Master thesis
Computing Science

Radboud University

Extending Iris with Inductive predicates using Elpi

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

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

Assessor:
dr. Freek Wiedijk
`freek@cs.ru.nl` …

June 14, 2024

Abstract

Separation logic gives a framework for defining and proving specifications for programs. In separation logic, representation predicates are used to relate a data structure in the heap to an object in the logic. When dealing with recursive data structures, inductive representation predicates are needed to represent them.

Software verification systems that embed separation logic into a proof assistant have to embed the separation logic into the logic of the proof assistant. Consequently, inductive predicates must be derived from the base logic. This derivation can be done using three distinct fixpoints, each having downsides. Both using the fixpoint of the base logic and using Banach fixpoints impose significant limitations on the inductive predicates that can be defined. Using the least fixpoint, on the other hand, imposes many manual proofs.

This thesis develops a command and tactics to automate inductive predicates using the least fixpoint in Iris[Jun+18], which is embedded in the Coq proof assistant as the Iris Proof Mode (IPM) [Kre+18]. We use the Coq meta-programming language Coq-Elpi [Tas18] to generate the least fixpoint and prove the induction principle of this inductive predicate. Furthermore, we introduce tactics that automate applying using the inductive predicate. Lastly, we use our system of commands and tactics to redefine the total weakest precondition in Iris.

During the creation of our system, we reimplement a significant part of the tactics from the IPM and evaluate if Elpi would be a good fit for reimplementing the full IPM.

Contents

Chapter 1

Introduction

Induction on inductive predicates is a fundamental aspect of reasoning about recursive structures within a logic. Separation logic [ORY01; Rey02] has proven to be a promising basis for program verification. It employs an affine logic with additional connectives to reason about the heap of a program. Inductive predicates are an essential part of this logic, they allow one to reason about the recursive data structures present in the program.

We make use of an embedding of separation logic in an interactive proof assistant. As a result, inductive predicates in the separation logic cannot be an axiom and have to follow from the base logic of the proof assistant. Three major approaches have been found to define inductive predicates: structural recursion, the Banach fixpoint [Jun+18], the least fixpoint [App14].

- Structural recursion defines an inductive predicate by recursion on an inductive type in the base logic, e.g., defining an inductive predicate by recursion on lists defined by the proof assistant.
- The Banach fixpoint defines inductive predicates by guarding the recursion behind the step-indexing present in some separation logics.
- The least fixpoint is the most general approach, it takes a monotone function, the *pre fixpoint function*, describing the behavior of the inductive predicate. Then, the least fixpoint of this function corresponds to the inductive predicate. The least fixpoint also allows for proving total correctness, whereas Banach fixpoints only allow proving partial correctness. Thus, in this thesis, we focus on the least fixpoint approach.

Separation logic has been implemented several times in proof assistants [AR; RKV21; Chl11; BJB12]. We make use of the separation logic Iris [Jun+15; Jun+16; Kre+17; Jun+18], implemented in the proof assistant Coq as the Iris Proof Mode/MoSeL [KTB17; Kre+18]. Iris has been applied for verification of Rust [Jun+17; Dan+19; Mat+22], Go [Cha+19], Scala [Gia+20], C [Sam+21], and WebAssemble [Rao+23]

Defining inductive predicates using the least fixpoint in Iris is a very manual process. Several trivial proofs must be performed, and several intermediary objects must be defined. Furthermore, using the inductive predicates in a proof requires additional manual steps.

This thesis aims to solve this problem by adding several commands and tactics to Coq that simplify and streamline working with inductive predicates. We implement our

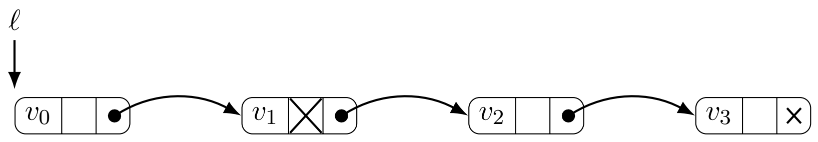


Figure 1.1: A node is shown here as three boxes next to each other, the first box contains a value. The second box is a boolean. The box is true, thus marked, if it is crossed out. The third box is a pointer, denoted by either a cross, a null pointer, or a circle with an arrow pointing to the next node.

commands and tactics in the λProlog dialect Elpi [Dun+15; GCT19]. Elpi is a λProlog dialect [MN86; Mil+91; BBR99; MN12]. To use Elpi as a Coq meta-programming language, there exists the Elpi Coq connector, Coq-Elpi [Tas18].

1.1 Central example

Marked linked lists, (MLLs), developed by Harris [Har01], are non-blocking concurrent linked lists. They are the central example used in this thesis. We will use them here to give a preview of the system we developed.

MLLs, are linked lists where each node has an additional mark bit. When a node is marked, and thus the bit is set, the node is considered deleted. An example of a MLL can be found in figure 1.1. An MLL allows for deleting a node out of a list without modifying any of the other nodes, helping with concurrent usages.

In order to reason about MLLs in separation logic, we relate a heap containing an MLL to a list in the separation logic in the IPM. Using our newly developed system, this can be achieved similarly to writing any other inductive predicate.

```coq
eiInd
Inductive is_MLL : val → list val → iProp :=
    | empty_is_MLL : is_MLL NONEV []
    | mark_is_MLL v vs l tl :
      l ↦ (v, #true, tl) -∗ is_MLL tl vs -∗
      is_MLL (SOMEV #l) vs
    | cons_is_MLL v vs tl l :
      l ↦ (v, #false, tl) -∗ is_MLL tl vs -∗
      is_MLL (SOMEV #l) (v :: vs).
```

The command `eiInd` (for "Elpi Iris Inductive") is prepended to the inductive statement and defines the inductive predicate `is_MLL`, together with the unfolding lemmas, constructor lemmas, and induction lemma. When we have a goal requiring induction on an `is_MLL` statement, we can simply call the `eiInduction` tactic on it. We then get goals for all the cases in the inductive predicate with the proper induction hypotheses.

1.2 Approach

To generate inductive properties from their inductive definition using the least fixpoint, we will take the following approach. We create the command, `eiInd`, as shown above.

Which, given an inductive definition in Coq, generates the pre fixpoint function, proves it monotone, and defines the fixpoint for the arity of the pre fixpoint function. Next, it proves the fixpoint properties of the defined fixpoint and generates constructor lemmas. Lastly, it generates and proves the induction lemma.

To use the inductive predicate, we create two tactics. The `eiInduction` tactic applies the induction lemma on the specified hypothesis. The `eiDestruct` tactic eliminates an inductive predicate into its possible constructors.

To accomplish these goals, we reimplement a subset of the IPM tactics in Elpi as *proof generators*, i.e., taking a goal in Elpi and producing a proof term that inhabits this goal. We also use these proof generators to reimplement several other IPM tactics, namely `eiIntros`, `eiSplit`, `eiEvalIn`, `eiModIntro`, `eiExFalso`, `eiClear`, `eiPure`, `eiApply` (without full specialization), `eiIntuitionistic`, and `eiExact`.

1.3 Contributions

This thesis contains the following contributions.

Generation of Iris inductive predicates We develop a system written in Elpi that, given an inductive definition in Coq, defines the inductive predicate with associated unfolding, constructor, and induction lemmas. In addition, tactics are created that automate unfolding the inductive predicate and applying the induction lemma. *(Ch. 5)*

Modular tactics in Elpi We present a way to define steps in a tactic, called *proof generators*, such that they can easily be composed. Allowing one to define simple proof generators that can be reused in many tactics. *(Sec. 4.7)*

Generate monotonicity proof of n-ary predicates We present an algorithm which given an n-ary predicate can find a proof of monotonicity. *(Sec. 3.3)*

Evaluation of Elpi Lastly, we evaluate Elpi with Coq-Elpi as a meta-language for Coq. We also discuss replacing LTaC with Elpi in IPM. *(Ch. 6)*

1.4 Outline

We start by giving a background on Separation logic in chapter 2. The chapter discusses the Iris separation logic while specifying and proving a program on MLLs. Next, in chapter 3, we discuss defining representation predicates in a separation logic using least fixpoints. Thus, we show how to define a representation predicate as an inductive predicate, and then give a novel algorithm to prove it monotone. In chapter 4, we give a tutorial on Elpi by implementing an IPM tactic, `iIntros`, in Elpi. Building on the foundations of chapter 4, we create the command and tactics to define inductive predicates in chapter 5. In chapter 6, we evaluate what was usefull in Elpi and what could be improved. We also discuss how and if Elpi can be used in IPM. Lastly, we discuss related work in chapter 7 and show the capabilities and shortcomings of the created commands and tactics in chapter 8, together with any future work.

Notation During the thesis, we will be working in two different programming languages. To always distinguish between them, the inline displays have a different color. Any `Coq displays` have a light green line next to them. Any `Elpi displays` have a light blue line next to them. Full-width listings also differentiate using green and blue lines, respectively.

Chapter 2

Background on separation logic

In this chapter we give a background on separation logic by specifying and proving the correctness of a program on marked linked lists (MLLs), as seen in chapter 1. First, we set up the running example in section 2.1. Next, we introduce the relevant features of separation logic in section 2.2. Then, we show how to give specifications using Hoare triples and weakest preconditions in section 2.3. In section 2.4, we show how Hoare triples and weakest preconditions relate to each other. In the process, we explain persistent propositions. Next, we show how we can create a predicate used to represent a data structure for our example in section 2.5. Lastly, we finish the specification and proof of a program manipulating marked linked lists in section 2.6.

2.1 Setup

Our running example is a program that deletes an element at an index in a MLL. This program is written in HeapLang, a higher order, untyped, ML-like language. HeapLang supports many concepts around both concurrency and higher-order heaps (storing closures on the heap). However, we will not need any of these features. These features are thus omitted. The language can be treated as a basic ML-like language. The syntax can be found in figure 2.1. For more information about HeapLang one can reference the Iris technical reference [Iri23].

We use several pieces of syntactic sugar to simplify notation. Lambda expressions, $\lambda x.\, e$, are defined using rec expressions. We write let statements, **let** $x = e$ **in** e', using lambda expressions $(\lambda x.\, e')(e)$. Let statements with tuples as binder are defined using combinations of **fst** and **snd**. Expression sequencing is written as $e; e'$, this is defined as **let** _ $= e$ **in** e'. The keywords **none** and **some** are just **inl** and **inr** respectively, both in values and in the match statement. We define the short circuit and, $e_1 \&\& e_2$, using the following if statement, **if** e_1 **then** e_2 **else** **false**. Lastly, when writing named functions, they are defined as names for anonymous functions.

Our running example deletes an index out of a list by marking that node, logically

$$
\begin{aligned}
v, w \in \mathit{Val} ::= {} & z \mid \mathbf{true} \mid \mathbf{false} \mid () \mid \ell \mid \qquad\qquad (z \in \mathbb{Z}, \ell \in \mathit{Loc}) \\
& (v, w) \mid \mathbf{inl}(v) \mid \mathbf{inr}(v) \mid \\
& \mathbf{rec}\ f(x) = e \\
e \in \mathit{Expr} ::= {} & v \mid x \mid e_1(e_2) \mid \circledcirc_1 e \mid e_1 \circledcirc_2 e_2 \mid \\
& \mathbf{rec}\ f(x) = e \mid \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \\
& (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \\
& \mathbf{inl}(e) \mid \mathbf{inr}(e) \mid \\
& \mathbf{match}\ e\ \mathbf{with}\ (\mathbf{inl}(x) \Rightarrow e_1 \mid \mathbf{inr}(y) \Rightarrow e_2)\ \mathbf{end} \mid \\
& \mathbf{ref}(e) \mid !\,e \mid e_1 \leftarrow e_2 \\
\circledcirc_1 ::= {} & - \mid \dots \\
\circledcirc_2 ::= {} & + \mid - \mid = \mid \dots
\end{aligned}
$$

Figure 2.1: Relevant fragment of the syntax of HeapLang

deleting it.

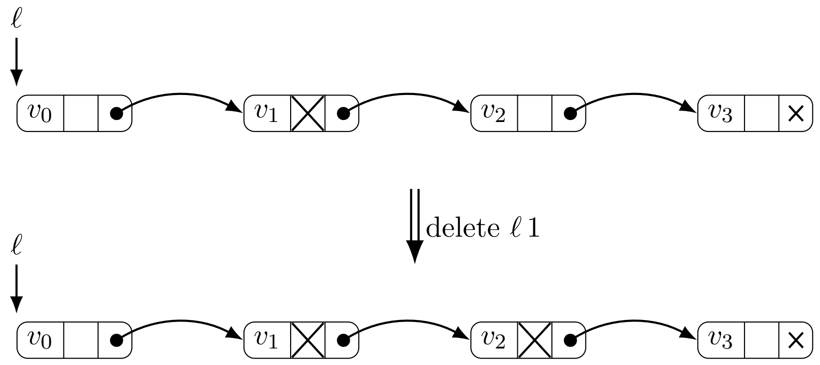
```
delete $hd\, i$ = match $hd$ with
                    none  $\Rightarrow ()$
                  | some $\ell \Rightarrow$ let $(x, mark, tl) = !\,\ell$ in
                              if $mark$ = false && $i = 0$ then
                                $\ell \leftarrow (x,$ true$, tl)$
                              else if $mark$ = false then
                                delete $tl\ (i-1)$
                              else
                                delete $tl\ i$
                  end
```

The example is a recursive function called delete, the function has two arguments. HeapLang has no null pointers, and thus we wrap a pointer in **none**, the null pointer, **some** ℓ, a non-null pointer pointing to ℓ. The first argument *hd* is either a null pointer, for the empty list, or a pointer to an MLL. The second argument, i, is the index in the MLL to delete. The first step this recursive function taken is checking whether we are deleting from the empty list. To accomplish this, we perform a match on *hd*. When *hd* is the null pointer, the list is empty, and we return unit. When *hd* is a pointer to ℓ, the list is not empty. We load the first node and save it in the three variables x, *mark* and *tl*. Now, x contains the first element of the list, *mark* tells us whether the element is marked, thus logically deleted, and *tl* contains the reference to the tail of the list. We now have three different branches we might take.

- If our index is zero and the element is not marked, thus logically deleted, we want to delete it. We write the node to the ℓ pointer, but with the mark bit set to **true**, thus logically deleting it.
- If the mark bit is **false**, but the index to delete, i, is not zero. The current node has not been deleted, and thus we want to decrease i by one and recursively call our function f on the tail of the list.

- If the mark bit is set to **`true`**, we want to ignore this node and continue to the next one. We thus call our recursive function f without decreasing i.

The expression delete $\ell\,1$ thus applies the transformation below.



When viewing this in terms of lists, the expression delete $\ell\,1$ deletes from the list $[v_0, v_2, v_3]$ the element v_2, thus resulting in the list $[v_0, v_3]$. This idea of representing an MLL using a mathematical structure is discussed more formally in section 2.5. However, to understand this we first need a basis of separation logic. This is discussed in the next section.

2.2 Separation logic

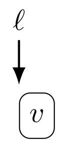
We make use of a subset of Iris [Jun+18] as our separation logic. This subset includes separation logic as first presented by Ishtiaq et al. and **reynoldsSeparationLogicLogic2002** [**reynoldsSeparationLogicLogic2002**; IO01], together with higher order connectives, persistent propositions and weakest preconditions as introduced by Iris. This logic is presented below, starting with the syntax.

$$\begin{aligned} P \in \mathit{iProp} ::= {} & \mathsf{False} \mid \mathsf{True} \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \\ & \ulcorner \phi \urcorner \mid \ell \mapsto v \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box\, P \mid \mathsf{wp}\; e\; [\Phi] \end{aligned}$$

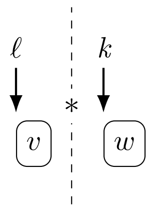
Separation logic contains all the usual higher order predicate logic connectives as seen on the first line. The symbol τ, represents any type we have seen, including *iProp* itself. The second row contains separation logic specific connectives. The *pure* connective, $\ulcorner \phi \urcorner$, embeds any Coq proposition, also called a pure proposition, into separation logic. Coq propositions include common connectives like equality, list manipulations and set manipulations. Whenever it is clear from context that a statement is pure, we may omit the pure brackets. The next two connectives, $\ell \mapsto v$ and $P * P$, are discussed in this section. The last three connectives, $P \mathrel{-\!\!*} P$, $\Box\, P$ and $\mathsf{wp}\; e\; [\Phi]$, are discussed when they become relevant in section 2.3 and section 2.4.

Separation logic reasons about ownership in heaps. Thus, a statement in separation logic describes a set of heaps for which the statement holds. Whenever a location exists in such a heap this is interpreted as owning that location with the unique permission to access its value. Using this semantic model of separation logic we give an intuition of the connectives.

The statement $\ell \mapsto v$, called ℓ *maps to* v, holds for any heap in which we own a location ℓ, which has the value v. We represent such a heap using the below diagram.



To describe two values in memory we could try to write $\ell \mapsto v \wedge k \mapsto w$. However, this does not ensure that ℓ and k are not the same location. The above diagram would still be a valid state of memory for the statement $\ell \mapsto v \wedge k \mapsto w$. Thus, we introduce a second form of conjunction, the separating conjunction, $P * Q$. For $P * Q$ to hold for a heap we have to split it in two disjoint parts, P should hold while owning only locations in the first part and Q should hold with only the second part.



$$\ell \mapsto v * k \mapsto w$$

To reason about statements in separation logic we make use of the notation $P \vdash Q$, called *entailment*. Intuitively, the heap described by Q has to be a subset of the heap described by P. The notation $P \dashv\vdash Q$ is entailment in both directions. Using this notation, the separating conjunction has the following set of rules.

$$\begin{aligned} \mathsf{True} * P &\dashv\vdash P \\ P * Q &\vdash Q * P \\ (P * Q) * R &\vdash P * (Q * R) \end{aligned}$$

$$\textsc{$*$-mono}\quad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}$$

The separating conjunction is commutative, associative and respects True as identity element. Instead of an introduction and elimination rule, like the normal conjunction, there is the $*$-MONO rule. This rule introduces the separating conjunction but also splits the hypotheses over the introduced propositions. The separating conjunction is not duplicable. Thus, the following rule is missing, $P \vdash P * P$. This makes intuitive sense since if $\ell \mapsto v$ holds, we could not split the memory in two, such that $\ell \mapsto v * \ell \mapsto v$ holds. We cannot have two disjoint sections of a heap where ℓ resides in both. Indeed, we have $\ell \mapsto v * \ell \mapsto v \vdash \mathsf{False}$.

2.3 Writing specifications of programs

In this section, we discuss how to specify actions of a program, we use two different methods, the Hoare triple and the weakest precondition. In the next section, section 2.4, we show how they are related.

Hoare triples Our goal when we specify a program is total correctness. Thus, given some precondition holds, the program does not crash, it terminates and afterward the postcondition holds. For our first attempt at total correctness, we use total Hoare triples, abbreviated to Hoare triples in this thesis.

$$[P]\ e\ [\Phi]$$

The Hoare triple consists of three parts, the precondition, P, the expression, e, and the postcondition, Φ. This Hoare triple states that, given that P holds beforehand, e does not crash, and it terminates. Afterward, for return value v, $\Phi(v)$ holds. Thus, Φ is a predicate taking a value as its argument. Whenever we write out the predicate, we omit the λ and write $[P]\, e\, [v.\, Q]$ instead. Whenever we assume v to be a certain value, v', instead of writing $[P]\, e\, [v.\, v = v' * Q]$ we just write $[P]\, e\, [v'.\, Q]$. Lastly, if we assume the return value is the unit, $()$, we leave it out entirely. Thus, $[P]\, e\, [v.\, v = () * Q]$ is equivalent to $[P]\, e\, [Q]$. This often happens as quite a few programs return $()$. We now look at an example of a specification for a basic program.

$$[\ell \mapsto v]\, \ell \leftarrow w\, [\ell \mapsto w]$$

This program assigns to location ℓ the value w. The precondition is, $\ell \mapsto v$. Thus, we own a location ℓ, and it has value v. Next, the specification states that we can execute $\ell \leftarrow w$, and it will not crash and will terminate. The program will return $()$ and afterward $\ell \mapsto w$ holds. Thus, we still own ℓ, and it now points to the value w. The specification for delete follows the same principle.

$$[\mathsf{isMLL}\, hd\, \vec{v}]\ \text{delete}\ hd\ i\ [\mathsf{isMLL}\, hd\, (\mathsf{remove}\, i\, \vec{v})]$$

The predicate $\mathsf{isMLL}\, hd\, \vec{v}$ holds if the MLL starting at hd contains the mathematical list $\vec{v}$. This predicate is explained further in section 2.5. The purely mathematical function remove gives the list $\vec{v}$ with index i removed. If the index is larger than the size of the list, the original list is returned. We thus specify the program by relating its actions to operations on a mathematical list.

Weakest precondition Hoare triples allow us to easily specify a program. However, in a proof, they are sometimes harder to work with when used in conjunction with predicates like isMLL. Especially when we will look at induction on this predicate in section 2.5 Hoare triples no longer suffice. Instead, we introduce the total weakest precondition, $\mathsf{wp}\ e\ [\Phi]$, abbreviated to weakest precondition from now on. The weakest precondition can be considered a Hoare triple without its precondition. Thus, $\mathsf{wp}\ e\ [\Phi]$ states that e does not crash and that it terminates. Afterward, for any return value v, the postcondition $\Phi(v)$ holds. We make use of the same abbreviations when writing the predicate of the weakest precondition, as with the Hoare triple.

We still need a precondition when working with the specification of a program. Thus, we embed this in the logic using the magic wand.

$$P \mathrel{-\!\!*} \mathsf{wp}\ e\ [\Phi]$$

The magic wand acts like the normal implication while considering the heap. The statement, $Q \mathrel{-\!\!*} R$, describes the state of memory where if we add the memory described by Q we get R. The below rule expresses this property.

$$\begin{array}{l}\mathrel{-\!\!*}\text{I-E} \\ \dfrac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}\end{array}$$

If we have as assumption P and need to prove $Q \mathrel{-\!\!*} R$, We can add Q to our assumptions to prove R. Thus, if we add ownership of the heap as described by Q, we can prove R.

Note that this rule works both ways, as signified by the double lined rule. It is both the introduction and the elimination rule.

We can now rewrite the specification of $\ell \leftarrow v$ using the weakest precondition.

$$\ell \mapsto v \twoheadrightarrow \mathsf{wp}\ \ell \leftarrow w\ [\ell \mapsto w]$$

This specification holds from WP-STORE in figure 2.2. The rules in this diagram follow a different style than is expected. We could have used the above specification of $\ell \leftarrow v$ as the rule. However, we make use of a "backwards" style [**reynoldsSeparationLogicLogic2002**; IO01], where we reason from conclusion to the assumptions. This is also the style used in the Coq implementation of Iris, and allows for more easy application of the rules. These rules can, however, be simplified to the style used above. The rules are listed in figure 2.2. We will now highlight the rules shortly.

For reasoning about the language constructs, we have three rules for the three different operations that deal with the memory and one rule for all pure operation.

- The rule WP-ALLOC defines the following. For $\mathsf{wp}\ \mathbf{ref}(v)\ [\Phi]$ to hold, $\Phi(\ell)$ should hold for a new ℓ with $\ell \mapsto v$.
- The rule WP-LOAD defines that for $\mathsf{wp}\ !\,\ell\ [\Phi]$ to hold, we need ℓ to point to v and separately if we add $\ell \mapsto v$, $\Phi(v)$ holds. Note that we need to add $\ell \mapsto v$ with the wand to the predicate, since the statement is not duplicable. Thus, if we know $\ell \mapsto v$, we have to use it to prove the first part of the WP-LOAD rule. But, at this point, we lose that $\ell \mapsto v$. Then, the WP-LOAD rule adds that we know $\ell \mapsto v$ using the magic wand to the postcondition.
- The rule WP-STORE works similar to WP-LOAD, but changes the value stored in ℓ for the postcondition.
- The rule WP-PURE defines that for any pure step we just change the expression in the weakest precondition

For reasoning about the general structure of the language and the weakest precondition itself we also have four rules.

- The rule WP-VALUE defines that if the expression is just a value, it is sufficient to prove the postcondition with the value filled in.
- The rule WP-MONO allows for changing the postcondition as long as this change holds for any value.
- The rule WP-FRAME allows for adding any propositions we have as assumption into the postcondition of a weakest precondition we have as assumption.
- The rule WP-BIND allows for extracting the expressions in the head position of a program. This is done by wrapping the head expression in a context as defined at the bottom of figure 2.2. The contexts as defined in figure 2.2 ensure a right to left, call-by-value evaluation of expressions. The verification of the rest of the program is delayed by moving it into the postcondition of the head expression.

An example where some of these rules can be found in section 2.4 and section 2.6

General rules.

$$\text{WP-VALUE}\quad \Phi(v) \vdash \mathsf{wp}\ v\ [\Phi]$$

$$\text{WP-MONO}\quad \frac{\forall v. \Phi(v) \vdash \Psi(v)}{\mathsf{wp}\ e\ [\Phi] \vdash \mathsf{wp}\ e\ [\Psi]}$$

$$\text{WP-FRAME}\quad Q * \mathsf{wp}\ e\ [x.\, P] \vdash \mathsf{wp}\ e\ [x.\, Q * P]$$

$$\text{WP-BIND}\quad \mathsf{wp}\ e\ [x.\, \mathsf{wp}\ K[x]\ [\Phi]] \vdash \mathsf{wp}\ K[e]\ [\Phi]$$

Rules for basic language constructs.

$$\text{WP-ALLOC}\quad \frac{}{\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell) \vdash \mathsf{wp}\ \mathbf{ref}(v)\ [\Phi]}$$

$$\text{WP-LOAD}\quad \frac{}{\ell \mapsto v * \ell \mapsto v \mathrel{-\!\!*} \Phi(v) \vdash \mathsf{wp}\ !\,\ell\ [\Phi]}$$

$$\text{WP-STORE}\quad \frac{}{\ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} \Phi()) \vdash \mathsf{wp}\ (\ell \leftarrow w)\ [\Phi]}$$

$$\text{WP-PURE}\quad \frac{e \longrightarrow_{\text{pure}} e'}{\mathsf{wp}\ e'\ [\Phi] \vdash \mathsf{wp}\ e\ [\Phi]}$$

Pure reductions.

$$(\mathbf{rec}\ f(x) = e)v \longrightarrow_{\text{pure}} e[v/x][fx := e/f] \qquad \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow_{\text{pure}} e_1$$

$$\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow_{\text{pure}} e_2 \qquad \mathbf{fst}(v_1, v_2) \longrightarrow_{\text{pure}} v_1$$

$$\mathbf{snd}(v_1, v_2) \longrightarrow_{\text{pure}} v_2 \qquad \frac{\circledcirc_1 v = w}{\circledcirc_1 v \longrightarrow_{\text{pure}} w} \qquad \frac{v_1 \circledcirc_2 v_2 = v_3}{v_1 \circledcirc_2 v_2 \longrightarrow_{\text{pure}} v_3}$$

$$\mathbf{match}\ \mathbf{inl}\, v\ \mathbf{with}\ \mathbf{inl}\, x \Rightarrow e_1 \mid \mathbf{inr}\, x \Rightarrow e_2\ \mathbf{end} \longrightarrow_{\text{pure}} e_1[v/x]$$

$$\mathbf{match}\ \mathbf{inr}\, v\ \mathbf{with}\ \mathbf{inl}\, x \Rightarrow e_1 \mid \mathbf{inr}\, x \Rightarrow e_2\ \mathbf{end} \longrightarrow_{\text{pure}} e_2[v/x]$$

Context rules

$$\begin{aligned} K \in \mathit{Ctx} ::= {} & \bullet \mid e\,K \mid K\,v \mid \circledcirc_1 K \mid e \circledcirc_2 K \mid K \circledcirc_2 v \mid \mathbf{if}\ K\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \\ & (e, K) \mid (K, v) \mid \mathbf{fst}(K) \mid \mathbf{snd}(K) \mid \\ & \mathbf{inl}(K) \mid \mathbf{inr}(K) \mid \mathbf{match}\ K\ \mathbf{with}\ \mathbf{inl} \Rightarrow e_1 \mid \mathbf{inr} \Rightarrow e_2\ \mathbf{end} \mid \\ & \mathbf{AllocN}(e, K) \mid \mathbf{AllocN}(K, v) \mid \mathbf{Free}(K) \mid !\,K \mid e \leftarrow K \mid K \leftarrow v \mid \end{aligned}$$

Figure 2.2: Rules for the weakest precondition assertion.

2.4 Persistent propositions and nested Hoare triples

In this section, first we define Hoare triples using the weakest precondition and in the process explain persistent propositions. Next we show how Hoare triples can be nested, and we end with a verification of an example where the persistence of Hoare triples is key.

$$\text{HOARE-DEF}\quad [P]\ e\ [\Phi] \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ [\Phi])$$

We replace the previous definition of Hoare triples with this one. This definition is very similar to how we used weakest preconditions with a precondition. However, we wrap the weakest precondition with precondition in a persistence modality, $\Box$.

Persistent propositions In separation logic, many propositions we often use are ephemeral. They denote specific ownership and can't be duplicated. However, there are some statements in separation logic that do not denote ownership. These are statements like, True, $\ulcorner 1 = 1 \urcorner$ and program specifications. For propositions such as these, it would be very useful if we could duplicate them. These propositions are called *persistent* in Iris terminology.

PRESISTENCE
$$\mathsf{persistent}(P) \triangleq P \vdash \Box\, P$$

Persistence is defined using the persistence modality, and is closed under (separating) conjunction, disjunction and quantifiers. Any proposition under the persistence modality can be duplicated, as can be seen in the rule $\Box$-DUP below. To prove a proposition under a persistence modality, we are only allowed to use the persistent propositions in our assumptions, as can be seen in the rule $\Box$-MONO below.

$\Box$-DUP
$$\Box\, P \dashv\vdash \Box\, P * \Box\, P$$

$\Box$-SEP
$$\Box\,(P * Q) \dashv\vdash \Box\, P * \Box\, Q$$

$\Box$-MONO
$$\frac{P \vdash Q}{\Box\, P \vdash \Box\, Q}$$

$\Box$-E
$$\Box\, P \vdash P$$

$\Box$-CONJ
$$\Box\, P \wedge Q \vdash \Box\, P * Q$$

$$\ulcorner \phi \urcorner \vdash \Box \ulcorner \phi \urcorner$$
$$\mathsf{True} \vdash \Box\, \mathsf{True}$$

$$\Box\, P \vdash \Box\,\Box\, P$$
$$\forall x.\ \Box\, P \vdash \Box\, \forall x.\, P$$
$$\Box\, \exists x.\, P \vdash \exists x.\ \Box\, P$$

From the above rules we can derive the following rule for introducing persistent propositions.

$\Box$-I
$$\frac{\mathsf{persistent}(P) \qquad P \vdash Q}{P \vdash \Box\, Q}$$

We keep that the assumption is persistent and are thus still allowed to duplicate the assumption.

Nested Hoare triples In HeapLang we functions are first class citizens. Thus values can contain function, at that point often called closures. Closures can be passed to functions and can be returned and stored on the heap. When we have a closure, we can use it multiple times and thus might need to duplicate the specification of the closure multiple times. This is why Hoare triples are persistent. Take the following example with its specification.

$$\begin{aligned} &\text{refadd} \ := \ \lambda n.\, \lambda \ell.\, \ell \leftarrow\, !\,\ell + n \\ &[\mathsf{True}]\ \text{refadd}\ n\,[f.\, \forall \ell.\ [\ell \mapsto m]\ f\,\ell\,[\ell \mapsto m + n]] \end{aligned}$$

This program takes a value n and then returns a closure which we can call with a pointer to add n to the value of that pointer. The specification of refadd has as postcondition another Hoare triple for the returned closure. We just need one more derived rule before we can apply this specification of refadd in a proof.

$$\frac{\overset{\text{WP-APPLY}}{}P \vdash [R]\,e\,[\Psi] \qquad Q \vdash R * \forall v.\, \Psi(v) \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ [\Phi]}{P * Q \vdash \mathsf{wp}\ K[e]\ [\Phi]}$$

This rule expresses that to prove a weakest precondition of an expression in a context, while having a Hoare triple for that expression. We can apply the Hoare triple and use the postcondition to infer a value for the continued proof of the weakest precondition. This rule is derived by using the WP-FRAME, WP-MONO and WP-BIND rules.

We now give an example where a returned function is used twice, thus where the persistence of Hoare triples is needed.

Lemma 2.1

Given that the following Hoare triples holds

$$[\mathsf{True}]\ \text{refadd}\ n\ [f.\, \forall \ell.\ [\ell \mapsto m]\ f\ \ell\ [\ell \mapsto m+n]]$$

This specification holds.

$$\begin{array}{l} [\mathsf{True}] \\ \quad \mathbf{let}\ g = \text{refadd}\ 10\ \mathbf{in} \\ \quad \mathbf{let}\ \ell = \mathbf{ref}\ 0\ \mathbf{in} \\ \quad g\,\ell; g\,\ell; !\,\ell \\ [20.\ \mathsf{True}] \end{array}$$

Proof. We use HOARE-DEF and introduce the persistence modality and wand. We now need to prove the following.

$$\mathsf{wp}\left(\begin{array}{l} \mathbf{let}\ g = \text{refadd}\ 10\ \mathbf{in} \\ \mathbf{let}\ \ell = \mathbf{ref}\ 0\ \mathbf{in} \\ g\,\ell; g\,\ell; !\,\ell \end{array}\right)[20.\ \mathsf{True}]$$

We apply the WP-BIND rule with the following context

$$K = \begin{array}{l} \mathbf{let}\ g = \bullet\ \mathbf{in} \\ \mathbf{let}\ \ell = \mathbf{ref}\ 0\ \mathbf{in} \\ g\,\ell; g\,\ell; !\,\ell \end{array}$$

Resulting in the following weakest precondition we need to prove.

$$\mathsf{wp}\ \text{refadd}\ 10\left[v.\,\mathsf{wp}\left(\begin{array}{l} \mathbf{let}\ g = v\ \mathbf{in} \\ \mathbf{let}\ \ell = \mathbf{ref}\ 0\ \mathbf{in} \\ g\,\ell; g\,\ell; !\,\ell \end{array}\right)[20.\ \mathsf{True}]\right]$$

We now use the WP-APPLY to get the following statement we need to prove.

$$\mathsf{wp}\left(\begin{array}{l} \mathbf{let}\ g = f\ \mathbf{in} \\ \mathbf{let}\ \ell = \mathbf{ref}\ 0\ \mathbf{in} \\ g\,\ell; g\,\ell; !\,\ell \end{array}\right)[20.\ \mathsf{True}]$$

With as assumption, the following.

$$\forall \ell.\ [\ell \mapsto m]\, f\,\ell\, [\ell \mapsto m + 10]$$

Applying WP-PURE gets us the following statement to prove.

$$\mathsf{wp}\ \begin{pmatrix} \textbf{let}\,\ell = \textbf{ref}\,0\,\textbf{in} \\ f\,\ell; f\,\ell; !\,\ell \end{pmatrix}\ [20.\ \mathsf{True}]$$

Using WP-BIND and WP-ALLOC reaches the following statement to prove.

$$\mathsf{wp}\ \left(\ f\,\ell; f\,\ell; !\,\ell\ \right)\ [20.\ \mathsf{True}]$$

With as added assumption that, $\ell \mapsto 0$ holds. We can now duplicate the Hoare triple about f we have as assumption. We use WP-BIND with the first instance of the Hoare triple and the assumption about ℓ applied using WP-APPLY. This is repeated, and we reach the following proof state.

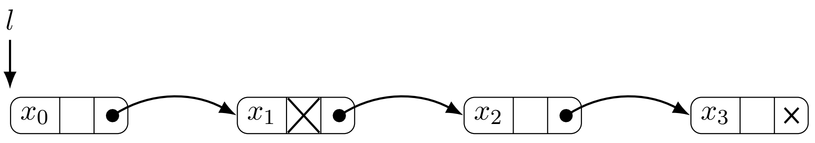
$$\mathsf{wp}\ !\,\ell\,[20.\ \mathsf{True}]$$

With as assumption that $\ell \mapsto 20$ holds. We can now use the WP-LOAD rule to prove the statement.

□

2.5 Representation predicates

We have shown in the previous three sections how one can represent simple states of the heap in separation logic and reason about it together with the program. However, this strategy of does not work for defining predicated for complicated data types. One such data type is the MLL. We want to connect an MLL in memory to a mathematical list. In section 2.3, we used the predicate $\mathsf{isMLL}\ hd\,\vec{v}$. In the next chapter we show how such a predicate can be defined, in this section we show how such a predicate can be used. We start with an example of how isMLL is used.



We want to reason about the above state of memory. Using the predicate isMLL, we state that it represents the list $[x_0, x_2, x_3]$. This is expressed as, $\mathsf{isMLL}\ (\textbf{some}\,\ell)\ [x_0, x_2, x_3]$.

In order to demonstrate how isMLL functions, we provide the inductive property listed below. In chapter 3 we will show how isMLLis defined and that it has the below property.

$$\begin{aligned} \mathsf{isMLL}\ hd\,\vec{v} = \quad & hd = \textbf{none} * \vec{v} = [\,] \vee \\ & (\exists \ell, v', tl.\ hd = \textbf{some}\,l * l \mapsto (v', \textbf{true}, tl) * \mathsf{isMLL}\ tl\,\vec{v}) \vee \\ & \left(\exists \ell, v', \vec{v}'', tl.\ \begin{matrix} hd = \textbf{some}\,l * l \mapsto (v', \textbf{false}, tl) * \\ \vec{v} = v' :: \vec{v}'' * \mathsf{isMLL}\ tl\,\vec{v}'' \end{matrix}\right) \end{aligned}$$

The predicate isMLLfor a hd and $\vec{v}$ holds if either of the below three options are true, as signified by the disjunction.

- The hd is **none** and thus the mathematical list, $\vec{v}$ is also empty
- The hd contains a pointer to some node, this node is marked as deleted and the tail is a MLL represented by the original list $\vec{v}$. Note that the location ℓ cannot be used again in the list, as it is disjoint by use of the separating conjunction.
- The value hd contains a pointer to some node, and this node is not marked as deleted. The list $\vec{v}$ now starts with the value v' and ends in the list $\vec{v}''$. Lastly, the value tl is a MLL represented by this mathematical list $\vec{v}''$

Since isMLL is an inductive predicate, we can define an induction principle. In chapter 3, we will show how this induction principle can be derived from the definition of isMLL.

$$\frac{\begin{array}{c}\textsf{isMLL-}\textsc{ind}\\ \mathsf{True} \vdash \Phi\ \mathbf{none}\ [] \qquad l \mapsto (v', \mathbf{true}, tl) * (\mathsf{isMLL}\ tl\ \vec{v} \wedge \Phi\ tl\ \vec{v}) \vdash \Phi\ (\mathbf{some}\ l)\ \vec{v} \\ l \mapsto (v', \mathbf{false}, tl) * (\mathsf{isMLL}\ tl\ \vec{v} \wedge \Phi\ tl\ \vec{v}) \vdash \Phi\ (\mathbf{some}\ l)\ (v' :: \vec{v})\end{array}}{\mathsf{isMLL}\ hd\ \vec{v} \vdash \Phi\ hd\ \vec{v}}$$

To use this rule, we need two things. We need to have an assumption of the shape $\mathsf{isMLL}\ hd\ \vec{v}$, and we need to prove a predicate Φ that takes these same hd and $\vec{v}$ as variables. We then need to prove that Φ holds for the three cases of the induction principle of isMLL.

Case Empty MLL: This is the base case, we have to prove Φ with **none** and the empty list.

Case Marked Head: This is the first inductive case, we have to prove Φ for a head containing a pointer ℓ and the list $\vec{v}$. We have the assumption that ℓ points to a node that is marked as deleted and contains a possible null pointer tl. We also have the following induction hypothesis: the tail, tl, is a MLL represented by $\vec{v}$, and Φ holds for tl and $\vec{v}$.

Case Unmarked head: This is the second inductive case, we have to prove Φ for a head containing a pointer ℓ and a list with as first element v' and the rest of the list is named $\vec{v}$. We have the assumption that ℓ points to a node that is marked as not deleted, and the node contains a possible null pointer tl. We also have the following induction hypothesis: the tail, tl, is a MLL represented by $\vec{v}$, and Φ holds for tl and $\vec{v}$.

The induction hypothesis in the last two cases is different from statements we have seen so far in separation logic, it uses the normal conjunction. We use the normal conjunction, since both $\mathsf{isMLL}\ tl\ \vec{v}$ and $\Phi\ tl\ \vec{v}$ reason about the section of memory containing tl. We thus cannot split the memory in two for these statements. This also has a side effect on how we use the induction hypothesis. We can only use one side of the conjunction in any one branch of the proof. We see this in practice in the next section, section 2.6.

2.6 Proof of delete in MLL

In this section, we prove the specification of delete. Recall the definition of delete.

```
delete $hd\,i$ = match $hd$ with
                   none  ⇒ ()
                 | some $\ell$ ⇒ let $(x, mark, tl) = !\,\ell$ in
                              if $mark = $ false && $i = 0$ then
                                $\ell \leftarrow (x,$ true$, tl)$
                              else if $mark = $ false then
                                delete $tl\ (i-1)$
                              else
                                delete $tl\ i$
                 end
```

Lemma 2.2

For any index $i \geq 0$, $\vec{v} \in \mathit{List}(\mathit{Val})$ and $hd \in \mathit{Val}$,

$$[\mathsf{isMLL}\ hd\ \vec{v}]\ \text{delete}\ hd\ i\ [\mathsf{isMLL}\ hd\ (\mathsf{remove}\ i\ \vec{v})]$$

Proof. We first use the definition of a Hoare triple, HOARE-DEF, to obtain the associated weakest precondition.

$$\Box(\mathsf{isMLL}\ hd\ \vec{v} \twoheadrightarrow \mathsf{wp}\ \text{delete}\ hd\ i\ [\mathsf{isMLL}\ hd\ (\mathsf{remove}\ i\ \vec{v})])$$

Since we have only pure assumptions we can assume $\mathsf{isMLL}\ hd\ \vec{v}$, and we now have to prove:

$$\mathsf{wp}\ \text{delete}\ hd\ i\ [\mathsf{isMLL}\ hd\ (\mathsf{remove}\ i\ \vec{v})]$$

We do strong induction on $\mathsf{isMLL}\ hd\ \vec{v}$ as defined by rule isMLL-IND. For Φ we take:

$$\Phi\ hd\ \vec{v} \triangleq \forall i.\ \mathsf{wp}\ \text{delete}\ hd\ i\ [\mathsf{isMLL}\ hd\ (\mathsf{remove}\ i\ \vec{v})]$$

We need to prove three cases:

Empty MLL: We need to prove the following

$$\mathsf{wp}\ \text{delete}\ \textbf{none}\ \ i\ [\mathsf{isMLL}\ \textbf{none}\ (\mathsf{remove}\ i\ [])]$$

We can now repeatedly use the WP-PURE rule and finish with the rule WP-VALUE to arrive at the following statement that we have to prove:

$$\mathsf{isMLL}\ \textbf{none}\ (\mathsf{remove}\ i\ [])$$

This follows from the definition of isMLL

Marked Head: We know that $\ell \mapsto (v', \textbf{true}, tl)$ with disjointly as IH the following:

$$(\forall i.\ \mathsf{wp}\ \text{delete}\ tl\ i\ [\mathsf{isMLL}\ tl\ (\mathsf{remove}\ i\ \vec{v})]) \wedge \mathsf{isMLL}\ tl\ \vec{v}$$

And, we need to prove that:

$$\mathsf{wp}\ \text{delete}\,(\texttt{some}\,\ell)\ i\,[\mathsf{isMLL}\,(\texttt{some}\,\ell)\ (\mathsf{remove}\,i\,\vec{v})]$$

By using the WP-PURE rule, we get that we need to prove:

$$\mathsf{wp}\left(\begin{array}{l}\texttt{let}\,(x, mark, tl) = !\,\ell\ \texttt{in}\\ \texttt{if}\ mark = \texttt{false}\ \&\&\ i = 0\ \texttt{then}\\ \quad \ell \leftarrow (x, \texttt{true}, tl)\\ \texttt{else if}\ mark = \texttt{false then}\\ \quad \text{delete}\ tl\ (i-1)\\ \texttt{else}\\ \quad \text{delete}\ tl\ i\end{array}\right)[\mathsf{isMLL}\,(\texttt{some}\,\ell)\ (\mathsf{remove}\,i\,\vec{v})]$$

We can now use WP-BIND and WP-LOAD with $\ell \mapsto (v, \texttt{true}, tl)$ to get our new statement that we need to prove:

$$\mathsf{wp}\left(\begin{array}{l}\texttt{let}\,(x, mark, tl) = (v, \texttt{true}, tl)\ \texttt{in}\\ \texttt{if}\ mark = \texttt{false}\ \&\&\ i = 0\ \texttt{then}\\ \quad \ell \leftarrow (x, \texttt{true}, tl)\\ \texttt{else if}\ mark = \texttt{false then}\\ \quad \text{delete}\ tl\ (i-1)\\ \texttt{else}\\ \quad \text{delete}\ tl\ i\end{array}\right)[\mathsf{isMLL}\,(\texttt{some}\,\ell)\ (\mathsf{remove}\,i\,\vec{v})]$$

We now repeatedly use WP-PURE to reach the following:

$$\mathsf{wp}\ \text{delete}\ tl\ i\,[\mathsf{isMLL}\,(\texttt{some}\,\ell)\ (\mathsf{remove}\,i\,\vec{v})]$$

Which is the left-hand side of our IH.

Unmarked head: We know that $\ell \mapsto (v', \texttt{false}, tl)$ with disjointly as IH the following:

$$\forall i.\ \mathsf{wp}\ \text{delete}\ tl\ i\,\left[\mathsf{isMLL}\ tl\,(\mathsf{remove}\,i\,\vec{v}'')\right] \wedge \mathsf{isMLL}\ tl\,\vec{v}''$$

And, we need to prove that:

$$\mathsf{wp}\ \text{delete}\,(\texttt{some}\,\ell)\ i\,\left[\mathsf{isMLL}\,(\texttt{some}\,\ell)\ (\mathsf{remove}\,i\,(v' :: \vec{v}''))\right]$$

We repeat the steps from the previous case, except for using $\ell \mapsto (v, \texttt{false}, tl)$ with the WP-LOAD rule, until we repeatedly use WP-PURE. We instead use WP-PURE once to reach the following statement:

$$\mathsf{wp}\left(\begin{array}{l}\texttt{if}\ \texttt{false} = \texttt{false}\ \&\&\ i = 0\ \texttt{then}\\ \quad \ell \leftarrow (v', \texttt{true}, tl)\\ \texttt{else if}\ \texttt{false} = \texttt{false then}\\ \quad \text{delete}\ tl\ (i-1)\\ \texttt{else}\\ \quad \text{delete}\ tl\ i\end{array}\right)\left[\mathsf{isMLL}\,(\texttt{some}\,\ell)\ (\mathsf{remove}\,i\,(v' :: \vec{v}''))\right]$$

Here we do a case distinction on whether $i = 0$, thus, if we want to delete the current head of the MLL.

Case $i = 0$: We repeatedly use WP-PURE until we reach:

$$\mathsf{wp}\ \ell \leftarrow (v, \mathbf{true}, tl)\ \left[\mathsf{isMLL}\,(\mathbf{some}\,\ell)\ (\mathsf{remove}\,0\,(v' :: \vec{v}''))\right]$$

We then use WP-STORE with $\ell \mapsto (v, \mathbf{true}, tl)$, which we retained after the previous use of WP-LOAD, and $-\!\!*$I-E. We now get that $\ell \mapsto (v', \mathbf{false}, tl)$, and we need to prove:

$$\mathsf{wp}\ ()\ \left[\mathsf{isMLL}\,(\mathbf{some}\,\ell)\ (\mathsf{remove}\,0\,(v' :: \vec{v}''))\right]$$

We use WP-VALUE to reach:

$$\mathsf{isMLL}\,(\mathbf{some}\,\ell)\ (\mathsf{remove}\,0\,(v' :: \vec{v}''))$$

This now follows from the fact that $(\mathsf{remove}\,0\,(v' :: \vec{v}'')) = \vec{v}''$ together with the definition of isMLL, $\ell \mapsto (v', \mathbf{false}, tl)$ and the IH.

Case $i > 0$: We repeatedly use WP-PURE until we reach:

$$\mathsf{wp}\ \text{delete}\ tl\ (i-1)\ \left[\mathsf{isMLL}\,(\mathbf{some}\,\ell)\ (\mathsf{remove}\,(i-1)\,(v' :: \vec{v}''))\right]$$

We use WP-MONO with as assumption our the left-hand side of the IH. We now need to prove the following:

$$\mathsf{isMLL}\ tl\,(\mathsf{remove}\,i\ \vec{v}'') \vdash \mathsf{isMLL}\,(\mathbf{some}\,\ell)\ (\mathsf{remove}\,(i-1)\,(v' :: \vec{v}''))$$

This follows from the fact that $(\mathsf{remove}\,(i-1)\,(v' :: \vec{v}'')) = v' :: (\mathsf{remove}\,i\ \vec{v}'')$ together with the definition of isMLL and $\ell \mapsto (v, \mathbf{false}, tl)$, which we retained from WP-LOAD. $\square$

Chapter 3

Fixpoints for representation predicates

In this chapter we show how non-structurally recursive representation predicates can be defined using least fixpoints. In section 3.1, we explain why it is difficult to define non-structurally recursive predicates and generally explain the approach that is taken. Next, in section 3.2, we show the way least fixpoints are defined in Iris. Lastly, in section 3.3 we explain the improvements we made to the approach of Iris in order for the process to be automated.

3.1 Problem statement

In order to define a recursive predicate, we have to prove it actually exists. One way of defining recursive predicates is by structural recursion. Thus, every recursive call in the predicate has to be on a structurally smaller part of the arguments.

The candidate argument for structural recursion in isMLL would be the list of values used to represent the MLL. However, this does not work given the second case of the recursion.

$$\mathsf{isMLL}\, hd\, \vec{v} = \cdots \vee (\exists \ell, v', tl.\ hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{true}, tl) * \mathsf{isMLL}\, tl\, \vec{v}) \vee \cdots$$

Here, the list of values is passed straight onto the recursive call to isMLL. Thus, it is not structurally recursive.

We need another approach to define non-structurally recursive predicates such as these. Iris has several approaches to resolve this problem, as is discussed in chapter 7. The approach we use as the basis of `eiInd` is the least fixpoint, inspired by the Knaster-Tarski fixpoint theorem [Tar55]. Given a monotone function on predicates, there exists a least fixpoint of this function. We can now choose a function such that the fixpoint corresponds to the recursive predicate we wanted to design. This procedure is explained thoroughly in the next section, section 3.2.

3.2 Least fixpoint in Iris

To define a least fixpoint in Iris, the first step is to have a monotone function.

Definition 3.1: Monotone function

Function $\mathsf{F}\colon (A \to \mathit{iProp}) \to A \to \mathit{iProp}$ is monotone when for any $\Phi, \Psi\colon A \to \mathit{iProp}$, it holds that

$$\Box(\forall y.\, \Phi\, y \mathrel{-\!\!*} \Psi\, y) \vdash \forall x.\, \mathsf{F}\, \Phi\, x \mathrel{-\!\!*} \mathsf{F}\, \Psi\, x$$

In other words, F is monotone in its first argument.

This definition of monotone follows the definition of monotone in other fields, with one exception. The assumption has an additional restriction, it has to be persistent. The persistence is necessary since F could use its monotone argument multiple times.

Example 3.2

Take the following function.

$$\begin{aligned}\mathsf{F}\, \Phi\, v \triangleq{}& (v = \mathtt{none}) \vee \\ & (\exists \ell_1, \ell_2, v_1, v_2.\, v = \mathtt{some}(\ell_1, \ell_2) * \ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 * \Phi\, v_1 * \Phi\, v_2)\end{aligned}$$

This is the function for binary trees. The value v is either empty, and we have an empty tree. Or v contains two locations, for the two branches of the tree. Each location points to a value and Φ is holds for both of these values. The fixpoint, as is discussed in theorem 3.3, of this function holds for a value containing a binary tree. However, before we can take the fixpoint we have to prove it is monotone.

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) \vdash \forall v.\, \mathsf{F}\, \Phi\, v \mathrel{-\!\!*} \mathsf{F}\, \Psi\, v$$

Proof. We start by introducing v and the wand.

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) * \mathsf{F}\, \Phi\, v \vdash \mathsf{F}\, \Psi\, v$$

We now unfold the definition of F and eliminate and introduce the disjunction, resulting in two statements to prove.

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) * v = \mathtt{none} \vdash v = \mathtt{none}$$

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) * \left(\exists \ell_1, \ell_2, v_1, v_2.\, \begin{array}{l} v = \mathtt{some}(\ell_1, \ell_2) * \ell_1 \mapsto v_1 * \\ \ell_2 \mapsto v_2 * \Phi\, v_1 * \Phi\, v_2 \end{array}\right) \vdash$$
$$\left(\exists \ell_1, \ell_2, v_1, v_2.\, \begin{array}{l} v = \mathtt{some}(\ell_1, \ell_2) * \ell_1 \mapsto v_1 * \\ \ell_2 \mapsto v_2 * \Psi\, v_1 * \Psi\, v_2 \end{array}\right)$$

The first statement holds directly. For the second statement, we eliminate the existentials in the assumption and use the created variables to introduce the existentials in the conclusion.

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) * \begin{array}{l} v = \mathtt{some}(\ell_1, \ell_2) * \\ \ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 * \\ \Phi\, v_1 * \Phi\, v_2 \end{array} \vdash \begin{array}{l} v = \mathtt{some}(\ell_1, \ell_2) * \\ \ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 * \\ \Psi\, v_1 * \Psi\, v_2 \end{array}$$

Any sub propositions that occur both on the left and right-hand side are canceled out using $*$-MONO.

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) * \Phi\, v_1 * \Phi\, v_2 \vdash \Psi\, v_1 * \Psi\, v_2$$

We want to split the conclusion and premise in two, such that we get the following statements, with $i \in \{1, 2\}$.

$$\Box(\forall w.\, \Phi\, w \mathrel{-\!\!*} \Psi\, w) * \Phi\, v_i \vdash \Psi\, v_i$$

To achieve this split, we duplicate the persistent premise and then split using $*$-MONO again. Both these statements hold trivially. □

In the previous proof it was essential that the premise of monotonicity is persistent. This occurs any time we have a data structure with more than one branch.

Now that we have a definition of a function, we can prove that a least fixpoint of a monotone function always exists.

Theorem 3.3: Least fixpoint

Given a monotone function $\mathsf{F} \colon (A \to \mathit{iProp}) \to A \to \mathit{iProp}$, called the *pre fixpoint function*, there exists a least fixpoint $\mu\mathsf{F} \colon A \to \mathit{iProp}$ such that

1. The fixpoint equality holds

$$\mu\mathsf{F}\, x \dashv\vdash \mathsf{F}\,(\mu\mathsf{F})\, x$$

2. The iteration property holds

$$\Box\, \forall y.\, \mathsf{F}\, \Phi\, y \mathrel{-\!\!*} \Phi\, y \vdash \forall x.\, \mu\mathsf{F}\, x \mathrel{-\!\!*} \Phi\, x$$

Proof. Given a monotone function $\mathsf{F} \colon (A \to \mathit{iProp}) \to A \to \mathit{iProp}$ we define $\mu\mathsf{F}$ as

$$\mu\mathsf{F}\, x \triangleq \forall \Phi.\ \Box(\forall y.\, \mathsf{F}\, \Phi\, y \mathrel{-\!\!*} \Phi\, y) \mathrel{-\!\!*} \Phi\, x$$

We now prove the two properties of the least fixpoint

1. The right to left direction follows from monotonicity of F. The left to right direction follows easily from monotonicity of F and the right to left direction.

2. This follows directly from unfolding the definition of $\mu\mathsf{F}$. □

The first property of theorem 3.3, fixpoint equality, defines that the least fixpoint is a fixpoint. The second property of theorem 3.3, iteration, ensures that this fixpoint is the least of the possible fixpoints. The iteration property is a simpler version of the induction principle. The induction hypothesis during iteration is simpler. It only ensures that Φ holds under F. Full induction requires that we also know that the fixpoint holds under F in the induction hypothesis.

Lemma 3.4: Induction principle

Given a monotone predicate $\mathsf{F} \colon (A \to \mathit{iProp}) \to (A \to \mathit{iProp})$, it holds that

$$\Box(\forall x.\, \mathsf{F}\,(\lambda y.\, \Phi\, y \wedge \mu\mathsf{F}\, y)\, x \mathrel{-\!\!*} \Phi\, x) \mathrel{-\!\!*} \forall x.\, \mu\mathsf{F}\, x \mathrel{-\!\!*} \Phi\, x$$

Proof. The induction principle for a Ψ holds by the iteration property with $\Phi x = \Psi\, x \wedge \mu\mathsf{F}\, x$ □

This lemma follows from monotonicity and the least fixpoint properties.
We can now use the above steps to define isMLL

Example 3.5: Iris least fixpoint of isMLL

We want to create a least fixpoint such that it has the following inductive property.

$$\begin{aligned}\mathsf{isMLL}\; hd\, \vec{v} = \quad & hd = \mathtt{none} * \vec{v} = [] \vee \\ & (\exists \ell, v', tl.\; hd = \mathtt{some}\, l * l \mapsto (v', \mathtt{true}, tl) * \mathsf{isMLL}\; tl\, \vec{v}) \vee \\ & \left(\exists \ell, v', \vec{v}'', tl.\; \begin{matrix} hd = \mathtt{some}\, l * l \mapsto (v', \mathtt{false}, tl) * \\ \vec{v} = v' :: \vec{v}'' * \mathsf{isMLL}\; tl\, \vec{v}'' \end{matrix}\right)\end{aligned}$$

The first step is creating the pre fixpoint function. We accomplish this by adding an argument to isMLL and then transforming it into a function. Next, we substitute any recursive calls to isMLL with this argument.

$$\begin{aligned}\mathsf{isMLL_F}\; \Phi\; hd\, \vec{v} \triangleq \quad & hd = \mathtt{none} * \vec{v} = [] \vee \\ & (\exists \ell, v', tl.\; hd = \mathtt{some}\, l * l \mapsto (v', \mathtt{true}, tl) * \Phi\; tl\, \vec{v}) \vee \\ & \left(\exists \ell, v', \vec{v}'', tl.\; \begin{matrix} hd = \mathtt{some}\, l * l \mapsto (v', \mathtt{false}, tl) * \\ \vec{v} = v' :: \vec{v}'' * \Phi\; tl\, \vec{v}'' \end{matrix}\right)\end{aligned}$$

This has created a function, $\mathsf{isMLL_F}$. The function applies the predicate, Φ, on the tail of any possible MLL, while ensuring the head is part of an MLL. Next, we want to prove that $\mathsf{isMLL_F}$ is monotone. However, $\mathsf{isMLL_F}$ has the following type.

$$\mathsf{isMLL_F} : (\mathit{Val} \to \mathit{List\; Val} \to \mathit{iProp}) \to \mathit{Val} \to \mathit{List\; Val} \to \mathit{iProp}$$

But, definition 3.1 only works for functions of type

$$\mathsf{F} : (A \to \mathit{iProp}) \to A \to \mathit{iProp}$$

This is solved by uncurrying $\mathsf{isMLL_F}$

$$\mathsf{isMLL'_F}\; \Phi\, (hd, \vec{v}) \triangleq \mathsf{isMLL_F}\; \Phi\; hd\, \vec{v}$$

The function $\mathsf{isMLL'_F}$ now has the type

$$\mathsf{isMLL'_F} : (\mathit{Val} \times \mathit{List\; Val} \to \mathit{iProp}) \to \mathit{Val} \times \mathit{List\; Val} \to \mathit{iProp}$$

And we can prove $\mathsf{isMLL_F}$ is monotone.

$$\begin{gathered}\Box(\forall (hd, \vec{v}).\; \Phi\, (hd, \vec{v}) \mathrel{-\!\!*} \Psi\, (hd, \vec{v})) \\ \vdash \forall (hd, \vec{v}).\; \mathsf{isMLL'_F}\; \Phi\, (hd, \vec{v}) \mathrel{-\!\!*} \mathsf{isMLL'_F}\; \Psi\, (hd, \vec{v})\end{gathered}$$

Proof. We use a similar proof as in example 3.2. It involves more steps as we have more branches, but the same ideas apply. $\Box$

Given that $\mathsf{isMLL'_F}$ is monotone, we now know from theorem 3.3 that the least fixpoint exists of $\mathsf{isMLL'_F}$. By uncurrying we can create the final definition of isMLL.

$$\mathsf{isMLL}\; hd\, \vec{v} \triangleq \mu(\mathsf{isMLL'_F})\, (hd, \vec{v})$$

This definition of isMLL has the inductive property as described in section 2.5. That property is the fixpoint equality. After expanding any currying, we get the below induction principle for isMLL from lemma 3.4.

$$\begin{aligned}&\Box(\forall hd, \vec{v}.\, \mathsf{isMLL_F}\, (\lambda hd', \vec{v}'.\, \Phi\, hd'\, \vec{v}' \wedge \mathsf{isMLL}\, hd'\, \vec{v}')\, hd\, \vec{v} \mathrel{-\!\!*} \Phi\, hd\, \vec{v})\\ &\mathrel{-\!\!*} \forall hd, \vec{v}.\, \mathsf{isMLL}\, hd\, \vec{v} \mathrel{-\!\!*} \Phi\, hd\, \vec{v}\end{aligned}$$

The induction principle from section 2.5 is also derivable from lemma 3.4. The three cases of the induction principle follow from the disjunctions in $\mathsf{isMLL_F}$.

3.3 Syntactic monotone proof search

As we discussed in chapter 1, the goal of this thesis is to show how to automate the definition of representation predicates from inductive definitions. The major hurdle in this process can be seen in example 3.5, proving a function monotone. In this section, we show how a monotonicity proof can be found by using syntactic proof search.

We base our strategy on the work by Sozeau [Soz09]. They create a system for rewriting expressions in goals in Coq under generalized relations, instead of just equality. Many definitions are equal, but do them in separation logic instead of the logic of Coq. The proof search itself is not based on the generalized rewriting of Sozeau.

We take the following strategy. We prove the monotonicity of all the connectives once. We now prove the monotonicity of the function by making use of the monotonicity of the connectives with which it is built.

Monotone connectives We don't want to uncurry every connective when using that it is monotone, and thus we take a different approach to what is monotone. For every connective we give a signature telling us how it is monotone. We show a few of these signatures below.

Connective	Type	Signature
$*$	$iProp \to iProp \to iProp$	$(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$
$\vee$	$iProp \to iProp \to iProp$	$(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$
$\mathrel{-\!\!*}$	$iProp \to iProp \to iProp$	$\mathsf{flip}(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$
$\exists$	$(A \to iProp) \to iProp$	$((=) \Longrightarrow (\mathrel{-\!\!*})) \Longrightarrow (\mathrel{-\!\!*})$

We make use of the Haskell prefix notation, $(\mathrel{-\!\!*})$, to turn an infix operator into a prefix function. The signature of a connective defines the requirements for monotonicity a connective has. The signatures are based on building relations, which we can apply on the connectives.

Definition 3.6: Relation in *iProp*

A relation in separation logic on type A is defined as

$$iRel\ A \triangleq A \to A \to iProp$$

The combinators used to build signatures now build relations.

Definition 3.7: Respectful relation

The respectful relation $R \Longrightarrow R' \colon \mathit{iRel}\ (A \to B)$ of two relations $R \colon \mathit{iRel}\ A$, $R' \colon \mathit{iRel}\ B$ is defined as

$$R \Longrightarrow R' \triangleq \lambda f, g.\, \forall x, y.\, R\, x\, y \mathrel{-\!\!*} R'\, (f\, x)\, (g\, y)$$

Definition 3.8: Flipped relation

The flipped relation $\mathsf{flip}\, R \colon \mathit{iRel}\ A$ of a relation $R \colon \mathit{iRel}\ A$ is defined as

$$\mathsf{flip}\, R \triangleq \lambda x, y.\, R\, y\, x$$

Given a signature we can define when a connective has a signature.

Definition 3.9: Proper element of a relation

Given a relation $R \colon \mathit{iRel}\ A$ and an element $x \in A$, x is a proper element of R if $R\, x\, x$

We define how a connective is monotone by the signature it is a proper element of. The proofs that the connectives are the proper elements of their signature are fairly trivial, but we will highlight the existential qualifier.

Question: I want to expand the first two signatures, but I don't have anything interesting to say about it except for showing the expanded version

We can unfold the definitions in the signature and fill in the existential quantification in order to get the following statement,

$$\forall \Phi, \Psi.\, (\forall x, y.\, x = y \mathrel{-\!\!*} \Phi\, x \mathrel{-\!\!*} \Psi\, y) \mathrel{-\!\!*} (\exists x.\, \Phi\, x) \mathrel{-\!\!*} (\exists x.\, \Psi\, x)$$

This statement can be easily simplified by substituting y for x in the first relation.

$$\forall \Phi, \Psi.\, (\forall x.\, \Phi\, x \mathrel{-\!\!*} \Psi\, x) \mathrel{-\!\!*} (\exists x.\, \Phi\, x) \mathrel{-\!\!*} (\exists x.\, \Psi\, x)$$

We create a new combinator for signatures, the pointwise relation, to include the above simplification in signatures.

Definition 3.10: Pointwise relation

The pointwise relation $\gtrdot R$ is a special case of a respectful relation defined as

$$\gtrdot R \triangleq \lambda f, g.\, \forall x.\, R\, (f\, x)\, (g\, y)$$

The new signature for the existential quantification becomes

$$\gtrdot(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$$

Monotone functions To create a monotone function for the least fixpoint we need to be able to at least define definition 3.1 in terms of the proper element of a signature. We already have most of the combinators needed, but we are missing a way to mark a relation as persistent.

Definition 3.11: Persistent relation

The persistent relation $\square R\colon \mathit{iRel}\ A$ for a relation $R\colon \mathit{iRel}\ A$ is defined as

$$\square R \triangleq \lambda x, y.\ \square(R\,x\,y)$$

Thus we can create the following signature for definition 3.1.

$$\square(\gtrdot(-\!\!*)) \Longrightarrow \gtrdot(-\!\!*)$$

Filling in an F as the proper element, we get the following statement.

$$\square(\forall y.\, \mathit{\Phi}\, y -\!\!* \Psi\, y) -\!\!* \forall x.\, \mathsf{F}\,\mathit{\Phi}\, x -\!\!* \mathsf{F}\,\Psi\, x$$

Which is definition 3.1 but using only wands, instead of entailments. We use the same structure for the signature of $\mathsf{isMLL_F}$. But we add an extra pointwise to the left and right-hand side of the respectful relation for the extra argument.

$$\square(\gtrdot \gtrdot (-\!\!*)) \Longrightarrow \gtrdot \gtrdot (-\!\!*)$$

We can thus write down the monotonicity of a function without explicit currying and uncurrying.

Monotone proof search The monotone proof search is based on identifying the top-level relation and the top-level function beneath it. Thus, in the below proof state, the wand is the top-level relation and the disjunction is the top-level function.

$$\begin{array}{c}\text{Top-level function}\\ \square(\cdots) \vdash (\cdots \overset{\swarrow}{\vee} \cdots) \underset{\substack{\uparrow\\ \text{Top-level relation}}}{-\!\!*} (\cdots \overset{\searrow}{\vee} \cdots)\end{array}$$

Using these descriptions we show a proof using our monotone proof search. Then, we outline the steps we took in this proof.

Example 3.12: $\mathsf{isMLL_F}$ is monotone

The predicate $\mathsf{isMLL_F}$ is monotone in its first argument. Thus, $\mathsf{isMLL_F}$ is a proper element of

$$\square(\gtrdot \gtrdot (-\!\!*)) \Longrightarrow \gtrdot \gtrdot (-\!\!*)$$

In other words

$$\square(\forall hd\,\vec{v}.\, \mathit{\Phi}\, hd\,\vec{v} -\!\!* \Psi\, hd\,\vec{v}) -\!\!* \forall hd\,\vec{v}.\, \mathsf{isMLL_F}\,\mathit{\Phi}\, hd\,\vec{v} -\!\!* \mathsf{isMLL_F}\,\Psi\, hd\,\vec{v}$$

Proof. We assume any premises, $\square(\forall hd\,\vec{v}.\, \mathit{\Phi}\, hd\,\vec{v} -\!\!* \Psi\, hd\,\vec{v})$. We omit the premises in future proof states, but it is always there since it is persistent. Next, we introduce the universal quantifiers. After unfolding $\mathsf{isMLL_F}$, we have to prove the following.

$$(\cdots \vee \cdots \mathit{\Phi} \cdots) -\!\!* (\cdots \vee \cdots \Psi \cdots)$$

Thus, the top-level connective is the wand and the one below it is the disjunction. We now search for a signature ending on a magic wand and which has the disjunction as a proper element. We find the signature $(\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*})$ with $(\vee)$. We apply $((\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}) \Longrightarrow (\mathrel{-\!\!*}))(\vee)(\vee)$, resulting in two statements to prove.

$$\begin{gathered}(hd = \mathbf{none} * \vec{v} = []) \mathrel{-\!\!*} (hd = \mathbf{none} * \vec{v} = []) \\ (\cdots \Phi \cdots \vee \cdots \Phi \cdots) \mathrel{-\!\!*} (\cdots \Psi \cdots \vee \cdots \Psi \cdots)\end{gathered}$$

The first statement follows directly from reflexivity of the magic wand. The second statement utilizes the same disjunction signature again. Thus, we just show the result of applying it.

$$\begin{gathered}(\exists \ell, v', tl.\ \cdots \Phi \cdots) \mathrel{-\!\!*} (\exists \ell, v', tl.\ \cdots \Psi \cdots) \\ (\exists \ell, v', \vec{v}'', tl.\ \cdots \Phi \cdots) \mathrel{-\!\!*} (\exists \ell, v', \vec{v}'', tl.\ \cdots \Psi \cdots)\end{gathered}$$

Both statements have as top-level relation $(\mathrel{-\!\!*})$ with below it $\exists$. We apply the signature of $\exists$ with as result.

$$\begin{gathered}\forall \ell.\ (\exists v', tl.\ \cdots \Phi \cdots) \mathrel{-\!\!*} (\exists v', tl.\ \cdots \Psi \cdots) \\ \forall \ell.\ (\exists v', \vec{v}'', tl.\ \cdots \Phi \cdots) \mathrel{-\!\!*} (\exists v', \vec{v}'', tl.\ \cdots \Psi \cdots)\end{gathered}$$

We introduce ℓ and repeat these steps until the existential quantification is no longer the top-level function.

$$\begin{gathered}(hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{true}, tl) * \Phi\, tl\, \vec{v}) \mathrel{-\!\!*} \\ (hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{true}, tl) * \Psi\, tl\, \vec{v}) \\ \begin{pmatrix} hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{false}, tl) * \\ \vec{v} = v' :: \vec{v}'' * \Phi\, tl\, \vec{v}'' \end{pmatrix} \mathrel{-\!\!*} \\ \begin{pmatrix} hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{false}, tl) * \\ \vec{v} = v' :: \vec{v}'' * \Psi\, tl\, \vec{v}'' \end{pmatrix}\end{gathered}$$

We can now repeatedly apply the signature of $(*)$ and apply reflexivity to any created propositions without Φ or Ψ. This leaves us with

$$\begin{gathered}\square\, (\forall hd\, \vec{v}.\, \Phi\, hd\, \vec{v} \mathrel{-\!\!*} \Psi\, hd\, \vec{v}) \vdash \Phi\, tl\, \vec{v} \mathrel{-\!\!*} \Psi\, tl\, \vec{v} \\ \square\, (\forall hd\, \vec{v}.\, \Phi\, hd\, \vec{v} \mathrel{-\!\!*} \Psi\, hd\, \vec{v}) \vdash \Phi\, tl\, \vec{v}'' \mathrel{-\!\!*} \Psi\, tl\, \vec{v}''\end{gathered}$$

These hold from the assumption. □

The strategy we use for proof search consists of two steps. We have a normalization step, and we have an application step.

Normalization Introduce any universal quantifiers, extra created wands and modalities. Afterward, do an application step.

Application We apply the first option that works.

1. If the left and right-hand side of the relation are equal, and the relation is reflexive, apply reflexivity.

2. Check if the conclusion follows from a premise, and then apply it.
3. Look for a signature of the top-level function where the last relation matches the top-level relation of the conclusion. Apply it if we find one. Next, do a normalization step.

We start the proof with the normalization step and continue until all created branches are proven.

Generating the fixpoints theorem Given the above proof of monotonicity of $\mathsf{isMLL_F}$, theorem 3.3 does not give a least fixpoint for $\mathsf{isMLL_F}$. We change the definition to add an arbitrary number of arguments to the fixpoint and its properties.

$$\mu\mathsf{F}\, x_1 \cdots x_n \triangleq \forall \Phi.\ \Box(\forall y_1, \cdots, y_n.\, \mathsf{F}\, \Phi\, y_1 \cdots y_n \twoheadrightarrow \Phi\, y_1 \cdots y_n) \twoheadrightarrow \Phi\, x_1 \cdots x_n$$

This is not a valid definition in our logic for an arbitrary n. Thus, we create a least fixpoint theorem for any function we want to take a fixpoint of.

Example 3.13: isMLL least fixpoint theorem

We have the monotone function

$$\mathsf{isMLL_F} \colon (\mathit{Val} \to \mathit{List\,Val} \to \mathit{iProp}) \to \mathit{Val} \to \mathit{List\,Val} \to \mathit{iProp}$$

We use the above definition of the least fixpoint with $n = 2$.

$$\mu\mathsf{isMLL_F}\ hd\ \vec{v} \triangleq \forall \Phi.\ \Box(\forall hd', \vec{v}'.\ \mathsf{isMLL_F}\, \Phi\, hd'\, \vec{v}' \twoheadrightarrow \Phi\, hd'\, \vec{v}') \twoheadrightarrow \Phi\, hd\, \vec{v}$$

For the induction principle, we apply the same strategy. With isMLL, we get the induction principle as described in example 3.5.

Chapter 4

Implementing an Iris tactic in Elpi

In this chapter we will show how Elpi together with Coq-Elpi is used to create new Iris Proof Mode (IPM) tactics in Coq. This chapter explains the relevant inner working of IPM, give a tutorial on how Elpi works and how to create a tactic using Coq-Elpi, and finally set up the necessary functions for the commands and tactics around inductive predicates we will define in chapter 5.

In section 4.1, we give a short recap of how the `iIntros` tactic functions. Next in section 4.2 we explain how the Iris context is implemented in IPM. Next, in section 4.3, we explain the Iris lemmas we use as the building blocks for the Elpi version of the tactic. In section 4.4 we explain how to use Elpi and Coq-Elpi while developing the `iIntros` tactic.

4.1 `iIntros` example

The IPM `iIntros` tactic acts as the `intros` tactic but on Iris propositions and the Iris contexts. The `intros` tactic takes as its first argument instructions in a domain-specific language (DSL). Based on these instructions, it performs several proof steps. The `iIntros` implements a similar DSL as the Coq tactic. A few expansions were added as inspired by ssreflect [HKP97; GMT16], they are used to perform other common initial proof steps such as `simpl`, `done` and others. We will show two examples of how `iIntros` is used to help prove lemmas.

We have seen in chapter 2 how we have two types of propositions as our assumptions during a proof. There are persistent and non-persistent (also called spatial from now on) propositions. In the IPM there are two corresponding contexts, the persistent and spatial context. Consider the following Coq lemma:

```Coq
Lemma example1 : P -∗ □ Q -∗ P.
```

After applying `iIntros "HP #HQ"` we get

```Coq
P, Q: iProp
============
"HQ" : Q
```

```
------------□                                        Coq
"HP" : P
------------*
P
```

The tactic `iIntros "HP #HQ"` consist of two introduction patters applied after each other. `HP` introduces `P` intro the spatial context with the name `"HP"`. The `#HQ` introduces the next wand, but because of the `#` it is introduced into the persistent context (This fails if the proposition is not persistent).

The `iIntros` tactic also applies to universal quantifications, existential quantifications, separating conjunctions and disjunctions. Take the following proof state,

```
P: nat → iProp                                       Coq
===============================================
-----------------------------------------------*
∀ x : nat, (∃ y : nat, P x ∗ P y) ∨ P 0 -∗ P 1
```

We again use one application of `iIntros` to introduce and eliminate the premise.

`iIntros "%x [[%y [Hx ?]] | H0]"`

When applied we get two proof states, one for each side of the disjunction elimination.

```
(1/2)                                                Coq
P: nat → iProp
x, y: nat
==================
"Hx" : P x
"_" : 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 universal quantifier introduction or wand introduction. We then interpret the intro pattern for the introduced hypothesis. A few of the possible intro patterns are:

- `"?"` uses an anonymous identifier for the hypothesis.
- `"H"` names the hypothesis 'H' in the spatial context.
- `"#H"` names the hypothesis 'H' in the persistent context.
- `"%H"` introduces the the hyptothesis into the Coq context with name 'H'
- `"[IPL | IPR]"` performs a disjunction elimination on the hypothesis. The two contained introduction patterns are recursively applied.

- `"[IPL IPR]"` performs a separating conjunction elimination on the hypothesis. The two contained introduction patterns are recursively applied.

- `"[%x IP]"` performs existential quantifier introduction on the hypothesis. The variable is name ‘x’ and `IP` is applied recursively. Note that this introduction pattern overlaps with previous pattern. This pattern is tried first.

We 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

Before describing our implementation of the Elpi `eiIntros` tactic, we need a quick interlude about how the Iris contexts and entailment are defined in Coq.

The IPM creates the context using the following definitions

```coq
Inductive ident :=
  | IAnon : positive → ident
  | INamed :> string → ident.

Inductive env : Type :=
  | Enil : env
  | Esnoc : env → ident → iProp → env.

Record envs := Envs {
  env_persistent : env;
  env_spatial : env;
  env_counter : positive;
}.
```

An identifier is either anonymous and given only a number, or a name using a string. Identifiers are mapped to propositions using `env`. This is a reversed linked list. Hence, new assumptions in an environment get added to the end of the list using `Esnoc`. The context consists of two such maps, one for the persistent hypotheses and one for the spatial hypotheses. Lastly, it contains a counter for creating fresh anonymous identifiers.

We now define how a context is interpreted in an entailment.

```coq
Definition envs_entails
    (Δ : envs iProp) (Q : iProp) : Prop :=
      ⌜envs_wf (env_intuitionistic Δ) (env_spatial Δ)⌝
    ∧ □ [∧] (env_intuitionistic Δ)
    ∧ [∗] (env_spatial Δ)
```

```Coq
  ⊢ Q.
```

The persistent and spatial context are transformed into a proposition. The persistent context is combined using the iterated conjunction and surrounded by a persistence modality. The spatial context is simply combined using the iterated separating conjunction. Lastly, `envs_wf` ensures that every identifier only occurs once in the context.

Using `of_envs`, `envs_entails` defines entailment where the assumption is a context. Note that `envs_entails` is a Coq predicate, not a separation logic predicate. An `envs_entailment` statement is displayed as in section 4.1.

4.3 Tactics

To create the IPM tactics, lemmas are defined that apply a proof rule but transforms an `envs_entails` into another `envs_entails`.

```Coq
1 Lemma tac_wand_intro Δ i P Q :
2   match envs_app false (Esnoc Enil i P) Δ with
3   | None => False
4   | Some Δ' => envs_entails Δ' Q
5   end →
6   envs_entails Δ (P -∗ Q).
```

The structure of wand introduction is still the same, if `P ⊢ Q` holds one line 4, `(P -∗ Q)` holds on line 6. However, the IPM needs to add `P` to the context, `Δ`, and handle the case when the chosen name, `i`, has already been used in the context. To add `P` to the context, the IPM uses the function `envs_app`. The first argument tells us to which context the second argument should be appended, `true` for the persistent context, and `false` for the spatial context. The second argument is the environment to append, and the third argument is the context to which we append. We first create a new environment containing just `P` with name `i` using `Esnoc`. Next, we add this environment to the existing context, `Δ`. This results in either `None`, when the name already exists in `Δ`, or `Some Δ'`, when we successfully add the new proposition. This new context is then used as the context for proving `Q`. A similar tactic is made for introducing persistent propositions, but it checks if `P` is also persistent and then adds it to that context.

Many more lemmas such as these are in the IPM. They are the core of many of the tactics we create in section 4.7 and chapter 5.

4.4 Elpi

Our Elpi implementation `eiIntros` consists of three parts, as seen in figure 4.1. The first two parts interpret the DSL used to describe the proofs steps to be taken. Then, the last part applies these proofs steps. 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.

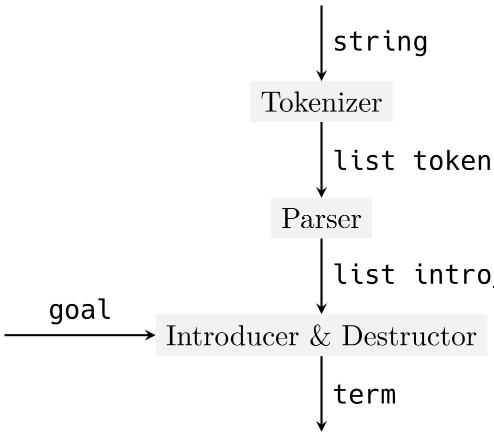


Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

4.5 Tokenizer

The tokenizer takes as input a string, which the tokenizer transforms into a list of tokens. 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.

4.5.1 Data types

The introduction patterns are separated 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".

Elpi

```
kind token type.

type tBar, tBracketL, tBracketR, tParenL, tParenR,
     tAmp, tAnon, tSimpl, tDone, tForall, tAll token.
type tName string -> token.
type tPure option string -> token.
```

We first define a new type called token using the `kind` keyword, where `type` specifies the kind of our new type. Next, 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 followed the type of the 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 kinded types, like `option`.

Elpi
```
kind option type -> type.
type none option A.
type some A -> option A.
```

Creating types of kind `type -> type` is done using the `kind` directive and passing in a more complicated kind as shown above. `list` is implemented similarly with standard notation.

Using the above types, we represent a given string as a list of tokens. Thus, given the string `"[H %H']"` we represent it as the following Elpi list of tokens

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

4.5.2 Predicates

Programs in Elpi consist of predicates. Every predicate has several rules to describe the relation between its arguments.

Elpi
```
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 of input and output is given. This predicate has only one rule, defined on line 2. The variable `S` has type `string` . The variable `O` has type `list token` . By calling predicates after the `:-` symbol, we define the relation between the arguments. The first predicate we call, `rex.split` , splits the second argument by delimiters matching the regular expression in the first argument. The result is stored in the third argument. It has the following type

Elpi
```
pred rex.split i:string, i:string, o:list string.
```

We split the input string using the delimiter `""`, resulting in splitting the string into a list of its characters. Strings in Elpi are native data types and cannot be matched on, and thus we need to split it. The next line, line 4, calls the recursive tokenizer, `tokenizer.rec` [1], on the list of split strings 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 are sometimes better considered 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

[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.

discuss in section 4.5.5. To execute a predicate, we thus find the first rule whose 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, 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[2].

The predicate `tokenize.rec` uses matching and unification to solve most cases.

```
Elpi
 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] [tFresh | TS] :- !,
 5   tokenize.rec SL TS.
 6 tokenize.rec ["/", "/", "=" | SL]
 7              [tSimpl, tDone | TS] :- !,
 8   tokenize.rec SL TS.
 9 tokenize.rec ["/", "/" | SL] [tDone | TS] :- !,
10   tokenize.rec SL TS.
```

The full predicate has rules for all tokens, a few rules are considered here. All rules use the *cut*, `!`, to prevent backtracking, see section 4.5.5, for now they can be ignored. When calling this predicate, the first rule is used when the first argument matches `[]` and if the second argument unifies with `[]`. The difference is that, for a value to match an argument, the value has to be equal or more specific than the argument. In other words, the value can only contain a variable if the argument also contains a variable at that place in the value. Thus, the only valid value for the first argument of the first rule is `[]`. When unifying two values, we allow the variable given to a predicate to be less specific than the argument. If that is the case, the variables are filled in until they match. 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 take a list with the first few elements set. The output is unified with a list starting with the token that corresponds to the string we match on. The tails of the input and output are recursively computed.

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 9 would both match the value `["/", "/", "="]` as first argument. But, we interpret this using the rule on line 6 since it is before the rule on line 9. This results in our list of strings being tokenized as `[tSimpl, tDone]`.

4.5.4 Functional programming in Elpi

While Elpi 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

[2]A fun side effect of outputs being just variables we pass to a predicate is that we can also easily create a reversible function. 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.

to write the entry point of the tokenizer as defined in section 4.5.2 without the need for temporary variables to be passed around.

```Elpi
pred tokenize o:string, o:list token.
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.

The second useful feature is how lambda expressions are first class citizens of the language. The `pred` statement is a wrapper around a constructor definition using `type`, with the addition of denoting arguments as inputs or outputs. When defining a predicate using `type`, all arguments are outputs. The following predicates have the same type.

```Elpi
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 can thus write predicates that accept other predicates as arguments.

```Elpi
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.

```Elpi
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 until 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. This rule is only considered if the first rule fails, thus when `Pred X` 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. Any unification that happened because of the first rule is undone, and the next rule is tried. This will be the rule on line 4 and returns the input as the second output of the predicate.

Now, it might happen that the second rule also fails. If the second output variable does not unify with its input, the rule fails. This would let the whole execution of the predicate fail. Thus, the call on line 4 could fail, which would cause backtracking and an incorrect split of the input, `Pred X` holds but rule 2 is used. Thus, we make use of a cut, `!`, stopping backtracking. When a cut happens, any other possible rules in that execution of a predicate are discarded.

We use `take-while-split` to define the rule for the token `tName`.

Elpi
```
1 tokenize.rec SL [tName S | TS] :-
2   take-while-split SL is-identifier S' SL',
3   { std.length S' } > 0, !,
4   std.string.concat "" S' S,
5   tokenize.rec SL' TS.
6 tokenize.rec XS _ :- !,
7   coq.say "unrecognized tokens" XS, fail.
```

To tokenize a name, we first call `take-while-split` with as predicate `is-identifier`, which checks if a string is a 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. 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.

We also add a rule to give an error message when a token is not recognized on line 6. To ensure this rule is only called on the exact token that is not recognized, we need to not backtrack when a character is recognized, but the rest of the string is not. Thus, we add a cut to every rule when we know a token is correct.

4.6 Parser

The Parser uses the same language features as were used in the tokenizer. Thus, we won't go into detail of its workings. We create a type, `intro_pat`, to store the parse tree.

Elpi
```
kind ident type.
type iNamed string -> ident.
type iAnon term -> ident.

kind intro_pat type.
type iFresh, iSimpl, iDone intro_pat.
type iIdent ident -> intro_pat.
type iList list (list intro_pat) -> intro_pat.
```

Next, we use reductive descent parsing to parse the following grammar into the above data structure.

$$
\begin{array}{lcl}
\langle \textit{intropattern\_list} \rangle & ::= & \epsilon \\
 & | & \langle \textit{intropattern} \rangle\ \langle \textit{intropattern\_list} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{intropattern} \rangle & ::= & \langle \textit{ident} \rangle \\
 & | & \text{‘}\texttt{?}\text{’} \mid \text{‘}\texttt{/=}\text{’} \mid \text{‘}\texttt{//}\text{’} \\
 & | & \text{‘}\texttt{[}\text{’}\ \langle \textit{intropattern\_list} \rangle\ \text{‘}\texttt{]}\text{’} \\
 & | & \text{‘}\texttt{(}\text{’}\ \langle \textit{intropattern\_conj\_list} \rangle\ \text{‘}\texttt{)}\text{’}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{intropattern\_list} \rangle & ::= & \epsilon \\
 & | & \langle \textit{intropattern} \rangle\ \text{‘}\texttt{|}\text{’}\ \langle \textit{intropattern\_list} \rangle \\
 & | & \langle \textit{intropattern} \rangle\ \langle \textit{intropattern\_list} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{intropattern\_conj\_list} \rangle & ::= & \epsilon \\
 & | & \langle \textit{intropattern} \rangle\ \text{‘}\texttt{\&}\text{’}\ \langle \textit{intropattern\_conj\_list} \rangle
\end{array}
$$

In order to make the parser be properly performant, it is important to minimize backtracking. Backtracking is necessary when implementing the second and third case of the ⟨*intropattern_list*⟩ parser. Backtracking might incur significant slowdowns due to reparsing frequently.

4.7 Applier

While creating the tokenizer and parser so far, we have only had to use standard Elpi. We will now be creating the applier. The applier will get a parsed intro pattern and use this to apply steps on the goal. Thus, we now have to communicate with Coq. We make use of Coq-Elpi [Tas18] to get a Coq API in Elpi.

To create a proof in Elpi we take the approach of building one large proof term. We apply this proof term to the goal at the end of the created tactic. We get into more details on this approach in section 4.7.3.

Before we get to building proofs, we first discuss how Coq terms and the Coq context are represented in Elpi in section 4.7.1. Lastly, we show how quotation and anti-quotation are used when building Coq terms in Elpi in section 4.7.2. Using the concepts in these sections, we explain creating proofs in Elpi in section 4.7.3. We discuss the structure of `eiIntros` in section 4.7.4. Lastly, in section 4.7.5 we show how a tactic is called and how a created proof is applied.

4.7.1 Coq-Elpi HOAS

Coq-Elpi makes use of Higher-order abstract syntax (HOAS) [PE88] in order to represent Coq terms in Elpi. Thus, it makes use of the binders in Elpi to represent binders in Coq terms. In this section we will discuss the structure of this HOAS and show how to call the Coq type checker in Elpi.

Take the following Coq term: `0+1`, which when expanding any notation becomes `Nat.add 0 (S 0)`. In Elpi this term is represented as follows.

```elpi
app [global (const «Nat.add»),
    global (indc «O»),
    app [global (indc «S»), global (indc «O»)]]
```

References to Coq object cannot be directly written as `«Nat.add»`. We will discuss how to create these objects in section 4.7.2.

The above Elpi term consists of several constructors. The first constructor is `app`, it is application of Coq terms. It gets a list, the tail of the list are the arguments and the head is what we are applying them to. Next, we have the `global` constructor. It takes a global reference of a Coq object and turns it into a term. Lastly, we have `const` and `indc`, these create a global reference of a constant or inductive constructor respectively.

Coq function terms work again similarly. Take the Coq term `fun (n: nat), n + 1`. This is represented in Elpi as follows.

```elpi
fun `n` (global (indt «nat»))
          (n \ app [global (indt «sum»),
                     n, app [global (indc «S»),
                             global (indc «O»)]])
```

The `fun` constructor takes three arguments. The name of the binder, here `n`. A term containing the type of the binder, `(global (indt «nat»))`. And, a function that produces a term, indicated by the lambda expression with as binder `n`. This is where the HOAS is applied. We use the Elpi lambda expression to encode the argument in the body of the function. Thus, `fun` has the following type definition.

```elpi
type fun name -> term -> (term -> term) -> term.
```

The type `name` is a special type of string. Names in Elpi are special strings which are convertible to any other string. Thus, any name equals any other name. Other Coq terms like `forall`, `let` and `fix` work in the same way.

Given that functions generating bodies of terms are integral to the Coq-Elpi data structures, we need the ability to move under a binder. To solve this, Elpi provides the `pi x\` quantifier. It allows us to introduce a fresh constant `c` any time the expression is evaluated. Take the following example, where we assign the above Coq function to the variable `FUN`.

```elpi
1 FUN = fun _ _ F,
2 pi x\ F x = app [A, B x, C]
```

On line 1 we store the function inside `FUN` in the variable `F`. Remember that the left and right-hand side of the equals sign are unified. Thus, we unify `FUN` with `fun _ _ F` and assign the function inside the `fun` constructor to `F`. On the next line, we create a fresh constant `x`, we now unify `F x` with `app [A, B x, C]`. The first and third element in the list of `app` are assigned to `A` and `C`. The second element of `app` is the binder of the function. Since `x` only exists in the scope of `pi x\`, we cannot just assign it to `B`. It might be used outside the scope of the `pi` quantifier. Thus, we make it a function. We unify `B x` with `x`, and `B` becomes the identity function.

We can call the Coq type checker from inside Elpi on any term. For the type checker to know the type of any binders we are under, it checks if a type is declared, `decl x N T`. Thus, we look for any `decl` rules which have as term `x` and store the name and type of `x` in `N` and `T`. However, now we need to add a rule when entering a binder to store the name and type of that binder. In the below code, `NAT` has the value `(global (indt «nat»))`.

```Elpi
pi x\ decl x `n` NAT
        => coq.typecheck (F x) Type ok.
```

We make use of `=>` connective. The rule in front of `=>` is added on top of the known rules while executing the expressions behind `=>`. Thus, in the scope of `coq.typecheck`, we know that `x` has type `nat`. After type checking, `Type` has value `nat`.

4.7.2 Quotation and anti-quotation

To create terms, Coq-Elpi implements quotation and anti-quotation. This allows for writing Coq terms in Elpi. The Coq terms are parsed by the Coq parser in the context where the Elpi code is loaded in.

```Elpi
FUN = {{ fun (n: nat), n + 1 }}
```

Now `FUN` has the value.

```Elpi
fun `n` (global (indt «nat»))
          (n \ app [global (indt «sum»),
                    n, app [global (indc «S»),
                            global (indc «O»)]])
```

Coq-Elpi also allows for putting Elpi variables back into a Coq term. This is called anti-quotation.

```Elpi
FUN = {{ fun (n: nat), n + lp:C }}
```

We extract the right-hand side of the plus operator in `FUN` into the variable `C` [3]. It thus has the same effect as what we did in the previous section to extract values out of a term. We can of course also use anti-quotation to insert previously calculated values into a term we are constructing.

These two ways of using anti-quotation will see much use when we create proofs in the next section, section 4.7.3. Where we create a proof term:

```Elpi
Proof = {{ tac_wand_intro _ lp:T _ _ _ _ _ }}
```

After unifying `Proof` with the goal, we want to extract any newly created proof variables.

[3]We cannot do the same for the left-hand side of the addition. It contains a binder and thus can only be examined using the method seen in the previous section, section 4.7.1

```elpi
Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:NewProof }},
```

The new proof variable is extracted in the variable `NewProof`.

4.7.3 Proof steps in Elpi

Now that we have a solid foundation on how to work with Coq terms in Elpi we can start creating proof terms. Proof steps in Elpi are built by creating one big term which has the type of the goal. Any leftover holes in this term are new goals in Coq. To facilitate this process, we create a new type called `hole`.

```elpi
kind hole type.
type hole term -> term -> hole.
```

A `hole` contains two arguments. The goal, also called the type, is the first argument. The second argument is the proof variable, the variable to which we assign the proof term. Predicates that take and return holes are called *proof generators*. Take the following proof generator, it applies the iris ex falso rule to the current hole.

```elpi
1 pred do-iExFalso i:hole, o:hole.
2 do-iExFalso (hole Type Proof)
3             (hole FalseType FalseProof) :-
4   coq.elaborate-skeleton
5     {{ tac_ex_falso _ _ _ }} Type Proof ok,
6   Proof = {{ tac_ex_falso _ _ lp:FalseProof }},
7   coq.typecheck FalseProof FalseType ok.
```

The proof makes use of a variant of the ex falso rule, which is aware of contexts.

```coq
Lemma tac_ex_falso Δ Q :
  envs_entails Δ False →
  envs_entails Δ Q.
```

Thus, `tac_ex_falso` takes three arguments, the context, what we want to prove and a proof for `envs_entails Δ False`.

The Elpi code on lines 4-7 are the normal steps to apply a lemma. We make use of the Coq-Elpi API call, `coq.elaborate-skeleton` to apply this lemma to the hole. It elaborates the first argument against the type. The fully elaborated term is stored in the variable `Proof`. In this instance, `Proof` is the lemma with the Iris context filled in and a variable where the proof for `envs_entails Δ False` goes. Furthermore, the type information of any holes is added to the Elpi context. We extract this new proof variable on line 4. The proof variable is type checked to get the associated type of the proof variable using `coq.typecheck`. Together, these two variables for the new hole.

This is the structure of the most basic proof generators we use in our tactics. The concept of a hole allows for very composable proof generators. We will now discuss some more difficult proof generators. They will deal more directly with the iris context or introduce variables in the Coq context, and thus we need to create the rest of the proof under a binder.

Iris context counter

In section 4.2, we saw how anonymous assumptions are created in the iris context. We keep a counter in the context to ensure we can create a fresh anonymous identifier. This counter is convertible, allowing us to change it without doing changing the proof. In Elpi it is easier to keep track of this counter outside the context. We thus introduce a new type for an Iris hole.

Elpi
```
1  kind ihole type.
2  type ihole term -> hole -> ihole. % ihole counter hole
```

When we start the proof step, we take the current counter and store it. At then end of the proof, we set it again before returning it to Coq.

In a proof generator, we now simply use the counter in the `ihole` to generate a new identifier for an assumption. In any new `ihole`, we increase the counter by one.

Elpi
```
1  pred do-iIntro-anon i:ihole, o:ihole.
2  do-iIntro-anon (ihole N (hole Type Proof))
3                 (ihole N' (hole IType IProof)) :-
4    coq.reduction.vm.norm {{ Pos.succ lp:N }} _ N',
5    coq.elaborate-skeleton
6      {{ tac_wand_intro _ (IAnon lp:N) _ _ _ _ _ }}
7      Type Proof ok, !,
8    Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:IProof }},
9    coq.typecheck IProof IType' ok,
10   pm-reduce IType' IType.
```

The above proof generator introduces a wand into an anonymous hypothesis. On line 4 we increase the counter. Since the counter is a Coq term, we create a Coq term that increases the counter and execute it using `coq.reduction.vm.norm`. Next, using the old context counter, we create the identifier `(IAnon lp:N)`. We apply the lemma to the type of the hole and extract the new proof variable and type. Lastly, the created new proof types are often not fully normalized. The lemma we have applying has the following type.

Coq
```
1  Lemma tac_wand_intro Δ i P Q R :
2    FromWand R P Q →
3    match envs_app false (Esnoc Enil i P) Δ with
4    | None => False
5    | Some Δ' => envs_entails Δ' Q
6    end →
7    envs_entails Δ R.
```

The proof variable thus gets the type on lines 3-6. We normalize this using `pm-reduce`[4] to just `envs_entails Δ' Q` as long as the name was not already used.

[4] `pm-reduce` is also fully written in Elpi and is made extendable after definition of the tactics. To accomplish this Coq-Elpi databases are used with commands to add extra reduction rules to the database.

Continuation Passing Style

When introducing a universal quantifier in Coq, the proof term is a function. The new hole in the proof is now in the function. Thus, we are forced to continue the proof under the binder of the function in the proof term. To compose proof generators, we make use of continuation passing style (CPS) for these proof generators.

```elpi
pred do-intro i:string, i:hole, i:(hole -> prop).
do-intro ID (hole Type Proof) C :-
  coq.id->name ID N,
  coq.elaborate-skeleton (fun N _ _) Type Proof ok,
  Proof = (fun _ T IntroFProof),
  pi x\ decl x N T =>
    coq.typecheck (IntroFProof x) (FType x) ok,
    C (hole (FType x) (IntroFProof x)).
```

This proof generator introduces a Coq universal quantifier into the Coq context with the name `ID`. It first transforms the name, an Elpi string, into a Coq string term called `N`. Next we elaborate the proof term `fun (x: _), _` on `Type`. We extract the type of the binder in `T` and the function containing the new proof variable in `IntroFProof`. To move under the binder of the function we use the `pi` connective and then declare the name and type of `x` to the Coq context. Now can get the type of the proof variable. This might also depend on `x`, and thus it is also a function. Lastly, we call the continuation function with the new type and proof variable.

The unfortunate part of using CPS is that any predicates that use `do-intro` often also need to use CPS. Thus, we only use it when absolutely necessary.

4.7.4 Applying intro patterns

Now that we have defined multiple proof generators, we execute them depending on our intro patterns.

```elpi
pred do-iIntros i:(list intro_pat),
                i:ihole, i:(ihole -> prop).
do-iIntros [] IH C :- !, C IH.
do-iIntros [iFresh | IPS] IH C :- !,
  do-iIntro-anon IH IH', !,
  do-iIntros IPS IH' C.
do-iIntros [iPure (some X) | IPS] (ihole N H) C :-
  do-iForallIntro H H',
  do-intro X H
    (h\ sigma IntroProof\ sigma IntroType\
        sigma NormType\
        h = hole IntroType IntroProof,
        pm_reduce IntroType NormType, !,
        do-iIntros IPS
                   (ihole N (hole NormType IntroProof))
                   C
    ).
do-iIntros [iList IPS | IPSS] (ihole N H) C :- !,
  do-iIntro-anon (ihole N H) IH, !,
```

Elpi

```
20    do-iDestruct (iAnon N) (iList IPS) IH (ih'\ !,
21      do-iIntros IPSS ih' C
22    ).
```

This is a selection of the rules of the `do-iIntros` proof generator. The generator iterates over the intro patterns in the list. In the base case on line 3 it simply calls the continuation function. The second case, on line 4-6, simply calls a proof generator, in this case introducing an anonymous Iris assumption. Then, it continuous executing the rest of the intro patterns.

The third case, on lines 7-17, has three steps. First, it calls a proof generator that puts an Iris universal quantifier at the front of the goal as a Coq universal quantifier. This does not interact with the fresh counter, and thus we only give it a normal hole. Next we call `do-intro` as defined in section 4.7.3. This takes a continuation function which we define in lines 10-17. The hole this function gets, `h`, is not fully normalized. We thus need to access the type in the hole and reduce it. However, if we would just do `h = hole IntroType IntroProof` to extract the type from the hole, Elpi would give an error. By default, variables are created at the level of the predicate they are defined in. However, a predicate can only contain constants, by `pi x\`, created before they are defined. Thus, we make use of the quantifier `sigma X\` to instead define the variable in the continuation function. This ensures that the binder we are moving under is in scope when defining the variable. Once we have resolved that issue, we call `do-iIntros` on the rest of the intro patterns.

Question: The binders in variables is quite complicated, and I was hoping to skip this. But it is quite a downside of the CPS. I put it in for now. But maybe remove it.

For the fourth case, we will not go into too much detail, but just give an outline of what happens. This case covers the destruction intro patterns. These were parsed into an `iList` containing the destruction pattern. We first introduce the assumption we want to destroy with an anonymous name. Next, we call `do-iDestruct` to do the destruction. This can create multiple holes in the process, and the continuation function we pass it will be executed at the end of all of them. The predicate `do-iDestruct` has the same structure as `do-iIntros`, and we will see it in **??** when we discuss the destruction of inductive predicates.

4.7.5 Starting the tactic

The entry point of a tactic in Elpi is the `solve` predicate.

Elpi

```
solve (goal _ _ Type Proof [str Args]) GS :-
  tokenize Args T, !,
  parse_ipl T IPS, !,
  do-iStartProof (hole Type Proof) IH, !,
  do-iIntros IPS IH (ih\ set-ctx-count-proof ih _), !,
  coq.ltac.collect-goals Proof GL SG,
  all (open pm-reduce-goal) GL GL',
  std.append GL' SG GS.
```

The entry point takes a goal, which contains the type of the goal, the proof variable, and any arguments we gave. We then tokenize and parse the argument such that we have an intro pattern to apply. We use the start proof, proof generator to transform the goal into an `envs_entails` goal and get the context counter. And we are ready to use `do-iIntros` to apply the intro pattern. At the end, set the correct context counter in the proof. We now have a proof term in the `Proof` variable that we want to return to Coq. We make use of several Coq-Elpi predicates to accomplish this. First, collect all holes in the proof term and transform them into objects of the type `goal` in the lists `GL`, `SG`. The two lists are the normal goals and the shelved goals, goals Coq expects to be solved during proving of the normal goals[5]. This step uses type checking to create the type of the goals, and thus they are not normalized, on line 7 we normalize all main goals. Lastly, we combine the two lists again and return then to Coq using the variable `GS`.

[5]Goals in Coq-Elpi can either be sealed or opened. A sealed goal contains all binders for the context of the goal in the goal. A goal is opened by going under all the binders and adding all the types of the binders as rules. The sealing of goals to pass them around is necessary when you can make no assumptions on what happens to the context of a goal, and is thus the model used for the entry point of Coq-Elpi. However, in our proof generators we know when new things are added to the context, and thus we can take a more specialized approach using CPS.

Chapter 5

Elpi implementation of Inductive

We discuss the implementation of the `eiInd` command together with integrations in the `eiIntros` tactic and the `eiInduction` tactic.

The `eiInd` command mirrors the steps taken in chapter 3. These steps are outlined in figure 5.1, with their associated section. It starts by interpreting the Coq inductive statement and producing the pre fixpoint function. Next, we prove monotonicity and construct the fixpoint. Then, we create and prove the fixpoint properties and the constructor lemmas. Lastly, we create and prove the induction lemma.

In sections 5.7 and 5.8 we discuss how the tactics to use an inductive predicate are made. We first discuss the `eiInduction` tactic in section 5.7, which performs induction on the specified inductive predicate. Next, in section 5.8, we outline the extensions to the `eiIntros` tactic concerning inductive predicates.

In section 5.9 we generalize the previously created commands and tactics to support parameters on the inductive. Lastly, in section 5.10 we show how the `eiInd` command can be used to define the total weakest precondition.

5.1 Constructing the pre fixpoint function

The `eiInd` command is called by writing a Coq inductive statement and prepending it with the `eiInd` command. The below inductive statement implements the isMLL inductive predicate from chapter 3 in Coq.

```coq
eiInd
Inductive is_MLL : val → list val → iProp :=
    | empty_is_MLL : is_MLL NONEV []
    | mark_is_MLL v vs l tl :
      l ↦ (v, #true, tl) -∗ is_MLL tl vs -∗
      is_MLL (SOMEV #l) vs
    | cons_is_MLL v vs tl l :
      l ↦ (v, #false, tl) -∗ is_MLL tl vs -∗
      is_MLL (SOMEV #l) (v :: vs).
```

The inductive statement is received in Elpi as the following value of type `indt-decl`, unimportant fields of constructors are filled in with an `_`.

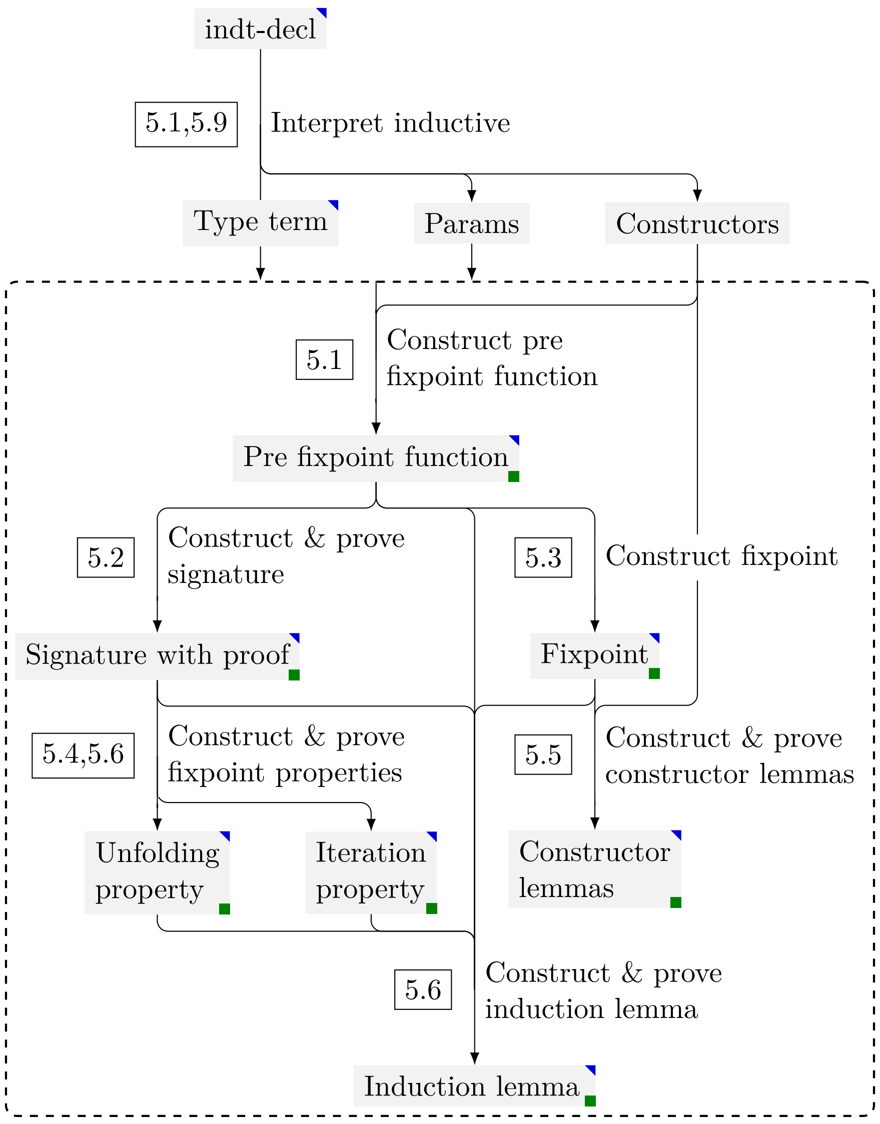


Figure 5.1: The structure of the `eiInd` command. Arrows are steps in the command, and boxes are the objects that are being created. If a box has a green box, it is defined in Coq. If a box has a blue triangle, it is stored in the Elpi database. All arrows reference the section in which they are explained.

```elpi
inductive `is_MLL` _
  (arity {{ val -> list val -> iProp }})
  (f \ [constructor `empty_is_MLL`
          (arity {{ lp:f NONEV [] }}),
        constructor `mark_is_MLL`
          (parameter `v` _ Type (v\
            (parameter `vs` _ Type (vs\
              (parameter `l` _ Type (l\
                (parameter `tl` _ Type (tl\
                  {{ lp:l ↦ (lp:v, #true, lp:tl) -∗
                     lp:f lp:tl lp:vs -∗
```

Elpi

```
12                        lp:f (SOMEV #(lp:l)) lp:vs }}
13               )))))))),
14        constructor `cons_is_MLL` …])
```

The inductive consists of its name, `` `is_MLL` ``, its type, and a function containing the constructors with their names. Both the constructors and the type contain possible Coq binders. The constructor `mark_is_MLL` has the Coq binders `v`, `vs`, `l` and `tl`, these are represented with the parameter constructor on lines 6-9. The parameter constructor takes the name, type, and a function giving the rest of the term.

By recursing through the inductive, we generate the following pre fixpoint function.

Coq

```
λ (F : val → list val → iProp) (v : val) (vs : list val),
  (⌜v = InjLV #()⌝ ∗ ⌜vs = []⌝
    ∨ (∃ (v' : val) (vs' : list val) l (tl : val),
          l ↦ (v', #true, tl) ∗ F tl vs' ∗
          ⌜v = InjRV #l⌝ ∗ ⌜vs = vs⌝)
    ∨ ∃ (v' : val) (vs' : list val) (tl : val) l,
          l ↦ (v', #false, tl) ∗ F tl vs' ∗
          ⌜v = InjRV #l⌝ ∗ ⌜vs = v' :: vs⌝)
```

This function is produced by applying the following transformations: We replace the top-level wands with separating conjunctions. We transform the binders of the constructors into Iris existential quantifiers. We replace the last recursive call in a constructor with equalities for each of its arguments. Concatenate the constructors with disjunctions. And, lastly, wrap the combined constructors into a function taking the recursive call `F`, and the arguments, `v` and `vs`.

The pre fixpoint function is defined as `is_MLL_pre`.

5.2 Creating and proving proper signatures

In this section we describe how a proper is created and proven for the previously defined function. This section implements the theory as defined in section 3.3.

Proper definition in Coq Proper elements of relations are defined using type classes and named `IProper`. Respectful relations, `R ==> R`, pointwise relations, `.> R` and persistent relations, `□> R` are defined with accompanying notations. Any signatures are defined as global instances of `IProper`.

To easily find the `IProper` instance for a given connective and relation, an additional type class is added.

Coq

```
Class IProperTop {A} {B} (R : iRelation A) (m : B)
                 (f: iRelation A → iRelation B) :=
  iProperTop : IProper (f R) m.
```

Given a relation `R` and connective `m`, we find a function `f` that transforms the relation into the proper relation for that connective. For example, given the `IProper` instance for separating conjunctions, we get the `IProperTop` instance.

```coq
Global Instance sep_IProper :
  IProper _ (bi_wand ==> bi_wand ==> bi_wand)
           bi_sep.

Global Instance sep_IProperTop :
  IProperTop bi_wand (bi_sep)
             (fun F => bi_wand ==> bi_wand ==> F).
```

Creating a signature Using these Coq definitions, we transform the type into an `IProper`. A Proper relation for a pre fixpoint function will always have the shape `(□> R ==> R)`. The relation `R` is constructed by wrapping a wand with as many pointwise relations as there are arguments in the inductive predicate. The full `IProper` term is constructed by giving this relation to `IProper` together with the pre fixpoint function. Any parameters are quantified over and given to the fixpoint function.

```coq
IProper (□> .> .> bi_wand ==> .> .> bi_wand)
        (is_MLL_pre)
```

Proving a signature To prove a signature, we implement the recursive algorithm as defined in section 3.3. We use the proof generators from section 4.7 to create a proof term for the signature. We will highlight the interesting step of applying an `IProper` instance.

A relevant `IProperTop` instance can be found by giving the top-level relation and top-level function of the current goal. However, some `IProperTop` instances are defined on partially applied functions. Take the existential quantifier. It has the type `∀ {A : Type}, (A → iProp) → iProp`. The `IProper` and `IProperTop` instances are defined with an arbitrary `A` filled in.

```coq
Global Instance exists_IProper {A} :
  IProper (.> bi_wand ==> bi_wand)
          (@bi_exist A).
Global Instance exists_IProperTop {A} :
  IProperTop (bi_wand) (@bi_exist A)
             (fun F => .> bi_wand ==> F).
```

Thus, when searching for the instance, we also have to fill in the `A`. The number of arguments we have to fill in when searching for an `IProperTop` instance differs per connective. We take the following approach.

```elpi
pred do-steps.do i:ihole, i:term, i:term, i:term.
do-steps.do IH R (app [F | FS]) _ :-
  std.exists { std.iota {std.length FS} }
             (n\ std.take n FS FS'),
  do-iApplyProper IH R (app [F | FS']) HS, !,
  std.map HS (x\r\ do-steps x) _.
```

The `do-steps.do` predicate contains rules for all options in the application step in the proof search algorithm. The rule highlighted here applies to an `IProper` instance. It gets the Iris hole `IH`, the top-level relation `R`, and the top-level function `app [F | FS]`. The last argument is not relevant to this rule.

Next, on line 3, we first create a list of integers from one until the length of the arguments of the top-level function with `std.iota`. Next, the `std.exists` predicate tries to execute its second argument for every element of this list until one succeeds. The second argument then just takes the first `n` arguments of the top-level function and stores it in the variable `FS'`. This obviously always succeeds, however the predicate on line 4 does not. `do-iApplyProper` takes the Iris hole, relation and now partially applied top-level function and tries to apply the appropriate `IProper` instance. However, when this predicate fails because it can't find an `IProper` instance, we backtrack into the previous predicate. This is `std.exists`, and we try the next rule there, and we take the next element of the list and try again. This internal backtracking ensures we try every partial application of the top-level function until we find an `IProperTop` instance that works. If there are none, we can try another rule of `do-steps.do`.

Lastly, on line 6, we continue the algorithm. We would rather not backtrack into the `std.exists` when something goes wrong in the rest of the algorithm, and thus we include a cut after successfully applying the `IProper` instance.

The predicate `do-iApplyProper` follows the same pattern as the other Iris proof generators we defined in section 4.7. It mirrors a simplified version of the IPM `iApply` tactic while also finding the appropriate `IProper` instance to apply.

Adding monotonicity proofs to Coq With the signature and the proof term for monotonicity of the pre fixpoint function we define a new lemma in Coq called `is_MLL_pre_mono`. Thus allowing any further proof in the command and outside it to make use of the monotonicity of `is_MLL_pre`.

5.3 Constructing the fixpoint and storing the definitions

The command `eiInd` generates the fixpoint as defined in section 3.3.

Coq
```
1 λ (v : val) (l : list val),
2   (∀ F : val → list val → iProp,
3     □ (∀ (v' : val) (l' : list val),
4           is_MLL_pre F v' l' -∗ F v' l')
5     -∗ F v l)
```

The fixpoint is generated by recursing through the type term. For every dependent product in the type, we generate a lambda function, as on line 1. Next, we add the universal quantifier on an `F` on line 2. We again recurse through the type term to generate the left-hand side of the wand on lines 3 and 4. Lastly, we apply the binders of the lambda functions to `F` on line 5.

This results in creating the following fixpoint statement, defined as `is_MLL`. Note that we do not have a separate definition of the fixpoint not yet applied to a concrete pre fixpoint function, as was the case in section 3.2.

Coq-Elpi database Coq-Elpi provides a way to store data between executions of tactics and commands, this is called the database. We define predicates whose rules are stored in the database.

```
Elpi Db induction.db lp:{{                                    Coq
  pred inductive-pre o:gref, o:gref.
  pred inductive-mono o:gref, o:gref.
  pred inductive-fix o:gref, o:gref.
  pred inductive-unfold o:gref, o:gref, o:gref,
                        o:gref, o:int.
  pred inductive-iter o:gref, o:gref.
  pred inductive-ind o:gref, o:gref.
  pred inductive-type o:gref, o:indt-decl.
}}.
```

The rules are always defined such that the fixpoint definition is the first argument and the objects we want to associate to it are next. We store the references to any objects we create after any of the previous or following steps. We also include some extra information in some rules. `inductive-unfold` includes the number of constructors the fixpoint has, and `inductive-type` contains only the Coq inductive. When retrieving information about an object, we can simply check in the database by calling the appropriate predicate. Thus allowing further invocations of tactics to retrieve the necessary definitions concerning a fixpoint.

5.4 Unfolding property

In this section we prove the unfolding property of the fixpoint from theorem 3.3. This proof is generated for every new inductive predicate to account for the different possible arities of inductive predicates. The proof of the unfolding property is split into three parts, separate proofs of the two directions and finally the combination of the directions into the unfolding property. We explain how the proof of one direction is created in the section. Any other proofs generated in this or other sections follow the same strategy and will not be explained in as much detail.

Generating the proof goal is done by recursing over the type term, this results in the following statements to prove. Where the other unfolding lemmas either flip the entailment or replace it with a double entailment.

```
∀ (v : val) (l : list val),                                   Coq
    is_MLL_pre (is_MLL) v l
  ⊢ is_MLL v l
```

The proof term is generated by chaining proof generators such that no holes exist in the proof term. We thus use our tactics defined in chapter 4, tactics not mentioned in chapter 4 follow the same strategy as ones defined in that chapter.

```
pred mk-unfold.r->l i:int, i:int,                             Elpi
                    i:term, i:term, i:hole.
mk-unfold.r->l Ps N Proper Mono (hole Type Proof) :-
  do-intros-forall (hole Type Proof)
                   (mk-unfold.r->l.1 Ps N Proper Mono).
```

This predicate performs the first step in the proof generation before calling the next step. It takes the number of parameters, `Ps`, which we discuss section 5.9, the number of arguments the fixpoint takes, `N`, the `IProper` signature, `Proper`, a reference to the monotonicity proof `Mono` and the hole for the proof. It then introduces any universal quantifiers at the start of the proof. The rest of the proof has to happen under the binder of these quantifiers, and thus we use CPS to continue the proof in the predicate `mk-unfold.r->l.1`.

Elpi
```
1  pred mk-unfold.r->l.1 i:int, i:int,
2                        i:term, i:term, i:hole.
3  mk-unfold.r->l.1 Ps N Proper Mono H :-
4    do-iStartProof H IH, !,
5    do-iIntros [iIdent (iNamed "HF"), iPure none,
6                iIntuitionistic (iIdent (iNamed "HI")),
7                iHyp "HI"] IH
8               (mk-unfold.r->l.2 Ps N Proper Mono).
```

This proof generator performs all steps possible using the `do-iIntros` proof generator. It takes the same arguments as `mk-unfold.r->l`. On line 3, it initializes the Iris context and thus creates an Iris hole, `IH`. Next, we apply several proof steps using de `do-iIntros` proof generator. This again results in a continuation into a new proof generator. We are now in the following proof state.

Coq
```
1  "HI" : ∀ (v : val) (l : list val),
2          is_MLL_pre F v l -∗ F v l
3  ------------------------------------------------□
4  "HF" : is_MLL_pre (is_MLL) l' v'
5  ------------------------------------------------∗
6  is_MLL_pre F l' v'
```

We need to apply monotonicity of `is_MLL_pre` on the goal and `"HF"`.

Elpi
```
1  pred mk-unfold.r->l.2 i:int, i:int,
2                           i:term, i:term, i:ihole.
3  mk-unfold.r->l.2 Ps N Proper Mono IH :-
4    ((copy {{ @IProper }} {{ @iProper }} :- !) =>
5      copy Proper IProper'),
6    type-to-fun IProper' IProper,
7    std.map {std.iota Ps} (x\r\ r = {{ _ }}) Holes, !,
8    do-iApplyLem (app [IProper | Holes]) IH [
9      (h\ sigma PType\ sigma PProof\
10         sigma List\ sigma Holes2\ !,
11       h = hole PType PProof,
12       std.iota Ps List,
13       std.map List (x\r\ r = {{ _ }}) Holes2,
14       coq.elaborate-skeleton (app [Mono | Holes2])
15                              PType PProof ok,
16     )] [IH1, IH2],
17   do-iApplyHyp "HF" IH2 [], !,
18   std.map {std.iota N} (x\r\ r = iPure none) Pures, !,
19   do-iIntros
20     {std.append [iModalIntro | Pures]
21                 [iIdent (iNamed "H"), iHyp "H",
22                     iModalIntro, iHyp "HI"]}
23     IH1 (ih\ true).
```

We won't discuss this last proof generator in full detail but explain what is generally accomplished by the different lines of code. The proof generator again takes the same arguments as the previous two steps. Lines 4-7 transform the signature of the pre fixpoint function into a statement we can apply to the goal. The complexity comes from dealing with parameters, which we discuss in section 5.9.

Question: Long version: On lines 4 and 5 to signature is transformed into a term that can be applied to the current hole. The type class is replaced by the type class constructor and any universal quantifiers are transformed into lambda expressions with the same type binder. On line 7 a list of holes is generated to append to the monotonicity statement we apply. These holes fill in the parameter arguments in the statement. We are thus applying the following statement.

```
iProper (□> .> .> bi_wand ==> .> .> bi_wand)
        (is_MLL_pre)
```

Line 8 applies this statement, resulting in 3 holes we need to solve. The first hole is a non-Iris hole that resulted from transforming the goal into an Iris entailment. This hole has to be solved in CPS. This is done in lines 9-15. Lines 9-15 apply the proof of monotonicity to solve the `IProper` condition[1].

Line 17 ensures that the monotonicity is applied on `"HF"`. Next, lines 18-23 solve the following goal using another instance of the `do-iIntros` proof generator.

```
"HI" : ∀ (v : val) (vs : list val),
        is_MLL_pre f v vs -∗ f v vs
------------------------------------------------□
(□> .> .> bi_wand) is_MLL f
```

Thus proving the right to left unfolding property. This proof together with the other two proofs of this section are defined as `is_MLL_unfold_1`, `is_MLL_unfold_2` and `is_MLL_unfold`.

5.5 Constructor lemmas

The constructors of the inductive predicate are transformed into lemmas that can be applied during a proof utilizing inductive predicates. By again recursing on the type term, a lemma is generated per constructor.

```
∀ (v : val) (vs : list val),
  ⌜v = InjLV #()⌝ ∗ ⌜vs = []⌝ -∗ is_MLL v vs

∀ (v : val) (vs : list val),
  (∃ (v : val) (vs : list val) l (tl : val),
        l ↦ (v, #true, tl) ∗ is_MLL tl vs ∗
        ⌜v = InjRV #l⌝ ∗ ⌜vs = vs⌝)
     -∗ is_MLL v vs

∀ (v : val) (vs : list val),
  (∃ (v : val) (vs : list val) (tl : val) l,
```

[1]This section of code can't make use of spilling, thus creating many more lines and temporary variables. We can't use spilling since the hidden temporary variables created by spilling are defined at the top level of the predicate. Thus, they can't hold any binders that we might be under. So to solve this, we define any temporary variables ourselves using the `sigma X\` connective.

```coq
        l ↦ (v, #false, tl) ∗ is_MLL tl vs ∗
        ⌜v = InjRV #l⌝ ∗ ⌜vs = v :: vs⌝)
      -∗ is_MLL v vs
```

Both constructor lemmas are a wand of the associated constructor to the fixpoint. They are defined with the name of their respective constructor, `empty_is_MLL`, `mark_is_MLL` and `cons_is_MLL`.

Question: Mention that equalities could be resolved, but that it is not done?

5.6 Iteration and induction lemmas

The iteration and induction lemmas follow the same strategy as the previous sections. The iteration property that we prove is:

```coq
∀ Φ : val → list val → iProp,
     □ (∀ (v : val) (vs : list val),
          is_MLL_pre Φ v vs -∗ Φ v vs)
  -∗ ∀ (v : val) (vs : list val), is_MLL v vs -∗ Φ v vs
```

The induction lemma that we prove is:

```coq
∀ Φ : val → list val → iProp,
     □ (∀ (v : val) (vs : list val),
             is_MLL_pre
                (λ (v' : val) (vs' : list val),
                  Φ v' vs' ∧ is_MLL v' vs')
                v vs
          -∗ Φ v vs)
  -∗ ∀ (v : val) (vs : list val), is_MLL v vs -∗ Φ v vs
```

These both mirror the iteration property and induction lemma from section 3.3. They are defined as `is_MLL_iter` and `is_MLL_ind`.

Question: Maybe explain in more detail? But they are basically the same as in 3.3. If I do switch to isMLL for this chapter, would it then be fine?

5.7 `eiInduction` tactic

The `eiInduction` tactic will apply the induction lemma and perform follow-up proof steps such that we get base and inductive cases to prove. We first show an example of applying the induction lemma and then show how the `eiInduction` tactic implements the same and more.

Example 5.1

We show how to apply the induction lemma in a Coq lemma. We take as an example lemma 2.2.

Coq

```
Lemma MLL_delete_spec (vs : list val)
                      (i : nat) (hd : val) :
  [[{ is_MLL hd vs }]]
    MLL_delete hd #i
  [[{ RET #(); is_MLL hd (delete i vs) }]].
Proof.
```

The proof of this Hoare triple was by induction. Thus, we first prepare for the induction step, resulting in the following proof state.

Coq

```
vs: list val
hd: val
---------------------------------------
"His" : is_MLL hd vs
---------------------------------------∗
∀ (P : val → iPropI Σ) (i : nat),
  (is_MLL hd (delete i vs) -∗ P #()) -∗
  WP MLL_delete hd #i [{ v, P v }]
```

Here `"His"` is the assumption we apply induction on. As `Φ` we choose the function:

Coq

```
λ (hd: val) (vs: list val),
  ∀ (P : val → iPropI Σ) (i : nat),
    (is_MLL hd (delete i vs) -∗ P #()) -∗
    WP MLL_delete hd #i [{ v, P v }]
```

Allowing us to apply the induction lemma.

The `eiInduction` tactic is called as `eiInduction "His" as "[...]"`. It takes the name of an assumption and an optional introduction pattern.

Elpi

```
1  pred do-iInduction i:ident, i:intro_pat, i:ihole,
2                     o:(ihole -> prop).
3  do-iInduction ID IP (ihole _ (hole Type _) as IH) C :-
4    find-hyp ID Type (app [global GREF | Args]),
5    inductive-ind GREF INDLem, !,
6    inductive-type GREF T, !,
7    do-iInduction.inner ID IP T (app [global INDLem])
8                          Args IH C.
```

This is the proof generator for induction proofs. It takes the identifier of the induction assumption and the introduction pattern. If there is no introduction pattern given, `IP` is `iAll`. Lastly, the proof generator takes the iris hole to apply induction in.

On line 3 we get the fixpoint object and its arguments. Next, on line 4 and 5, we search in the database for the induction lemma and Coq inductive object associated with this fixpoint. This information is all given to the inner function.

The inner predicate is used to recursively descent through the inductive data structure and apply any parameters to the induction lemma. Next, the conclusion of the Iris entailment is taken out of the goal. It is transformed into a function over the remaining arguments of the induction assumption. And we apply the induction lemma with the applied parameters and the function.

The resulting goal first gets general introduction steps and then either applies the introduction pattern given or just destructs into the base and induction cases.

5.8 `eiIntros` integrations

The `eiIntros` tactic gets additional cases for destructing induction predicates. Whenever a disjunction elimination introduction pattern is used, the tactic first checks if the connective to destruct is an inductive predicate. If this is the case, it first applies the unfolding lemma before doing the disjunction elimination.

We also added a new introduction pattern `"**"`. This introduction pattern checks if the current top-level connective is an inductive predicate. If this is the case, it uses unfolding and disjunction elimination to eliminate the predicate.

5.9 Parameters

The `eiInd` command can handle Coq binders for the whole Coq inductive statement, also called *parameters* in this chapter. Consider this modified inductive predicate for MLL.

```coq
EI.ind
Inductive is_R_MLL {A} (R : val -> A -> iProp) :
                   val → list A → iProp :=
  | empty_is_R_MLL : is_R_MLL R NONEV []
  | mark_is_R_MLL v xs l tl :
      l ↦ (v, #true, tl) -∗ is_R_MLL R tl xs -∗
      is_R_MLL R (SOMEV #l) xs
  | cons_is_R_MLL v x xs tl l :
      l ↦ (v, #false, tl) -∗ R v x -∗
      is_R_MLL R tl xs -∗
      is_R_MLL R (SOMEV #l) (x :: xs).
```

Instead of equating the values in the MLL to a list of values, we instead use an explicit relation to relate the values in the MLL to the list. To accomplish this, we add two parameters, `{A}` and `(R : val -> A -> iProp)`. These values of `is_R_MLL` do not change during the inductive, and thus they are handled differently.

When receiving the inductive value in the command, the `inductive` constructor is wrapped in binders for each parameter. Thus, when interpreting the inductive statement, we keep track of all binders of parameters and add the type of the binder to the Elpi type context.

Now, whenever we make a term which we define in Coq, we have to put add the parameters. Consider the pre fixpoint function of `is_R_MLL` before adding the fixpoints.

```elpi
F' = {{
  λ (F : val → list lp:a → iProp)
    (v : val) (xs : list lp:a),
    …
  ∨ ∃ v' x xs' tl l,
      l ↦ (v', #false, tl) ∗ lp:r v' x ∗ F tl xs' ∗
      ⌜v = InjRV #l⌝ ∗ ⌜xs = x :: xs'⌝
}}
```

We only consider the interesting constructor. The term still contains Elpi binders, which are not bound in the term. We solve this problem using the following Elpi predicate.

```elpi
pred replace-params-bo i:list param, i:term, o:term.
replace-params-bo [] T T.
replace-params-bo [(par ID _ Type C) | Params]
                  Term (fun N Type FTerm) :-
  replace-params-bo Params Term Term',
  (pi x\ (copy C x :- !) => copy Term' (FTerm x)),
  coq.id->name ID N.
```

It takes a list of parameters containing the name, type, and binder of the constant, and the term we want to bind parameters in. If there are still parameters left to bind, we first recursively bind the rest of the parameters. Next, we copy the term with the other parameters bound into the function `FTerm`, however when we encounter the parameter during copying we instead use the binder of `FTerm`. Lastly, we fix the type of the name of the parameter. The returned term is a Coq function based on `FTerm` and the name and type of the parameter. We have a similar predicate, `replace-params-ty` to bind parameters in dependent products, instead of lambda functions.

We make use of the above predicate to transform `F'` into the pre fixpoint function.

```coq
λ (A : Type) (R : val → A → iProp)
  (F : val → list A → iProp)
  (H : val) (H0 : list A),
    (⌜H = InjLV #()⌝ ∗ ⌜H0 = []⌝)
  ∨ (∃ (v : val) (xs : list A) l (tl : val),
      l ↦ (v, #true, tl) ∗ F tl xs ∗
      ⌜H = InjRV #l⌝ ∗ ⌜H0 = xs⌝)
  ∨ ∃ (v : val) (x : A) (xs : list A) (tl : val) l,
     l ↦ (v, #false, tl) ∗ R v x ∗ F tl xs ∗
     ⌜H = InjRV #l⌝ ∗ ⌜H0 = x :: xs⌝
```

We use `replace-params-bo` and `replace-params-ty` to bind parameters in any terms created during `eiInd`. During proof generation, we also need to keep parameters in mind. When applying lemmas generated during creation of the inductive predicate, we have to add holes for any parameters of the inductive predicate. An example of this procedure can be found on line 7 of `mk-unfold.r->l.2` in section 5.4.

5.10 Application to other inductive predicates

In this section we show how the system we developed for defining inductive predicates in Iris is applicable to a more real-world example than MLLs.

In the IPM, the total weakest precondition proof rules are not axioms. They are derived from the definition of the total weakest precondition, and, the total weakest precondition is defined in terms of the base Iris logic. This definition is a fixpoint following the procedure in section 3.2.

We can fully define the total weakest precondition using the following `eiInd` command.

Coq
```
1  eiInd
2  Inductive twp (s : stuckness) :
3      coPset -> expr Λ ->
4      (val Λ -> iProp Σ) -> iProp Σ :=
5    | twp_some E v e1 Φ :
6      (|={E}=> Φ v) -∗
7      ⌜to_val e1 = Some v⌝ -∗
8      twp s E e1 Φ
9    | twp_none E e1 Φ :
10     (∀ σ1 ns κs nt,
11        state_interp σ1 ns κs nt ={E,∅}=∗
12        ⌜if s is NotStuck then reducible_no_obs e1 σ1
13                          else True⌝ ∗
14        ∀ κ e2 σ2 efs,
15          ⌜prim_step e1 σ1 κ e2 σ2 efs⌝ ={∅,E}=∗
16          ⌜κ = []⌝ ∗
17          state_interp σ2 (S ns) κs (length efs + nt) ∗
18          twp s E e2 Φ ∗
19          [∗ list] ef ∈ efs, twp s ⊤ ef fork_post)
20     -∗ ⌜to_val e1 = None⌝
21     -∗ twp s E e1 Φ.
```

It contains several Iris connectives we have not seen so far, and thus need to provide a signature for them. On line 11 and 15 we have the fancy update connective, `={_,_}=∗`, and on line 19 we have the iterated separating conjunction, `[∗ list]`. That is the only addition to the commands and tactics we need.

Resulting from this inductive statement, we get all the properties of the fixpoint and the induction lemma for the total weakest precondition. These allow us to define all the proof rules.

We can thus use `eiInd` with the associated tactics to define useful and large inductive predicates and provide proofs about them.

Chapter 6

Evaluation of Elpi

In this chapter, we evaluate Elpi based on our experiences during this work. We first discuss where our work benefited from Elpi and Coq-Elpi in section 6.1. Next, in section 6.2, we discuss where Elpi could be improved and where difficulties lie in using it as a commands and tactics meta-programming language. Lastly, we discuss if Elpi can be used to replace LTaC as the meta-programming language for the IPM in section 6.3.

6.1 Advantages of Elpi

We will highlight the advantages of using Elpi as a meta-programming language for Coq. We will discuss how logic programming is used and how Elpi interacts with Coq. Lastly, we discuss the documentation of Elpi.

Logic programming in Elpi Elpi is a logic programming language, similar to Prolog. It works best when making full use of the features of logic programming languages. This includes structuring predicates around backtracking and fully utilizing unification.

Debugging can be a challenge with programs that require extensive backtracking. It often happens that an error only surfaces after backtracking a few times. However, Elpi includes the excellent Elpi tracer and a Elpi trace browser extension [TW23] for the editor Visual Studio Code. It enables one to visually examine all paths taken by the interpreter while executing the program. This greatly helps in understanding where backtracking happened wrongly. This is helpful when starting with a new programming paradigm like logic programming.

Several other additions that Elpi has made to λProlog, made it easier and more concise to program. By spilling the use of explicit intermediary variables is greatly reduced. Warnings for variables that are only used once help reduce typos. Finally, the Elpi database allows for more modular tactics and commands.

Interacting with Coq Coq-Elpi has worked very well in facilitating the interaction between Coq and Elpi. Quotation and anti-quotation allow for easily creating and extracting Coq terms and greatly reduces noise by embracing the Coq notations.

When using term constructors, the HOAS structure works well. Writing recursive functions to create or interpret terms creates clean and readable code, even though binders can behave unexpectedly, as we will touch upon in the next section.

When creating Coq terms, it is essential to make sure they are properly typed. Elpi does not have a typed and an untyped term, like other Coq meta-programming languages such as LTaC have. But, since you have complete control over when to call the Coq type checker, you often type-check a term right before using it in Coq. This strongly reduces unnecessary type checking. Furthermore, encoding the type of binders using the `decl` predicate allows one to circumvent the type checker entirely when possible.

Lastly, when the type checker fails and backtracking is properly handled, the type checking error is automatically shown with the failure of the tactic. This greatly improves the experience for the user when a command or tactic does not work.

Documentation Getting started in Elpi is made easy with the excellent tutorials on writing Elpi code, and creating either a command or tactic. They explain step by step how the logic programming language can be used. They explain some major cautions and pitfalls and ensure that small programs are easily developed.

The documentation of the standard library of Elpi and Coq-Elpi consists of comments in the source code of the standard libraries. These comments are thorough and help explain most of the standard library, but they do make the whole process less accessible than either a document or a website containing the documentation for the standard libraries.

6.2 Issues with Elpi

In this section, we will discuss the challenges we encountered while interacting with Elpi. Despite Elpi's strengths, there are certain areas where it encounters issues. We first discuss how Elpi and LTaC interact. Next, we discuss the difficulties with using binders in Elpi. We then show why anonymous predicates in Elpi are prone to bugs. Lastly, we discuss why debugging large programs in Elpi is difficult.

Disadvantages of combining Elpi with LTaC In Elpi you can call LTaC code with the needed arguments like terms, strings, and other types. However, integrating LTaC tactics into an Elpi proof often poses significant challenges.

Since the Coq context is declared by adding rules to the Elpi context, a proof state does not simply consist of a proof variable and a type. It also consists of all the constants and their declared types. When creating proofs in Elpi we incrementally increase the Coq context and thus the Elpi context. However, when calling an LTaC tactic on a proof variable with arguments, the resulting goal has no relation to the old binders used in the proof. This makes passing values throughout the proof very difficult and frequently results in obscure errors surrounding binders and variables.

The result of these issues is that it is only really feasible to use LTaC tactics when they finish a branch of the proof. Only when no terms have to be passed to subsequent sections of the proof can you use LTaC code in between[1][2].

Binders in Elpi One of the main sources of trouble in the previous paragraph were binders in Elpi. While they are an essential part of the HOAS structure of Coq terms

[1]We have successfully allowed for calling the `simpl` tactic using `eiIntros`, however, any more complicated LTaC tactics have to be managed carefully.

[2]Our first attempt at the commands and tactics described in this thesis was based on calling LTaC a lot more, it also called the Elpi proof generators as if they were LTaC tactics, thus creating binders for every step of the proof. This resulted in many hard to debug errors and weird behaviors. Therefore, we switched directions from these LTaC like tactics towards what we now call holes.

in Elpi, they can work in un-intuitive ways. Every Elpi variable is quantified over all binders it is under at declaration. A variable can thus only contain binders over which it is quantified. This leads to a myriad of errors when returning terms created under a binder or when a variable gets quantified over a binder twice[3].

Binders are an essential and powerful part of Elpi. However, they are also quite unintuitive and may hinder the features that depend on them.

Anonymous predicates Any anonymous predicates containing intermediary variables are susceptible to errors. As described in section 4.7.4 and above, variables are bound in the uppermost predicate they are defined in. Thus, when creating an anonymous predicate where either the predicate is used multiple times, or it is used under a binder, the predicate fails and backtracks when executed. This is mitigated by adding `sigma X\` for every variable `X` at the start of an anonymous predicate. However, when using spilling in an anonymous predicate, you do not have access to the intermediary variable. Therefore, it is generally not possible to use spilling in anonymous predicates.

The problems described above make anonymous predicates only useful when they are small. Any other predicates should be created using the normal `pred` keyword at the top level. However, especially when using CPS, you often need a small predicate that is only used once. Here, an anonymous predicate would be useful, as seen by the listing in section 4.7.4. There, we still used anonymous predicates and worked around the issues described here.

Debugging large programs We have previously discussed the advantages of Elpi's tracer in debugging small programs. However, currently, the tracer does not function properly in larger programs. The tracer significantly increases the execution time of a program. Furthermore, the created traces are too large for the Visual Studio Code extension to ingest. You can limit traces to only a few predicates. However, this is frequently not enough to fully grasp the execution, given the amount of backtracking. Given that the tracer is no longer usable when debugging programs, the difficulty of debugging Elpi programs becomes apparent.

Elpi programs creating large terms need to print them often during debugging. These large terms are even longer to print as Elpi constructors and when printing using the Coq pretty printer, important details can be missed. Investigating why unclear error messages occur becomes a lot harder without full introspection in the program trace.

Thus, either you split a program up into multiple stages during development, or you endure the slower and more laborious process of print debugging while backtracking.

6.3 Elpi as the meta programming language for the IPM

In this section, we will discuss the benefits and downsides of using Elpi as the meta-programming language for the IPM of Iris.

Firstly, Elpi works best when the entire system is written in Elpi. Thus, when implementing the IPM in Elpi, the entire IPM needs to be ported to Elpi. A representative portion of the tactics in the IPM have already been implemented as part of this thesis. Therefore, the switch to Elpi should be possible.

[3]This is a bug in Elpi that has been reported.

Switching to Elpi could also come with several benefits. The Elpi database could allow for more modular tactics. For example, the tactic `pm_reduce` reduces a term on only pre-determined definitions. Using the Elpi database, definitions could be added to `pm_reduce` whenever they are defined. Currently, these definitions need to happen before `pm_reduce` is defined. Furthermore, deeper introspection into the goal and proof term could allow for removing workarounds and creating more powerful tactics. Instead of keeping a fresh anonymous identifier counter in the Iris context, one could search through the used identifiers and choose one that has not been used. Given that no type checking or elaboration needs to be done during such an operation, this should not induce a significant slowdown. Thus, Elpi could allow for more powerful and modular tactics by making use of Elpi specific features.

However, using Elpi also imposes some drawbacks besides the all-or-nothing approach. Elpi proof generators do not mimic the Coq syntax as closely as the current implementation of IPM tactics does. This raises the barrier to entry when creating new tactics or porting existing ones to Elpi. Additionally, creating tactics in Elpi requires a certain base understanding of the inner workings of Coq, which is not necessary with the current implementation. Ultimately, this results in a harder to parse code base with more verbosity.

Porting the IPM to Elpi could be a net benefit if all of the IPM is ported to Elpi. Elpi is continuously getting improved, and there are possibilities for features to be added to Elpi to aid in the transition of the IPM to Elpi.

Chapter 7

Related work

7.1 Other projects using Elpi

There have been several projects that have used Elpi to create commands and tactics. Both Derive [Tas19] and Hierarchy Builder [CST20] center around creating definitions and do not involve creating tactics. The project Trocq [CCM24] creates commands and a tactic to facilitate proof transfer in Coq. They do not focus on creating modular proof generators and only generate one large proof term.

7.2 Inductive predicates in program verification systems

We will discuss the different approaches to program verification and how they represent inductive predicates. There have been various approaches to program verification used in the past 30 years. They can be roughly categorized into three categories when looking at inductive predicates. Program verifiers that don't use separation logic. Program verifiers that use separation logic, but in their own verifier. And program verifiers that embed separation logic in an interactive proof assistant.

Program verifier without separation logic Program verifiers in this category are, for example, Dafny [Lei10], and Spec# [BLS04; LM10]. These program verifiers do not have to deal with representation predicates and thus do not create inductive predicates.

Separation logic program verifiers without a proof assistant These program verifiers do not have to embed the separation logic into another logic. Thus, they add inductive predicates and induction as axioms to the separation logic. Projects in this category are VeriFast [Jac+11], Viper [MSS16; SM18], and Smallfoot [BCO05].

Separation logic program verifier in a proof assistant These program verifiers embed the separation logic into the logic of the proof assistant. This can be done in several ways. Both works by Appel and Rocquencourt [AR], and Rouvoet, Krebbers, and Visser [RKV21], embed separation logic as propositions from a concrete heap to the proof assistant propositions. Thus, they can both use the inductive definition components of the respective proof assistant for defining inductive predicates in the separation logic.

The work by Chlipala [Chl11] and Bengtson, Jensen, and Birkedal [BJB12], both embed a separation logic in Coq. They use embeddings of separation logics, which don't allow for using the Coq `Inductive` statement. They only support defining representation predicates using the Coq `Fixpoint`. Thus, using structural recursion.

Lastly, Appel [App14] defines inductive predicates similarly to Iris. Thus, they define a monotone pre fixpoint function and take the fixpoint.

7.3 Other implementations of the IPM

In this thesis, we reimplemented several tactics of the IPM. This replication of [KTB17] has been done various times before in the meta programming languages LTac2 and MTac2. And in the proof assistant Lean. The implementation in LTac2 was done in the master thesis of Liesnikov [Lie20]. They keep the same structure in their tactics as the IPM while also adding some tactics of their own.

The MTac2 meta programming language creates fully typed tactics. In the paper introducing MTac2 by Kaiser et al. [Kai+18] some tactics of the IPM were reimplemented in MTac2. This implementation focused on showing the capabilities of MTac2 by solving a few known faults in the original IPM.

Lastly, the IPM was also implemented in Lean by König [Kön22]. Unlike the previous two reimplementations of the IPM, this instance also had to replicate all definitions and lemmas since it uses a different proof assistant as its base logic.

All three reimplementations of the IPM did not consider inductive predicates. The first two reimplementations can make use of the same strategy of defining inductive separation logic predicates as used in Iris. The last reimplementations of the IPM can utilize the Lean structural recursion or a similar fixpoint construction as in Iris to define inductive predicates.

7.4 Algorithms based on proper elements and signatures

The concept of proper elements and signatures was taken from the work by Sozeau [Soz09]. They use proper elements and signatures (called Propers in their work) to create a tactic for generalized rewriting in Coq. This tactic extends the existing `rewrite` tactic from Coq by allowing one to rewrite lemmas under terms for which an appropriate `Proper` instance is given.

This is a fairly different use of the same base definitions of signatures and respectful, and pointwise relations. But, it informed our approach to automatically proving monotonicity pre fixpoint functions.

Chapter 8

Conclusion

In this thesis we showed how to create inductive predicates automatically in the Iris logic in Coq using Elpi. To accomplish this, we created a command, `eiInd`, which, given a standard Coq inductive statement on the Iris separation logic, defines the inductive predicate with its associated lemmas. Next, we created tactics that allow one to easily eliminate the inductive predicate, apply constructors, and perform induction. These tactics were integrated into a novel partial reimplementation of the IPM in Elpi to allow the inductive predicates to be tightly integrated. Lastly, we showed that the system created for defining inductive predicates can define complicated predicates like the total weakest precondition, as originally defined by the IPM.

8.1 Future work

We see three possible directions for feature work. Implement more advanced tactics and definitions related to inductive predicates in the Elpi implementation of the IPM. Add the non-expansive property to relevant definitions and lemmas. Generalize the fixpoint generation to coinductive predicates and the Banach fixpoint.

Advanced inductive definitions and tactics The Coq inductive command has support for mutually defined inductive types. These are two inductive predicates that are mutually dependent on each other, i.e., you cannot define one before you define the other. Creating these types of inductive predicates is not currently possible in the system we developed. Adding this to our system might require features from Elpi that are not yet implemented. Another feature of Coq inductive predicates is the `inversion` tactic. This tactic derives the possible constructors with which an inductive predicate was created given its arguments [CT96]. The `inversion` tactic is an essential part of many Coq proofs about inductive predicates and could be interesting to implement for Iris inductive predicates.

Non-expansive inductive predicates The Iris definitions for the fixpoint included a non-expansive requirement for the pre fixpoint function. A pre fixpoint function, F, is non-expansive if $\forall \Phi, \Psi.\ (\forall x.\ \Phi\, x \iff \Psi\, x) \mathrel{-\!\!*} (\forall x.\ \mathsf{F}\, \Phi\, x \iff \mathsf{F}\, \Psi\, x)$. This property has to be expanded to any arity as was done for monotonicity and automatically solved. This would allow for a larger class of inductive predicates to be defined, namely ones where the fixpoint properties depend on the fixpoint being non-expansive.

Coinductive and Banach inductive predicates Besides the inductive predicates we defined based on the least fixpoint, there are two other non-automated classes of (co)inductive predicates available in Iris. These are the inductive predicates based on the Banach fixpoint, and coinductive predicates based on the greatest fixpoint. These allow for more types of (co)inductive predicates and other notions of (co)induction on these predicates.

Bibliography

[App14] Andrew W. Appel. *Program Logics for Certified Compilers.* 2014. DOI: 10.1017/CBO9781107256552.

[AR] Andrew W Appel and INRIA Rocquencourt. "Tactics for Separation Logic". In: ().

[BBR99] Catherine Belleannée, Pascal Brisset, and Olivier Ridoux. "A Pragmatic Reconstruction of λProlog". In: *The Journal of Logic Programming* 41.1 (Oct. 1, 1999), pp. 67–102. DOI: 10.1016/S0743-1066(98)10038-9.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *Proc. 4th Int. Conf. Form. Methods Compon. Objects.* FMCO'05. Nov. 1, 2005, pp. 115–137. DOI: 10.1007/11804192_6.

[BJB12] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. "Charge!" In: *Interact. Theorem Proving.* 2012, pp. 315–331. DOI: 10.1007/978-3-642-32347-8_21.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# Programming System: An Overview". In: *Proc. 2004 Int. Conf. Constr. Anal. Safe Secure Interoper. Smart Devices.* CASSIS'04. Mar. 10, 2004, pp. 49–69. DOI: 10.1007/978-3-540-30569-9_3.

[CCM24] Cyril Cohen, Enzo Crance, and Assia Mahboubi. "Trocq: Proof Transfer for Free, With or Without Univalence". In: *Program. Lang. Syst.* 2024, pp. 239–268. DOI: 10.1007/978-3-031-57262-3_10.

[Cha+19] Tej Chajed et al. "Verifying Concurrent, Crash-Safe Systems with Perennial". In: *Proc. 27th ACM Symp. Oper. Syst. Princ.* SOSP '19. Oct. 27, 2019, pp. 243–258. DOI: 10.1145/3341301.3359632.

[Chl11] Adam Chlipala. "Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic". In: *SIGPLAN Not.* 46.6 (June 4, 2011), pp. 234–245. DOI: 10.1145/1993316.1993526.

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. "Hierarchy Builder: Algebraic Hierarchies Made Easy in Coq with Elpi". In: FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction. 167. June 29, 2020, 34:1. DOI: 10.4230/LIPIcs.FSCD.2020.34.

[CT96] Cristina Cornes and Delphine Terrasse. "Automating Inversion of Inductive Predicates in Coq". In: *Types Proofs Programs.* 1996, pp. 85–104. DOI: 10.1007/3-540-61780-9_64.

[Dan+19] Hoang-Hai Dang et al. “RustBelt Meets Relaxed Memory”. In: *Proc. ACM Program. Lang.* 4 (POPL Dec. 20, 2019), 34:1–34:29. DOI: `10.1145/3371102`.

[Dun+15] 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`.

[GCT19] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. “Implementing Type Theory in Higher Order Constraint Logic Programming”. In: *Math. Struct. Comput. Sci.* 29.8 (Sept. 2019), pp. 1125–1150. DOI: `10.1017/S0960129518000427`.

[Gia+20] Paolo G. Giarrusso et al. “Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris”. In: *Proc. ACM Program. Lang.* 4 (ICFP Aug. 3, 2020), 114:1–114:29. DOI: `10.1145/3408996`.

[GMT16] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. “A Small Scale Reflection Extension for the Coq System”. PhD thesis. Inria Saclay Ile de France, 2016. URL: `https://inria.hal.science/inria-00258384/document`.

[Har01] Timothy L. Harris. “A Pragmatic Implementation of Non-blocking Linked-lists”. In: *Distrib. Comput.* 2001, pp. 300–314. DOI: `10.1007/3-540-45414-4_21`.

[HKP97] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. “The Coq Proof Assistant a Tutorial”. In: *Rapp. Tech.* 178 (1997). URL: `http://www.itpro.titech.ac.jp/coq.8.2/Tutorial.pdf`.

[IO01] Samin S. Ishtiaq and Peter W. O’Hearn. “BI as an Assertion Language for Mutable Data Structures”. In: *SIGPLAN Not.* 36.3 (Jan. 1, 2001), pp. 14–26. DOI: `10.1145/373243.375719`.

[Iri23] The Iris Team. “The Iris 4.1 Reference”. In: (Nov. 10, 2023), pp. 51–56. URL: `https://plv.mpi-sws.org/iris/appendix-4.1.pdf`.

[Jac+11] Bart Jacobs et al. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *Proc. Third Int. Conf. NASA Form. Methods.* NFM’11. Apr. 18, 2011, pp. 41–55.

[Jun+15] 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`.

[Jun+16] Ralf Jung et al. “Higher-Order Ghost State”. In: *SIGPLAN Not.* 51.9 (Sept. 4, 2016), pp. 256–269. DOI: `10.1145/3022670.2951943`.

[Jun+17] Ralf Jung et al. “RustBelt: Securing the Foundations of the Rust Programming Language”. In: *Proc. ACM Program. Lang.* 2 (POPL Dec. 27, 2017), 66:1–66:34. DOI: `10.1145/3158154`.

[Jun+18] 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`.

[Kai+18] Jan-Oliver Kaiser et al. “Mtac2: Typed Tactics for Backward Reasoning in Coq”. In: *Proc. ACM Program. Lang.* 2 (ICFP July 30, 2018), 78:1–78:31. DOI: `10.1145/3236773`.

[Kön22] Lars König. *An Improved Interface for Interactive Proofs in Separation Logic.* 2022. DOI: `10.5445/IR/1000153230`.

[Kre+17] 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`.

[Kre+18] 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`.

[KTB17] 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`.

[Lei10] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Log. Program. Artif. Intell. Reason.* 2010, pp. 348–370. DOI: `10.1007/978-3-642-17511-4_20`.

[Lie20] Bohdan Liesnikov. *Extending and Automating Iris Proof Mode with Ltac2.* Saarland University Faculty of Mathematics and Computer Science, Dec. 7, 2020. URL: `https://github.com/liesnikov/msc-thesis/releases/tag/posterity-build`.

[LM10] K. Rustan M. Leino and Peter Müller. “Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs”. In: *Advanced Lectures on Software Engineering: LASER Summer School 2007/2008.* Jan. 1, 2010, pp. 91–139.

[Mat+22] Yusuke Matsushita et al. “RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code”. In: *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implement.* PLDI 2022. June 9, 2022, pp. 841–856. DOI: `10.1145/3519939.3523704`.

[Mil+91] Dale Miller et al. “Uniform Proofs as a Foundation for Logic Programming”. In: *Annals of Pure and Applied Logic* 51.1 (Mar. 14, 1991), pp. 125–157. DOI: `10.1016/0168-0072(91)90068-W`.

[MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic.* 2012. DOI: `10.1017/CBO9781139021326`.

[MN86] Dale A. Miller and Gopalan Nadathur. “Higher-Order Logic Programming”. In: *Third Int. Conf. Log. Program.* Lecture Notes in Computer Science. 1986, pp. 448–462. DOI: `10.1007/3-540-16492-8_94`.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification Model Checking Abstr. Interpret.* 2016, pp. 41–62. DOI: `10.1007/978-3-662-49122-5_2`.

[ORY01] Peter O’Hearn, John Reynolds, and Hongseok Yang. “Local Reasoning about Programs That Alter Data Structures”. In: *Comput. Sci. Log.* 2001, pp. 1–19. DOI: `10.1007/3-540-44802-0_1`.

[PE88] F. Pfenning and C. Elliott. “Higher-Order Abstract Syntax”. In: *Proc. ACM SIGPLAN 1988 Conf. Program. Lang. Des. Implement.* PLDI ’88. June 1, 1988, pp. 199–208. DOI: `10.1145/53990.54010`.

[Rao+23] Xiaojia Rao et al. “Iris-Wasm: Robust and Modular Verification of WebAssembly Programs”. In: *Proc. ACM Program. Lang.* 7 (PLDI June 6, 2023), 151:1096–151:1120. DOI: `10.1145/3591265`.

[Rey02] J.C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Proc. 17th Annu. IEEE Symp. Log. Comput. Sci.* Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. July 2002, pp. 55–74. DOI: `10.1109/LICS.2002.1029817`.

[RKV21] Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. “Intrinsically Typed Compilation with Nameless Labels”. In: *Proc. ACM Program. Lang.* 5 (POPL Jan. 4, 2021), 22:1–22:28. DOI: `10.1145/3434303`.

[Sam+21] Michael Sammler et al. “RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types”. In: *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implement.* PLDI 2021. June 18, 2021, pp. 158–174. DOI: `10.1145/3453483.3454036`.

[SM18] Alexander J. Summers and Peter Müller. “Automating Deductive Verification for Weak-Memory Programs”. In: *Tools Algorithms Constr. Anal. Syst.* 2018, pp. 190–209. DOI: `10.1007/978-3-319-89960-2_11`.

[Soz09] Matthieu Sozeau. “A New Look at Generalized Rewriting in Type Theory”. In: *J. Formaliz. Reason.* 2.1 (1 2009), pp. 41–62. DOI: `10.6092/issn.1972-5787/1574`.

[Tar55] Alfred Tarski. “A Lattice-Theoretical Fixpoint Theorem and Its Applications”. In: *Pac. J. Math.* 5.2 (June 1, 1955), pp. 285–309. URL: `https://msp.org/pjm/1955/5-2/p11.xhtml`.

[Tas18] Enrico Tassi. “Elpi: An Extension Language for Coq (Metaprogramming Coq in the Elpi λProlog Dialect)”. Jan. 2018. URL: `https://inria.hal.science/hal-01637063`.

[Tas19] Enrico Tassi. “Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq”. In: *DROPS-IDNv2document104230LIPIcsITP201929.* 10th International Conference on Interactive Theorem Proving (ITP 2019). 2019. DOI: `10.4230/LIPIcs.ITP.2019.29`.

[TW23] Enrico Tassi and Julien Wintz. *LPCIC/Elpi-Lang.* Version v0.2.6. λProlog and the Calculus of Inductive Constructions, Aug. 21, 2023. URL: `https://github.com/LPCIC/elpi-lang`.