixed point** of f iff

$$d \sqsubseteq f(d).$$

Let D be a set, $(D, \sqsubseteq)$ be a poset and f be a function $f : D \to D$.

Definition (Greatest post-fixed point)

The greatest post-fixed of f is greatest among the post-fixed points of f. That is, $d \in D$ is the greatest post-fixed point of f iff:

- $d \sqsubseteq f(d)$ and
- $\forall x.\, x \sqsubseteq f(x) \Rightarrow x \sqsubseteq d$.

Let D be a set, $(D, \sqsubseteq)$ be a poset and f be a function $f : D \to D$.

Definition (Greatest post-fixed point)

The greatest post-fixed of f is greatest among the post-fixed points of f. That is, $d \in D$ is the greatest post-fixed point of f iff:

- $d \sqsubseteq f(d)$ and
- $\forall x.\, x \sqsubseteq f(x) \Rightarrow x \sqsubseteq d$.

Theorem

If $d \in D$ is the greatest post-fixed point of f, then d is the greatest fixed-point of f [Ésik 2009, Proposition 2.1].

Co-Inductive Types

Remark

The inductive/co-inductive types can be defined/represented as least/greatest fixed-points of appropriated functions (functors).

Recall that the least and greatest fixed-points of a unary function f are denoted by $\mu x.f(x)$ and $\nu x.f(x)$, respectively.

Example

Let 1 be the unity type, and $+$ and $\times$ be the operators for disjoint union and Cartesian product, respectively. Then

$$\mathsf{Nat} := \mu X.1 + X, \qquad \mathsf{Conat} := \nu X.1 + X,$$

$$\mathsf{List}\ A := \mu X.1 + (A \times X), \qquad \mathsf{Colist}\ A := \nu X.1 + (A \times X),$$

$$\mathsf{Stream}\ A := \nu X.A \times X.$$

Co-Inductive Types

Remark

'Due to the coincidence of least and greatest fixed-point types [Smyth and Plotkin 1982] in lazy languages such as Haskell, the distinction between inductive and coinductive types is blurred in partial functional programming.' [Abel 2014, p. 148]

Co-Recursive Functions Guarded by Constructors

Definition

Recursion function: functions from an inductive type

Co-recursive function: functions into an co-inductive type

Co-Recursive Functions Guarded by Constructors

Definition

Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

'we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation.' [Gibbons and Hutton 2005, p. 1]

Co-Recursive Functions Guarded by Constructors

Definition

Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

'we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation.' [Gibbons and Hutton 2005, p. 1]

'we use the term corecursive program for a function whose range is a type defined recursively as the greatest solution of some equation.' [Gibbons and Hutton 2005, p. 1]

Co-Recursive Functions Guarded by Constructors

Definition

Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

'we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation.' [Gibbons and Hutton 2005, p. 1]

'we use the term corecursive program for a function whose range is a type defined recursively as the greatest solution of some equation.' [Gibbons and Hutton 2005, p. 1]

Remark

Alternative names for co-recursion could be 'non-wellfounded recursion' or 'baseless recursion' [Moss and Danner 1997].

Co-Recursive Functions Guarded by Constructors

Condition

‘Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them.’ [Giménez 1995, p. 51]

Co-Recursive Functions Guarded by Constructors

Condition

'Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them.' [Giménez 1995, p. 51]

Example

```
CoFixpoint from (n : nat) : Stream nat := n :: from (S n).
```

Co-Recursive Functions Guarded by Constructors

Condition

'Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them.' [Giménez 1995, p. 51]

Example

```
CoFixpoint from (n : nat) : Stream nat := n :: from (S n).
```

Example

```
CoFixpoint alter : Stream bool := true :: false :: alter.
```

Co-Recursive Functions Guarded by Constructors

Example (counterexample)

```
CoFixpoint
  filter (A : Type)(P : A → bool)(xs : Stream A) : Stream A :=
match xs with x' :: xs' =>
  if P x' then x' :: filter P xs' else filter P xs'
end.
```

The `filter` function is not guarded by constructors because there is not constructor to guard the recursive call in the `else` branch.

Co-Recursive Functions Guarded by Constructors

Auxiliary definition

```
Definition tail (A : Type)(xs : Stream A) : Stream A :=
match xs with _ :: xs' => xs' end.
```

Co-Recursive Functions Guarded by Constructors

Auxiliary definition

```
Definition tail (A : Type)(xs : Stream A) : Stream A :=
match xs with _ :: xs' => xs' end.
```

Example (counterexample)

```
CoFixpoint zeros : Stream nat := 0 :: tail zeros.
```

The `zeros` function is not guarded by constructors because there is a function (`tail`) applied to the recursive call which is not a constructor.

Co-Recursive Functions Guarded by Constructors

Example

From `nat` to `Conat` (recursive version).

```
Fixpoint nat2conat (n : nat) : Conat :=
  match n with
    | O    => cozero
    | S n' => cosucc (nat2conat n')
  end.
```

Co-Recursive Functions Guarded by Constructors

Example

From nat to Conat (recursive version).

```
Fixpoint nat2conat (n : nat) : Conat :=
  match n with
    | O    => cozero
    | S n' => cosucc (nat2conat n')
  end.
```

From nat to Conat (co-recursive version).

```
CoFixpoint nat2conat (n : nat) : Conat :=
  match n with
    | O    => cozero
    | S n' => cosucc (nat2conat n')
  end.
```

Equality

Suitable notions of equality between potentially infinite terms can be defined as binary co-inductive relations.

Equality

Suitable notions of equality between potentially infinite terms can be defined as binary co-inductive relations.

Auxiliary definition

```
Definition head (A : Type)(xs : Stream A) : A :=
match xs with x' :: _ => x' end.
```

Example (equality on streams)

The equality between streams is defined by the co-inductive bisimilarity relation [Turner 1995].

```
CoInductive EqStream (A : Type) : Stream A → Stream A → Prop :=
  eqS : ∀ xs ys : Stream A,
        head xs = head ys →
        EqStream (tail xs) (tail ys) →
        EqStream xs ys.
```

Example (equality on streams)

The equality between streams is defined by the co-inductive bisimilarity relation [Turner 1995].

```
CoInductive EqStream (A : Type) : Stream A → Stream A → Prop :=
  eqS : ∀ xs ys : Stream A,
        head xs = head ys →
        EqStream (tail xs) (tail ys) →
        EqStream xs ys.
Notation "xs ≈ ys" :=
  (EqStream xs ys) (at level 70, no associativity).
```

Co-Induction Principle

Co-induction principle, greatest fixed-point induction or Park's rule

Let $F(X)$ be a functor, then

$$\forall X.X \sqsubseteq F(X) \Rightarrow X \sqsubseteq \nu X.F(X)$$

is the co-induction principle associated to $F(X)$ [Dybjer and Sander 1989; Giménez and Casterán 2007].

Co-Induction Principle

Example (co-induction principle associated to $\approx$)

The functor (bisimulation):

$$F(X, xs, ys) := \mathsf{head}\ xs = \mathsf{head}\ ys \wedge X(\mathsf{tail}\ xs, \mathsf{tail}\ ys)$$

Co-Induction Principle

Example (co-induction principle associated to $\approx$)

The functor (bisimulation):

$$F(X, xs, ys) := \mathsf{head}\ xs = \mathsf{head}\ ys \wedge X(\mathsf{tail}\ xs, \mathsf{tail}\ ys)$$

The co-induction principle:

$$\forall X.(\forall xs\, \forall ys. X(xs, ys) \Rightarrow F(X, xs, ys)) \Rightarrow \forall xs\, \forall ys. X(xs, ys) \Rightarrow \nu X.F(X, xs, ys)$$

Example (co-induction principle associated to $\approx$)

The functor (bisimulation):

$$F(X, xs, ys) := \text{head } xs = \text{head } ys \wedge X(\text{tail } xs, \text{tail } ys)$$

The co-induction principle:

$$\forall X.(\forall xs\, \forall ys. X(xs, ys) \Rightarrow F(X, xs, ys)) \Rightarrow \forall xs\, \forall ys. X(xs, ys) \Rightarrow \nu X. F(X, xs, ys)$$

The Coq type:

```
co_ind : ∀ A : Type, ∀ R : Stream A → Stream A → Prop,
         (∀ xs ys : Stream A, R xs ys →
           head xs = head ys ∧ R (tail xs) (tail ys)) →
         ∀ xs ys : Stream A, R xs ys → xs ≈ ys
```

Co-Induction Principle

Example (the map-iterate property)

The property states that [Gibbons and Hutton 2005; Giménez and Casterán 2007]

```
map f (iterate f x) ≈ iterate f (f x).
```

where

Co-Induction Principle

Example (the map-iterate property)

The property states that [Gibbons and Hutton 2005; Giménez and Casterán 2007]

```
map f (iterate f x) ≈ iterate f (f x).
```

where

```
CoFixpoint
  map (A B : Type)(f : A → B)(xs : Stream A) : Stream B:=
  match xs with x' :: xs' => f x' :: map f xs' end.
CoFixpoint iterate (A : Type)(f : A → A)(a : A) : Stream A :=
  a :: iterate f (f a).
```

Co-Induction Principle

Example (the map-iterate property)

The property states that [Gibbons and Hutton 2005; Giménez and Casterán 2007]

```
map f (iterate f x) ≈ iterate f (f x).
```

where

```
CoFixpoint
  map (A B : Type)(f : A → B)(xs : Stream A) : Stream B:=
  match xs with x' :: xs' => f x' :: map f xs' end.
CoFixpoint iterate (A : Type)(f : A → A)(a : A) : Stream A :=
  a :: iterate f (f a).
```

See the proof in the source code in the course web page.

References

Abel, Andreas (2014). Programming and Reasoning with Infinite Structures Using Copatterns and Sized Types. In: Software Engineering Workshops 2014 (SE-WS 2014). Ed. by Schmid, Klaus, Böhm, Wolfgang, Heinrich, Robert, Herrmann, Andrea, Hoffmann, Anne, Landes, Dieter, Konersmann, Marco, Ruhroth, Thomas, Sander, Oliver, Stolz, Volker, Trancón-Widemann, Baltasar and Weißbach, Rüdiger. Vol. 1129. CEUR Workshop Proceedings. CEUR-WS.org, pp. 148–150 (cit. on p. 19).

Dybjer, Peter and Sander, Herbert P. (1989). A Functional Programming Approach to the Specification and Verification of Concurrent Systems. Formal Aspects of Computing 1, pp. 303–319 (cit. on p. 36).

Ésik, Zoltán (2009). Fixed Point Theory. In: Handbook of Weighted Automata. Ed. by Droste, Manfred, Kuich, Werner and Vogler, Heiko. Monographs in Theoretical Computer Science. An EATCS Series. Springer. Chap. 2 (cit. on pp. 15, 16).

Gibbons, Jeremy and Hutton, Graham (2005). Proof Methods for Corecursive Programs. Fundamenta Informaticae XX, pp. 1–14 (cit. on pp. 20–23, 40–42).

Giménez, Eduardo (1995). Codifying Guarded Definitions with Recursive Schemes. In: Types for Proofs and Programs (TYPES 1994). Ed. by Dybjer, Peter, Nordström, Bengt and Smith, Jan. Vol. 996. Lecture Notes in Computer Science. Springer, pp. 39–59 (cit. on pp. 24–26).

References

Giménez, Eduardo and Casterán, Pierre (2007). A Tutorial on [Co-]Inductive Types in Coq. URL: http://coq.inria.fr/documentation (visited on 29/07/2014) (cit. on pp. 36, 40–42).

Hrbacek, Karel and Jech, Thomas (1999). Introduction to Set Theory. Third Edition, Revised and Expanded. Marcel Dekker (cit. on pp. 2, 3).

Milner, Robin and Tofte, Mads (1991). Co-induction in Relational Semantics. Theoretical Computer Science 87.1, pp. 209–220. DOI: 10.1016/0304-3975(91)90033-X (cit. on pp. 2, 3).

Moss, Lawrence S. and Danner, Norman (1997). On the Foundation of Corecursion. Logic Journal of the IGPL 5.2, pp. 231–257 (cit. on pp. 20–23).

Smyth, M. B. and Plotkin, G. D. (1982). The Category-Theoretic Solution of Recursive Domain Equations. SIAM Journal on Computing 11.4, pp. 761–783 (cit. on p. 19).

The Coq Development Team (2016). The Coq Proof Assistant. Reference Manual. Version 8.5pl2. (Cit. on pp. 4, 5).

Turner, D. A. (1995). Elementary Strong Functional Programming. In: Functional Programming Languages in Education (FPLE 1995). Ed. by Hartel, Pieter H. and Plasmeijer, Rinus. Vol. 1022. Lecture Notes in Computer Science. Springer, pp. 1–13 (cit. on pp. 34, 35).