Bachelor thesis
Computing Science

Radboud University

Iris Induction or something

Author:
Luko van der Maas
luko.vandermaas@ru.nl
s1010320

Supervisor:
dr. Robbert Krebbers
robbert@cs.ru.nl

Assessor:
...
...

October 19, 2023

Abstract

This is an abstract. It is very abstract. And now a funny pun about Iris from github copilot: ”Why did the mathematician bring Iris to the formal methods conference? Because they wanted to be a ’proof-essional’ with the most ’Irisistible’ Coq proofs!”

Contents

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 [4, 2, 7, 3]. 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 hold. Thus, our logic is substructural, namely linear. To be able to reason about this logic in coq a tactics language has been added called the Iris Proof Mode (IPM) [5, 6]. In the IPM a new context has been added that represents the contexts containing resources. One context represents all disjunct resources and one represents all invariant 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 consists of a DSL for quickly introducing different logical connectives in Iris. It has also been extended to deal with often seen initial steps in a proof, like **simpl**, **done** and others. We will show a few examples of how iIntros can be used to help proof lemma's.

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 with an and to the context, the magic wand adds the left-hand side with a separating and to the resource context.

Question: How to name the different resources

```
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, an exists, 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
```

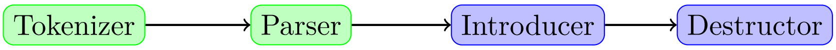

Figure 1.1: Structure of `eiIntros`

We break down the statement `iIntros "%x [[%y [Hx Hy]] | H0]"` intro its components. The first `"%x"` introduces a pure Coq variable for the `∀ x : nat`. Next `"[H | H]"` first introduces the magic wand and then eliminates the or, resulting in two goals. `"[%y H]"` eliminates the exists and adds the `y` to the Coq context. Lastly, `"[Hx Hy]"` eliminates the seperating and in the Iris context by splitting it into two assumptions `"Hx"` and `"Hy"`.

There are more patterns available to introduce complicated goals, these can be found in [5].

1.2 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 [8]. We will use these languages to implement the Elpi variant of `iIntros`, called `eiIntros`.

1.2.1 Structure of `eiIntros`

Our Elpi tactic 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 next two parts will apply the interpreted DSL. Interpreting consists of first tokenizing the input string, then we parse the tokens into a recursive Elpi data structure. We use this structure to recurse over it and apply steps when necessary. Lastly whenever we have to eliminate some hypothesis we call the destructor. Each of these parts will he explained in a section below. In each section we will expand more elements part of Elpi, such that you won't need any more knowledge of Elpi.

Tokenizer

Elpi is a λProlog language. This is a combination of two concepts, the predicate based language of Prolog and lambda application. Elpi also has constraint handling in it, but we will focus on that in section 1.2.3. For the tokenizer we will only need the more basic aspects of Elpi. We will highlight these while describing the tokenizer.

The tokenizer

1.2.2 Parser

1.2.3 Applier

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 subgoals, 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⟩


⟨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'],
4    (
5      open (refine {{ @tac_impl_intro _ _ lp:T _ _ _ _ _ _ _ }}) G' [GRes
6        (
7          open (refine {{ @tac_wand_intro _ _ lp:T _ _ _ _ _ }}) G' [G'']
8          open (pm_reduce) G'' [G'''],
9          open (false-error {calc ("eiIntro: " ^ X ^ " not fresh")}) G'''
10       );
11     (!, coq.ltac.fail 0 {calc ("eiIntro: " ^ X ^ " could not introduce"
12   ),
13   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 []. 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.

Chapter 2

Inductive

2.1 fixpoint predicate

2.2 Monotone

2.3 Fixpoint

2.4 Unfold lemma

2.5 Induction or Coinduction scheme

2.6 `iConstructor`, `iDestruct` abd `iInductive`

Bibliography

[1] Cvetan Dunchev et al. “ELPI: Fast, Embeddable, λProlog Interpreter”. In: *Log. Program. Artif. Intell. Reason.* Lecture Notes in Computer Science. 2015, pp. 460–468. DOI: 10.1007/978-3-662-48899-7_32.

[2] Ralf Jung et al. “Higher-Order Ghost State”. In: *SIGPLAN Not.* 51.9 (Sept. 4, 2016), pp. 256–269. DOI: 10.1145/3022670.2951943.

[3] Ralf Jung et al. “Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic”. In: *J. Funct. Program.* 28 (Jan. 2018), e20. DOI: 10.1017/S0956796818000151.

[4] Ralf Jung et al. “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning”. In: *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.* POPL ’15. Jan. 14, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980.

[5] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive Proofs in Higher-Order Concurrent Separation Logic”. In: *SIGPLAN Not.* 52.1 (Jan. 1, 2017), pp. 205–217. DOI: 10.1145/3093333.3009855.

[6] Robbert Krebbers et al. “MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic”. In: *Proc. ACM Program. Lang.* 2 (ICFP July 30, 2018), 77:1–77:30. DOI: 10.1145/3236772.

[7] Robbert Krebbers et al. “The Essence of Higher-Order Concurrent Separation Logic”. In: *Program. Lang. Syst.* Lecture Notes in Computer Science. 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26.

[8] Enrico Tassi. “Elpi: An Extension Language for Coq (Metaprogramming Coq in the Elpi λProlog Dialect)”. Jan. 2018. URL: https://inria.hal.science/hal-01637063.