Chapter 1

Elpi tactic for iIntros

1.1 Intros

In Coq proofs often start with the same few tactics. They start with some ∀ introductions, some → introductions and some destructs of existential quantifiers, \/, /\ and others. Because these happen so often, a little DSL has been made to quickly write these steps down, and are called intro patterns. These intro patterns are included in many tactics to quickly deal with the result of these tactics, but we will focus on `intros`.

1.1.1 Coq `intros`

We look at a subset of the total intro pattern syntax that is used in coq. Our subset is shown here

$$\begin{array}{lcl}
\langle\textit{intropattern}\rangle & ::= & `\texttt{*}' \\
 & | & `\texttt{**}' \\
 & | & \langle\textit{simple\_intropattern}\rangle \\
 & & \\
\langle\textit{simple\_intropattern}\rangle & ::= & \langle\textit{naming\_intropattern}\rangle \\
 & | & `\texttt{\_}' \\
 & | & \langle\textit{or\_and\_intropattern}\rangle \\
 & | & \langle\textit{equality\_intropattern}\rangle \\
 & & \\
\langle\textit{naming\_intropattern}\rangle & ::= & \langle\textit{ident}\rangle \\
 & | & `\texttt{?}' \\
 & | & `\texttt{?}'\langle\textit{ident}\rangle \\
 & & \\
\langle\textit{or\_and\_intropattern}\rangle & ::= & `\texttt{[}' \ ( \ \langle\textit{intropattern}\rangle^{\star} \ )^{\star}_{`|'} \ `\texttt{]}' \\
 & | & `\texttt{(}' \ \langle\textit{intropattern}\rangle^{\star}_{`\&'} \ `\texttt{)}'
\end{array}$$

⟨*equality_ intropattern*⟩ ::= '`->`'
| '`<-`'
| '`[=`' ⟨*intropattern*⟩* '`]`'

...

1.1.2 Iris `iIntros`

Iris has in its logic several more connectives that behave like /\ and \/, but are not them. This combined with the separate environments that iris adds, result in us not being able to use the coq `intros` tactic. Thus, we have written our own tactic that can deal with the Iris logic. We call this tactic `iIntros`.

...

1.1.3 Elpi implementation of iIntros

We implement our tactic in the λProlog programming language Elpi [1]. Elpi implements λprolog and adds constraint handling rules to it. To use it as a coq meta programming language we make use of the Elpi Coq connector, coq-elpi [2].

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` `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⟩
```

$\langle \mathit{intropattern\_conj\_list} \rangle ::= \epsilon$
$\quad\quad\quad\quad\quad\quad\quad\quad | \quad \langle \mathit{intropattern} \rangle$ '&' $\langle \mathit{intropattern\_conj\_list} \rangle$

With the caveat that a $\langle \mathit{intropattern\_conj\_list} \rangle$ 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 $\langle \mathit{intropattern\_conj\_list} \rangle$ 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.

Executing an intro pattern The intro patterns are executed by descending through them recursively. Thus, the intro pattern executor 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 execute the remaining intro patterns on the new goal. These are cases like '//', '/=' and '%a'. We show '/=' here as an example, it should execute 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. Coq-elpi has a refine built in that allows us to refine a goal using a lemma with holes in it. 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. 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. The identifier executor has all these aspects.

```
go_iIntros [iIdent ID | IPS] G GL :- !,
  ident->term ID X T,
  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''' [GRes]
      );
    (!, coq.ltac.fail 0 {calc ("eiIntro: " ^ X ^ " could not introduce")}, fail)
  ),
  go_iIntros IPS GRes GL.
```