Chapter 5

Elpi implementation of Inductive

We discuss the implementation of the `eiInd` command together with integrations in the `eiIntros` tactic and the `eiInduction` tactic.

Structure of `eiInd` The `eiInd` command consists of several steps we have outlined in figure 5.1. Each of these steps are explained in the sections referenced in the diagram. TODO: Insert diagram

Inductive tactics In the last two sections we discuss how the tactics to use an inductive predicate are made. We first discuss the `eiInduction` tactic in section 5.8, which performs induction on the specified inductive predicate. Next, in section 5.9, we outline the extensions to the `eiIntros` tactic concerning inductive predicates.

5.1 Parsing inductive data structure

The `eiInd` command is called by writing a Coq inductive statement and prepending it with the `eiInd` command. We will use the below inductive statement as an example for this and any subsequent sections.

```Coq
eiInd
Inductive is_R_list {A} (R : val → A → iProp) :
                    val → list A → iProp :=
  | empty_is_R_list : is_R_list R NONEV []
  | cons_is_R_list l v tl x xs :
      l ↦ (v,tl) -∗ R v x -∗ is_R_list R tl xs -∗
      is_R_list R (SOMEV #l) (x :: xs).
```

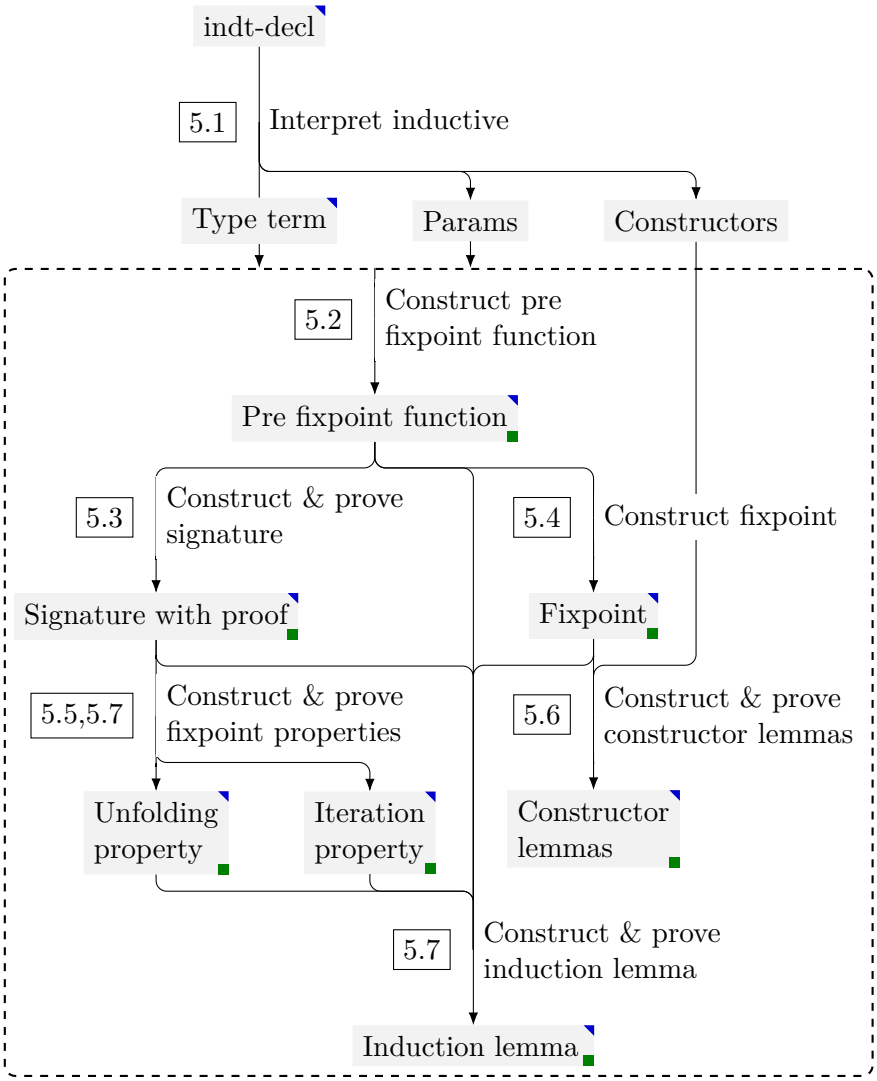

Figure 5.1: The structure of the `eiInd` command. Arrows are steps in the command and boxes are the objects that are being created. If a box has a `green box`, it is defined in Coq. If a box has a `blue triangle`, it is stored in the Elpi database. All arrows reference the section in which they are explained.

This inductive predicate relates a linked list to a Coq list by relation `R`. Since the Coq list can have an arbitrary type `A` and the predicate `R` is constant, we add them as parameters to the predicate.

In Elpi this is received as a data structure of type `indt-decl`.

Question: Should this be a more detailed explanation?

```
1  parameter `A` maximal                                          Elpi
2    TYPE
3    (a \ parameter `R` explicit
4          {{ val -> lp:a -> iProp }}
5          (r \ inductive `is_R_list` tt
6                (arity {{ val -> list lp:a -> iProp }})
7                (f \ [constructor `empty_is_R_list`
8                        (arity …),
9                      constructor `cons_is_R_list`
10                       (arity …)])))
```

The contents of the constructors is removed from this example for conciseness. A parameter consists of the name, if it should be maximally inserted, the type, and a function that given a binder returns the rest of the term. The first parameter has the name `` `A` ``, is maximally inserted, and the type is not yet calculated, thus a variable. The second parameter `R` depends on the first parameter in its type, `{{ val -> lp:a -> iProp }}`.

The first step of `eiInd` is interpreting the inductive into usable terms. This interpretation is done by descending into the binders. During the recursive steps, we keep track of the binders, together with the name and type of each binder. Later, whenever we construct a term we use this list of parameters to abstract the term over the parameters.

The inductive starts on line 5. It consists of its name, `` `is_R_list` ``, if its inductive, `tt`, its type, and a function containing the constructors. The next step in `eiInd` is normalizing the type to a term from a term with its own parameters. A lot of further steps are based on this term containing the type of the inductive predicate, called the *type term* from now on.

We now a have the list of parameters, the type of the inductive predicate and a list of the constructors under a binder. These are the ingredients needed to construct any further terms and proofs.

Question: This is a bit of a cluttered, structure. But how to add more structure?

5.2 Constructing the pre fixpoint function

In this step we want to transform the inductive definition of `is_R_list` into the function we take a fixpoint of called the pre fixpoint function. This step results in a function of the form of $\mathsf{isMLL}_\mathsf{F}$ in example 3.5thesis.pdf. From the previous step we got a function producing the two constructors.

```
1  f \ [                                                          Elpi
2    {{ lp:f (InjLV #()) [] }},
3    {{ l v tl x xs : l ↦ (v, tl) -* R v x -* lp:f tl xs
4                    -* lp:f (InjRV #l) (x :: xs) }}
5  ]
```

Note that the parameter arguments to `is_R_list'` are included in the function `f`. The first step is to transform any Coq binders in a constructor, such as `l'`, `v'` or `tl'` into Iris existential quantifiers. Next, the magic wands in the constructors are replaced with separating conjunctions. Lastly, we connect all constructors using Iris disjunctions. This results in the following intermediate Elpi function.

```
Body = f \ {{                                                    Elpi
    lp:f (InjLV #()) []
  ∨ ∃ l v tl x xs,
      l ↦ (v, tl) ∗ R v x ∗ lp:f tl xs ∗
      lp:f (InjRV #l) (x :: xs)
}}
```

Note that no types have been calculated at this point, as can be seen by the absence of types in the existential quantifiers. The next step is to transform this Elpi function into a term containing a Coq function over `f` and the two arguments of the inductive predicate. The type is transformed into a function by replacing the `prod` constructor with the `fun` constructor in the type term. The function binders are now created as follows, where `TypeTerm` is the term containing the type of the inductive predicate and `FunTerm` is that type transformed into a function. Thus, `FunTerm` has type `term -> term`, it takes a term and inserts that term as the body of the function.

```
F' = (fun `F` TypeTerm (f \ FunTerm (Body f)))                   Elpi
```

We create a Coq function that takes the recursive argument, `F'`. The body of the function is the previously created `FunTerm`. We fill the body with the created `Body`, where we fill in the recursive argument.

Lastly, we need to replace the final recursive call of every constructor with equalities. They relate the arguments of function to the values used in the constructor. They are created by descending into the functions and keeping track of the binders. By recursively descending into each constructor and always taking the right side of a separating conjunction we can find the last recursive call. We then replace it with an equality for each of its arguments.

If necessary, we abstract the created term over the parameters of the inductive. We type check this term and get the following pre fixpoint function.

```
λ (A : Type) (R : val → A → iProp)                               Coq
  (F : val → list A → iProp) (H : val) (H0 : list A),
    ⌜H = InjLV #()⌝ ∗ ⌜H0 = []⌝
  ∨ ∃ l (v tl : val) (x : A) (xs : list A),
```

```
        l ↦ (v, tl) ∗ R v x ∗ F tl xs ∗
        ⌜H = InjRV #l⌝ ∗ ⌜H0 = x :: xs⌝
```
Coq

This Coq term is defined as `is_R_list_pre`.

5.3 Creating and proving proper signatures

In this section we describe how a proper is created and proven for the previously defined function. This implements the theory as defined in section 3.3thesis.pdf.

Proper definition in Coq Proper elements of relations are defined using type classes and named `IProper`. Respectful relations, `R ==> R`, pointwise relations, `.> R` and persistent relations, `□> R` are defined with accompanying notations. Any signatures are defined as global instances of `IProper`.

To easily find the `IProper` instance for a given connective and relation an additional type class is added.

```
Class IProperTop {A} {B}
                 (R : iRelation A) (m : B)
                 f := iProperTop : IProper (f R) m.
```
Coq

Given a relation `R` and connective `m` we find a function `f` that transforms the relation into the proper relation for that connective. For example, given the `IProper` instance for separating conjunctions we get the `IProperTop` instance.

```
Global Instance sep_IProper :
  IProper _ (bi_wand ==> bi_wand ==> bi_wand)
           bi_sep.

Global Instance sep_IProperTop :
  IProperTop bi_wand (bi_sep)
             (fun F => bi_wand ==> bi_wand ==> F).
```
Coq

Creating a signature Using these Coq definitions we transform the type into an `IProper`. A Proper relation for a function as described above will always have the shape `(□> R ==> R)`. The relation `R` is constructed by wrapping a wand with as many pointwise relations as there are arguments in the inductive predicate. The full `IProper` term is constructed by giving this relation to `IProper` together with the pre fixpoint function. Any parameters are quantified over and given to the fixpoint function.

```Coq
∀ (A : Type) (R : val → A → iProp),
  IProper (□> .> .> bi_wand ==> .> .> bi_wand)
          (is_R_list_pre A R)
```

Proving a signature To prove a signature we implement the recursive algorithm as defined in section 3.3thesis.pdf. We use the proof generators from section 4.7thesis.pdf to create a proof term for the signature. We will highlight the interesting step of applying an `IProper` instance.

A relevant `IProperTop` instance can be found by giving the top level relation and top level function of the current goal. However, some `IProperTop` instances are defined on partially applied functions. Take the existential quantifier. It has the type `∀ {A : Type}, (A → iProp) → iProp`. The `IProper` and `IProperTop` instances are defined with an arbitrary `A` filled in.

```Coq
Global Instance exists_IProper {A} :
  IProper (.> bi_wand ==> bi_wand)
          (@bi_exist A).
Global Instance exists_IProperTop {A} :
  IProperTop (bi_wand) (@bi_exist A)
             (fun F => .> bi_wand ==> F).
```

Thus, when searching for the instance we also have to fill in this argument. The amount of arguments we have to fill in when searching for an `IProperTop` instance differs per connective. We take the following approach.

```Elpi
pred do-steps.do i:ihole, i:term, i:term, i:term.
do-steps.do IH R (app [F | FS]) _ :-
  std.exists { std.iota {std.length FS} }
             (n\ std.take n FS FS'),
  do-iApplyProper IH R (app [F | FS']) HS, !,
  std.map HS (x\r\ do-steps x) _.
```

The `do-steps.do` predicate contains rules for every possibility in the proof search algorithm. The rule highlighted here applies an `IProper` instance. It gets the Iris hole `IH`, the top level relation `R`, and the top level function `app [F | FS]`. The last argument is not relevant for this rule.

Next, on line 3, we first create a list of integers from 1 till the length of the arguments of top level function with `std.iota`. Next, the `std.exists` predicate tries to execute its second argument for every element of this list until one succeeds. The second argument then just takes the first `n` arguments of the top level function and stores it in the variable `FS'`.

This obviously always succeeds, however the predicate on line 4 does not. `do-iApplyProper` takes the Iris hole, relation and now partially applied top level function and tries to apply the appropriate `IProper` instance. However, when this predicate fails because it can't find an `IProper` instance, we backtrack into the previous predicate. This is `std.exists`, and we try the next rule there, and we take the next element of the list and try again. This internal backtracking ensures we try every partial application of the top level function until we find an `IProperTop` instance that works. If there are none, we can try another rule of `do-steps.do`.

Lastly on line 6, we continue the algorithm. We don't want to backtrack into the `std.exists` when something goes wrong in the rest of the algorithm, thus we include a cut after successfully applying the `IProper` instance.

The predicate `do-iApplyProper` follows the same pattern as the other Iris proof generators we defined in section 4.7thesis.pdf. It mirrors a simplified version of the IPM `iApply` tactic while also finding the appropriate `IProper` instance to apply.

Defining the pre fixpoint function monotone With the signature and the proof term for monotonicity of the pre fixpoint function we define a new lemma in Coq called `is_R_list_pre_mono`. Thus allowing any further proof in the command and outside it to make use of the monotonicity of `is_R_list_pre`.

5.4 Constructing the fixpoint

`eiInd` generates the fixpoint as defined in section 3.3thesis.pdf. The fixpoint is generated by recursing through the type term multiple times using the ideas of the previous sections. Afterwards we abstract over the parameters of the inductive. This results in creating the following fixpoint statement defined as `is_R_list`.

```
1 λ (A : Type) (R : val → A → iProp)                                    Coq
2   (v : val) (l : list A),
3   (∀ F : val → list A → iProp,
4     □ (∀ (v' : val) (l' : list A),
5           is_R_list_pre A R F v' l' -∗ F v' l')
6     -∗ F v l)
```

Coq-Elpi database Coq-Elpi provides a way to store data between executions of tactics and commands, this is called the database. We define predicates whose rules are stored in the database.

```
Elpi Db induction.db lp:{{                                    Coq
  pred inductive-pre o:gref, o:gref.
  pred inductive-mono o:gref, o:gref.
  pred inductive-fix o:gref, o:gref.
  pred inductive-unfold o:gref, o:gref, o:gref,
                        o:gref, o:int.
  pred inductive-iter o:gref, o:gref.
  pred inductive-ind o:gref, o:gref.
  pred inductive-type o:gref, o:indt-decl.
}}.
```

The rules are always defined such that the fixpoint definition is the first argument and the objects we want to associate to it are next. Thus, to store the pre fixpoint function of `is_R_list` in the database we add the rule:

```
inductive-pre (const «is_R_list»)                             Elpi
              (const «is_R_list_pre»)
```

Where instead of `«···»` we insert the variable containing the actual reference. We store the references to any objects we create after any of the previous of following steps. We also include some extra information in some rules, like `inductive-unfold` includes the amount of constructors the fixpoint has, and `inductive-type` contains only the Coq inductive. When retrieving information about an object, we can simply check in the database by calling the appropriate predicate.

5.5 Unfolding property

In this section we prove the unfolding property of the fixpoint from theorem 3.3thesis.pdf. This proof is generated for every new inductive predicate to account for the different possible arities of inductive predicates. The proof of the unfolding property is split into three parts, separate proofs of the two directions and finally the combination of the directions into the unfolding property. We explain how the proof of one direction is created in the section. Any other proofs generated in this or other sections follow the same strategy and will not be explained in as much detail.

Generating the proof goal is done by recursing over the type term, this results in the following statements to prove. Where the other unfolding lemmas either flip the entailment flipped or replace it with a double entailment.

```
∀ (A : Type) (R : val → A → iProp)                            Coq
  (v : val) (l : list A),
    is_R_list_pre A R (is_R_list A R) v l
  ⊢ is_R_list A R v l
```

The proof term is generated by chaining proof generators such that no holes exist in the proof term.

```
pred mk-unfold.r->l i:int, i:int,                                    Elpi
                    i:term, i:term, i:hole.
mk-unfold.r->l Ps N Proper Mono (hole Type Proof) :-
  do-intros-forall (hole Type Proof)
                   (mk-unfold.r->l.1 Ps N Proper Mono).
```

This predicate performs the first step in the proof generation before calling the next step. It takes the amount of parameters, `Ps`, the amount of arguments the fixpoint takes, `N`, the `IProper` signature, `Proper`, a reference to the monotonicity proof `Mono` and the hole for the proof. It then introduces any universal quantifiers at the start of the proof. The rest of the proof has to happen under the binder of these quantifiers, thus we use CPS to continue the proof in the predicate `mk-unfold.r->l.1`.

```
pred mk-unfold.r->l.1 i:int, i:int,                                  Elpi
                      i:term, i:term, i:hole.
mk-unfold.r->l.1 Ps N Proper Mono H :-
  do-iStartProof H IH, !,
  do-iIntros [iIdent (iNamed "HF"), iPure none,
              iIntuitionistic (iIdent (iNamed "HI")),
              iHyp "HI"] IH
             (mk-unfold.r->l.2 Ps N Proper Mono).
```

This proof generator performs all steps possible using the `do-iIntros` proof generator. It takes the same arguments as `mk-unfold.r->l`. On line 3, it initializes the Iris context and thus creates an Iris hole, `IH`. Next, we apply several proof steps using de `do-iIntros` proof generator. This again results in a continuation into a new proof generator. We are now in the following proof state.

```
"HI" : ∀ (v : val) (l : list A),                                     Coq
       is_R_list_pre A R F v l -∗ F v l
--------------------------------------------------□
"HF" : is_R_list_pre A R (is_R_list A R) l' v'
--------------------------------------------------∗
is_R_list_pre A R F l' v'
```

We need to apply monotonicity of `is_R_list_pre` on the goal and `"HF"`.

```
1  pred mk-unfold-2.proof-2 i:int, i:int,                          Elpi
2                           i:term, i:term, i:ihole.
3  mk-unfold-2.proof-2 Ps N Proper Mono IH :-
4    ((copy {{ @IProper }} {{ @iProper }} :- !) =>
5      copy Proper IProper'),
6    type-to-fun IProper' IProper,
7    std.map {std.iota Ps} (x\r\ r = {{ _ }}) Holes, !,
8    do-iApplyLem (app [IProper | Holes]) IH [
9      (h\ sigma PType\ sigma PProof\
10         sigma List\ sigma Holes2\ !,
11       h = hole PType PProof,
12       std.iota Ps List,
13       std.map List (x\r\ r = {{ _ }}) Holes2,
14       coq.elaborate-skeleton (app [Mono | Holes2])
15                              PType PProof ok,
16   )] [IH1, IH2],
17   do-iApplyHyp "HF" IH2 [], !,
18   std.map {std.iota N} (x\r\ r = iPure none) Pures, !,
19   do-iIntros
20     {std.append [iModalIntro | Pures]
21                 [iIdent (iNamed "H"), iHyp "H",
22                  iModalIntro, iHyp "HI"]}
23     IH1 (ih\ true).
```

We won't discuss this last proof generator in full detail but explain what is generally accomplished by the different lines of code. The proof generator again takes the same arguments as the previous two steps. Lines 4-7 transform the signature of the pre fixpoint function into the following statement we can apply to the goal.

```
1  (λ (A : Type) (R : val → A → iProp),                            Coq
2    iProper (□> .> .> bi_wand ==> .> .> bi_wand)
3            (is_R_list_pre A R)
4  ) _ _
```

Question: Long version: On lines 4 and 5 to signature is transformed into a term that can be applied to the current hole. The type class is replaced by the type class constructor and any universal quantifiers are transformed into lambda expressions with the same type binder. On line 7 a list of holes is generated to append to the monotonicity statement we apply. These holes fill in the parameter arguments in the statement. We are thus applying the following statement.

Line 8 applies this statement resulting in 3 holes we need to solve. The first hole is a non-Iris hole that resulted from transforming the goal into an Iris entailment. This hole has to be solved in CPS. This is done in lines 9-15. Lines 9-15 apply the proof of monotonicity to solve the `IProper` condition[1].

Line 17 ensures that the monotonicity is applied on `"HF"`. Next, lines

[1]This section of code can't make use spilling, thus creating many more lines and temporary variables. We can't use spilling since the hidden temporary variables created by spilling are defined at the top level of the predicate. Thus, they can't hold any binders that we might be under. So solve this we define any temporary variables ourselves using the `sigma X\` connective.

18-23 solve the following goal using another instance of the `do-iIntros` proof generator.

```
"HI" : ∀ (H : val) (H0 : list A),
        is_R_list_pre A R a H H0 -∗ a H H0
--------------------------------------------□
(□> .> .> bi_wand)%i_signature (is_R_list A R) a
```

Thus proving the right to left unfolding property. This proof together with the other two proofs of this section are defined as `is_R_list_unfold_1`, `is_R_list_unfold_2` and `is_R_list_unfold`.

5.6 Constructor lemmas

The constructors of the inductive predicate are transformed into lemmas that can be applied during a proof utilizing inductive predicates. By again recursing on the type term a lemma is generated per constructor.

```
∀ (A : Type) (R : val → A → iProp)
  (v : val) (l : list A),
  ⌜v = InjLV #()⌝ ∗ ⌜l = []⌝ -∗ is_R_list A R v l

∀ (A : Type) (R : val → A → iProp)
  (v : val) (l : list A),
     (∃ l' (v' tl : val) (x : A) (xs : list A),
        l' ↦ (v', tl) ∗ R v' x ∗ is_R_list A R tl xs ∗
        ⌜v = InjRV #l'⌝ ∗ ⌜l = x :: xs⌝)
  -∗ is_R_list A R v l
```

Both constructor lemmas are an wand of the associated constructor to the fixpoint. They are defined with the name of their respective constructor, `empty_is_R_list` and `cons_is_R_list`.

Question: Mention that equalities could be resolved, but that it is not done?

5.7 Iteration and induction lemmas

The iteration and induction lemmas follow the same strategy as the previous sections. The iteration property that we prove is:

```
∀ (A : Type) (R : val → A → iProp)
  (Φ : val → list A → iProp),
  □ (∀ (H : val) (H0 : list A),
       is_R_list_pre A R Φ H H0 -∗ Φ H H0) -∗
  ∀ (H : val) (H0 : list A),
    is_R_list A R H H0 -∗ Φ H H0
```

The induction lemma that we prove is:

```
∀ (A : Type) (R : val → A → iProp)
  (Φ : val → list A → iProp),
  □ (∀ (H : val) (H0 : list A),
      is_R_list_pre A R
        (λ (H1 : val) (H2 : list A),
           Φ H1 H2 ∧ is_R_list A R H1 H2) H
        H0 -∗ Φ H H0) -∗
  ∀ (H : val) (H0 : list A),
    is_R_list A R H H0 -∗ Φ H H0
```

Coq

These both mirror the iteration property and induction lemma from section 3.3thesis.pdf. They are defined as `is_R_list_iter` and `is_R_list_ind`.

Question: Maybe explain in more detail? But they are basically the same as in 3.3. If I do switch to isMLL for this chapter, would it then be fine?

5.8 `eiInductive` tactic

The `eiInduction` tactic will apply the induction lemma and perform follow-up proof steps such that we get base and inductive cases to prove. We first show an example of applying the induction lemma and then show how the `eiInduction` tactic implements the same and more.

Question: I switch example which is not so nice, but I don't know how to properly do fix this

Example 5.1

We show how to apply the induction lemma in a Coq lemma. We take as an example lemma 2.2thesis.pdf.

```
Lemma MLL_delete_spec (vs : list val)
                      (i : nat) (hd : val) :
  [[{ is_MLL hd vs }]]
    MLL_delete hd #i
  [[{ RET #(); is_MLL hd (delete i vs) }]].
Proof.
```

Coq

The proof of this Hoare triple was by induction, thus we first prepare for the induction step resulting in the following proof state.

```
vs: list val
hd: val
--------------------------------------
"His" : is_MLL hd vs
--------------------------------------*
∀ (P : val → iPropI Σ) (i : nat),
  (is_MLL hd (delete i vs) -* P #()) -*
  WP MLL_delete hd #i [{ v, P v }]
```
Coq

Here `"His"` is the assumption we apply induction on. As `Φ` we choose the function:

```
λ (hd: val) (vs: list val),
  ∀ (P : val → iPropI Σ) (i : nat),
    (is_MLL hd (delete i vs) -* P #()) -*
    WP MLL_delete hd #i [{ v, P v }]
```
Coq

Allowing us to apply the induction lemma.

The `eiInduction` tactic is called as `eiInduction "His" as "[...]"`. It takes the name of an assumption and an optional introduction pattern.

```
pred do-iInduction i:ident, i:intro_pat, i:ihole,
                   o:(ihole -> prop).
do-iInduction ID IP (ihole _ (hole Type _) as IH) C :-
  find-hyp ID Type (app [global GREF | Args]),
  inductive-ind GREF INDLem, !,
  inductive-type GREF T, !,
  do-iInduction.inner ID IP T (app [global INDLem])
                      Args IH C.
```
Elpi

This is the proof generator for induction proofs. It takes the identifier of the induction assumption and the introduction pattern. If there is no introduction pattern given, `IP` is `iAll`. Lastly, the proof generator takes the iris hole to apply induction in.

On line 3 we get the fixpoint object and its arguments. Next, on line 4 and 5, we search in the database for the induction lemma and Coq inductive object associated with this fixpoint. This information is all given to the inner function.

The inner predicate is used to recursively descent through the inductive data structure and apply any parameters to the induction lemma. Next, the conclusion of the Iris entailment is taken out of the goal. It is transformed into a function over the remaining arguments of the induction assumption. And we apply the induction lemma with the applied parameters and the function.

The resulting goal first gets general introduction steps and then either applies the introduction pattern given or just destructs into the base and induction cases.

5.9 `eiIntros` integrations

The `eiIntros` tactic gets additional cases for destructing induction predicates. Whenever a disjunction elimination introduction pattern is used, the tactic first checks if the connective to destruct is an inductive predicate. If this is the case, it first applies the unfolding lemma before doing the disjunction elimination.

We also add a new introduction pattern `"**"`. This introduction pattern checks if the current top level connective is an inductive predicate. If this is the case, it uses unfolding and disjunction elimination to eliminate the predicate.