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. 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.

$$\square P * Q \vdash R$$

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

```
1 P, Q, R: iProp
2 ============
3 "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

4.3 Tactics

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 ?.

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 define our tokens. Next we show how to define a predicate that transform our string into the tokens we defined.

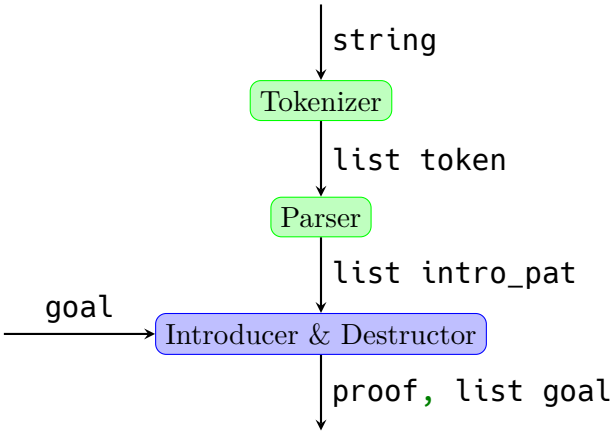

Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

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".

```
1  kind token type.
2
3  type tAnon, tFrame, tBar, tBracketL, tBracketR, tAmp,
4       tParenL, tParenR, tBraceL, tBraceR, tSimpl,
5       tDone, tForall, tAll token.
6  type tName string -> token.
7  type tNat int -> token.
8  type tPure option string -> token.
9  type tArrow direction -> token.
10
11 kind direction type.
12 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`.

```
1 kind option type -> type.
2 type none option A.
3 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`:

```
1 [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:

```
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.

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

[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.

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.

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.

```
1 pred tokenize i:string, o:list token.
2 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 ?.

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.

```
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.

```
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

```
1 tokenize.rec XS _ :- !,
2   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

4.6.1 Data structure

4.6.2 Reductive descent parsing

4.6.3 Danger of backtracking

4.7 Applier

4.7.1 Elpi coq HOAS

4.7.2 Coq context in Elpi

4.7.3 Quotation and anti-quotation

4.7.4 Proofs in Elpi

Iris context counter

4.7.5 Continuation Passing Style

4.7.6 Backtracking in proofs

4.7.7 Starting the tactic

4.8 Writing commands