IndProp: Inductively Defined Propositions (`IndProp.v`)

Keywords

- `Inductive` Inductive declaration, where each constructor corresponds to an inference rule

Tactics

- `inversion E as [| n' E'].`
- `inversion`
  - applied to equalities: `discriminate` and `injection` (also does necessary `intros + rewrite`)
  - applied to analyze evidence for inductively defined propositions
- `induction E as [|n' E' IH].` induction on `E` which is an inductively defined proposition
- Calling `destruct` has the effect of replacing all occurrences of the property argument by the values that correspond to each constructor.
- `rewrite H in *: *`

Defintions and Theorems

Inductively Defined Propositions

- `Inductive even : nat -> Prop := | ev_0 : even 0 | ev_SS (n : nat) (H : even n) : even (S (S n)).`
- `Theorem even_4 : even 4.` Use `apply` with the rule names
- `Theorem even_4' : even 4.` Use function application syntax
- `Theorem ev_plus4 : forall n, even n -> even (4 + n).`

- `Theorem ev_double : forall n, even (double n).`

Using Evidence in Proofs

- `Theorem ev_inversion : forall (n : nat), even n -> (n = 0) \/ (exists n', n = S (S n') /\ even n').`
- `Theorem ev_minus2 : forall n, even n -> even (pred (pred n)).`
- `Theorem evSS_ev : forall n, even (S (S n)) -> even n.`
- `Theorem evSS_ev' : forall n, even (S (S n)) -> even n.`
- `Theorem one_not_even : ~ even 1.`
- `Theorem one_not_even' : ~ even 1.`
- `Theorem SSSSev_even : forall n, even (S (S (S (S n)))) -> even n.`
- `Theorem even5_nonsense : even 5 -> 2 + 2 = 9.`

Induction on Evidence

- `Lemma ev_even : forall n, even n -> exists k, n = double k.`
- `Theorem ev_even_iff : forall n, even n <-> exists k, n = double k.`
- `Theorem ev_sum : forall n m, even n -> even m -> even (n + m).`
- `Inductive even' : nat -> Prop :=`
- `Theorem even'_ev : forall n, even' n <-> even n.`
- `Theorem ev_ev__ev : forall n m, even (n + m) -> even n -> even m.`
- `Theorem ev_plus_plus : forall n m p, even (n + m) -> even (n + p) -> even (m + p).`

Inductive Relations

- `Module Playground.`
- `Inductive le : nat -> nat -> Prop := | le_n n : le n n | le_S n m (H: le n m) : le n (S m).`
- `Notation "m <= n" := (le m n).`
- `End Playground.`
- `Definition lt (n m:nat) := le (S n) m.`
- `Notation "m < n" := (lt m n).`
- `Inductive square_of : nat -> nat -> Prop := | sq n : square_of n (n * n).`

- `Inductive next_nat : nat -> nat -> Prop := | nn n : next_nat n (S n).`
- `Inductive next_even : nat -> nat -> Prop := | ne_1 n : even (S n) -> next_even n (S n) | ne_2 n (H : even (S (S n))) : next_even n (S (S n)).`
- `Lemma le_trans : forall m n o, m <= n -> n <= o -> m <= o.`
- `Theorem O_le_n : forall n, 0 <= n.`
- `Theorem n_le_m__Sn_le_Sm : forall n m, n <= m -> S n <= S m.`
- `Theorem Sn_le_Sm__n_le_m : forall n m, S n <= S m -> n <= m.`
- `Theorem le_plus_l : forall a b, a <= a + b.`
- `Theorem plus_lt : forall n1 n2 m, n1 + n2 < m -> n1 < m /\ n2 < m.`
- `Theorem lt_S : forall n m, n < m -> n < S m.`
- `Theorem leb_complete : forall n m, n <=? m = true -> n <= m.`
- `Theorme leb_correct : forall n m, n <= m -> n <=? m = true.`
- `Theorem leb_true_trans : forall n m o, n <=? m = true -> m <=? o = true -> n <=? o = true.`
- `Theorem leb_iff : forall n m, n <=? m = true <-> n <= m.`
- `Module R.`
- `Inductive R : nat -> nat -> nat -> Prop`
- `Theorem R_equiv_fR : forall m n o, R m n o <-> fR m n = o.`
- `End R.`
- `Inductive subseq : list nat -> list nat -> Prop := | ss0 l : subseq [] l | ss1 h t1 t2 (H: subseq t1 t2) : subseq (h::t1) (h::t2) | ss2 l1 h2 t2 (Ht: subseq l1 t2) : subseq l1 (h2::t2).`
- `Theorem subseq_refl : forall (l : list nat), subseq l l.`
- `Theorem subseq_app : forall (l1 l2 l3 : list nat), subseq l1 l2 -> subseq l1 (l2 ++ l3).`
- `Theorem subseq_trans : forall (l1 l2 l3 : natlist), subseq l1 l2 -> subseq l2 l3 -> subseq l1 l3.`

Case Study: Regular Expression

- `inductive reg_exp {T : Type} : Type := | EmptySet | EmptyStr | Char (t : T) | App (r1 r2 : reg_exp) | Union (r1 r2 : reg_exp) | Star (r : reg_exp).`
- `induction exp_match {T : Type} : list T -> reg_exp -> Prop`
- `Notation "s =~ re" := (exp_match s re) (at level 80).`
- `Example reg_exp_ex1 : [1] =~ Char 1.`
- `Example reg_exp_ex2 : [1; 2] =~ App (Char 1) (Char 2).`

- `Example reg_exp_ex3 : ~ ([1; 2] =~ Char 1).` Using `inversion`, we can show that certain strings do not match a regular expression.
- `Fixpoint reg_exp_of_list {T} (l : list T)`
- `Lemma MStar1 : forall T s (re : @reg_exp T), s =~ re -> s =~ Star re.`
- `Lemma empty_is_empty : forall T (s : list T), ~ (s =~ EmptySet).`
- `Lemma MUnion' : forall T (s : list T) (re1 re2 : @reg_exp T), s =~ re1 \/ s =~ re2 -> s =~ Union re1 re2.`
- `Lemma MStar' : forall T (ss : list (list T)) (re : reg_exp), (forall s, In s ss -> s =~ re) -> fold app ss [] =~ Star re.`
- `Lemma reg_exp_of_list_spec : forall T (s1 s2 : list T), s1 =~ reg_exp_of_list s2 <-> s1 = s2.`
- `Fixpoint re_chars {T} (re : reg_exp) : list T`
- `Theorem in_re_match : forall T (s : list T) (re : reg_exp) (x : T), s =~ re -> In x s -> In x (re_chars re).`
- `Fixpoint re_not_empty {T : Type} (re : @reg_exp T) : bool`

Problems

- `Theorem ev_ev__ev : forall n m, even(n + m) -> even n -> even m.`
  `induction En as [| n' Hn' IHn']` Why is `IHn' : even(n' + m) -> even m`?
- `Theorem plus_lt : forall n1 n2 m, n1 + n2 < m -> n1 < m /\ n2 < m.` Easy proof?
- `Theorem leb_complete : forall n m, n <=? m = true -> n <= m.` Explain it.
- `Theorem subseq_trans : forall (l1 l2 l3 : natlist), subseq l1 l2 -> subseq l2 l3 -> subseq l1 l3.` How to prove it?

> Written with [StackEdit](https://stackedit.io/).