Chapter 1

Elpi introduction

In this chapter we will show how Elpi together with Coq-Elpi can be used to create new tactics. We will do this by giving a tutorial on how to create the `iIntros` tactic from Iris.

1.1 Iris `iIntros`

Iris is a separation logic [Jun+15; Jun+16; Kre+17; Jun+18]. Predicates can be seen as propositions over resources, *e.g.,* heaps. Thus, there are a number of extra logical connectives such as `P * Q`, which represents that `P` and `Q` split up the resources into two disjoints in which they respectively hold. Moreover, hypothesis in our logic can often be used only once when proving something, they represent recourses that we consume when used. To be able to reason about this logic in Coq a tactics language has been added called the Iris Proof Mode (IPM) [KTB17; Kre+18]. In the IPM a new context has been added that represents the contexts containing resources. One context represents all disjoint resources and one represents all persistent resources. The tactics are build to replicate many of the behaviors of the Coq tactics while manipulating the Iris context. We will be specifically looking at the `iIntros` tactic. First, we will describe how `iIntros` works, and then we will describe how `iIntros` can be created using Elpi.

1.1.1 `iIntros` example

`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, only it was expanded, as inspired by ssreflect [HKP97; GMT16], to also 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 begin with a lemma about the magic wand. The magic wand can be seen as the implication of separation logic which also takes into account the separation of resources. Thus, where a normal implication introduction adds the left-hand side to the coq context, the magic wand adds the left-hand side to the disjoint resource context.

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

When using `iIntros "HP HR"`, this transforms into

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

We have introduced the two separation logic propositions into the Iris 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 forall existential, an exists existential, a conjunction and a disjunction. We can again use one application of `iIntros` to introduce and eliminate the premise. `iIntros "%x [[%y [Hx Hy]] | H0]"` takes the proof to the following state of two goals

```
(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. They can have the following interpretations:

`"H"` is the name pattern. It is just a name, and we rename the hypothesis to that name, like in the previous example.

`"\%H"` is the pure pattern. It is a percent sign followed by a name, the hypothesis is then interpreted as a Coq hypothesis if possible, and added to the coq context.

`"[H | H]"` is the or pattern. This introduction pattern is two intro patterns separated by a bar and surrounded by square brackets. We perform a disjunction elimination on the introduced hypothesis.

`"[H1 H2]"` is the separating and pattern. This introduction pattern is two intro patterns surrounded by square brackets but not separated by a bar. We perform a separating conjunction elimination.

`"[\%x H]"` is the exists pattern. If first element of a separating and pattern is a pure elimination we first try to eliminate an exists in the 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 or, 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 [KTB17].

1.2 Elpi implementation of iIntros

We implement our tactic in the λProlog language Elpi [Dun+15; GCT19]. Elpi implements λprolog [MN86; Mil+91; BBR99; MN12] and adds con-

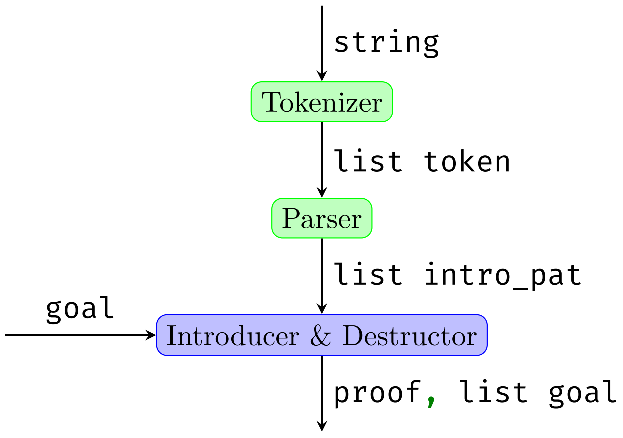

Figure 1.1: Structure of `eiIntros` with the input and output types on the edges.

straint handling rules to it [Mon11]. These constraint handling rules allow us to define invariants and constraints for variables by allowing us to defer the execution of a program when a variable gets assigned a value.

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

Our Elpi implementation `eiIntros` consists of four parts as seen in figure 1.1. The first two parts will interpret the DSL used to describe what we want to introduce. Then, the next two parts will apply the interpreted DSL. In section 1.2.1 we describe how a string is tokenized by the tokenizer. In section 1.2.2 we describe how a list of tokens is parsed into a list of intro patterns. In section 1.2.3 we describe how we use an intro pattern to introduce a wand or forall existential. Lastly, in section 1.2.4 we describe how a hypothesis is destructed according to the intro pattern. In every section we describe more parts of the Elpi programming language and the Coq-Elpi connector starting the base concepts of the language.

1.2.1 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.

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. 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 having 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 them. `tName` for example has type `string -> token`, thus containing a string. Besides `string` there are a few more basic types in Elpi suchs 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.

We can now represent a string as a list of these tokens. Given the string `"[H %H']"` we can represent it as the following list of type `token`:

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

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 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. String support is very minimal in, and we can't pattern match on the first character of a string. Thus, we need this workaround.

The next line, line 4, calls the recursive tokenizer, `tokenizer.rec`[1], on the split list of 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. 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

[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.

and what happens when a rule does not hold, we will discuss in the next two sections.

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 has a premise and a cut as its conclusion, we will discuss cuts in section 1.2.1, for now they can be ignored. The 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, in that case the variable will get the value of the constructor. Thus, we can either pass `[]` to the second argument, or some variable `V`. After which the variable `V` will be equal to `[]`.

The effect of matching on input arguments is that they can't be variable we set, thus they have to already have a value checking if they have the correct value. Whereas the output arguments, because of unification, get assigned a value if they don't have one already when checking if they have the correct 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 as [tSimpl, tDone], since that is the rule that comes first.

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 string representing this list of tokens.

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

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.

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

Spilling allows us to write the entry point of the tokenizer as defined in section 1.2.1 without the need of the temporary variable to pass the list of strings around.

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 operationally equal to the previous version, but just written shorter. We do have to be careful when spilling as the context for the spilled variable will be the outer rule we are defining. We will come across this in section ?

TODO: Refer to 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 `type`, only all aguments 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.

```
pred map i:list A, i:(A -> B -> prop), o:list B.
map [] _ [].
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.

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), o:list A, o:list A.
2 take-while-split [X|XS] Pred [X|YS] ZS :- Pred X,
3   take-while-split XS Pred YS ZS.
4 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

```
type tName string -> token.

tokenize.rec SL [tName S | TS] :-
  take-while-split SL is-identifier S' SL',
  { std.length S' } > 0, !,
  std.string.concat "" S' S,
  tokenize.rec SL' TS.
```

To tokenize a name we first call `take-while-split` with as predicate a predicate that checks if a string is valid identifier character, wether it is either a letter or one of a few symbols allowed in identifiers. It thus splits up the input string list into an 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 our `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.ltac.fail 0 "unrecognized tokens" XS.
```

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.

1.2.2 Parser

1.2.3 Introducer

1.2.4 Destructor

1.3 Old text

Elpi goals Goals in coq-elpi are represented as three main parts. A context of existential variables (evars) together with added rules assigning a type or definition to each variable. A goal, represented as a unification variable applied on all evars, together with a pending constraint typing the goal as the type of the goal. Lastly, a list of arguments applied to a tactic is given as part of every goal. The arguments are part of the goal, since they can reference the evars, and thus can't be taken out of the scope of the existential variables. Thus, a tactic invocation on the left is translated to an Elpi goal on the right.

```
P : Prop   pi c1\ decl c1 `P` (sort prop) ⇒
H : P          pi c2\ decl c2 `H` c1 ⇒
===========      declare_constraint (evar (T c1 c2)
P                                         c1
                                          (P c1 c2))
tac (P) asdf 12                     on (T c1 c2),
                 solve (goal [decl c1 `P` (sort prop),
                              decl c2 `H` c1]
                             (T c1 c2)
                             c1
                             (P c1 c2)
                             [trm c1, str "asdf", int 123])
```

This setup of the goal allows us to unify the trigger `T` with a proof term, which will trigger the elaboration of `T` against the type (here `c1`) and unification of the resulting term with our proof variable `P`. This resulting proof term will likely contain more unification variables, representing sub-goals, which we can collect as our resulting goal list (Elpi has the built-in `coq.ltac.collect-goals` predicate, that does this for us).

We do have a problem with these goals. They are not very portable. Since they need existential variables to have been created and rules to be assumed, we can't just pass around a goal without being very careful about the context it is in. This problem was solved by adding a sealed-goal. A sealed goal is a lambda function that takes existential variables for each element in its context. The arguments of the lambda functions can then be used in place of the existential variables in the goal. This allows us to pass around goals without having to worry about the context they are in. The sealed goal is then opened by applying it to existential variables. This is done by

```
pred open i:open-tactic, i:sealed-goal, o:list sealed-goal.
```

It opens a sealed goal and then applies the `open-tactic` to the opened goal. The resulting list of sealed goals is unified with the last argument.

Sealed goals allow us to program our tactics in separate steps, where each step is an `open-tactic`. This is especially useful since we have to call quite some LTac code on our goals within Elpi to solve side-goals.

Calling LTac There are built-in API's in coq-elpi to call LTac code by name. When calling LTac code we can give arguments by setting the arguments in our goal. This does mean we have to be careful to remove the arguments of our tactics from the goal before we give it to any called tactics. Also, this limits us to arguments of which coq-elpi has a type, and mapping. Thus, for now we are only able to pass strings, numbers, terms and lists of these to LTac tactics. We are not able to call any tactics that use coq intro patterns or any other syntax, until support has been added for these in coq-elpi.

[Structure of iIntros]Structure of `iIntros` `iIntros` is based on the multi goal tactic from coq-elpi. Since with multi goal tactics we get a sealed goal as input which we open when necessary, instead of having an already opened goal. We first call a predicate `parse_args` to parse the arguments and unset any arguments in the goal. Next we call the predicate `go_iIntros` which will interpret the intro pattern structure created and apply the necessary lemma's and tactics. `go_iIntros` will defer to other tactics in Elpi to apply the intro patterns.

Parsing intro patterns We parse our intro pattern in two steps. We first tokenize the input string. Furthermore, we then parse the list of generated tokens. Tokenizing uses a simple linear recursive predicate. We do no backtracking in the tokenizer.

We come across the first larger snag of Elpi here, its string handling. Strings in Elpi are `cstring`, and not lists of chars. Our first step is thus to split the string into a list of strings of length 1. Luckily, `rex.split "" I OS` allows us to split our input string `IS` on every character giving us our `OSS`, list of strings.

Another thing we have to be careful with is the naming of our constants. We first define a type token:

```
kind token type.
```

And then give different inhabitants of that type.

```
type tAnon, tFrame, tBar, tBracketL, ··· token.
type tName string -> token.
        ⋮
```

But we can actually write unification variables instead of names after `type`. This is valid Elpi and allows us to .... But when porting coq code to Elpi, if you aren't careful you might end up with some Pascal Case names and no warning or error from Elpi, except for broken syntax highlighting.

Now that we have our tokenized input, we can start parsing it. We use a reductive descent parser as the basis of our parser. The syntax we parse is

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

⟨intropattern⟩           ::= ⟨ident⟩
                          |  ‘_’ | ‘?’ | ‘$’ | ‘*’ | ‘**’ | ‘/=’ | ‘//’ | ‘!%’
                          |  ‘!>’ | ‘->’ | ‘<-’
                          |  ‘[’ ⟨intropattern_list⟩ ‘]’
                          |  ‘(’ ⟨intropattern_conj_list⟩ ‘)’
                          |  ‘%’ ⟨ident⟩
                          |  ‘#’ ⟨intropattern⟩ % Wait this one is weird
                          |  ‘-#’ ⟨intropattern⟩
                          |  ‘>’ ⟨intropattern⟩

⟨intropattern_list⟩      ::= ϵ
                          |  ⟨intropattern⟩ ‘|’ ⟨intropattern_list⟩
                          |  ⟨intropattern⟩ ⟨intropattern_list⟩

⟨intropattern_conj_list⟩ ::= ϵ
                          |  ⟨intropattern⟩ ‘&’ ⟨intropattern_conj_list⟩
```

With the caveat that a ⟨*intropattern_conj_list*⟩ has to have at least length 2.

The nice thing about reductive decent parsers, is that we can keep the structure of the syntax in BNF as the structure of the program. Thus, the parser for ⟨*intropattern_conj_list*⟩ becomes.

```
pred parse_conj_ilist i:list token, o:list token, o:list intro_pat.
parse_conj_ilist [tParenR | R] [tParenR | R] [IP].
parse_conj_ilist TS R [IP | L'] :-
  parse_ip TS [tAmp | RT] IP,
  parse_conj_ilist RT R L'.
```

Any parser should be interpreted as taking a list of tokens to parse and giving back a list of tokens that are left over after parsing and a list of intro patterns that got made after parsing. And because we unify our predicates we can pattern match on the output list of tokens, and we fail as soon as possible.

After the parsing we get a list of intro patterns of the following type

```
kind intro_pat type.
type iFresh, iDrop, iFrame, ··· intro_pat.
type iIdent ident -> intro_pat.
type iList list (list intro_pat) -> intro_pat.
type iPure option string -> intro_pat.
type iIntuitionistic intro_pat -> intro_pat.
type iSpatial intro_pat -> intro_pat.
type iModalElim intro_pat -> intro_pat.
type iRewrite direction -> intro_pat.
type iCoqIntro ltac1-tactic -> intro_pat.
```

`iList` represents a list of lists of intro patterns. The outer list is the disjunction intro pattern and the inner list the conjunction intro pattern. `iCoqIntro` is used when we pass a pure intro to `iIntros ( ... )`. It is thus never parsed for now and only added separately afterwards. However, this would be the place to allow pure intro patterns to be added in the middle of an Iris intro pattern.

Applying an intro pattern The intro patterns are applied by descending through them recursively. Thus, the intro pattern applier looks like

```
type go_iIntros (list intro_pat) -> tactic.
```

`tactic` is an abbreviation for the type `sealed-goal -> sealed goal`. We have a few interesting cases we will highlight here.

The simplest intro patterns are ones that just call a piece of LTac code and then apply the remaining intro patterns on the new goal. These are cases like '//', '/=' and '%a'. We show '/=' here as an example, it should apply `simpl` on the goal.

```
go_iIntros [iSimpl | IPS] G GL :-
    open (coq.ltac.call "simpl" []) G [G'],
    go_iIntros IPS G' GL.
```

A lot of intro patterns also require us to apply some Iris lemma. We will use the below example to show how a single step is build. Coq-elpi has a `refine` built in that allows us to refine a goal using a lemma with holes in it, as can be seen in line 5 and 7. In the drop and identifier intro patterns we also have to try several lemmas and when none work give an error message. We do this by making use of the backtracking capabilities of Elpi. We first try refining with implication intro, if that does not produce one goal we try wand intro and if none work we give an error. When a lemma is successfully applied we sometimes have to deal with some side goals that have to be solved. However, we don't want to backtrack any more after finding the correct branch to enter. Thus, we cut the backtracking after applying the lemma to make sure we surface the correct error message as can be seen on line 7.

```
1 go_iIntros [iIdent ID | IPS] G GL :- !,
2   ident->term ID X T,
3   open startProof G [G'],
  (
    open (refine {{ @tac_impl_intro _ _ lp:T _ _ _ _ _ _ _ }}) G' [GRes
      (
        open (refine {{ @tac_wand_intro _ _ lp:T _ _ _ _ _ }}) G' [G'']
        open (pm_reduce) G'' [G'''],
        open (false-error {calc ("eiIntro: " ^ X ^ " not fresh")}) G'''
      );
    (!, coq.ltac.fail 0 {calc ("eiIntro: " ^ X ^ " could not introduce"
  ),
  go_iIntros IPS GRes GL.
```

Now we get to the most interesting case. The destruct cases.

```
1 go_iIntros [iList IPS | IPSS] G GL :- !,
2   open startProof G [StartedGoal],
3   open (go_iFresh N) StartedGoal [FreshGoal],
4   go_iIntros [iIdent (iAnon N)] FreshGoal [IntroGoal],
5   go_iDestruct (iAnon N) (iList IPS) IntroGoal GL',
6   all (go_iIntros IPSS) GL' GL.
```

To destruct a goal the first step is to get an anonymous identifier and introduce the goal with it as on line 3. Getting a fresh identifier is quite easy using Elpi as we just have to extract the current counter and increase it by one.

```
type go_iFresh term -> open-tactic.
go_iFresh N (goal Ctx Trigger
                  {{ envs_entails (Envs lp:DP lp:DS lp:N) lp:Q }}
                  Proof Args)
            [seal (goal Ctx Trigger
                      {{ envs_entails (Envs lp:DP lp:DS (Pos.succ lp:N)
                      Proof Args)].
```

Next we can introduce using our previously defined identifier introduction step. Lastly we call the destruct applier with the identifier of the hypothesis we just introduced.

`go_iDestruct` has a pretty similar construction as `go_iIntros`. We identify the case we are interested about, we use the specific applier for that case and handle any side goals or error messages. Lastly, we call `go_iDestruct` on any nested intro patterns and merge the resulting goal lists.

Improvements possible using Elpi

We will list some possible improvements that could be done or have been done to by using Elpi. Most improvements found so far are about not needing weird workaround anymore that where uses by the LTac tactics. Elpi allows easier introspection into goals and passing around values found in goals. Also, backtracking depending on steps made looks to be easier. This allows us to remove the workarounds in `go_iExistDestruct` for remembering the name of the value to destruct. Also, the separating and destruct with a pure left side no longer needs to be resolved through a case in the type classes to destruct a separating and. Instead, we can backtrack when the exists destruct does not work. …

Downsides of Elpi

One problem that often occurs when using a meta-programming language is the need to transform between data types in the original and meta-programming language [**<empty citation>**]. This adds some overhead to quite a few actions that have to be done.

Also, not all API's are implemented yet in Elpi. For this project the most obvious missing thing is the support for coq intro patterns. It would be very helpful to be able to manipulate coq intro patterns and pass them to other tactics. Furthermore, support for the coq parser would add the possibility for a lot of added functionality.

Another mayor downside is the current lack of documentation. A few happy paths are documented fairly well by a few tutorials. However, there are a lot of library functions and possibilities that exist in Elpi and coq-elpi that have little to no documentation and make you have to read the source code to understand them. This point is somewhat lessened by the excellent support by the creator and other enthusiasts in the Zullip chat. A lot of the more pressing issues are resolved by asking them there. Also, the language is still young which explains a lot of these pain points.

Debugging is still not that simple in Elpi. Especially because of backtracking, it often occurs that a mistake in your program will not surface the error at the location where the error is. Even misspelling a variable often won't give an error and can result in hard to find mistakes. Typing a comma instead of a point will often not generate an error, even though your code is not getting run after the point. These can sometimes lead to quite long hunts for stupid mistakes.

Upsides of Elpi

Elpi as a language works quite well and the combination of λprolog with constraint handling allows for some very nice code. By using functional programming basics it is easy to create most pure functions and when making full use of the unification and backtracking some very elegant code can be created.

The base tactic programming language is well put together and allows for powerful manipulation of goals. It is possible to take values out of goals manipulate them. Apply tactics dynamically on any goals. This makes it work quite well for writing out more complicated tactics in a fairly readable manner.

The data type system is well put together and allows for easy construction of large complicated data structures.

The code tracer is an excellent tool that allows for much easier introspection into what is actually going on. It helps find many small bugs and also just gives a great feeling for how the language works.

The quotation system allows for fairly easy inclusion of coq terms into Elpi programs. This is very powerful tool allowing for matching of goals to take out values, constructing proof terms for use with `refine`, and creating coq data types.