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.

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 5.7, which performs induction on the specified inductive predicate. Next, in section 5.8, we outline the extensions to the `eiIntros` tactic concerning inductive predicates.

5.1 Constructing the pre fixpoint function

The `eiInd` command is called by writing a Coq inductive statement and prepending it with the `eiInd` command. The below inductive statement implements the isMLL inductive predicate from chapter 3thesis.pdf in Coq.

```Coq
eiInd
Inductive is_MLL : val → list val → iProp :=
    | empty_is_MLL : is_MLL NONEV []
    | mark_is_MLL v vs l tl :
      l ↦ (v, #true, tl) -* is_MLL tl vs -*
      is_MLL (SOMEV #l) vs
    | cons_is_MLL v vs tl l :
      l ↦ (v, #false, tl) -* is_MLL tl vs -*
      is_MLL (SOMEV #l) (v :: vs).
```

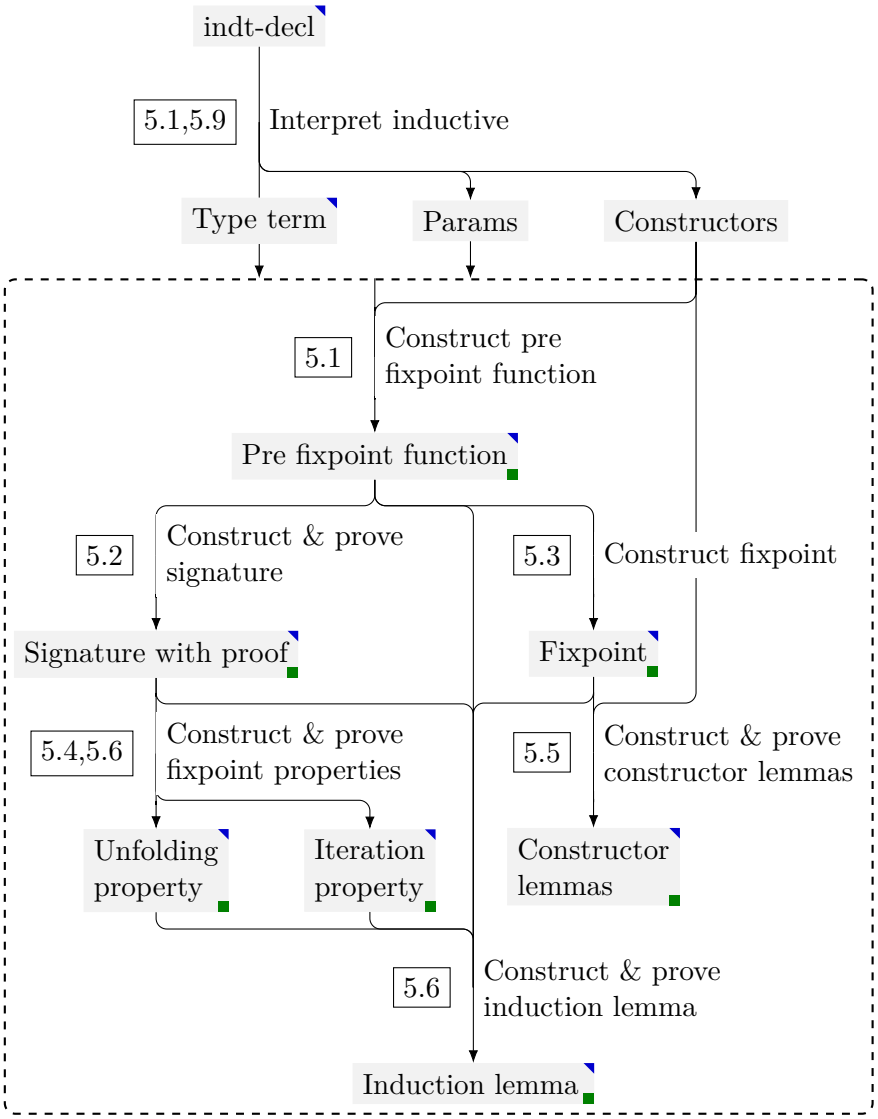

Figure 5.1: The structure of the `eiInd` command. Arrows are steps in the command and boxes are the objects that are being created. If a box has a green box, it is defined in Coq. If a box has a blue triangle, it is stored in the Elpi database. All arrows reference the section in which they are explained.

The inductive statement is received in Elpi as the following value of type `indt-decl`, unimportant fields of constructors are filled in with an `_`.

```
inductive `is_MLL` _
  (arity {{ val -> list val -> iProp }})
  (f \ [constructor `empty_is_MLL`
```

Elpi

```
Elpi
          (arity {{ lp:f NONEV [] }}),
        constructor `mark_is_MLL`
          (parameter `v` _ Type (v\ …)),
        constructor `cons_is_MLL`
          (parameter `v` _ Type (v\ …))])
```

The inductive consists of its name, `` `is_MLL` ``, its type, and a function containing the constructors with their names. Both the constructors and the type contain possible Coq binders. The constructor `mark_is_MLL` has the Coq binders `v`, `vs`, `l` and `tl`, these are represented with the parameter constructor. The parameter constructor takes the name, type and a function giving the rest of the term.

We first normalize the type and constructors into terms. Any Coq binders in the type are prepended as depended products, any Coq binders in the constructors are transformed into Iris existential quantifiers. The normalized type is referenced as the *type term* in this chapter. The resulting constructors under a binder become:

```
Elpi
f \ [
  {{ lp:f (InjLV #()) [] }},
  {{ ∃ v vs l tl, l ↦ (v, #true, tl) -* lp:f tl vs -*
                  lp:f (SOMEV #l) vs }},
  {{ ∃ v vs l tl, l ↦ (v, #false, tl) -* lp:f tl vs -*
                  lp:f (SOMEV #l) (v :: vs) }},
]
```

Next, we transform the constructors into the pre fixpoint function. This results in a function of the form of isMLL_F in example 3.5thesis.pdf. The first step is to replace the magic wands in the constructors with separating conjunctions. Then, we connect all constructors using Iris disjunctions. This results in the following intermediate Elpi function.

```
Elpi
Body = f \ {{
    lp:f (InjLV #()) []
  ∨ ∃ v vs l tl, l ↦ (v, #true, tl) ∗ lp:f tl vs ∗
                   lp:f (SOMEV #l) vs
  ∨ ∃ v vs l tl, l ↦ (v, #false, tl) ∗ lp:f tl vs ∗
                  lp:f (SOMEV #l) (v :: vs)
}}
```

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 term is transformed into a function by replacing dependent products, the `prod` constructor, with function, the `fun` constructor, in the type term. The function binders are now created as follows, where `FunTerm` is

the type term transformed into a function.

```
FunTerm b = {{ λ (v : val) (vs : list val), lp:b }},
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`, in which we fill 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. This results in the following term.

```
λ (F : val → list val → iProp) (v : val) (vs : list val),
  (⌜v = InjLV #()⌝ ∗ ⌜vs = []⌝
    ∨ (∃ (v' : val) (vs' : list val) l (tl : val),
          l ↦ (v', #true, tl) ∗ F tl vs' ∗
          ⌜v = InjRV #l⌝ ∗ ⌜vs = vs⌝)
    ∨ ∃ (v' : val) (vs' : list val) (tl : val) l,
          l ↦ (v', #false, tl) ∗ F tl vs' ∗
          ⌜v = InjRV #l⌝ ∗ ⌜vs = v' :: vs⌝)
```
Coq

This term is defined as `is_MLL_pre`.

5.2 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.

```
IProper (□> .> .> bi_wand ==> .> .> bi_wand)
        (is_MLL_pre)
```
Coq

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.

```
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).
```
Coq

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.

```
1 pred do-steps.do i:ihole, i:term, i:term, i:term.                    Elpi
2 do-steps.do IH R (app [F | FS]) _ :-
3   std.exists { std.iota {std.length FS} }
4              (n\ std.take n FS FS'),
5   do-iApplyProper IH R (app [F | FS']) HS, !,
6   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_MLL_pre_mono`. Thus allowing any further proof in the command and outside it to make use of the monotonicity of `is_MLL_pre`.

5.3 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_MLL`.

```
λ (v : val) (l : list val),
  (∀ F : val → list val → iProp,
    □ (∀ (v' : val) (l' : list val),
          is_MLL_pre F v' l' -∗ F v' l')
    -∗ F v l)
```

Coq

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:{{
  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.
}}.
```

Coq

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_MLL` in the database we add the rule:

```
inductive-pre (const «is_MLL»)
              (const «is_MLL_pre»)
```

Elpi

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.

5.4 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.

```Coq
∀ (v : val) (l : list val),
    is_MLL_pre (is_MLL) v l
  ⊢ is_MLL v l
```

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

```Elpi
pred mk-unfold.r->l i:int, i:int,
                    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`, which we discuss section 5.9, 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`.

```Elpi
pred mk-unfold.r->l.1 i:int, i:int,
                      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 val),                                    Coq
        is_MLL_pre F v l -∗ F v l
--------------------------------------------------□
"HF" : is_MLL_pre (is_MLL) l' v'
--------------------------------------------------∗
is_MLL_pre F l' v'
```

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

```
pred mk-unfold.r->l.2 i:int, i:int,                                   Elpi
                      i:term, i:term, i:ihole.
mk-unfold.r->l.2 Ps N Proper Mono IH :-
  ((copy {{ @IProper }} {{ @iProper }} :- !) =>
    copy Proper IProper'),
  type-to-fun IProper' IProper,
  std.map {std.iota Ps} (x\r\ r = {{ _ }}) Holes, !,
  do-iApplyLem (app [IProper | Holes]) IH [
    (h\ sigma PType\ sigma PProof\
        sigma List\ sigma Holes2\ !,
      h = hole PType PProof,
      std.iota Ps List,
      std.map List (x\r\ r = {{ _ }}) Holes2,
      coq.elaborate-skeleton (app [Mono | Holes2])
                             PType PProof ok,
  )] [IH1, IH2],
  do-iApplyHyp "HF" IH2 [], !,
  std.map {std.iota N} (x\r\ r = iPure none) Pures, !,
  do-iIntros
    {std.append [iModalIntro | Pures]
                [iIdent (iNamed "H"), iHyp "H",
                 iModalIntro, iHyp "HI"]}
    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 a statement we can apply to the goal. The complexity comes from dealing with parameters, which we discuss in section 5.9.

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.

```Coq
iProper (□> .> .> bi_wand ==> .> .> bi_wand)
        (is_MLL_pre)
```

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

```Coq
"HI" : ∀ (v : val) (vs : list val),
        is_MLL_pre f v vs -* f v vs
--------------------------------------------□
(□> .> .> bi_wand)%i_signature (is_MLL) f
```

Thus proving the right to left unfolding property. This proof together with the other two proofs of this section are defined as `is_MLL_unfold_1`, `is_MLL_unfold_2` and `is_MLL_unfold`.

5.5 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.

```Coq
∀ (v : val) (vs : list val),
  ⌜v = InjLV #()⌝ * ⌜vs = []⌝ -* is_MLL v vs

∀ (v : val) (vs : list val),
  (∃ (v : val) (vs : list val) l (tl : val),
        l ↦ (v, #true, tl) * is_MLL tl vs *
        ⌜v = InjRV #l⌝ * ⌜vs = vs⌝)
     -* is_MLL v vs

∀ (v : val) (vs : list val),
  (∃ (v : val) (vs : list val) (tl : val) l,
        l ↦ (v, #false, tl) * is_MLL tl vs *
```

[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.

```
          ⌜v = InjRV #l⌝ ∗ ⌜vs = v :: vs⌝)
      -∗ is_MLL v vs
```
Coq

Both constructor lemmas are a wand of the associated constructor to the fixpoint. They are defined with the name of their respective constructor, `empty_is_MLL`, `mark_is_MLL` and `cons_is_MLL`.

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

5.6 Iteration and induction lemmas

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

```
∀ Φ : val → list val → iProp,
    □ (∀ (v : val) (vs : list val),
         is_MLL_pre Φ v vs -∗ Φ v vs)
  -∗ ∀ (v : val) (vs : list val), is_MLL v vs -∗ Φ v vs
```
Coq

The induction lemma that we prove is:

```
∀ Φ : val → list val → iProp,
    □ (∀ (v : val) (vs : list val),
           is_MLL_pre
              (λ (v' : val) (vs' : list val),
                Φ v' vs' ∧ is_MLL v' vs')
              v vs
         -∗ Φ v vs)
  -∗ ∀ (v : val) (vs : list val), is_MLL v vs -∗ Φ v vs
```
Coq

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

5.7 `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.

```Coq
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.
```

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

```Coq
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 }]
```

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

```Coq
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 }]
```

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.

```Elpi
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.
```

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.

5.8 `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.

5.9 Parameters

The `eiInd` command is able to handle Coq binders for the whole Coq inductive statement, also called *parameters* in this chapter. Consider this modified inductive predicate for MLL.

```
EI.ind                                                              Coq
Inductive is_R_MLL {A} (R : val -> A -> iProp) :
                   val → list A → iProp :=
  | empty_is_R_MLL : is_R_MLL R NONEV []
  | mark_is_R_MLL v xs l tl :
      l ↦ (v, #true, tl) -∗ is_R_MLL R tl xs -∗
      is_R_MLL R (SOMEV #l) xs
  | cons_is_R_MLL v x xs tl l :
      l ↦ (v, #false, tl) -∗ R v x -∗
      is_R_MLL R tl xs -∗
      is_R_MLL R (SOMEV #l) (x :: xs).
```

Instead of equating the values in the MLL to a list of values, we instead

use an explicit relation to relate the values in the MLL to the list. To accomplish this we add two parameters, `{A}` and `(R : val -> A -> iProp)`. These values of `is_R_MLL` do not change during the inductive, and thus they are handled differently.

When receiving the inductive value in the command, the `inductive` constructor is wrapped in binders for each parameter. Thus, when interpreting the inductive statement we keep track of all binders of parameters and add the type of the binder to the Elpi type context.

Now, whenever we make a term which we define in Coq, we have to put add the parameters. Consider the pre fixpoint function of `is_R_MLL` before adding the fixpoints.

```
F' = {{                                                    Elpi
  λ (F : val → list lp:a → iProp)
    (v : val) (xs : list lp:a),
    …
  ∨ ∃ v' x xs' tl l,
      l ↦ (v', #false, tl) ∗ lp:r v' x ∗ F tl xs' ∗
      ⌜v = InjRV #l⌝ ∗ ⌜xs = x :: xs'⌝
}}
```

We only consider the interesting constructor. The term still contains Elpi binders which are not bound in the term. We solve this problem using the following Elpi predicate.

```
pred replace-params-bo i:list param, i:term, o:term.       Elpi
replace-params-bo [] T T.
replace-params-bo [(par ID _ Type C) | Params]
                  Term (fun N Type FTerm) :-
  replace-params-bo Params Term Term',
  (pi x\ (copy C x :- !) => copy Term' (FTerm x)),
  coq.id->name ID N.
```

It takes a list of parameters containing the name, type, and binder of the constant, and the term we want to bind parameters in. If there are still parameters left to bind, we first recursively bind the rest of the parameters. Next, we copy the term with the other parameters bound into the function `FTerm`, however when we encounter the parameter during copying we instead use the binder of `FTerm`. Lastly, we fix the type of the name of the parameter. The returned term is a Coq function based on `FTerm` and the name and type of the parameter. We have a similar predicate, `replace-params-ty` to bind parameters in dependent products, instead of lambda functions.

We make use of the above predicate to transform `F'` into the pre fixpoint function.

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

We use `replace-params-bo` and `replace-params-ty` to bind parameters in any terms created during `eiInd`. During proof generation we also need to keep parameters in mind. When applying lemmas generated during creation of the inductive predicate, we have to add holes for any parameters of the inductive predicate. An example of this procedure can be found on line 7 of `mk-unfold.r->l.2` in section 5.4.