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:
...
...

May 21, 2024

Abstract

Field, current gap, direction of solution, Results, Genererailzation of results and where else to apply it.

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

Contents

Chapter 1

Introduction

Iris is a separation logic [Jun+15; Jun+16; Kre+17; Jun+18]. It is implemented in Coq in what is called the Iris Proof Mode (IPM) [KTB17; Kre+18].

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 langugae 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, $\texttt{let}\, x = e\, \texttt{in}\, e'$, 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 $\texttt{let}\, \_ = e\, \texttt{in}\, e'$. The keywords **none** and **some** are just **inl** and **inr** respectively, both in values

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

Figure 2.1: Relevant fragment of the syntax of HeapLang

and in the match statement. We define the short circuit and, $e_1 \&\& e_2$, using the following if statement, $\mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, \mathbf{false}$. 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 deleting it.

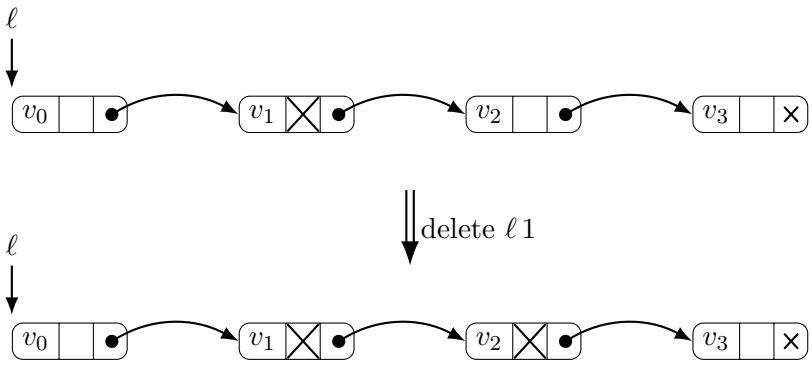
$$
\begin{array}{l}
\text{delete}\ \mathit{hd}\, i = \mathbf{match}\, \mathit{hd}\, \mathbf{with} \\
\quad\ \ \mathbf{none}\ \ \Rightarrow () \\
\quad \mid \mathbf{some}\, \ell \Rightarrow \mathbf{let}\, (x, \mathit{mark}, \mathit{tl}) = !\, \ell\, \mathbf{in} \\
\qquad\qquad \mathbf{if}\ \mathit{mark} = \mathbf{false}\ \&\&\ i = 0\, \mathbf{then} \\
\qquad\qquad\quad \ell \leftarrow (x, \mathbf{true}, \mathit{tl}) \\
\qquad\qquad \mathbf{else\ if}\ \mathit{mark} = \mathbf{false}\, \mathbf{then} \\
\qquad\qquad\quad \text{delete}\ \mathit{tl}\ (i - 1) \\
\qquad\qquad \mathbf{else} \\
\qquad\qquad\quad \text{delete}\ \mathit{tl}\ i \\
\quad\ \ \mathbf{end}
\end{array}
$$

The example is a recursive function called delete, the function has two arguments. HeapLang has no null pointers, thus we wrap a pointer in **none**, the null pointer, $\mathbf{some}\, \ell$, 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 do 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.



A tuple is shown here as three boxes next to each other, the first box contains a value. The second box is a boolean, it 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.

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 seperation logiv. This subset includes separation logic as first presented by Ishtiaq et al. and Reynolds

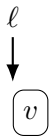
[IO01; Rey02], together with higher order connectives, persistent propositions and weakest preconditions as introduced by Iris. This logic is presented below, starting with the syntax.

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

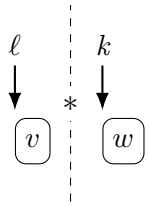
Separation logic contains all the usual higher order predicate logic connectives as seen on the first line. The symbol τ, represents any type we have seen, including *iProp* itself. The second row contains separation logic specific connectives. The *pure* connective, $\ulcorner \phi \urcorner$, embeds 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.

$$\begin{aligned} \mathsf{True} * P &\dashv\vdash P \\ P * Q &\vdash Q * P \\ (P * Q) * R &\vdash P * (Q * R) \end{aligned} \qquad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \; {}^{*\text{-MONO}}$$

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 afterwards the postcondition holds. To do this we first 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. Afterwards, 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 very simple 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 afterwards $\ell \mapsto w$ 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 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 seen as a Hoare triple without its precondition. Thus, $\mathsf{wp}\ e\ [\Phi]$ states that e does not crash and that it terminates. Afterwards, 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 taking into account the heap. The statement, $Q \mathrel{-\!\!*} R$, describes the state of memory where if we add the memory described by Q we get R. This property is expressed by the below rule.

$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}\ {\scriptstyle \mathrel{-\!\!*}\text{I-E}}$$

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

$$\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}\ \ell \leftarrow w\ [\ell \mapsto w]$$

This specification holds from WP-STORE in figure 2.2. The rules in this diagram follow a different style then is expected. We could have use the above specification of $\ell \leftarrow v$ as the rule. However, we make use of a "backwards" style [IO01; Rey02], 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.

General rules.

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

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

Rules for basic language constructs.

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

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

Pure reductions.

$$(\mathbf{rec}\ f(x) = e)v \longrightarrow_{\text{pure}} e[v/x][fx := e/f]$$

$$\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow_{\text{pure}} e_1$$

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

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

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

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

Context rules

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

Figure 2.2: Rules for the weakest precondition assertion.

- 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

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.

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

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 $\square$-MONO below.

$$\square\text{-DUP}\quad \square P \dashv\vdash \square P * \square P \qquad \square\text{-SEP}\quad \square (P * Q) \dashv\vdash \square P * \square Q \qquad \square\text{-MONO}\quad \frac{P \vdash Q}{\square P \vdash \square Q}$$

$$\square\text{-E}\quad \square P \vdash P \qquad \square\text{-CONJ}\quad \square P \wedge Q \vdash \square P * Q \qquad \frac{\ulcorner \phi \urcorner \vdash \square \ulcorner \phi \urcorner}{\mathsf{True} \vdash \square\, \mathsf{True}}$$

$$\square P \vdash \square \square P$$
$$\forall x.\ \square P \vdash \square \forall x.\ P$$
$$\square \exists x.\ P \vdash \exists x.\ \square P$$

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

$$\square\text{-I}\quad \frac{\mathsf{persistent}(P) \qquad P \vdash Q}{P \vdash \square Q}$$

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.

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

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.

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

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

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

We apply the WP-BIND rule with the following context

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

Resulting in the following weakest precondition we need to prove.

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

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

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

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.

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

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

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

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 prove state.

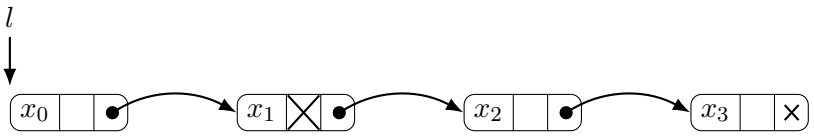
$$\mathsf{wp}\ !\,\ell\ [20.\,\mathsf{True}]$$

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, $\mathsf{isMLL}\,(\textbf{some}\,\ell)\,[x_0, x_2, x_3]$.

To illustrate how isMLL works we give the below inductive property. In chapter 3 we will show how isMLLis defined and that it has the below property.

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

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.

isMLL-IND

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

To use this rule we need two things. We need to have an assumption of the shape 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 name $\vec{v}$. 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.

$$
\begin{array}{l}
\text{delete } hd\, i = \mathbf{match}\ hd\ \mathbf{with} \\
\quad\ \mathbf{none} \ \Rightarrow () \\
\quad \mid \mathbf{some}\ \ell \Rightarrow \mathbf{let}\ (x, mark, tl) = !\,\ell\ \mathbf{in} \\
\qquad\qquad \mathbf{if}\ mark = \mathbf{false}\ \&\&\ i = 0\ \mathbf{then} \\
\qquad\qquad\quad \ell \leftarrow (x, \mathbf{true}, tl) \\
\qquad\qquad \mathbf{else\ if}\ mark = \mathbf{false}\ \mathbf{then} \\
\qquad\qquad\quad \text{delete } tl\ (i-1) \\
\qquad\qquad \mathbf{else} \\
\qquad\qquad\quad \text{delete } tl\ i \\
\quad \mathbf{end}
\end{array}
$$

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.

$$\square(\mathsf{isMLL}\ hd\ \vec{v} \mathrel{-\!\!*} \mathsf{wp}\ \text{delete } hd\, i\ [\mathsf{isMLL}\ hd\ (\mathsf{remove}\, i\ \vec{v})])$$

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

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

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:

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

This follows from the definition of isMLL

Marked Head: We know that $\ell \mapsto (v', \mathbf{true}, tl)$ 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:

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

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

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

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

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

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

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

Which is the left-hand side of our IH.

Unmarked head: We know that $\ell \mapsto (v', \mathbf{false}, tl)$ 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:

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

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

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

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. □

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 hard 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 fix this problem, as is discussed in chapter 6. 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

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

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.

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

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.

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

Proof. We start by introducing v and the wand.

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

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

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

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

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.

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

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 $\mathsf{F} \colon (A \to \mathit{iProp}) \to A \to \mathit{iProp}$, 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 $\mathsf{F} : (A \to \mathit{iProp}) \to (A \to \mathit{iProp})$, it holds that

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

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.

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

The first step is creating the function. We do this by adding an argument to isMLL transforming it into a function. We then substitute any recursive calls to isMLL with this argument.

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

This has created a function, $\mathsf{isMLL}_{\mathsf{F}}$. 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.

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

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

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

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

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

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. □

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, thus we take a different approach on 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
$*$	$iProp \to iProp \to iProp$	$(-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*)$
$\vee$	$iProp \to iProp \to iProp$	$(-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*)$
$-\!\!*$	$iProp \to iProp \to iProp$	$\mathsf{flip}(-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*)$
$\exists$	$(A \to iProp) \to iProp$	$((=) \Longrightarrow (-\!\!*)) \Longrightarrow (-\!\!*)$

We make use of the Haskell prefix notation, $(-\!\!*)$, 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 $R \Longrightarrow R'$: $iRel\ (A \to B)$ of two relations R: $iRel\ A$, R': $iRel\ B$ 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,

$$\forall \Phi, \Psi.\ (\forall x, y.\ x = y \twoheadrightarrow \Phi\, x \twoheadrightarrow \Psi\, y) \twoheadrightarrow (\exists x.\ \Phi\, x) \twoheadrightarrow (\exists x.\ \Psi\, x)$$

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

$$\forall \Phi, \Psi.\ (\forall x.\ \Phi\, x \twoheadrightarrow \Psi\, x) \twoheadrightarrow (\exists x.\ \Phi\, x) \twoheadrightarrow (\exists x.\ \Psi\, x)$$

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 $\gg R$ is a special case of a respectful relation defined as

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

The new signature for the existential quantification becomes

$$\gg (\twoheadrightarrow) \Longrightarrow (\twoheadrightarrow)$$

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 the combinators needed, but we are missing a way to mark a relation as persistent.

Definition 3.11: Persistent relation

The persistent relation $\square R$: *iRel A* for a relation R: *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.

$$\square(\mathrel{\ggg}(-\!\!*)) \Longrightarrow \mathrel{\ggg}(-\!\!*)$$

Filling in a F as the proper element get the following statement.

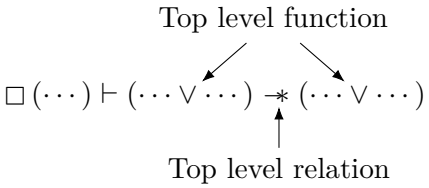
$$\square(\forall y.\, \Phi\, y \mathrel{-\!\!*} \Psi\, y) \mathrel{-\!\!*} \forall x.\, \mathsf{F}\, \Phi\, x \mathrel{-\!\!*} \mathsf{F}\, \Psi\, x$$

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.

$$\square(\mathrel{\ggg} \mathrel{\ggg} (-\!\!*)) \Longrightarrow \mathrel{\ggg} \mathrel{\ggg} (-\!\!*)$$

We are thus able to 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.



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

$$\square(\mathrel{\ggg} \mathrel{\ggg} (-\!\!*)) \Longrightarrow \mathrel{\ggg} \mathrel{\ggg} (-\!\!*)$$

In other words

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

Proof. We assume any premises, $\square\,(\forall hd\,\vec{v}.\,\Phi\,hd\,\vec{v} \mathrel{-\!\!*} \Psi\,hd\,\vec{v})$. 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.

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

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 $(-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*)$ with $(\vee)$. We apply $((-\!\!*) \Longrightarrow (-\!\!*) \Longrightarrow (-\!\!*))(\vee)(\vee)$ resulting in two statements to prove.

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

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 end results of applying it.

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

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

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

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

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

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

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

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. Afterwards, do an application step.

Application We apply the first option that works.

1. If 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 amount of arguments to the fixpoint and its properties.

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

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

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

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

$$\mu\mathsf{isMLL_F}\ \mathit{hd}\ \vec{v} \triangleq \forall \Phi.\ \Box(\forall \mathit{hd}', \vec{v}'.\, \mathsf{isMLL_F}\ \Phi\ \mathit{hd}'\ \vec{v}' \mathrel{-\!\!*} \Phi\ \mathit{hd}'\ \vec{v}') \mathrel{-\!\!*} \Phi\ \mathit{hd}\ \vec{v}$$

For the induction principle we apply the same strategy. For 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 can be used to create new (IPM) tactics in Coq. This chapter will function explain relevant inner working of the 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 the IPM. And, in section 4.3 we explain the Iris lemmas we use as the building blocks for our 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. It implements a similar domain specific language (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` can be 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) proposition. In the IPM there are two corresponding contexts, the persistent and spatial context. Consider the statement $\vdash P \mathrel{-\!\!*} \Box Q \mathrel{-\!\!*} P$. As a Coq goal this would be

```Coq
P, Q: iProp
============
------------*
P -* □ Q -* P
```

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

```Coq
P, Q: iProp
============
"HQ" : Q
------------□
"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).

This does not only work on the magic wand, we can also use this to introduce more complicated statements. Take the following proof state,

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

It consists of a universal quantification, an existential quantification, a separating conjunction and a disjunction. We can 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.

```Coq
(1/2)
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
```
Coq

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.

Thus, we can break down `iIntros "%x [[%y [Hx Hy]] | H0]"` into its components. We first forall introduce or first sub intro pattern `"%x"` and then perform the second case, introduce a pure Coq variable for the `∀ x : nat`. Next we wand introduce for the second sub intro pattern, `"[[%y [Hx Hy]] | H0]"` and interpret the outer pattern. it is the third case and eliminates the disjunction, resulting in two goals. The left patterns of the seperating conjunction pattern eliminates the exists and adds the `y` to the Coq context. Lastly, `"[Hx Hy]"` is the fourth case and eliminates the seperating conjunction in the Iris context by splitting it into two assumptions `"Hx"` and `"Hy"`.

There are more patterns available to introduce more complicated goals, these can be found in a paper written by Krebbers, Timany, and Birkedal [KTB17].

4.2 Contexts

Before starting the Elpi `eiIntros` tactic we need a quick interlude about how the Iris contexts and entailment are made 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 only a number, or a name. 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 of_envs
    (Γp Γs : env iProp) : iProp :=
  □ [∧] Γp ∧ [∗] Γs ∧ ⌜envs_wf Γp Γs⌝.

Definition envs_entails
    (Δ : envs iProp) (Q : iProp) : Prop :=
  of_envs (env_intuitionistic Δ) (env_spatial Δ) ⊢ Q.
```

The definition `of_envs` transforms the persistent and spatial context 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 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 preserve the context.

```
1 Lemma tac_wand_intro Δ i P Q :                                    Coq
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` hodls 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 tell 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 can then be 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. We will also make use of them many times while creating any tactics, and they will appear many times in section 4.7.

4.4 Elpi

We implement our tactic in the λProlog language Elpi [Dun+15; GCT19]. Elpi implements λprolog [MN86; Mil+91; BBR99; MN12] with a few additions. These additions are crucial for the workings of Coq-Elpi but won't be discussed here as they are not directly used the created tactics and commands.

To use Elpi as a Coq meta programming language, there exists the Elpi Coq connector, Coq-Elpi [Tas18]. We use Coq-Elpi to implement the Elpi variant of `iIntros`, named `eiIntros`.

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

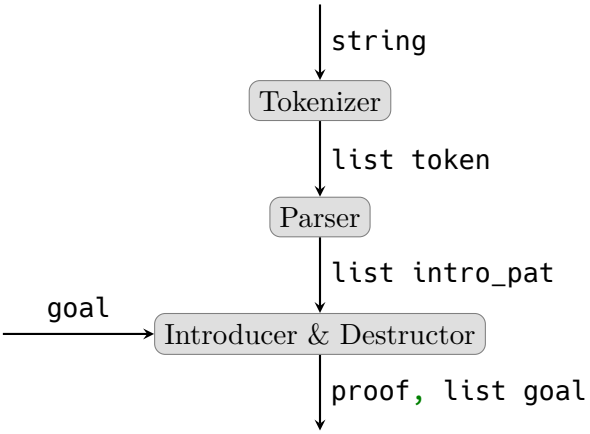


Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

In section 4.5 we describe how a string is tokenized by the tokenizer. In section 4.6 we describe how a list of tokens is parsed into a list of intro patterns. In section 4.7 we describe how we use an intro pattern to introduce and eliminate the needed connectives. In every section we describe more parts of the Elpi programming language and the Coq-Elpi connector starting with the base concepts of the language and working up to the mayor concepts of Elpi and Coq-Elpi.

4.5 Tokenizer

The tokenizer takes as input a string, 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 speperated 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".

```
kind token type.

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

```
6 type tPure option string -> token.
```
Elpi

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 and then the type of those constructors. The first set of constructors do not take any arguments, thus have type `token`, and just represent one or more constant characters. The next few constructors take an argument and produce a token, thus allowing us to store data in the tokens. For example, `tName` has type `string -> token`, thus containing a string. Besides `string`, there are a few more basic types in Elpi such as `int`, `float` and `bool`. We also have higher order types, like `option A`, and later on `list A`.

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

Creating types of kind `type -> type` can be done using the `kind` directive and passing in a more complicated kind as shown above.

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

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

4.5.2 Predicates

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

```
1 pred tokenize i:string, o:list token.
2 tokenize S O :-
3   rex.split "" S SS,
4   tokenize.rec SS O.
```
Elpi

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 can define the relation between the arguments. The first predicate we call, `rex.split`, has the following type:

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

When we call it, we assign the empty string to its first argument, the string we want to tokenize to the second argument, and we store the output list of string in the new variable `SS`. This predicate allows us to split a string at a certain delimiter. We take as delimiter the empty string, thus splitting the string up in a list of strings of one character each. Strings in Elpi are based on OCaml strings and are not lists of characters. Since Elpi does not support pattern matching on partial strings, we need this workaround. The next line, line 4, calls the recursive tokenizer, `tokenizer.rec`[1], on the list of split string and assigns the output to the output variable `O`.

The reason predicates in Elpi are called predicates and not functions, is that they don't always have to take an input and give an output. They can sometimes better be seen as predicates defining for which values of their arguments they hold. Each rule defines a list of predicates that need to hold for their premise to hold. Thus, a predicate can have multiple values for its output, as long as they hold for all contained rules. These multiple possible values can be reached by backtracking, which we will discuss in section 4.5.5. To execute a predicate, we thus find the first rule for which its premise is sufficient for the arguments we supply. We then check if each of the predicates in the conclusion hold starting at the top. If they hold, 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
pred tokenize.rec i:list string, o:list token.
tokenize.rec [] [] :- !.
tokenize.rec [" " | SL] TS :- !, tokenize.rec SL TS.
```

[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.

[2] A fun side effect of outputs being just variables we pass to a predicate is that we can also easily create a function that is reversible. If we change the mode of our first argument to output and move rule 3 to the bottom, we can pass in a list of tokens and get back a list of strings representing this list of tokens.

```
4  tokenize.rec ["?" | SL] [tFresh | TS] :- !,                    Elpi
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.
```

This predicate has rules for all tokens, a few rules are considered here. All rules have a cut, `!`, as part of their conclusion, we will discuss cuts in section 4.5.5, for now they can be ignored. When calling this predicate, the first rule can be 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 our language is based on predicates, we still often defer to a functional style of programming. The first language feature that is very useful for this goal is spilling. Spilling allows us to write the entry point of the tokenizer as defined in section 4.5.2 without the need for temporary variables to be passed around.

```
1  pred tokenize o:string, o:list token.                          Elpi
2  tokenize S O :- tokenize.rec {rex.split "" S} O.
```

We spill the output of a predicate into the input of another predicate by using the `{ }` syntax. We don't specify the last argument of the predicate

and only the last argument of a predicate can be spilled.

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.

```
1 pred tokenize i:string, o:list token.                    Elpi
2 type tokenize string -> list token -> prop.
```

The `prop` type is the type of propositions, and with arguments they become predicates. We are thus able to write predicates that accept other predicates as arguments.

```
1 pred map i:list A, i:(A -> B -> prop), o:list B.         Elpi
2 map [] _ [].
3 map [X|XS] F [Y|YS] :- F X Y, map XS F YS.
```

`map` takes as its second argument a predicate on `A` and `B`. On line 3 we map this predicate to the variable `F`, and we then use it to either find a `Y` such that `F X Y` holds, or check if for a given `Y`, `F X Y` holds. We can use the same strategy to implement many of the common functional programming higher order functions.

4.5.5 Backtracking

In this section we will finally describe what happens when a rule fails to complete halfway through. We start with a predicate which will be of much use for the last part of our tokenizer.

```
1 pred take-while-split i:list A, i:(A -> prop),           Elpi
2                       o:list A, o:list A.
3 take-while-split [X|XS] Pred [X|YS] ZS :- Pred X, !,
4   take-while-split XS Pred YS ZS.
5 take-while-split XS _ [] XS.
```

`take-while-split` is a predicate that should take elements of its input list till its input predicate no longer holds and then output the first part of input in its third argument and the last part of the input in its fourth argument.

The predicate contains two rules. The first rule, defined on lines 2 and 3, recurses as long as the input predicate, `Pred` holds for the input list, `[X|XS]`. The second rule returns the last part of the list. 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 can 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 a backtrack 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 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.
```
Elpi

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

$$
\begin{aligned}
\langle intropattern\_list \rangle \quad ::= \; & \epsilon \\
| \; & \langle intropattern \rangle \; \langle intropattern\_list \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle intropattern \rangle \quad ::= \; & \langle ident \rangle \\
| \; & \text{`?'} \mid \text{`/='} \mid \text{`//'} \\
| \; & \text{`[' } \langle intropattern\_list \rangle \text{ `]'} \\
| \; & \text{`(' } \langle intropattern\_conj\_list \rangle \text{ `)'}
\end{aligned}
$$

$$
\begin{aligned}
\langle intropattern\_list \rangle \quad ::= \; & \epsilon \\
| \; & \langle intropattern \rangle \text{ `|' } \langle intropattern\_list \rangle \\
| \; & \langle intropattern \rangle \; \langle intropattern\_list \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle intropattern\_conj\_list \rangle \quad ::= \; & \epsilon \\
| \; & \langle intropattern \rangle \text{ `\&' } \langle intropattern\_conj\_list \rangle
\end{aligned}
$$

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 $\langle intropattern\_list \rangle$ parser. Backtracking can 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 can 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 can be 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 can be applied.

4.7.1 Coq-Elpi HOAS

Coq-Elpi makes use of Higher-order abstract syntax (HOAS) 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 O (S O)`. In Elpi this term is represented as follows.

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

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.

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

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.

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

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 in 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`.

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

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 can't 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»))`.

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

We make use of `=>` connective. The rule in front of `=>` is added on top of the know 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.

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

Now `FUN` has the value.

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

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

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

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:

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

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

```
Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:NewProof }};
```
Elpi

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

4.7.3 Proof steps in Elpi

Now that we have a solid foundation how to work with Coq terms in Elpi we can start creating proof terms. Proof steps in Elpi are build 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`.

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

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

A `hole` contains the goal as its first argument, together with the proof variable we need to assign the proof to as its second argument. Take the following proof term generator that applies the iris ex falso rule to the current hole.

```elpi
pred do-iExFalso i:hole, o:hole.
do-iExFalso (hole Type Proof)
            (hole FalseType FalseProof) :-
  coq.elaborate-skeleton
    {{ tac_ex_falso _ _ _ }} Type Proof ok,
  Proof = {{ tac_ex_falso _ _ lp:FalseProof }},
  coq.typecheck FalseProof FalseType ok.
```
Elpi

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.
```
Coq

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 case `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 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 assumption 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.

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

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

In a proof generator we can 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.

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

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.

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

The proof variable thus gets the type on lines 3-6. We can normalize

this using `pm-reduce`[4] to just `envs_entails Δ' Q` as long as the name was not already used.

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.

```
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)).
```
Elpi

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 can execute them depending on our intro patterns.

```
pred do-iIntros i:(list intro_pat),
                i:ihole, i:(ihole -> prop).
do-iIntros [] IH C :- !, C IH.
do-iIntros [iFresh | IPS] IH C :- !,
```
Elpi

[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.

```
5   do-iIntro-anon IH IH', !,                                    Elpi
6   do-iIntros IPS IH' C.
7 do-iIntros [iPure (some X) | IPS] (ihole N H) C :-
8   do-iForallIntro H H',
9   do-intro X H
10    (h\ sigma IntroProof\ sigma IntroType\
11        sigma NormType\
12        h = hole IntroType IntroProof,
13        pm_reduce IntroType NormType, !,
14        do-iIntros IPS
15                   (ihole N (hole NormType IntroProof))
16                   C
17    ).
18 do-iIntros [iList IPS | IPSS] (ihole N H) C :- !,
19   do-iIntro-anon (ihole N H) IH, !,
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, 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 fixed that issue, we can 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 in to 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.

```
1 solve (goal _ _ Type Proof [str Args]) GS :-                    Elpi
2   tokenize Args T, !,
3   parse_ipl T IPS, !,
4   do-iStartProof (hole Type Proof) IH, !,
5   do-iIntros IPS IH (ih\ set-ctx-count-proof ih _), !,
6   coq.ltac.collect-goals Proof GL SG,
7   all (open pm-reduce-goal) GL GL',
8   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 en 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 normal 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.

Structure of `eiInd` The `eiInd` command consists of several steps we have outlined in figure 5.1. Each of these steps are explained in the sections referenced in the diagram. TODO: Insert diagram

Inductive tactics In the last two sections we discuss how the tactics to use an inductive predicate are made. We first discuss the `eiInduction` tactic in section 5.8, which performs induction on the specified inductive predicate. Next, in section 5.9, we outline the extensions to the `eiIntros` tactic concerning inductive predicates.

5.1 Parsing inductive data structure

The `eiInd` command is called by writing a Coq inductive statement and prepending it with the `eiInd` command. We will use the below inductive statement as an example for this and any subsequent sections.

```Coq
eiInd
Inductive is_R_list {A} (R : val → A → iProp) :
                    val → list A → iProp :=
  | empty_is_R_list : is_R_list R NONEV []
  | cons_is_R_list l v tl x xs :
      l ↦ (v,tl) -∗ R v x -∗ is_R_list R tl xs -∗
      is_R_list R (SOMEV #l) (x :: xs).
```

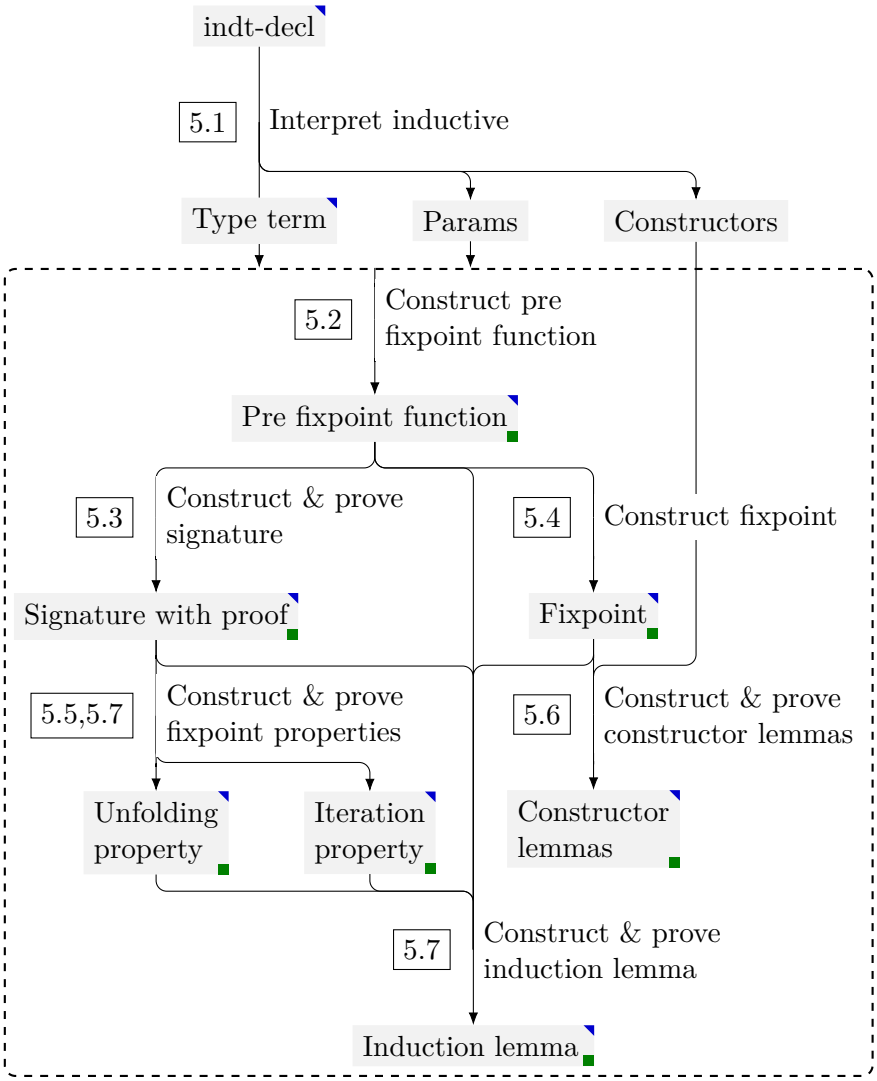


Figure 5.1: The structure of the `eiInd` command. Arrows are steps in the command and boxes are the objects that are being created. If a box has a `green box`, it is defined in Coq. If a box has a `blue triangle`, it is stored in the Elpi database. All arrows reference the section in which they are explained.

This inductive predicate relates a linked list to a Coq list by relation `R`. Since the Coq list can have an arbitrary type `A` and the predicate `R` is constant, we add them as parameters to the predicate.

In Elpi this is received as a data structure of type `indt-decl`.

Question: Should this be a more detailed explanation?

```
1  parameter `A` maximal                                          Elpi
2    TYPE
3    (a \ parameter `R` explicit
4          {{ val -> lp:a -> iProp }}
5          (r \ inductive `is_R_list` tt
6                (arity {{ val -> list lp:a -> iProp }})
7                (f \ [constructor `empty_is_R_list`
8                        (arity …),
9                      constructor `cons_is_R_list`
10                       (arity …)])))
```

The contents of the constructors is removed from this example for conciseness. A parameter consists of the name, if it should be maximally inserted, the type, and a function that given a binder returns the rest of the term. The first parameter has the name `` `A` ``, is maximally inserted, and the type is not yet calculated, thus a variable. The second parameter `R` depends on the first parameter in its type, `{{ val -> lp:a -> iProp }}`.

The first step of `eiInd` is interpreting the inductive into usable terms. This interpretation is done by descending into the binders. During the recursive steps, we keep track of the binders, together with the name and type of each binder. Later, whenever we construct a term we use this list of parameters to abstract the term over the parameters.

The inductive starts on line 5. It consists of its name, `` `is_R_list` ``, if its inductive, `tt`, its type, and a function containing the constructors. The next step in `eiInd` is normalizing the type to a term from a term with its own parameters. A lot of further steps are based on this term containing the type of the inductive predicate, called the *type term* from now on.

We now a have the list of parameters, the type of the inductive predicate and a list of the constructors under a binder. These are the ingredients needed to construct any further terms and proofs.

Question: This is a bit of a cluttered, structure. But how to add more structure?

5.2 Constructing the pre fixpoint function

In this step we want to transform the inductive definition of `is_R_list` into the function we take a fixpoint of called the pre fixpoint function. This step results in a function of the form of $\mathsf{isMLL}_\mathsf{F}$ in example 3.5. From the previous step we got a function producing the two constructors.

```
1  f \ [                                                          Elpi
2    {{ lp:f (InjLV #()) [] }},
3    {{ l v tl x xs : l ↦ (v, tl) -* R v x -* lp:f tl xs
4                    -* lp:f (InjRV #l) (x :: xs) }}
5  ]
```

Note that the parameter arguments to `is_R_list'` are included in the function `f`. The first step is to transform any Coq binders in a constructor, such as `l'`, `v'` or `tl'` into Iris existential quantifiers. Next, the magic wands in the constructors are replaced with separating conjunctions. Lastly, we connect all constructors using Iris disjunctions. This results in the following intermediate Elpi function.

```
Elpi
Body = f \ {{
    lp:f (InjLV #()) []
  ∨ ∃ l v tl x xs,
      l ↦ (v, tl) ∗ R v x ∗ lp:f tl xs ∗
      lp:f (InjRV #l) (x :: xs)
}}
```

Note that no types have been calculated at this point, as can be seen by the absence of types in the existential quantifiers. The next step is to transform this Elpi function into a term containing a Coq function over `f` and the two arguments of the inductive predicate. The type is transformed into a function by replacing the `prod` constructor with the `fun` constructor in the type term. The function binders are now created as follows, where `TypeTerm` is the term containing the type of the inductive predicate and `FunTerm` is that type transformed into a function. Thus, `FunTerm` has type `term -> term`, it takes a term and inserts that term as the body of the function.

```
Elpi
F' = (fun `F` TypeTerm (f \ FunTerm (Body f)))
```

We create a Coq function that takes the recursive argument, `F'`. The body of the function is the previously created `FunTerm`. We fill the body with the created `Body`, where we fill in the recursive argument.

Lastly, we need to replace the final recursive call of every constructor with equalities. They relate the arguments of function to the values used in the constructor. They are created by descending into the functions and keeping track of the binders. By recursively descending into each constructor and always taking the right side of a separating conjunction we can find the last recursive call. We then replace it with an equality for each of its arguments.

If necessary, we abstract the created term over the parameters of the inductive. We type check this term and get the following pre fixpoint function.

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

```
        l ↦ (v, tl) ∗ R v x ∗ F tl xs ∗
        ⌜H = InjRV #l⌝ ∗ ⌜H0 = x :: xs⌝
```
Coq

This Coq term is defined as `is_R_list_pre`.

5.3 Creating and proving proper signatures

In this section we describe how a proper is created and proven for the previously defined function. This 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.

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

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.

```
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).
```
Coq

Creating a signature Using these Coq definitions we transform the type into an `IProper`. A Proper relation for a function as described above 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
∀ (A : Type) (R : val → A → iProp),
  IProper (□> .> .> bi_wand ==> .> .> bi_wand)
          (is_R_list_pre A R)
```

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 this argument. The amount 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 every possibility in the proof search algorithm. The rule highlighted here applies 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 for this rule.

Next, on line 3, we first create a list of integers from 1 till the length of the arguments of 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 don't want to backtrack into the `std.exists` when something goes wrong in the rest of the algorithm, 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.

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

5.4 Constructing the fixpoint

`eiInd` generates the fixpoint as defined in section 3.3. The fixpoint is generated by recursing through the type term multiple times using the ideas of the previous sections. Afterwards we abstract over the parameters of the inductive. This results in creating the following fixpoint statement defined as `is_R_list`.

```Coq
λ (A : Type) (R : val → A → iProp)
  (v : val) (l : list A),
  (∀ F : val → list A → iProp,
    □ (∀ (v' : val) (l' : list A),
          is_R_list_pre A R F v' l' -* F v' l')
    -* F v l)
```

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. Thus, to store the pre fixpoint function of `is_R_list` in the database we add the rule:

```
inductive-pre (const «is_R_list»)                             Elpi
              (const «is_R_list_pre»)
```

Where instead of `«···»` we insert the variable containing the actual reference. We store the references to any objects we create after any of the previous of following steps. We also include some extra information in some rules, like `inductive-unfold` includes the amount 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.

5.5 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 flipped or replace it with a double entailment.

```
∀ (A : Type) (R : val → A → iProp)                            Coq
  (v : val) (l : list A),
    is_R_list_pre A R (is_R_list A R) v l
  ⊢ is_R_list A R v l
```

The proof term is generated by chaining proof generators such that no holes exist in the proof term.

```
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 amount of parameters, `Ps`, the amount 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, thus we use CPS to continue the proof in the predicate `mk-unfold.r->l.1`.

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

```
"HI" : ∀ (v : val) (l : list A),                                     Coq
       is_R_list_pre A R F v l -∗ F v l
--------------------------------------------------□
"HF" : is_R_list_pre A R (is_R_list A R) l' v'
--------------------------------------------------∗
is_R_list_pre A R F l' v'
```

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

```
pred mk-unfold-2.proof-2 i:int, i:int,                    Elpi
                         i:term, i:term, i:ihole.
mk-unfold-2.proof-2 Ps N Proper Mono IH :-
  ((copy {{ @IProper }} {{ @iProper }} :- !) =>
    copy Proper IProper'),
  type-to-fun IProper' IProper,
  std.map {std.iota Ps} (x\r\ r = {{ _ }}) Holes, !,
  do-iApplyLem (app [IProper | Holes]) IH [
    (h\ sigma PType\ sigma PProof\
        sigma List\ sigma Holes2\ !,
      h = hole PType PProof,
      std.iota Ps List,
      std.map List (x\r\ r = {{ _ }}) Holes2,
      coq.elaborate-skeleton (app [Mono | Holes2])
                             PType PProof ok,
  )] [IH1, IH2],
  do-iApplyHyp "HF" IH2 [], !,
  std.map {std.iota N} (x\r\ r = iPure none) Pures, !,
  do-iIntros
    {std.append [iModalIntro | Pures]
                [iIdent (iNamed "H"), iHyp "H",
                 iModalIntro, iHyp "HI"]}
    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 the following statement we can apply to the goal.

```
(λ (A : Type) (R : val → A → iProp),                      Coq
  iProper (□> .> .> bi_wand ==> .> .> bi_wand)
          (is_R_list_pre A R)
) _ _
```

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.

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

[1] This section of code can't make use 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 solve this we define any temporary variables ourselves using the `sigma X\` connective.

18-23 solve the following goal using another instance of the `do-iIntros` proof generator.

```
"HI" : ∀ (H : val) (H0 : list A),
        is_R_list_pre A R a H H0 -∗ a H H0
--------------------------------------------□
(□> .> .> bi_wand)%i_signature (is_R_list A R) a
```

Thus proving the right to left unfolding property. This proof together with the other two proofs of this section are defined as `is_R_list_unfold_1`, `is_R_list_unfold_2` and `is_R_list_unfold`.

5.6 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.

```
∀ (A : Type) (R : val → A → iProp)
  (v : val) (l : list A),
  ⌜v = InjLV #()⌝ ∗ ⌜l = []⌝ -∗ is_R_list A R v l

∀ (A : Type) (R : val → A → iProp)
  (v : val) (l : list A),
     (∃ l' (v' tl : val) (x : A) (xs : list A),
        l' ↦ (v', tl) ∗ R v' x ∗ is_R_list A R tl xs ∗
        ⌜v = InjRV #l'⌝ ∗ ⌜l = x :: xs⌝)
  -∗ is_R_list A R v l
```

Both constructor lemmas are an wand of the associated constructor to the fixpoint. They are defined with the name of their respective constructor, `empty_is_R_list` and `cons_is_R_list`.

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

5.7 Iteration and induction lemmas

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

```
∀ (A : Type) (R : val → A → iProp)
  (Φ : val → list A → iProp),
  □ (∀ (H : val) (H0 : list A),
       is_R_list_pre A R Φ H H0 -∗ Φ H H0) -∗
  ∀ (H : val) (H0 : list A),
    is_R_list A R H H0 -∗ Φ H H0
```

The induction lemma that we prove is:

```
∀ (A : Type) (R : val → A → iProp)
  (Φ : val → list A → iProp),
  □ (∀ (H : val) (H0 : list A),
      is_R_list_pre A R
        (λ (H1 : val) (H2 : list A),
           Φ H1 H2 ∧ is_R_list A R H1 H2) H
        H0 -∗ Φ H H0) -∗
  ∀ (H : val) (H0 : list A),
    is_R_list A R H H0 -∗ Φ H H0
```

Coq

These both mirror the iteration property and induction lemma from section 3.3. They are defined as `is_R_list_iter` and `is_R_list_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?

5.8 `eiInductive` 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.

Question: I switch example which is not so nice, but I don't know how to properly do fix this

Example 5.1

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

```
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.
```

Coq

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

```
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 }]
```
Coq

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

```
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 }]
```
Coq

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.

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

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.

5.9 `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 add 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.

Chapter 6

Related work

- Knaster Tarksi fixpoint theorem
- Generalized rewriting. They backtrack in their proof search, we don't

Chapter 7

Conclusion

7.1 Application

7.2 Evaluation of Elpi

7.3 Future work

Bibliography

[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.

[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.

[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.

[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.

[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+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.

[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.

[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.

[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.

[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.