Co-inductive predicates and bisimilarity

Coq'Art section 13.6–13.7

Koen Timmermans and Marnix Suilen

Definitions

Recall the definition of `LList`:

```
Set Implicit Arguments.

CoInductive LList (A:Set) : Set :=
   LNil : LList A |
   LCons : A -> LList A -> LList A.

Implicit Arguments LNil [A].
```

Definitions

Recall the definition of `LList`:

```
Set Implicit Arguments.

CoInductive LList (A:Set) : Set :=
   LNil : LList A |
   LCons : A -> LList A -> LList A.

Implicit Arguments LNil [A].
```

And the definition of `from`:

```
CoFixpoint from (n:nat) : LList nat := LCons n (from (S n)).
```

Definitions

Recall the definition of `LList`:

```
Set Implicit Arguments.

CoInductive LList (A:Set) : Set :=
   LNil : LList A |
   LCons : A -> LList A -> LList A.

Implicit Arguments LNil [A].
```

And the definition of `from`:

```
CoFixpoint from (n:nat) : LList nat := LCons n (from (S n)).
```

And of `repeat`:

```
CoFixpoint repeat (A:Set)(a:A) : LList A := LCons a (repeat a).
```

Recall from_unfold

```
Lemma from_unfold: forall n:nat, from n = LCons n (from (S n)).
Proof.
intro n.
LList_unfold (from n).
simpl.
reflexivity.
Qed.
```

Recall Guard conditions

A definition by `cofixpoint` is only accepted if all recursive calls occur inside one of the arguments of a constructor of the co-inductive type.

Co-inductive Predicates

- Used for properties on co-inductive types that cannot be defined inductively.

Co-inductive Predicates

- Used for properties on co-inductive types that cannot be defined inductively.
- Example: infiniteness of LLists.
  - Finiteness can be proven with a finite number of applications of `Finite_LCons` to a term obtained with `Finite_LNil`.

Co-inductive Predicates

- Used for properties on co-inductive types that cannot be defined inductively.
- Example: infiniteness of LLists.
  - Finiteness can be proven with a finite number of applications of `Finite_LCons` to a term obtained with `Finite_LNil`.
    An *inductive* predicate.

Co-inductive Predicates

- Used for properties on co-inductive types that cannot be defined inductively.
- Example: infiniteness of LLists.
  - Finiteness can be proven with a finite number of applications of `Finite_LCons` to a term obtained with `Finite_LNil`.
    An *inductive* predicate.
  - Infiniteness cannot be proven this way.
    It needs a *co-inductive* predicate.

Predicate for Infinite

This is a predicate that indicates that a LList is infinite.

```
CoInductive Infinite (A:Set) : LList A -> Prop :=
 Infinite_LCons :
  forall (a:A) (l : LList A), Infinite l -> Infinite (LCons a l).
```

Infinite proofs

- We want to prove that `from n` yields infinite lists for every natural number n.

Infinite proofs

- We want to prove that `from n` yields infinite lists for every natural number n.
- We do this by building an inhabitant of the type `forall n:nat, Infinite (from n)`.

Infinite proofs

- We want to prove that `from n` yields infinite lists for every natural number n.
- We do this by building an inhabitant of the type `forall n:nat, Infinite (from n)`.
- For this, we need a co-recursive function of which this is a fixpoint.

Infinite proofs

- We want to prove that `from n` yields infinite lists for every natural number n.
- We do this by building an inhabitant of the type `forall n:nat, Infinite (from n)`.
- For this, we need a co-recursive function of which this is a fixpoint.

We define

```
Definition F_from :
  (forall n:nat, Infinite (from n)) -> forall n:nat, Infinite (from n).
```

Infinite proofs

- We want to prove that `from n` yields infinite lists for every natural number n.
- We do this by building an inhabitant of the type `forall n:nat, Infinite (from n)`.
- For this, we need a co-recursive function of which this is a fixpoint.

We define

```
Definition F_from :
  (forall n:nat, Infinite (from n)) -> forall n:nat, Infinite (from n).
```

We have to prove that this satisfies the guard condition.

Infinite proofs

- We want to prove that `from n` yields infinite lists for every natural number n.
- We do this by building an inhabitant of the type `forall n:nat, Infinite (from n)`.
- For this, we need a co-recursive function of which this is a fixpoint.

We define

```
Definition F_from :
  (forall n:nat, Infinite (from n)) -> forall n:nat, Infinite (from n).
```

We have to prove that this satisfies the guard condition.

```
intro H.
intro n.
rewrite (from_unfold n).
split.
apply H.
Defined.
```

The `cofix` tactic

- The `cofix` tactic automates much of the above:

The `cofix` tactic

- The `cofix` tactic automates much of the above:

```
Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof cofix H : forall n:nat, Infinite (from n) := F_from H.
```

The `cofix` tactic

- The `cofix` tactic automates much of the above:

```
Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof cofix H : forall n:nat, Infinite (from n) := F_from H.
```

- To prove a property P, where P uses a co-inductive predicate, one should construct a term of the form `cofix H : P := t`.

The `cofix` tactic

- The `cofix` tactic automates much of the above:

```
Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof cofix H : forall n:nat, Infinite (from n) := F_from H.
```

- To prove a property P, where P uses a co-inductive predicate, one should construct a term of the form `cofix H : P := t`.
- Here, t has type P in the context with a hypothesis $H : P$.
  The term we obtain satisfies the guard condition.

The `cofix` tactic

- The `cofix` tactic automates much of the above:

```
Theorem from_Infinite_V0 : forall n:nat, Infinite (from n).
Proof cofix H : forall n:nat, Infinite (from n) := F_from H.
```

- To prove a property P, where P uses a co-inductive predicate, one should construct a term of the form `cofix H : P := t`.
- Here, t has type P in the context with a hypothesis $H : P$.
  The term we obtain satisfies the guard condition.
- This can also be done without explicitly mentioning P.

```
Theorem from_Infinite_V1 : forall n:nat, Infinite (from n).
Proof.
cofix H.
apply (F_from H).
Qed.
```

And we can use this tactic in an interactive way.

```
Theorem from_Infinite : forall n:nat, Infinite (from n).
Proof.
cofix H.
intro n.
rewrite (from_unfold n).
apply Infinite_LCons.
apply H.
Qed.
```

Guard condition violation

```
Lemma from_Infinite_buggy :
  forall n:nat, Infinite (from n).
Proof.
cofix H.
auto with llists.
```

Guard condition violation

```
Lemma from_Infinite_buggy :
  forall n:nat, Infinite (from n).
Proof.
cofix H.
auto with llists.
```

Proof completed.

Guard condition violation

```
Lemma from_Infinite_buggy :
  forall n:nat, Infinite (from n).
Proof.
cofix H.
auto with llists.
```

Proof completed.

```
Qed.
```

Error: Recursive definition of "H" is ill-formed. In environment

```
H: V n:nat, Infinite (from n)
unguarded recursive call in "H"
```

The `Guarded` tactic

Check for guard violations after using an auto command:

The `Guarded` tactic

Check for guard violations after using an auto command:

```
Lemma from_Infinite_buggy : ..
Proof.
cofix H.
auto with llists.
Guarded.
```

The `Guarded` tactic

Check for guard violations after using an auto command:

```
Lemma from_Infinite_buggy : ..
Proof.
cofix H.
auto with llists.
Guarded.
```

Error: Recursive definition of "H" is ill-formed.

The `Guarded` tactic

Check for guard violations after using an auto command:

```
Lemma from_Infinite_buggy : ..
Proof.
cofix H.
auto with llists.
Guarded.
```

Error: Recursive definition of "H" is ill-formed.

```
Undo.
intro n; rewrite (from_unfold n).
split; auto.
Guarded.
```

The `Guarded` tactic

Check for guard violations after using an auto command:

```
Lemma from_Infinite_buggy : ..
Proof.
cofix H.
auto with llists.
Guarded.
```

Error: Recursive definition of "H" is ill-formed.

```
Undo.
intro n; rewrite (from_unfold n).
split; auto.
Guarded.
```

The condition holds up to here

The `Guarded` tactic

Check for guard violations after using an auto command:

```
Lemma from_Infinite_buggy : ..
Proof.
cofix H.
auto with llists.
Guarded.
```

Error: Recursive definition of "H" is ill-formed.

```
Undo.
intro n; rewrite (from_unfold n).
split; auto.
Guarded.
```

The condition holds up to here

```
Qed.
```

`LNil` is not infinite

```
Theorem LNil_not_Infinite : forall (A:Set), ~Infinite (LNil (A:=A)).
Proof.
intros A H.
inversion H.
Qed.
```

Infiniteness of `repeat`

- We prove that `repeat a` yields an infinite `LList A` for any a of type A.
- For this, we need an auxiliary lemma

Infiniteness of `repeat`

- We prove that `repeat a` yields an infinite `LList A` for any a of type A.
- For this, we need an auxiliary lemma

```
Lemma repeat_unfold: forall A:Set, forall a:A,
                 repeat a = LCons a (repeat a).
Proof.
intro A.
intro a.
LList_unfold (repeat a).
simpl.
reflexivity.
Qed.
```

We can use this lemma to prove the following theorem

We can use this lemma to prove the following theorem

```
Lemma repeat_infinite : forall (A:Set) (a:A), Infinite (repeat a).
Proof.
intro A.
cofix a.
intro b.
```

We can use this lemma to prove the following theorem

```
Lemma repeat_infinite : forall (A:Set) (a:A), Infinite (repeat a).
Proof.
intro A.
cofix a.
intro b.
```

The proof state at this moment is

```
A : Set
a : forall a : A, Infinite (repeat a)
b : A
============================
 Infinite (repeat b)
```

```
A : Set
a : forall a : A, Infinite (repeat a)
b : A
============================
 Infinite (repeat b)
```

```
A : Set
a : forall a : A, Infinite (repeat a)
b : A
============================
 Infinite (repeat b)
```

We finish this by

```
rewrite (repeat_unfold b).
apply Infinite_LCons.
apply a.
Qed.
```

Bisimilarity

Weaker form of equality: two things are the same if they look/behave the same.
For LLists: two LList As are *bisimilar* if the first element of each LList A are equal, and the tails are bisimilar again:

Bisimilarity

Weaker form of equality: two things are the same if they look/behave the same.
For LLists: two LList As are *bisimilar* if the first element of each LList A are equal, and the tails are bisimilar again:

```
CoInductive bisimilar (A:Set) : LList A -> LList A -> Prop :=
  | bisim_LNil : bisimilar LNil LNil
  | bisim_LCons : forall (a:A)(l l' : LList A),
     bisimilar l l' -> bisimilar (LCons a l) (LCons a l').
```

`bisimilar` is an equivalence relation

We end by showing that `bisimilar` is an equivalence relation.
We use the built-in definitions from the `Relations` library.

```
Theorem bisimilar_equiv :
                forall (A:Set), equiv (LList A) (bisimilar (A:=A)).
```

We prove this theorem by introducing three lemmas, that claim that `bisimilar` as a relation is reflexive, symmetric and transitive.

See accompanying Coq file.