Chapter 1

Elpi tactic for iIntros

1.1 Intros

In coq proofs onften start with the same few tactics. They start with some forall introductions, some implication 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

$$
\begin{array}{lll}
\langle \textit{intropattern} \rangle & ::= & \text{`*'} \\
& | & \text{`**'} \\
& | & \langle \textit{simple\_intropattern} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle \textit{simple\_intropattern} \rangle & ::= & \langle \textit{naming\_intropattern} \rangle \\
& | & \text{`\_'} \\
& | & \langle \textit{or\_and\_intropattern} \rangle \\
& | & \langle \textit{equality\_intropattern} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle \textit{naming\_intropattern} \rangle & ::= & \langle \textit{ident} \rangle \\
& | & \text{`?'} \\
& | & \text{`?'} \langle \textit{ident} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle \textit{or\_and\_intropattern} \rangle & ::= & \text{`[' (} \langle \textit{intropattern} \rangle^{\star} \text{)}^{\star}_{|}\text{, `]'} \\
& | & \text{`(' } \langle \textit{intropattern} \rangle^{\star}_{\&}\text{, `)'}
\end{array}
$$

⟨*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 seperate 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 refrence 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 a 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 subgoals, which we can collect as our resulting goal list (elpi has the builtin `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 assumend, we can't just pass around a goal without being very carefull 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 opend goal. The resulting list of sealed goals is unified with the last argument.

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

Calling LTac There are builtin 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 carefull 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.