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 tactics in Coq. We will do this by giving a tutorial on how to implement the `iIntros` tactic from Iris.

4.1 `iIntros` example

The tactic `iIntros` is based on the Coq **`intros`** tactic. The Coq **`intros`** tactic makes use of a domain specific language (DSL) for quickly introducing different logical connective. In Iris this concept was adopted for the `iIntros` tactic, but modified to the Iris contexts. Also, a few expansions, as inspired by ssreflect [HKP97; GMT16], were added to perform other common initial proof steps such as **`simpl`**, **`done`** and others. We will show a few examples of how `iIntros` can be used to help prove lemmas.

We have seen in chapter 2 how we often 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 Coq assumption management is a very important part of writing proofs. Thus, in Coq implementation of the separation logic Iris, theses two types of assumptions have been made into two contexts, the persistent and the spatial context. Together with the Coq context, we thus have three context. As an example given we have the separation logic statement.

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

This would be shown in Iris as the following proof state.

```
P, Q, R: iProp
============
```

```
"HP" : P
------------□
"HR" : Q
------------∗
R
```

Above the double lined line we have the types of all our proof variables and any other statements in the Coq logic. Next we have a section of persistent proposition we have as assumptions, each one named. The assumption P is thus named `"HP"`. Following the persistent context we have the spatial context, where again each assumption is named. At the bottom we have the statement we want to prove. We will now show how the `iIntros` tactic modifies these contexts. Given the below proof state, we would want to introduce P and Q.

```
P, Q: iProp
============
------------∗
P -∗ Q -∗ P
```

We can use `iIntros "HP HQ"`, this will intelligently apply $-\!*$I-E twice.

```
P, Q: iProp
============
"HP" : P
"HQ" : Q
------------∗
P
```

We have introduced the two separation logic propositions into the spatial context. This does not only work on the magic wand, we can also use this to introduce more complicated statements. Take the following proof state,

```
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 seperating 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.

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

```
□ 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`. The contexts are encoded as pairs of identifiers with a proposition and put together in a record containing both contexts

```
Record envs (PROP : bi) := Envs {
  env_persistent : env PROP;
  env_spatial : env PROP;
  env_counter : positive;
}.
```

Question: Should I use *iProp* or `PROP` here?

Just the two contexts 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, as seen in the definition of `ident`.

```
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`.

```
Definition of_envs {PROP : bi}
    (Γp Γs : env PROP) : PROP :=
  (□ [∧] Γp ∧ [∗] Γs)%I.
```

The persistent context is combined using conjunctions and surrounded by a persistence modality. The spatial context is simply combined using separating conjunctions. Using the predicate we can create the final entailment from a context.

```
Definition envs_entails {PROP : bi}
    (Δ : envs PROP) (Q : PROP) : 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.

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

4.3 Tactics

The proof rules as defined in chapter 2 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.

```
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, 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` (this is just `consE` but backwards). 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

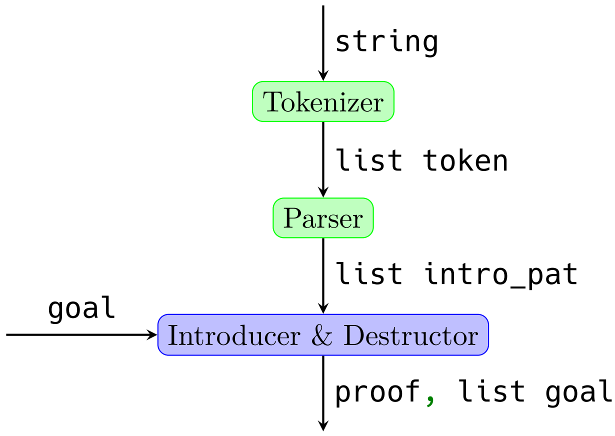

Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

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 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”.

```
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.

```
pred tokenize i:string, o:list token.
tokenize S O :-
  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:

```
1  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.

[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.

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.

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

```
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] [tSimpl, tDone | TS] :- !,
7   tokenize.rec SL TS.
8 tokenize.rec ["/", "/" | SL] [tDone | TS] :- !,
9   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 8 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 8. This results in our list of strings being tokenized as `[tSimpl, tDone]`.

A fun side effect of output 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.

Question: Don't know what to do with this, but is an interesting fact and shows the versatility, we might use it later.

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.

```
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 are thus able to write predicates that accept other predicates as arguments.

```
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.

```
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`.

```
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

```
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

```
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.

```
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.

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

In order to make the parser be properly performant it is important to minimize backtracking. Backtracking can incur significant slowdowns due to reparsing frequently.

4.7 Applier

- Only used standard Elpi so far
- Now use Coq-Elpi
- What Coq-Elpi adds
- Section overview

4.7.1 Elpi coq HOAS

- First step, represent Coq terms in Elpi
- Names and function application are just constructors

```
1+1
```

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

- Explain app, global, const, indc and «»
- Coq-Elpi uses higher-order abstract syntax (HOAS)
- functions in Coq are functions that produce terms in Coq-Elpi

```
fun (n: nat), n + 1
```

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

- fun constructor taking name, type and function producing term
- footnote about names all being convertible

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

- prod, let, fix work the same

4.7.2 Coq context in Elpi

- Looking at terms in functions becomes hard as we need to give the function an input to get the term
- introduce fresh constant using `pi x\`

```
FUN = fun _ _ F,
pi x\ F x = app [_, _, P],
P = app [global (indc «S»), global (indc «O»)]
```

- Take function out of constructor
- Fill in function with existential variable to inspect contents
- Take out number we add
- We lose type and name information about x

```
pred decl i:term, o:name, o:term.
decl x `n` (global (indt «nat»)).
```

- decl rule describes types and names of variables
- Lookup type using `decl x N T`
- We have to add the rule when we define x

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

- We add a rule to the top of the rules for the execution of the code after the `=>`
- During typechecking, `decl x N T` is executed resulting in ...
- `Type` becomes (global (indt «nat»))
- `=>` has many more uses later on

4.7.3 Quotation and anti-quotation

- Writing terms is a lot of work
- Coq-Elpi allows us to write Coq code that is translated immediately using imports in current file

```
{{ λ (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 putting Elpi vars in Coq terms (anti quotation)

```
{{ @envs_entails lp:PROP (@Envs lp:PROPE lp:CI lp:CS lp:N) lp:P }}
```

- Extract values from term
- Insert values in term, useful in proofs

```
{{ as_emp_valid_2 lp:Type _ (tac_start _ _) }}
```

- Lemma useful in next section
- Type is type of goal we want to proof
- Term becomes lemma we can apply to goal

4.7.4 Proofs in Elpi

- Proofs in Elpi built up proof term step by step
- Pass around Type of goal and variable to assign proof term to
- This is hole

```
kind hole type.
type hole term -> term -> hole. % hole Type Proof
```

- Proofs take a hole and often produce new holes
- Following proof step applies the ex-Falso proof step
- Replace type with False

```
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.
```

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

- Elaborate Lemma against type to generate proof term will be Lemma filled in with necessary values
- Next, extract New proof variable
- Get type of new proof variable

Iris context counter

- Iris can have anonymous hypotheses in context
- Keep track of number to assign to anon hypothesis
- Normally in Type
- Since we derive the type from the proof term we have to apply increases in this number in the proof term
- Instead we keep track of it separately

```
pred do-iStartProof i:hole, o:ihole.
do-iStartProof (hole Type Proof) (ihole N (hole NType NProof)) :-
  coq.elaborate-skeleton {{ as_emp_valid_2 lp:Type _ (tac_start _ _) 
  Proof = {{ as_emp_valid_2 _ _ (tac_start _ lp:NProof) }},
  coq.typecheck NProof NType ok,
  NType = {{ envs_entails (Envs _ _ lp:N) _}}.
```

- Start proof applies start proof lemma
- Next extracts current anon hypotheses count
- Stores it in hole using new type ihole

```
kind ihole type.
type ihole term -> hole -> ihole. % ihole iris hyp counter, (hole typ
```

- Counter is Coq positive since increasing it is fairly easy

```
pred increase-ctx-count i:term, o:term.
increase-ctx-count N NS :-
  coq.reduction.vm.norm {{ Pos.succ lp:N }} _ NS.
```

- We can increase counter and put it in the resulting `ihole` when necessary.

4.7.5 Continuation Passing Style

- When introducing a forall we need to add the variable to our context
- Next steps in the proof thus need the new value in the context
- We have to use continuation passing style

```
pred do-intro-anon i:hole, i:(hole -> prop).
do-intro-anon (hole Type Proof) C :-
  coq.ltac.fresh-id "a" {{ False }} ID,
  coq.id->name ID N,
  coq.elaborate-skeleton (fun N _ _) Type Proof ok,
  Proof = (fun _ T IntroFProof),
  @pi-decl N T x\
    coq.typecheck (IntroFProof x) (F x) ok,
    C (hole (F x) (IntroFProof x)).
```

- This introduces a variable without needing a name
- first two steps create the name of the variable
- Next we use a function as the proof term
- We extract the (term -> term) proof variable and the type
- Add the new variable to the context with the name
- Get the type of the new hole
- Call the continuation function on the hole in the context
- In our eiIntros tactic we will be calling predicates like `do-intro-anon` and thus we get a similar type

```
pred do-iIntros i:(list intro_pat), i:ihole, i:(ihole -> prop).
do-iIntros [] IH C :- !, C IH.
do-iIntros [iPure (none) | IPS] (ihole N (hole Type Proof)) C :-
  coq.elaborate-skeleton {{ tac_forall_intro_nameless _ _ _ _ _ _ }}
  Proof = {{ tac_forall_intro_nameless _ _ _ _ _ lp:IProof }},
  coq.typecheck IProof IType ok, !,
  do-intro-anon (hole IType IProof) (h\ sigma IntroProof\ sigma Intro
    h = hole IntroType IntroProof,
    coq.reduction.lazy.bi-norm IntroType NormType, !,
    do-iIntros IPS (ihole N (hole NormType IntroProof)) C
  ).
```

- The predicate `do-iIntros` gets a list of intro patterns, an ihole and the continuation function
- Base case calls the cont. predicate
- Pure intro case
- First transform goal to put forall at the top of goal
- Then use `do-intro-anon` to introduce that variable
- Lastly normalize the type and call iIntros on the new hole
- No anon Iris hypotheses introduced thus counter stays the same

4.7.6 Backtracking in proofs

Question: We don't actually need to backtrack here, we can just look at the type and see which case we need

```
pred do-iIntro-ident i:ident, i:ihole, o:ihole.
do-iIntro-ident ID (ihole N (hole Type Proof))
                   (ihole N (hole IType IProof)) :-
  ident->term ID _ T,
  coq.elaborate-skeleton
    {{ tac_impl_intro _ lp:T _ _ _ _ _ _ _ }}
    Type Proof ok, !,
  Proof =
    {{ tac_impl_intro _ _ _ _ _ _ _ _ lp:IProof }},
  coq.typecheck IProof IType' ok,
  pm-reduce IType' IType,
  if (IType = {{ False }})
     (coq.error "eiIntro: " X " not fresh")
     (true).
do-iIntro-ident ID (ihole N (hole Type Proof))
                   (ihole N (hole IType IProof)) :-
  ident->term ID _ T,
  coq.elaborate-skeleton
    {{ tac_wand_intro _ lp:T _ _ _ _ _ }}
    Type Proof ok, !,
  Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:IProof }},
  coq.typecheck IProof IType' ok,
  pm-reduce IType' IType,
  if (IType = {{ False }})
     (coq.error "eiIntro: " X " not fresh")
     (true).
do-iIntro-ident ID _ _ :-
  ident->term ID X _,
  coq.error "eiIntro: " X " could not introduce".
```

4.7.7 Starting the tactic

- Solve is the entry point
- Gets a goal with type proof and the arguments

```
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.
```

- First we parse the arguments
- Ten start proof and get the ihole
- Then start do-iIntros where at the end we put the context counter in the proof
- ...
- ...

4.8 Writing commands