形式化方法导引

第 6 章 案例分析

6.3 – Coq: A Prover based on Higher-order Logic

6.3.3 Inductive Declarations

黄文超
https://faculty.ustc.edu.cn/huangwenchao

代码链接

3. Inductive Declarations

Inductive Data Types

- A data-type *name* can be declared by specifying a set of constructors.

$$\text{Inductive } name : sort := c_1 : C_1 \mid \ldots \mid c_n : C_n .$$

  - *name* is the name of the type to be defined
  - *sort* is one of Set or Type (or even Prop)
  - c_i are the names of the constructors
  - C_i is the type of the constructor c_i

```
Print nat.
```

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


A data-type name can be declared by specifying a set of constructors. Each constructor ci is given a type Ci which declares the type of its expected argu- ments. A constructor possibly accepts arguments (which can be recursively of type name), and when applied to all its arguments, a constructor has type the inductive definition name itself. There are some syntactic restrictions over the type of constructors to make sure that the definition is well-founded.

3. Inductive Declarations

Inductive Data Types

- The declaration of an inductive definition
  - introduces new primitive objects for the type itself and its constructors;
  - generates theorems which provide induction principles to reason on objects in inductive types

```
Print bool.
```

```
Inductive bool : Set :=
    true : bool | false : bool.
```

```
Check bool_ind.
```

```
bool_ind
     : forall P : bool -> Prop,
       P true ->
       P false -> forall b : bool, P b
```

3. Inductive Declarations

Inductive Data Types

- The declaration of an inductive definition
  - introduces new primitive objects for the type itself and its constructors;
  - generates theorems which provide induction principles to reason on objects in inductive types

```
Print nat.
```

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

```
Check nat_ind.
```

```
nat_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

3. Inductive Declarations

Inductive Data Types

- The declaration of an inductive definition
  - introduces new primitive objects for the type itself and its constructors;
  - generates theorems which provide induction principles to reason on objects in inductive types

```
Print prod.
```

```
Inductive prod (A B : Type) : Type :=
    pair : A -> B -> A * B.
```

- It is a polymorphic definition, parametrized by two type A and B
- The constructor pair takes two arguments and pairs them in an object of type A * B
  - which is what is expected for a product representation.

3. Inductive Declarations

Inductive Types and Equality

- The constructors of an inductive type are injective and distinct, e.g.,
  - true $\neq$ false
  - $S\ n = S\ m \rightarrow n = m$ and $S\ n \neq 0$
- **Tactics** to prove for new inductive types:
  - discriminate H: prove **any** goal if H is a proof of $t_1 = t_2$ with t_1 and t_2 starting with different constructors

```
Goal (forall n, S (S n) = 1 -> 0=1).
intros n H.
discriminate H.
Qed.
```

3. Inductive Declarations

Inductive Types and Equality

- **Tactics** to prove for new inductive types:
  - injection H: assumes H is a proof of $t_1 = t_2$ with t_1 and t_2 starting with the same constructor
    - It will deduce equalities $u_1 = u_2, v_1 = v_2, \ldots$ between corresponding subterms and add these equalities as new hypotheses.

```
Goal (forall n m, S n = S (S m) -> 0<>n).
intros n m H.
injection H.
intro j.
intro k.
assert (0 = S m).
transitivity n.
assumption.
assumption.
discriminate H0.
```

3. Inductive Declarations

Inductive Propositions.

- All propositional connectives, except for negation, implication and universal quantifier, are declared using inductive definitions
  - False is a degenerated case where there are no constructors.

```
Print False.
Inductive False : Prop :=  .
```

```
Check False_ind.
False_ind
     : forall P : Prop, False -> P
```

  - True is the proposition with only one proof I

```
Print True.
Inductive True : Prop :=  I : True.
```

3. Inductive Declarations

Inductive Propositions.

- All propositional connectives, except for negation, implication and universal quantifier, are declared using inductive definitions
  - Conjunction of two propositions corresponds to the product type

```
Print and.
```

```
Inductive and (A B : Prop) : Prop :=
    conj : A -> B -> A /\ B.
```

```
Check and_ind.
```

```
and_ind
     : forall A B P : Prop,
       (A -> B -> P) -> A /\ B -> P
```

3. Inductive Declarations

Inductive Propositions.

- All propositional connectives, except for negation, implication and universal quantifier, are declared using inductive definitions
  - Disjunction to an inductive proposition with two constructors

```
Print or.
```

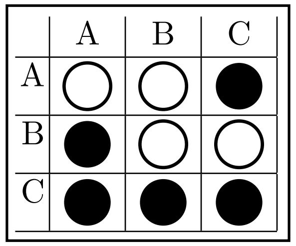
```
Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B
  | or_intror : B -> A \/ B.
```

```
Check or_ind.
```

```
or_ind
     : forall A B P : Prop,
       (A -> P) -> (B -> P) -> A \/ B -> P
```

3. Inductive Declarations

The Board Example



- At each step it is possible to choose one line or one column and to inverse the color of each token on that line or column.
- We want to study when a configuration is reachable from a starting configuration.

```
Inductive color : Type := White | Black.
Inductive pos : Type := A | B | C.
Inductive triple M := Triple : M -> M -> M -> triple M.
```

- A line White/Black/White will be represented by
  - the term Triple White Black White, with M to be explicitly given
  - So, the Coq internal term is Triple color White Black White
- To tell Coq to infer type arguments whenever possible

```
Set Implicit Arguments.
```

3. Inductive Declarations

The Board Example

- Introduce a special notation for triples:

```
Notation "[ x | y | z ]" := (Triple _ x y z).
```

- define a function:
  - Input: an element m in M
  - Output: a triple with the value m in the three positions.

```
Definition triple_x M (m:M) : triple M := [m | m | m].
```

3. Inductive Declarations

Definitions by pattern-matching

- When a term *t* belongs to some inductive type, it is possible to
  - build a new term by case analysis over the various constructors
    - which may occur as the head of *t* when it is evaluated.

$$\mathbf{match}\ term\ \ \mathbf{with}\ c_1\ args_1 \Rightarrow term_1\ \ \ldots\ c_n\ args_n \Rightarrow term_n\ \mathbf{end}$$

- In this construction, the expression term has an inductive type with *n* constructors $c_1, \ldots, c_n$.
- The term $term_i$ is the term to build when the evaluation of *t* produces the constructor c_i.

```
Definition iszero n :=
  match n with | O => true | S x => false end.
```

3. Inductive Declarations

Definitions by pattern-matching

- The board example
  - inverses a color

```
Definition turn_color (c: color) : color :=
  match c with | White => Black | Black => White end.
```

  - given a function *f*, and a triple *(a,b,c)*, applies *f* to all components

```
Definition triple_map M f (t: triple M) : triple M:=
  match t with (Triple _ a b c) => [(f a)|(f b)|(f c)] end.
```

  - expects a position and applies the function f at that position

```
Definition triple_map_select M f p t : triple M :=
  match t with (Triple _ a b c) =>
    match p with | A => [ (f a) | b | c ]
                 | B => [ a | (f b) | c ]
                 | C => [ a | b | (f c) ]
    end
  end.
```

3. Inductive Declarations

Generalized Pattern-Matching Definitions.

- Patterns can match several terms at the same time
  - they may contain the universal pattern which filters any expression
- Patterns are examined in a sequential way
  - they must cover the whole domain of the inductive type

```
Definition nozero n m :=
  match n, m with
    | O, _ => false | _, O => false | _, _ => true
  end.
```

3. Inductive Declarations

Some Equivalent Notations

- In the case of an inductive type with a single constructor C:

$$\textbf{let } (x_1, .., x_n) \textbf{:=} t \textbf{ in } u$$

  - is equivalent to $\textbf{match } t \textbf{ with } C x_1 .. x_n \Rightarrow u \textbf{ end}$

```
Definition fst (A B:Set) (p:A * B) := let (x, _) := p in x.
```

```
Definition fst (A B:Set) (H:A * B) := match H with
  | pair x y => x
end.
```

- In the case of an inductive type with two constructors (like booleans) c_1 and c_2

$$\textbf{if } t \textbf{ then } u_1 \textbf{ else } u_2$$

  - is equivalent to $\textbf{match } t \textbf{ with } c_1 \Rightarrow u_1 | c_2 \Rightarrow u_2 \textbf{ end}.$

3. Inductive Declarations

Fixpoint Definitions

- A function can be defined by fixpoint,
  - if one of its formal arguments, say x, has an inductive type and
  - if each **recursive call** is performed on a term which can be checked to be structurally smaller than x.

$$\textbf{Fixpoint}\ name\ (x_1 : type_1)\ \ldots\ (x_p : type_p)\{\texttt{struct}\ x_\mathrm{i}\} : type \ \texttt{:=}\ term$$

- The variable x_i following the **struct** keyword is the recursive argument.
- Its type $type_i$ must be an instance of an inductive type.

https://faculty.ustc.edu.cn/huangwenchao/zh_CN/zdylm/680196/list/index.htm

To define interesting functions over recursive data types, we use recursive functions. General fixpoints are not allowed since they lead to an unsound logic.

Only **structural recursion** is allowed. The basic idea is that x will usually be the main argument of a match and then recursive calls can be performed in each branch on some variables of the corresponding pattern.

3. Inductive Declarations

Fixpoint Definitions

- decreasing on 1st argument

```
Fixpoint plus1 (n m:nat) : nat :=
match n with | O => m | S p => S (plus1 p m) end.
```

- decreasing on 2nd argument

```
Fixpoint plus2 (n m:nat) : nat :=
match m with | O => n | S p => S (plus2 n p) end.
```

- **Error:** Cannot guess decreasing argument of fix.

```
Fixpoint test (b:bool) (n m:nat) : bool
  := match (n,m) with
  | (O,_) => true | (_,0) => false
  |(S p,S q)=> if b then test b p m else test b n q
end.
```

  - There should be one decreasing argument for each fixpoint

3. Inductive Declarations

Computing

- One can reduce a term and prints its normal form with **Eval** compute in term

```
Eval compute in (2 + 3)%nat.
```

```
Eval compute in (turn_color White).
```

```
Eval compute in (triple_map turn_color [Black|White|White]).
```

3. Inductive Declarations

Fixpoint Definitions

- Ackermann function

$$A(n,m) = \begin{cases} m+1 & \textbf{if } n=0 \\ A(n-1,1) & \textbf{if } m=0 \\ A(n-1,A(n,m-1)) & \textbf{otherwise} \end{cases}$$

```
Fixpoint ack (n m:nat) {struct n} : nat
  := match n with
       | 0 =>S m
       | S p => let fix ackn (m:nat) {struct m} :=
                  match m with 0 => ack p 1
                             | S q => ack p (ackn q)
                  end
                in ackn m
     end.
```

```
Goal forall n, ack (S n) 0 = ack n 1.
```

```
Goal forall n m, ack (S n) (S m) = ack n (ack (S n) m).
```


However, it is possible to define functions with more elaborated recursive schemes using higher order functions like the Ackermann function.
We may remark the internal definition of fixpoint using the let fix construction which defines the value of ack n as a new function ackn with one argument and a structurally smaller recursive call.
As an exercise, you may prove that the following equations are solved using reflexivity.

3. Inductive Declarations

Algorithms on lists

- How to prove algorithms on arrays?

```
Require Import List ZArith.
Open Scope Z_scope.
Open Scope list_scope.
Print list.
```

```
Inductive list (A : Type) : Type :=
    nil : list A
  | cons : A -> list A -> list A.
```

- Notations for lists include a::l for the operator cons and l1++l2 for the concatenation of two lists.

3. Inductive Declarations

Algorithms on lists

- Sum and maximum

```
Fixpoint sum (l : list Z) : Z :=
  match l with nil => 0 | a::m => a + sum m end.
```

```
Fixpoint max (l : list Z) : Z :=
  match l with nil => 0
             | a::nil => a
             | a::m => let b:= max m in
                         if Zle_bool a b then b else a
  end.
```

3. Inductive Declarations

Inductive Relations

$$\textbf{Inductive}\ name : arity := c_1 : C_1 \mid \ldots \mid c_n : C_n$$

- name: the name of the relation to be defined
- arity: its type
  - for instance nat->nat->Prop for a binary relation over natural numbers)
- as for data types, c_i and C_i are the names and types of constructors respectively.

- Example: $\forall n : \mathtt{nat}, 0 \leq n$ $\quad \forall n m : \mathtt{nat}, n \leq m \Rightarrow (\mathtt{S}\, n) \leq (\mathtt{S}\, m)$

```
Inductive LE : nat -> nat -> Prop :=
  | LE_O : forall n:nat, LE 0 n
  | LE_S : forall n m:nat, LE n m -> LE (S n) (S m).
```

The Board Example (Recall)

```
Inductive color : Type := White | Black.
Inductive pos : Type := A | B | C.
Inductive triple M := Triple : M -> M -> M -> triple M.

Set Implicit Arguments.

Notation "[ x | y | z ]" := (Triple _ x y z).

Definition triple_x M (m:M) : triple M := [m | m | m].

Definition turn_color (c: color) : color :=
  match c with | White => Black | Black => White end.

Definition triple_map M f (t: triple M) : triple M:=
  match t with (Triple _ a b c) => [(f a)|(f b)|(f c)] end.

Definition triple_map_select M f p t : triple M :=
  match t with (Triple _ a b c) =>
    match p with | A => [ (f a) | b | c ]
                 | B => [ a | (f b) | c ]
                 | C => [ a | b | (f c) ]
    end
  end.
```

https://faculty.ustc.edu.cn/huangwenchao/zh_CN/zdylm/680196/list/index.htm

The Board Example

```
Definition board := triple (triple color).
```

```
Definition start : board
  := [ [White | White | Black] |
       [Black | White | White] |
       [Black | Black | Black] ].
```

```
Definition target : board
  := [ [Black | Black | White] |
       [White | Black | Black] |
       [Black | Black | Black] ].
```

```
Definition turn_row (p: pos) : board -> board :=
  triple_map_select (triple_map turn_color) p.
```

```
Definition turn_col (p: pos) : board -> board :=
  triple_map (triple_map_select turn_color p).
```

The Board Example

- Define the relation

```
Definition move1 (b1 b2: board) : Prop :=
   (exists p : pos, b2=turn_row p b1)
\/ (exists p : pos, b2=turn_col p b1).
```

- An alternative direct inductive definition

```
Inductive move (b1:board) : board -> Prop :=
  move_row : forall (p:pos), move b1 (turn_row p b1)
| move_col : forall (p:pos), move b1 (turn_col p b1).
```

- Definition of reflexive-transitive closure

```
Inductive moves (b1:board): board -> Prop :=
  moves_init : moves b1 b1
| moves_step : forall b2 b3,
                 moves b1 b2 -> move b2 b3 -> moves b1 b3.
```

The Board Example

- Prove simple properties

```
Lemma move_moves : forall b1 b2, move b1 b2 -> moves b1 b2.
intros.
apply moves_step with b1.
apply moves_init.
assumption.
Qed.
```

- Prove reachability

```
Lemma reachable : moves start target.
apply moves_step with (turn_row A start).
apply move_moves.
apply move_row.
replace target with (turn_row B (turn_row A start)).
apply move_row.
auto.
Qed.
```

3. Inductive Declarations

Example: Needham-Schroeder Public Key protocol

```
Inductive agent : Set := A | B | I .
Inductive message : Set :=
  Name : agent -> message
        | Nonce : agent*agent -> message
        | SK : agent -> message
        | Enc : message -> agent -> message
        | P : message -> message -> message.

Section Protocol.
Variable X:agent.
```

```
Inductive send : agent -> message -> Prop :=
   init : send A (Enc (P (Nonce (A,X)) (Name A)) X)
 | trans1 : forall d Y,
    receive B (Enc (P (Nonce d) (Name Y)) B)
     -> send B (Enc (P (Nonce d) (Nonce (B,Y))) Y)
 | trans2 : forall d,
    receive A (Enc (P (Nonce (A,X)) (Nonce d)) A)
     -> send A (Enc (Nonce d) X)
 | cheat : forall m, known m -> send I m
with receive : agent -> message -> Prop :=
    link : forall m Y Z, send Y m -> receive Z m
with known : message -> Prop :=
    spy : forall m, receive I m -> known m
 | name : forall a, known (Name a)
 | secret_KI : known (SK I)
 | decomp_l : forall m m1, known (P m m1) -> known m
 | decomp_r : forall m m1, known (P m m1) -> known m1
 | compose : forall m m1, known m -> known m1 -> known (P m m1)
 | crypt : forall m a, known m -> known (Enc m a)
 | decrypt : forall m a, known (Enc m a) -> known (SK a) -> known m.
End Protocol.
```

```
Lemma flaw : receive I B (Enc (Nonce (B,A)) B).
```

```
Lemma flawB : known I (Nonce (B,A)).
```