Chapter 4

Implementing an Iris tactic in Elpi

In this chapter we will show how Elpi together with Coq-Elpi is used to create new Iris Proof Mode (IPM) tactics in Coq. This chapter explains the relevant inner working of IPM, give a tutorial on how Elpi works and how to create a tactic using Coq-Elpi, and finally set up the necessary functions for the commands and tactics around inductive predicates we will define in chapter 5thesis.pdf.

In section 4.1, we give a short recap of how the `iIntros` tactic functions. Next in section 4.2 we explain how the Iris context is implemented in IPM. Next, in section 4.3, we explain the Iris lemmas we use as the building blocks for the Elpi version of the tactic. In section 4.4 we explain how to use Elpi and Coq-Elpi while developing the `iIntros` tactic.

4.1 `iIntros` example

The IPM `iIntros` tactic acts as the `intros` tactic but on Iris propositions and the Iris contexts. The `intros` tactic takes as its first argument instructions in a domain-specific language (DSL). Based on these instructions, it performs several proof steps. The `iIntros` implements a similar DSL as the Coq tactic. A few expansions were added as inspired by ssreflect [HKP97; GMT16], they are used to perform other common initial proof steps such as `simpl`, `done` and others. We will show two examples of how `iIntros` is used to help prove lemmas.

We have seen in chapter 2thesis.pdf how we have two types of propositions as our assumptions during a proof. There are persistent and non-persistent (also called spatial from now on) propositions. In the IPM there are two corresponding contexts, the persistent and spatial context. Consider the following Coq lemma:

```coq
Lemma example1 : P -∗ □ Q -∗ P.
```

After applying `iIntros "HP #HQ"` we get

```coq
P, Q: iProp
============
"HQ" : Q
```

```
------------□
"HP" : P
------------∗
P
```

The tactic `iIntros "HP #HQ"` consist of two introduction patters applied after each other. `HP` introduces `P` intro the spatial context with the name `"HP"`. The `#HQ` introduces the next wand, but because of the `#` it is introduced into the persistent context (This fails if the proposition is not persistent).

The `iIntros` tactic also applies to universal quantifications, existential quantifications, separating conjunctions and disjunctions. Take the following proof state,

```
P: nat → iProp
==============================================
----------------------------------------------∗
∀ x : nat, (∃ y : nat, P x ∗ P y) ∨ P 0 -∗ P 1
```

We again use one application of `iIntros` to introduce and eliminate the premise.

```
iIntros "%x [[%y [Hx ?]] | H0]"
```

When applied we get two proof states, one for each side of the disjunction elimination.

```
(1/2)
P: nat → iProp
x, y: nat
==================
"Hx" : P x
"_" : P y
------------------∗
P 1

(2/2)
P: nat → iProp
x: nat
==================
"H0" : P 0
------------------∗
P 1
```

The intro pattern consists of multiple sub intro patterns. Each sub intro pattern starts with a universal quantifier introduction or wand introduction. We then interpret the intro pattern for the introduced hypothesis. A few of the possible intro patterns are:

- `"?"` uses an anonymous identifier for the hypothesis.
- `"H"` names the hypothesis 'H' in the spatial context.
- `"#H"` names the hypothesis 'H' in the persistent context.
- `"%H"` introduces the the hyptothesis into the Coq context with name 'H'
- `"[IPL | IPR]"` performs a disjunction elimination on the hypothesis. The two contained introduction patterns are recursively applied.

- `"[IPL IPR]"` performs a separating conjunction elimination on the hypothesis. The two contained introduction patterns are recursively applied.
- `"[%x IP]"` performs existential quantifier introduction on the hypothesis. The variable is name 'x' and `IP` is applied recursively. Note that this introduction pattern overlaps with previous pattern. This pattern is tried first.

We break down `iIntros "%x [[%y [Hx Hy]] | H0]"` into its components. We first forall introduce or first sub intro pattern `"%x"` and then perform the second case, introduce a pure Coq variable for the `∀ x : nat`. Next we wand introduce for the second sub intro pattern, `"[[%y [Hx Hy]] | H0]"` and interpret the outer pattern. it is the third case and eliminates the disjunction, resulting in two goals. The left patterns of the seperating conjunction pattern eliminates the exists and adds the `y` to the Coq context. Lastly, `"[Hx Hy]"` is the fourth case and eliminates the seperating conjunction in the Iris context by splitting it into two assumptions `"Hx"` and `"Hy"`.

There are more patterns available to introduce more complicated goals, these can be found in a paper written by Krebbers, Timany, and Birkedal [KTB17].

4.2 Contexts

Before describing our implementation of the Elpi `eiIntros` tactic, we need a quick interlude about how the Iris contexts and entailment are defined in Coq.

The IPM creates the context using the following definitions

```coq
Inductive ident :=
  | IAnon : positive → ident
  | INamed :> string → ident.

Inductive env : Type :=
  | Enil : env
  | Esnoc : env → ident → iProp → env.

Record envs := Envs {
  env_persistent : env;
  env_spatial : env;
  env_counter : positive;
}.
```

An identifier is either anonymous and given only a number, or a name using a string. Identifiers are mapped to propositions using `env`. This is a reversed linked list. Hence, new assumptions in an environment get added to the end of the list using `Esnoc`. The context consists of two such maps, one for the persistent hypotheses and one for the spatial hypotheses. Lastly, it contains a counter for creating fresh anonymous identifiers.

We now define how a context is interpreted in an entailment.

```coq
Definition envs_entails
    (Δ : envs iProp) (Q : iProp) : Prop :=
      ⌜envs_wf (env_intuitionistic Δ) (env_spatial Δ)⌝
    ∧ □ [∧] (env_intuitionistic Δ)
    ∧ [∗] (env_spatial Δ)
```

```
  ⊢ Q.
```

The persistent and spatial context are transformed into a proposition. The persistent context is combined using the iterated conjunction and surrounded by a persistence modality. The spatial context is simply combined using the iterated separating conjunction. Lastly, `envs_wf` ensures that every identifier only occurs once in the context.

Using `of_envs`, `envs_entails` defines entailment where the assumption is a context. Note that `envs_entails` is a Coq predicate, not a separation logic predicate. An `envs_entailment` statement is displayed as in section 4.1.

4.3 Tactics

To create the IPM tactics, lemmas are defined that apply a proof rule but transforms an `envs_entails` into another `envs_entails`.

```
1 Lemma tac_wand_intro Δ i P Q :
2   match envs_app false (Esnoc Enil i P) Δ with
3   | None => False
4   | Some Δ' => envs_entails Δ' Q
5   end →
6   envs_entails Δ (P -∗ Q).
```

The structure of wand introduction is still the same, if `P ⊢ Q` holds one line 4, `(P -∗ Q)` holds on line 6. However, the IPM needs to add `P` to the context, `Δ`, and handle the case when the chosen name, `i`, has already been used in the context. To add `P` to the context, the IPM uses the function `envs_app`. The first argument tells us to which context the second argument should be appended, `true` for the persistent context, and `false` for the spatial context. The second argument is the environment to append, and the third argument is the context to which we append. We first create a new environment containing just `P` with name `i` using `Esnoc`. Next, we add this environment to the existing context, `Δ`. This results in either `None`, when the name already exists in `Δ`, or `Some Δ'`, when we successfully add the new proposition. This new context is then used as the context for proving `Q`. A similar tactic is made for introducing persistent propositions, but it checks if `P` is also persistent and then adds it to that context.

Many more lemmas such as these are in the IPM. They are the core of many of the tactics we create in section 4.7 and chapter 5thesis.pdf.

4.4 Elpi

Our Elpi implementation `eiIntros` consists of three parts, as seen in figure 4.1. The first two parts interpret the DSL used to describe the proofs steps to be taken. Then, the last part applies these proofs steps. In section 4.5, we describe how a string is tokenized by the tokenizer. In section 4.6, we describe how a list of tokens is parsed into a list of intro patterns. In section 4.7, we describe how we use an intro pattern to introduce and eliminate the needed connectives. In every section we describe more parts of the Elpi programming language and the Coq-Elpi connector, starting with the base concepts of the language and working up to the mayor concepts of Elpi and Coq-Elpi.

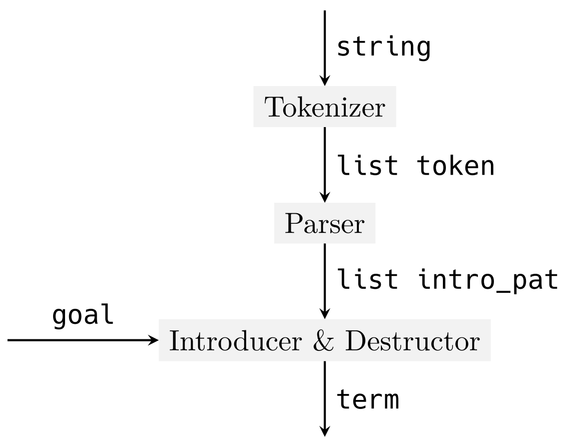

Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

4.5 Tokenizer

The tokenizer takes as input a string, which the tokenizer transforms into a list of tokens. Thus, the first step is to define our tokens. Next, we show how to define a predicate that transform our string into the tokens we defined.

4.5.1 Data types

The introduction patterns are separated into several distinct tokens. Most tokens just represent one or two characters, but some tokens also contain some data associated with that token. For example, `"H1"` is tokenized as the name token containing the string ”H1“.

Elpi

```
kind token type.

type tBar, tBracketL, tBracketR, tParenL, tParenR,
     tAmp, tAnon, tSimpl, tDone, tForall, tAll token.
type tName string -> token.
type tPure option string -> token.
```

We first define a new type called token using the `kind` keyword, where `type` specifies the kind of our new type. Next, we define several constructors for the token type. These constructors are defined using the `type` keyword, we specify a list of names for the constructors followed the type of the constructors. The first set of constructors do not take any arguments, thus have type `token`, and just represent one or more constant characters. The next few constructors take an argument and produce a token, thus allowing us to store data in the tokens. For example, `tName` has type `string -> token`, thus containing a string. Besides `string`, there are a few more basic types in Elpi such as `int`, `float` and `bool`. We also have higher kinded types, like `option`.

```elpi
kind option type -> type.
type none option A.
type some A -> option A.
```

Creating types of kind `type -> type` is done using the `kind` directive and passing in a more complicated kind as shown above. `list` is implemented similarly with standard notation.

Using the above types, we represent a given string as a list of tokens. Thus, given the string `"[H %H']"` we represent it as the following Elpi list of tokens

`[tBracketL, tName "H", tPure (some "H'"), tBracketR]`

4.5.2 Predicates

Programs in Elpi consist of predicates. Every predicate has several rules to describe the relation between its arguments.

```elpi
1  pred tokenize i:string, o:list token.
2  tokenize S O :-
3    rex.split "" S SS,
4    tokenize.rec SS O.
```

Line 1 describes the type of the predicate. The keyword `pred` starts the definition of a predicate. Next, we give the name of the predicate, "tokenize". Lastly, we give a list of arguments of our predicate. Each argument is marked as either `i:`, they act as an input or `o:`, they act as an output, in section 4.5.3 a more precise definition of input and output is given. This predicate has only one rule, defined on line 2. The variable `S` has type `string`. The variable `O` has type `list token`. By calling predicates after the `:-` symbol, we define the relation between the arguments. The first predicate we call, `rex.split`, splits the second argument by delimiters matching the regular expression in the first argument. The result is stored in the third argument. It has the following type

```elpi
pred rex.split i:string, i:string, o:list string.
```

We split the input string using the delimiter `""`, resulting in splitting the string into a list of its characters. Strings in Elpi are native data types and cannot be matched on, and thus we need to split it. The next line, line 4, calls the recursive tokenizer, `tokenizer.rec`[1], on the list of split strings and assigns the output to the output variable `O`.

The reason predicates in Elpi are called predicates and not functions, is that they don't always have to take an input and give an output. They are sometimes better considered as predicates, defining for which values of their arguments they hold. Each rule defines a list of predicates that need to hold for their premise to hold. Thus, a predicate can have multiple values for its output, as long as they hold for all contained rules. These multiple possible values can be reached by backtracking, which we will

[1]Names in Elpi can have special characters in them like `.`, `-` and `>`, thus, `tokenize` and `tokenize.rec` are fully separate predicates. It is just a convention that when creating a helper predicate we name it by adding a dot and a short name for the helper.

discuss in section 4.5.5. To execute a predicate, we thus find the first rule whose premise is sufficient for the arguments we supply. We then check if each of the predicates in the conclusion hold starting at the top. If they hold, we are done executing our predicate. How we determine when arguments are sufficient and what happens when a rule does not hold, we will discuss in the next two sections.

4.5.3 Matching and unification

The arguments of a predicate can be more than just a variable. We can supply a value containing variables and depending on the argument mode, input or output, we match or unify the input with the premise respectively[2].

The predicate `tokenize.rec` uses matching and unification to solve most cases.

Elpi

```
1  pred tokenize.rec i:list string, o:list token.
2  tokenize.rec [] [] :- !.
3  tokenize.rec [" " | SL] TS :- !, tokenize.rec SL TS.
4  tokenize.rec ["?" | SL] [tFresh | TS] :- !,
5    tokenize.rec SL TS.
6  tokenize.rec ["/", "/", "=" | SL]
7               [tSimpl, tDone | TS] :- !,
8    tokenize.rec SL TS.
9  tokenize.rec ["/", "/" | SL] [tDone | TS] :- !,
10   tokenize.rec SL TS.
```

The full predicate has rules for all tokens, a few rules are considered here. All rules use the *cut*, `!`, to prevent backtracking, see section 4.5.5, for now they can be ignored. When calling this predicate, the first rule is used when the first argument matches `[]` and if the second argument unifies with `[]`. The difference is that, for a value to match an argument, the value has to be equal or more specific than the argument. In other words, the value can only contain a variable if the argument also contains a variable at that place in the value. Thus, the only valid value for the first argument of the first rule is `[]`. When unifying two values, we allow the variable given to a predicate to be less specific than the argument. If that is the case, the variables are filled in until they match. Thus, we can either pass `[]` to the second argument, or some variable `V`. After the execution of the rule, the variable `V` will have the value `[]`.

The next four rules use the same principle. They take a list with the first few elements set. The output is unified with a list starting with the token that corresponds to the string we match on. The tails of the input and output are recursively computed.

When we encounter multiple rules that all match the arguments of a rule, we try the first one first. The rules on line 6 and 9 would both match the value `["/", "/", "="]` as first argument. But, we interpret this using the rule on line 6 since it is before the rule on line 9. This results in our list of strings being tokenized as `[tSimpl, tDone]`.

4.5.4 Functional programming in Elpi

While Elpi is based on predicates, we still often defer to a functional style of programming. The first language feature that is very useful for this goal is spilling. Spilling allows us

[2]A fun side effect of outputs being just variables we pass to a predicate is that we can also easily create a reversible function. If we change the mode of our first argument to output and move rule 3 to the bottom, we can pass in a list of tokens and get back a list of strings representing this list of tokens.

to write the entry point of the tokenizer as defined in section 4.5.2 without the need for temporary variables to be passed around.

Elpi
```
pred tokenize o:string, o:list token.
tokenize S O :- tokenize.rec {rex.split "" S} O.
```

We spill the output of a predicate into the input of another predicate by using the `{ }` syntax. We don't specify the last argument of the predicate, and only the last argument of a predicate can be spilled.

The second useful feature is how lambda expressions are first class citizens of the language. The `pred` statement is a wrapper around a constructor definition using `type`, with the addition of denoting arguments as inputs or outputs. When defining a predicate using `type`, all arguments are outputs. The following predicates have the same type.

Elpi
```
pred tokenize i:string, o:list token.
type tokenize string -> list token -> prop.
```

The `prop` type is the type of propositions, and with arguments they become predicates. We can thus write predicates that accept other predicates as arguments.

Elpi
```
1 pred map i:list A, i:(A -> B -> prop), o:list B.
2 map [] _ [].
3 map [X|XS] F [Y|YS] :- F X Y, map XS F YS.
```

`map` takes as its second argument a predicate on `A` and `B`. On line 3 we map this predicate to the variable `F`, and we then use it to either find a `Y` such that `F X Y` holds, or check if for a given `Y`, `F X Y` holds. We can use the same strategy to implement many of the common functional programming higher-order functions.

4.5.5 Backtracking

In this section we will finally describe what happens when a rule fails to complete halfway through. We start with a predicate which will be of much use for the last part of our tokenizer.

Elpi
```
1 pred take-while-split i:list A, i:(A -> prop),
2                       o:list A, o:list A.
3 take-while-split [X|XS] Pred [X|YS] ZS :- Pred X, !,
4   take-while-split XS Pred YS ZS.
5 take-while-split XS _ [] XS.
```

`take-while-split` is a predicate that should take elements of its input list until its input predicate no longer holds and then output the first part of input in its third argument and the last part of the input in its fourth argument.

The predicate contains two rules. The first rule, defined on lines 2 and 3, recurses as long as the input predicate, `Pred` holds for the input list, `[X|XS]`. The second rule returns the last part of the list. This rule is only considered if the first rule fails, thus when `Pred X` no longer holds.

The first rule destructs the input in its head `X` and its tail `XS`. It then checks if `Pred` holds for `X`, if it does, we continue the rule and call `take-while-split` on the tail while assigning X as the first element of the first output list and the output of the recursive call as the tail of the first output and the second output. However, if `Pred X` does not succeed, we backtrack. Any unification that happened because of the first rule is undone, and the next rule is tried. This will be the rule on line 4 and returns the input as the second output of the predicate.

Now, it might happen that the second rule also fails. If the second output variable does not unify with its input, the rule fails. This would let the whole execution of the predicate fail. Thus, the call on line 4 could fail, which would cause backtracking and an incorrect split of the input, `Pred X` holds but rule 2 is used. Thus, we make use of a cut, `!`, stopping backtracking. When a cut happens, any other possible rules in that execution of a predicate are discarded.

We use `take-while-split` to define the rule for the token `tName`.

Elpi

```
1 tokenize.rec SL [tName S | TS] :-
2   take-while-split SL is-identifier S' SL',
3   { std.length S' } > 0, !,
4   std.string.concat "" S' S,
5   tokenize.rec SL' TS.
6 tokenize.rec XS _ :- !,
7   coq.say "unrecognized tokens" XS, fail.
```

To tokenize a name, we first call `take-while-split` with as predicate `is-identifier`, which checks if a string is a valid identifier character, whether it is either a letter or one of a few symbols allowed in identifiers. It thus splits up the input string list into a list of string that is a valid identifier and the rest of the input. On line 5 we check if the length of the identifier is larger than 0. Next, on line 6, we concatenate the list of strings into one string, which will be our name. And on line 7, we call the tokenizer on the rest of the input, to create the rest of our tokens.

We also add a rule to give an error message when a token is not recognized on line 6. To ensure this rule is only called on the exact token that is not recognized, we need to not backtrack when a character is recognized, but the rest of the string is not. Thus, we add a cut to every rule when we know a token is correct.

4.6 Parser

The Parser uses the same language features as were used in the tokenizer. Thus, we won't go into detail of its workings. We create a type, `intro_pat`, to store the parse tree.

Elpi

```
kind ident type.
type iNamed string -> ident.
type iAnon term -> ident.

kind intro_pat type.
type iFresh, iSimpl, iDone intro_pat.
type iIdent ident -> intro_pat.
type iList list (list intro_pat) -> intro_pat.
```

Next, we use reductive descent parsing to parse the following grammar into the above data structure.

$$\begin{array}{lcl}
\langle \textit{intropattern\_list} \rangle & ::= & \epsilon \\
 & \mid & \langle \textit{intropattern} \rangle \ \langle \textit{intropattern\_list} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{intropattern} \rangle & ::= & \langle \textit{ident} \rangle \\
 & \mid & \text{‘}\texttt{?}\text{’} \mid \text{‘}\texttt{/=}\text{’} \mid \text{‘}\texttt{//}\text{’} \\
 & \mid & \text{‘}\texttt{[}\text{’} \ \langle \textit{intropattern\_list} \rangle \ \text{‘}\texttt{]}\text{’} \\
 & \mid & \text{‘}\texttt{(}\text{’} \ \langle \textit{intropattern\_conj\_list} \rangle \ \text{‘}\texttt{)}\text{’}
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{intropattern\_list} \rangle & ::= & \epsilon \\
 & \mid & \langle \textit{intropattern} \rangle \ \text{‘}\texttt{|}\text{’} \ \langle \textit{intropattern\_list} \rangle \\
 & \mid & \langle \textit{intropattern} \rangle \ \langle \textit{intropattern\_list} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{intropattern\_conj\_list} \rangle & ::= & \epsilon \\
 & \mid & \langle \textit{intropattern} \rangle \ \text{‘}\texttt{\&}\text{’} \ \langle \textit{intropattern\_conj\_list} \rangle
\end{array}$$

In order to make the parser be properly performant, it is important to minimize backtracking. Backtracking is necessary when implementing the second and third case of the ⟨*intropattern_list*⟩ parser. Backtracking might incur significant slowdowns due to reparsing frequently.

4.7 Applier

While creating the tokenizer and parser so far, we have only had to use standard Elpi. We will now be creating the applier. The applier will get a parsed intro pattern and use this to apply steps on the goal. Thus, we now have to communicate with Coq. We make use of Coq-Elpi [Tas18] to get a Coq API in Elpi.

To create a proof in Elpi we take the approach of building one large proof term. We apply this proof term to the goal at the end of the created tactic. We get into more details on this approach in section 4.7.3.

Before we get to building proofs, we first discuss how Coq terms and the Coq context are represented in Elpi in section 4.7.1. Lastly, we show how quotation and anti-quotation are used when building Coq terms in Elpi in section 4.7.2. Using the concepts in these sections, we explain creating proofs in Elpi in section 4.7.3. We discuss the structure of `eiIntros` in section 4.7.4. Lastly, in section 4.7.5 we show how a tactic is called and how a created proof is applied.

4.7.1 Coq-Elpi HOAS

Coq-Elpi makes use of Higher-order abstract syntax (HOAS) [PE88] in order to represent Coq terms in Elpi. Thus, it makes use of the binders in Elpi to represent binders in Coq terms. In this section we will discuss the structure of this HOAS and show how to call the Coq type checker in Elpi.

Take the following Coq term: `0+1`, which when expanding any notation becomes `Nat.add 0 (S 0)`. In Elpi this term is represented as follows.

```elpi
app [global (const «Nat.add»),
    global (indc «O»),
    app [global (indc «S»), global (indc «O»)]]
```

References to Coq object cannot be directly written as `«Nat.add»`. We will discuss how to create these objects in section 4.7.2.

The above Elpi term consists of several constructors. The first constructor is `app`, it is application of Coq terms. It gets a list, the tail of the list are the arguments and the head is what we are applying them to. Next, we have the `global` constructor. It takes a global reference of a Coq object and turns it into a term. Lastly, we have `const` and `indc`, these create a global reference of a constant or inductive constructor respectively.

Coq function terms work again similarly. Take the Coq term `fun (n: nat), n + 1`. This is represented in Elpi as follows.

```elpi
fun `n` (global (indt «nat»))
          (n \ app [global (indt «sum»),
                    n, app [global (indc «S»),
                            global (indc «O»)]])
```

The `fun` constructor takes three arguments. The name of the binder, here `n`. A term containing the type of the binder, `(global (indt «nat»))`. And, a function that produces a term, indicated by the lambda expression with as binder `n`. This is where the HOAS is applied. We use the Elpi lambda expression to encode the argument in the body of the function. Thus, `fun` has the following type definition.

```elpi
type fun name -> term -> (term -> term) -> term.
```

The type `name` is a special type of string. Names in Elpi are special strings which are convertible to any other string. Thus, any name equals any other name. Other Coq terms like `forall`, `let` and `fix` work in the same way.

Given that functions generating bodies of terms are integral to the Coq-Elpi data structures, we need the ability to move under a binder. To solve this, Elpi provides the `pi x\` quantifier. It allows us to introduce a fresh constant `c` any time the expression is evaluated. Take the following example, where we assign the above Coq function to the variable `FUN`.

```elpi
1 FUN = fun _ _ F,
2 pi x\ F x = app [A, B x, C]
```

On line 1 we store the function inside `FUN` in the variable `F`. Remember that the left and right-hand side of the equals sign are unified. Thus, we unify `FUN` with `fun _ _ F` and assign the function inside the `fun` constructor to `F`. On the next line, we create a fresh constant `x`, we now unify `F x` with `app [A, B x, C]`. The first and third element in the list of `app` are assigned to `A` and `C`. The second element of `app` is the binder of the function. Since `x` only exists in the scope of `pi x\`, we cannot just assign it to `B`. It might be used outside the scope of the `pi` quantifier. Thus, we make it a function. We unify `B x` with `x`, and `B` becomes the identity function.

We can call the Coq type checker from inside Elpi on any term. For the type checker to know the type of any binders we are under, it checks if a type is declared, `decl x N T`. Thus, we look for any `decl` rules which have as term `x` and store the name and type of `x` in `N` and `T`. However, now we need to add a rule when entering a binder to store the name and type of that binder. In the below code, `NAT` has the value `(global (indt «nat»))`.

```
pi x\ decl x `n` NAT
        => coq.typecheck (F x) Type ok.
```

Elpi

We make use of `=>` connective. The rule in front of `=>` is added on top of the known rules while executing the expressions behind `=>`. Thus, in the scope of `coq.typecheck`, we know that `x` has type `nat`. After type checking, `Type` has value `nat`.

4.7.2 Quotation and anti-quotation

To create terms, Coq-Elpi implements quotation and anti-quotation. This allows for writing Coq terms in Elpi. The Coq terms are parsed by the Coq parser in the context where the Elpi code is loaded in.

```
FUN = {{ fun (n: nat), n + 1 }}
```

Elpi

Now `FUN` has the value.

```
fun `n` (global (indt «nat»))
          (n \ app [global (indt «sum»),
                    n, app [global (indc «S»),
                            global (indc «O»)]])
```

Elpi

Coq-Elpi also allows for putting Elpi variables back into a Coq term. This is called anti-quotation.

```
FUN = {{ fun (n: nat), n + lp:C }}
```

Elpi

We extract the right-hand side of the plus operator in `FUN` into the variable `C` [3]. It thus has the same effect as what we did in the previous section to extract values out of a term. We can of course also use anti-quotation to insert previously calculated values into a term we are constructing.

These two ways of using anti-quotation will see much use when we create proofs in the next section, section 4.7.3. Where we create a proof term:

```
Proof = {{ tac_wand_intro _ lp:T _ _ _ _ _ }}
```

Elpi

After unifying `Proof` with the goal, we want to extract any newly created proof variables.

[3]We cannot do the same for the left-hand side of the addition. It contains a binder and thus can only be examined using the method seen in the previous section, section 4.7.1

```
Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:NewProof }},
```

Elpi

The new proof variable is extracted in the variable `NewProof`.

4.7.3 Proof steps in Elpi

Now that we have a solid foundation on how to work with Coq terms in Elpi we can start creating proof terms. Proof steps in Elpi are built by creating one big term which has the type of the goal. Any leftover holes in this term are new goals in Coq. To facilitate this process, we create a new type called `hole`.

```
kind hole type.
type hole term -> term -> hole.
```

Elpi

A `hole` contains two arguments. The goal, also called the type, is the first argument. The second argument is the proof variable, the variable to which we assign the proof term. Predicates that take and return holes are called *proof generators*. Take the following proof generator, it applies the iris ex falso rule to the current hole.

```
1 pred do-iExFalso i:hole, o:hole.
2 do-iExFalso (hole Type Proof)
3             (hole FalseType FalseProof) :-
4   coq.elaborate-skeleton
5     {{ tac_ex_falso _ _ _ }} Type Proof ok,
6   Proof = {{ tac_ex_falso _ _ lp:FalseProof }},
7   coq.typecheck FalseProof FalseType ok.
```

Elpi

The proof makes use of a variant of the ex falso rule, which is aware of contexts.

```
Lemma tac_ex_falso Δ Q :
  envs_entails Δ False →
  envs_entails Δ Q.
```

Coq

Thus, `tac_ex_falso` takes three arguments, the context, what we want to prove and a proof for `envs_entails Δ False`.

The Elpi code on lines 4-7 are the normal steps to apply a lemma. We make use of the Coq-Elpi API call, `coq.elaborate-skeleton` to apply this lemma to the hole. It elaborates the first argument against the type. The fully elaborated term is stored in the variable `Proof`. In this instance, `Proof` is the lemma with the Iris context filled in and a variable where the proof for `envs_entails Δ False` goes. Furthermore, the type information of any holes is added to the Elpi context. We extract this new proof variable on line 4. The proof variable is type checked to get the associated type of the proof variable using `coq.typecheck`. Together, these two variables for the new hole.

This is the structure of the most basic proof generators we use in our tactics. The concept of a hole allows for very composable proof generators. We will now discuss some more difficult proof generators. They will deal more directly with the iris context or introduce variables in the Coq context, and thus we need to create the rest of the proof under a binder.

Iris context counter

In section 4.2, we saw how anonymous assumptions are created in the iris context. We keep a counter in the context to ensure we can create a fresh anonymous identifier. This counter is convertible, allowing us to change it without doing changing the proof. In Elpi it is easier to keep track of this counter outside the context. We thus introduce a new type for an Iris hole.

Elpi
```
kind ihole type.
type ihole term -> hole -> ihole. % ihole counter hole
```

When we start the proof step, we take the current counter and store it. At then end of the proof, we set it again before returning it to Coq.

In a proof generator, we now simply use the counter in the `ihole` to generate a new identifier for an assumption. In any new `ihole`, we increase the counter by one.

Elpi
```
1  pred do-iIntro-anon i:ihole, o:ihole.
2  do-iIntro-anon (ihole N (hole Type Proof))
3                 (ihole N' (hole IType IProof)) :-
4    coq.reduction.vm.norm {{ Pos.succ lp:N }} _ N',
5    coq.elaborate-skeleton
6      {{ tac_wand_intro _ (IAnon lp:N) _ _ _ _ _ }}
7      Type Proof ok, !,
8    Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:IProof }},
9    coq.typecheck IProof IType' ok,
10   pm-reduce IType' IType.
```

The above proof generator introduces a wand into an anonymous hypothesis. On line 4 we increase the counter. Since the counter is a Coq term, we create a Coq term that increases the counter and execute it using `coq.reduction.vm.norm`. Next, using the old context counter, we create the identifier `(IAnon lp:N)`. We apply the lemma to the type of the hole and extract the new proof variable and type. Lastly, the created new proof types are often not fully normalized. The lemma we have applying has the following type.

Coq
```
1  Lemma tac_wand_intro Δ i P Q R :
2    FromWand R P Q →
3    match envs_app false (Esnoc Enil i P) Δ with
4    | None => False
5    | Some Δ' => envs_entails Δ' Q
6    end →
7    envs_entails Δ R.
```

The proof variable thus gets the type on lines 3-6. We normalize this using `pm-reduce`[4] to just `envs_entails Δ' Q` as long as the name was not already used.

[4] `pm-reduce` is also fully written in Elpi and is made extendable after definition of the tactics. To accomplish this Coq-Elpi databases are used with commands to add extra reduction rules to the database.

Continuation Passing Style

When introducing a universal quantifier in Coq, the proof term is a function. The new hole in the proof is now in the function. Thus, we are forced to continue the proof under the binder of the function in the proof term. To compose proof generators, we make use of continuation passing style (CPS) for these proof generators.

Elpi

```
pred do-intro i:string, i:hole, i:(hole -> prop).
do-intro ID (hole Type Proof) C :-
  coq.id->name ID N,
  coq.elaborate-skeleton (fun N _ _) Type Proof ok,
  Proof = (fun _ T IntroFProof),
  pi x\ decl x N T =>
    coq.typecheck (IntroFProof x) (FType x) ok,
    C (hole (FType x) (IntroFProof x)).
```

This proof generator introduces a Coq universal quantifier into the Coq context with the name `ID`. It first transforms the name, an Elpi string, into a Coq string term called `N`. Next we elaborate the proof term `fun (x: _), _` on `Type`. We extract the type of the binder in `T` and the function containing the new proof variable in `IntroFProof`. To move under the binder of the function we use the `pi` connective and then declare the name and type of `x` to the Coq context. Now can get the type of the proof variable. This might also depend on `x`, and thus it is also a function. Lastly, we call the continuation function with the new type and proof variable.

The unfortunate part of using CPS is that any predicates that use `do-intro` often also need to use CPS. Thus, we only use it when absolutely necessary.

4.7.4 Applying intro patterns

Now that we have defined multiple proof generators, we execute them depending on our intro patterns.

Elpi

```
pred do-iIntros i:(list intro_pat),
                i:ihole, i:(ihole -> prop).
do-iIntros [] IH C :- !, C IH.
do-iIntros [iFresh | IPS] IH C :- !,
  do-iIntro-anon IH IH', !,
  do-iIntros IPS IH' C.
do-iIntros [iPure (some X) | IPS] (ihole N H) C :-
  do-iForallIntro H H',
  do-intro X H
    (h\ sigma IntroProof\ sigma IntroType\
        sigma NormType\
        h = hole IntroType IntroProof,
        pm_reduce IntroType NormType, !,
        do-iIntros IPS
                   (ihole N (hole NormType IntroProof))
                   C
    ).
do-iIntros [iList IPS | IPSS] (ihole N H) C :- !,
  do-iIntro-anon (ihole N H) IH, !,
```

Elpi

```
20   do-iDestruct (iAnon N) (iList IPS) IH (ih'\ !,
21     do-iIntros IPSS ih' C
22   ).
```

This is a selection of the rules of the `do-iIntros` proof generator. The generator iterates over the intro patterns in the list. In the base case on line 3 it simply calls the continuation function. The second case, on line 4-6, simply calls a proof generator, in this case introducing an anonymous Iris assumption. Then, it continuous executing the rest of the intro patterns.

The third case, on lines 7-17, has three steps. First, it calls a proof generator that puts an Iris universal quantifier at the front of the goal as a Coq universal quantifier. This does not interact with the fresh counter, and thus we only give it a normal hole. Next we call `do-intro` as defined in section 4.7.3. This takes a continuation function which we define in lines 10-17. The hole this function gets, `h`, is not fully normalized. We thus need to access the type in the hole and reduce it. However, if we would just do `h = hole IntroType IntroProof` to extract the type from the hole, Elpi would give an error. By default, variables are created at the level of the predicate they are defined in. However, a predicate can only contain constants, by `pi x\`, created before they are defined. Thus, we make use of the quantifier `sigma X\` to instead define the variable in the continuation function. This ensures that the binder we are moving under is in scope when defining the variable. Once we have resolved that issue, we call `do-iIntros` on the rest of the intro patterns.

Question: The binders in variables is quite complicated, and I was hoping to skip this. But it is quite a downside of the CPS. I put it in for now. But maybe remove it.

For the fourth case, we will not go into too much detail, but just give an outline of what happens. This case covers the destruction intro patterns. These were parsed into an `iList` containing the destruction pattern. We first introduce the assumption we want to destroy with an anonymous name. Next, we call `do-iDestruct` to do the destruction. This can create multiple holes in the process, and the continuation function we pass it will be executed at the end of all of them. The predicate `do-iDestruct` has the same structure as `do-iIntros`, and we will see it in ?? when we discuss the destruction of inductive predicates.

4.7.5 Starting the tactic

The entry point of a tactic in Elpi is the `solve` predicate.

Elpi

```
solve (goal _ _ Type Proof [str Args]) GS :-
  tokenize Args T, !,
  parse_ipl T IPS, !,
  do-iStartProof (hole Type Proof) IH, !,
  do-iIntros IPS IH (ih\ set-ctx-count-proof ih _), !,
  coq.ltac.collect-goals Proof GL SG,
  all (open pm-reduce-goal) GL GL',
  std.append GL' SG GS.
```

The entry point takes a goal, which contains the type of the goal, the proof variable, and any arguments we gave. We then tokenize and parse the argument such that we have an intro pattern to apply. We use the start proof, proof generator to transform the goal into an `envs_entails` goal and get the context counter. And we are ready to use `do-iIntros` to apply the intro pattern. At the end, set the correct context counter in the proof. We now have a proof term in the `Proof` variable that we want to return to Coq. We make use of several Coq-Elpi predicates to accomplish this. First, collect all holes in the proof term and transform them into objects of the type `goal` in the lists `GL`, `SG`. The two lists are the normal goals and the shelved goals, goals Coq expects to be solved during proving of the normal goals[5]. This step uses type checking to create the type of the goals, and thus they are not normalized, on line 7 we normalize all main goals. Lastly, we combine the two lists again and return then to Coq using the variable `GS`.

[5]Goals in Coq-Elpi can either be sealed or opened. A sealed goal contains all binders for the context of the goal in the goal. A goal is opened by going under all the binders and adding all the types of the binders as rules. The sealing of goals to pass them around is necessary when you can make no assumptions on what happens to the context of a goal, and is thus the model used for the entry point of Coq-Elpi. However, in our proof generators we know when new things are added to the context, and thus we can take a more specialized approach using CPS.