Chapter 4

Implementing an Iris tactic in Elpi

In this chapter we will show how Elpi together with Coq-Elpi can be used to create new (IPM) tactics in Coq. This chapter will function explain relevant inner working of the 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 **??**.

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 the IPM. And, in section 4.3 we explain the Iris lemmas we use as the building blocks for our 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. It implements a similar domain specific language (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` can be 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) proposition. In the IPM there are two corresponding contexts, the persistent and spatial context. Consider the statement $\vdash P \twoheadrightarrow \square Q \twoheadrightarrow P$. As a Coq goal this would be

```Coq
P, Q: iProp
============
------------*
```

```Coq
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).

This does not only work on the magic wand, we can also use this to introduce more complicated statements. Take the following proof state,

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

It consists of a universal quantification, an existential quantification, a separating conjunction and a disjunction. We can again use one application of `iIntros` to introduce and eliminate the premise.

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

When applied we get two proof states, one for each side of the disjunction elimination. These different proof states are shown with the `(1/2)` and `(2/2)` prefixes.

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

(2/2)
P: nat → iProp
x: nat
```

```
==================                                                   Coq
"H0" : P 0
------------------*
P 1
```

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

- `"H"` represents renaming a hypothesis. The name given is used as the name of the hypothesis in the spatial context.

- `"%H"` represents pure elimination. The introduced hypothesis is interpreted as a Coq hypothesis, and added to the Coq context.

- `"[IPL | IPR]"` represents disjunction elimination. We perform a disjunction elimination on the introduced hypothesis. Then, we apply the two included intro patterns two the two cases created by the disjunction elimination.

- `"[IPL IPR]"` represents separating conjunction elimination. We perform a separating conjunction elimination. Then, we apply the two included intro patterns two the two hypotheses by the separating conjunction elimination.

- `"[%x IP]"` represents existential elimination. If first element of a separating conjunction pattern is a pure elimination we first try to eliminate an exists in the hypothesis and apply the included intro pattern on the resulting hypothesis. If that does not succeed we do a conjunction elimination.

Thus, we can 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 starting the Elpi `eiIntros` tactic we need a quick interlude about how the Iris contexts and entailment are made in Coq.

In separation logic we have the following statement

$$\Box P * Q \vdash R$$

This statement can be immediately written in Coq.

```coq
□ P ∗ Q ⊢ R
```

However, now we want to use named contexts as we saw in the previous section, thus give names to both `P` and `Q`. We start by creating an environment, giving names to propositions.

```coq
Inductive env (A : Type) : Type :=
  | Enil : env A
  | Esnoc : env A → ident → A → env A.
```

This is a reversed linked list. Hence, new assumptions in an environment get added to the end of the list using `Esnoc`. Using these environments we can define a context.

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

Just the two environments would allow us to give a context where all assumptions have names. However, it is also very useful to have anonymous assumptions. We thus allow our identifier to be either a name or a number.

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

To allow for creating fresh anonymous identifiers we have to know which numbers are already used. Thus, the context also contains a counter which holds the next available number for an anonymous assumption. This is the `env_counter`.

To allow for using this context as the assumption of an entailment we create a predicate `of_envs`.

```coq
Definition of_envs
    (Γp Γs : env iProp) : iProp :=
  □ [∧] Γp ∧ [∗] Γs.
```

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. Using the predicate we can create the final entailment from a context.

```coq
Definition envs_entails
    (Δ : envs iProp) (Q : iProp) : Prop :=
  of_envs (env_intuitionistic Δ) (env_spatial Δ) ⊢ Q.
```

Note `envs_entails` is a Coq predicate, not a separation logic predicate. It holds if the interpreted environment, `Δ`, entails the conclusion, `Q`. To allows for easily interpreting such an entailment it is written down as follows for our original statement.

```coq
P, Q, R: iProp
============
"HP" : P
------------□
"HR" : Q
------------∗
R
```

4.3 Tactics

The proof rules as defined in chapter 2thesis.pdf don't work easily with the new entailment we defined in the previous section. We thus define lemmas that work with the context once which can be used in further proofs. We have already seen one lemma that made the proof rules usable, WP-APPLY. This rule abstracted away the difference between Hoare triples and weakest preconditions. We now show how the wand introduction, −∗I-E, can be used with context.

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

The structure of wand introduction is still the same, given `Q` holds one line 4, `(P -∗ Q)` holds on line 6. However, Iris 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, Iris uses the function `envs_app`. The first argument tell 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 can then be 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 Iris in order to use the proof rules while also using the named context. We will also make use of them many times while creating any tactics, and they will appear many times in section 4.7.

4.4 Elpi

We implement our tactic in the λProlog language Elpi [Dun+15; GCT19]. Elpi implements λprolog [MN86; Mil+91; BBR99; MN12] and adds constraint handling rules to it [Mon11]. constraint handling will be explained in Section ?.

TODO: Defer constraint handling to later

To use Elpi as a Coq meta programming language, there exists the Elpi Coq connector, Coq-Elpi [Tas18]. We will use Coq-Elpi to implement the Elpi variant of `iIntros`, named `eiIntros`.

Our Elpi implementation `eiIntros` consists of three parts as seen in figure 4.1. The first two parts will interpret the DSL used to describe what we want to introduce. Then, the last part will apply the interpreted DSL. 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.

4.5 Tokenizer

The tokenizer takes as input a string. We will interpret every symbol in the string and produce a list of tokens from this string. Thus, the first step is to

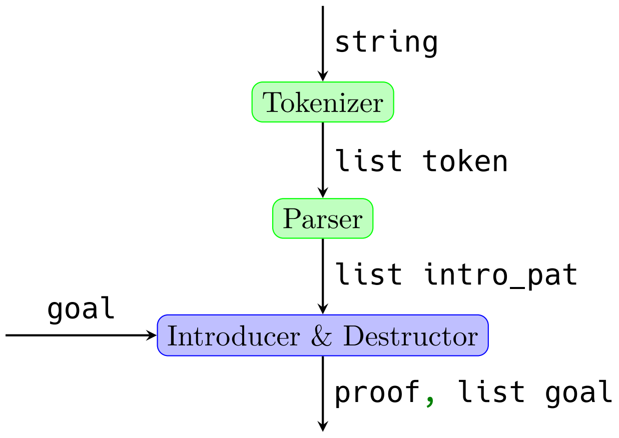

Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

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

We have separated the introduction patterns 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 tAnon, tFrame, tBar, tBracketL, tBracketR, tAmp,
     tParenL, tParenR, tBraceL, tBraceR, tSimpl,
     tDone, tForall, tAll token.
type tName string -> token.
type tNat int -> token.
type tPure option string -> token.
type tArrow direction -> token.

kind direction type.
type left, right direction.
```

We first define a new type called token using the `kind` keyword, where `type` specifies the kind of our new type. Then 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 and then the type of those 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 order types, like `option A`, and later on `list A`.

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

Creating types of kind `type -> type` can be done using the `kind` directive and passing in a more complicated kind as shown above.

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

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

4.5.2 Predicates

Programs in Elpi consist of predicates. Every predicate can have several rules to describe the relation between its inputs and outputs.

```
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 is given. In the only rule of our predicate, defined on line 2, we assign a variable to both of the arguments. `S` has type `string` and is bound to the first argument. `O` has type `list token` and is bound to the second argument. By calling predicates after the `:-` symbol we can define the relation between the arguments. The first predicate we call, `rex.split`, has the following type:

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

When we call it, we assign the empty string to its first argument, the string we want to tokenize to the second argument, and we store the output list of string in the new variable `SS`. This predicate allows us to split a string at a certain delimiter. We take as delimiter the empty string, thus splitting the string up in a list of strings of one character each. Strings in Elpi are based on OCaml strings and are not lists of characters. Since Elpi does not support pattern matching on partial strings, we need this workaround.

The next line, line 4, calls the recursive tokenizer, `tokenizer.rec` [1], on the list of split string 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 can sometimes better be seen 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 discuss in section 4.5.5. To execute a predicate, we thus find the first rule for which its 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, and we get a value for every output argument, 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] [tFrame | TS] :- !,
5   tokenize.rec SL TS.
6 tokenize.rec ["/", "/", "=" | SL]
7              [tSimpl, tDone | TS] :- !,
8   tokenize.rec SL TS.
```

[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.

[2]A fun side effect of outputs being just variables we pass to a predicate is that we can also easily create a function that is reversible. 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.

```
tokenize.rec ["/", "/" | SL] [tDone | TS] :- !,
  tokenize.rec SL TS.
```

This predicate has several rules, we chose a few to highlight here. The first rule, on line 2, has a premise and a cut as its conclusion, we will discuss cuts in section 4.5.5, for now they can be ignored. This rule can be used when the first argument matches `[]` and if the second argument unifies with `[]`. The difference is that, for two values to match they must have the exact same constructors and can only contain variables in the same places in the value. Thus, the only valid value for the first argument of the first rule is `[]`. When unifying two values we allow a variable to be unified with a constructor, when this happens the variable will get assigned the value of the constructor. 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 use the list pattern `[E1, ..., En | TL]`, where `E1` to `En` are the first n values and `TL` is the rest of the list, to match on the first few elements of the list. We unify the output with a list starting with the token that corresponds to the string we match on. The tails of the input and output we pass to the recursive call of the predicate to solve.

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 use 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 our language 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 to write the entry point of the tokenizer as defined in section 4.5.2 without the need of the temporary variable to pass the list of strings around.

```
pred tokenize i: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. It is mostly equal to the previous version, but just written shorter. There is one caveat, but it will be discussed in ?.

TODO: Refer to relevant section

The second useful feature is how lambda expressions are first class citizens of the language. A `pred` statement is a wrapper around a constructor definition using the keyword `type`, where all arguments are in output mode. The following predicate is equal to the type definition below it.

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

The `prop` type is the type of propositions, and with arguments they become predicates. We are thus able to 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 till 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 as soon as `Pred` 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 to the previous rule in our conclusion. Since there is no previous rule in the conclusion we instead undo any unification that has happened and try the next possible rule. This will be the rule on line 4 and returns the input as the second output of the predicate.

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

Elpi

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

To tokenize a name we first call `take-while-split` with as predicate `is-identifier`, which checks if a string is 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. We do this by spilling the length of `S'` into the `>` predicate. 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.

If our length check does not succeed we backtrack to next rule that matches, which is

Elpi

```
tokenize.rec XS _ :- !,
  coq.say "unrecognized tokens" XS, fail.
```

It prints an error messages saying that the input was not recognized as a valid token, after which it fails. The predicate thus does not succeed. There is one problem, if line 6 or 7 fails for some reason in the `tName` rule of the tokenizer, the current input starting at `X` is not unrecognized as we managed to find a token for the name at the start of the input. Thus, we don't want to backtrack to another rule of `tokenize.rec` when we have found a valid name token. This is where the cut symbol, `!`, comes in. It cuts the backtracking and makes certain that if we fail beyond that point we don't backtrack in this predicate.

If we take the following example

Elpi

```
1 tokenize.rec ["H","^"] TS
2                 ⇓ calls
3 tokenize.rec ["^"] TS'
```

When evaluating this predicate we would first apply the name rule of the `tokenize.rec` predicate. This would unify `TS` with `[tName "H" | TS']` and call line 3, `tokenize.rec ["^"] TS'`. Every rule of `tokenize.rec` fails including the last fail rule. This rule does first print `"unrecognized tokens ^"` but then also fails. Now when executing the rule of line 1, we have failed on the last predicate of the rule. If there was no cut before it, we would backtrack to the fail rule and also print `"unrecognized tokens [H, ^]"`. But, because there is a cut we don't print the faulty error message. Thus, we only print meaningful error message when we fail to tokenize an input.

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 make use a reductive descent parsing in order to parse the following grammar into the above data structure.

⟨*intropattern_list*⟩ ::= ϵ
  | ⟨*intropattern*⟩ ⟨*intropattern_list*⟩

⟨*intropattern*⟩ ::= ⟨*ident*⟩
  | '`?`' | '`/=`' | '`//`'
  | '`[`' ⟨*intropattern_list*⟩ '`]`'
  | '`(`' ⟨*intropattern_conj_list*⟩ '`)`'

⟨*intropattern_list*⟩ ::= ϵ
  | ⟨*intropattern*⟩ '`|`' ⟨*intropattern_list*⟩
  | ⟨*intropattern*⟩ ⟨*intropattern_list*⟩

$\langle$*intropattern_conj_list*$\rangle$ ::= ϵ
| $\langle$*intropattern*$\rangle$ '&' $\langle$*intropattern_conj_list*$\rangle$

In order to make the parser be properly performant it is important to minimize backtracking. Backtracking can 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 can 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 can be 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. In section 4.7.4 we show how we can create proofs which add things to the context. We discuss backtracking during proofs in **??**. Lastly, in section 4.7.6 we show how a tactic is called and how a created proof can be applied.

4.7.1 Coq-Elpi HOAS

Coq-Elpi makes use of Higher-order abstract syntax (HOAS) 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 typechecker in Elpi.

Take the following Coq term: `0+1`. In Elpi this term is created as follows.

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

This 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. The name they take is not a manipulable string from Elpi. We will discuss how these are made in section 4.7.2.

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

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

This time we get the `fun` constructor. It takes three arguments. The name of the binder, here `n`. The type of the binder, `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[3]. Other Coq terms like `forall`, `let` and `fix` work in the same way.

Now that functions generating bodies of terms are integral in the Coq-Elpi data structures we need the ability to move under a binder. In order to do this, Elpi provides the `pi x\` quantifier. It allows us to introduce a fresh constant `c` any time the expression is evaluated. Given the definition of `FUN` in the previous section, take the following piece of code that continuous where it left of.

```Elpi
5  FUN = fun _ _ F,
6  pi x\ F x = app [A, B x, C]
```

On line 5 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 can't just assign it to `B`. It might be used outside of the scope of the `pi` quantifier. Thus, we make it a function. We unify `B x` with `x`, and `B` becomes the identity function.

[3]Names in Elpi are special strings which are convertible to any other string. Thus, any name equals any other name. These are signified by the type `name`.

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.

```elpi
pi x\ decl x `n` (global (indt «nat»))
        => coq.typecheck (F x) Type ok.
```

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

4.7.2 Quotation and anti-quotation

Writing terms in Elpi is often overly verbose, thus Coq-Elpi implements quotation and anti-quotation, which 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.

```elpi
{{ fun (n: nat), n + 1 }} =
  fun `n` (global (indt «nat»)) c0 \
      app [global (indt «sum»),
          c0,
          app [global (indc «S»), global (indc «O»)]]
```

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

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

We extract the right-hand side of the plus operator in `FUN` into the variable `C`. 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:

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

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

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

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

4.7.3 Proof steps in Elpi

Now that we have a solid foundation how to work with Coq terms in Elpi we can start creating proof terms. Proof steps in Elpi are build 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. % hole Type Proof
```

A `hole` contains the type of what we are currently trying to prove in our code together with the proof variable we need to assign the proof for the type to. Any predicates we define that create some part of the proof term take a hole and depending on if they create new goals also return holes. Take the following proof term generator that applies the iris ex falso rule to the current hole.

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

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.
```

Thus, `tac_ex_falso` takes three arguments, the context, what we want to prove and a proof for `envs_entails Δ False`. 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. During the elaboration process a new term is created which is the fully elaborated term. In this case `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 context. We extract this new proof variable on line 4. We can type check the proof variable to get the associated type of the proof variable. Together these two variables for the new hole we return.

This is the structure of 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 assumption 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. This was necessary for the LTaC implementation of Iris. However, in Elpi it is a lot easier to pass around this counter outside the hole. 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 can set it in the type before returning it to Coq.

In a proof generator we can 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 context counter by normalizing the successor of the current counter. Since the counter is kept as a Coq term, we have to use the Coq successor. 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 3-5. We can normalize this using `pm-reduce` [4] to just `envs_entails Δ' Q`.

4.7.4 Continuation Passing Style

When introducing a universal quantifier in Coq the proof term is a function taking a variable of the type of the quantification. The new hole in the proof is now in the function. When creating proof terms such as these we are forced to continue the proof under the binder of the function in the proof term. To solve this problem 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 variable into the Coq context with the name `ID`. It first transforms the 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 again 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 need to also use CPS. Thus, we only use it when ab-

[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.

solutely necessary.

4.7.5 Applying intro patterns

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

```
                                                          Elpi
 1  pred do-iIntros i:(list intro_pat),
 2                  i:ihole, i:(ihole -> prop).
 3  do-iIntros [] IH C :- !, C IH.
 4  do-iIntros [iFresh | IPS] IH C :- !,
 5    do-iIntro-anon IH IH', !,
 6    do-iIntros IPS IH' C.
 7  do-iIntros [iPure (some X) | IPS] (ihole N H) C :-
 8    do-iForallIntro H H',
 9    do-intro X H
10      (h\ sigma IntroProof\ sigma IntroType\
11          sigma NormType\
12          h = hole IntroType IntroProof,
13          pm_reduce IntroType NormType, !,
14          do-iIntros IPS
15                    (ihole N (hole NormType IntroProof))
16                    C
17      ).
18  do-iIntros [iList IPS | IPSS] (ihole N H) C :- !,
19    do-iIntro-anon (ihole N H) IH, !,
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 anonymous assumption, thus we only give it a normal hole. Next we call `do-intro` as defined in section 4.7.4. 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

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.

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 fixed that issue, we can call `do-iIntros` on the rest of the intro patterns.

For the fourth case we will not go in to too much detail but just give an outline of what happens. This case covers all 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.

4.7.6 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 en 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

[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

type of the goals, and thus they are not normalized, on line 7 we normalize all normal goals. Lastly we combine the two lists again and return then to Coq using the variable `GS`.

more specialized approach using CPS.