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:
...
...

April 10, 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, **let** $x = e$ **in** e', using lambda expressions $(\lambda x.\, e')(e)$. Let statements with tuples as binder are defined using combinations of **fst** and **snd**. Expression sequencing is written as $e; e'$, this is defined as **let** $\_ = e$ **in** e'. The keywords **none** and **some** are just **inl** and **inr** respectively, both in values

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

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, **if** e_1 **then** e_2 **else false**. Lastly, when writing named functions, they are defined as names for anonymous functions.

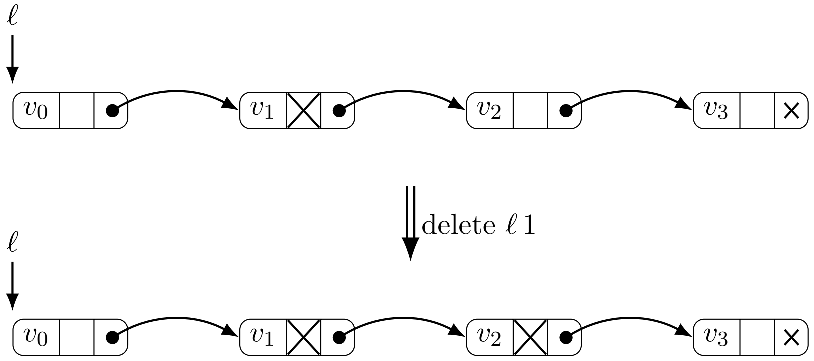
Our running example deletes an index out of a list by marking that node, logically deleting it.

```
delete hd i = match hd with
                none   ⇒ ()
              | some ℓ ⇒ let (x, mark, tl) = ! ℓ in
                         if mark = false && i = 0 then
                           ℓ ← (x, true, tl)
                         else if mark = false then
                           delete tl (i − 1)
                         else
                           delete tl i
              end
```

The example is a recursive function called delete, the function has two arguments. HeapLang has no null pointers, thus we wrap a pointer in **none**, the null pointer, **some** ℓ, a non-null pointer pointing to ℓ. The first argument *hd* is either a null pointer, for the empty list, or a pointer to an MLL. The second argument, i, is the index in the MLL to delete. The first step this recursive function taken is checking whether we are deleting from the empty list. To 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

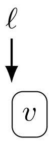
[IO01; Rey02], together with higher order connectives, persistent propositions and weakest preconditions as introduced by Iris. This logic is presented below, starting with the syntax.

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

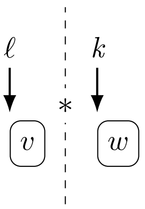
Separation logic contains all the usual higher order predicate logic connectives as seen on the first line. The symbol τ, represents any type we have seen, including *iProp* itself. The second row contains separation logic specific connectives. The *pure* connective, $\ulcorner \phi \urcorner$, embeds 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 $P \ast P$, are discussed in this section. The last three connectives, $P \mathrel{-\!\!\ast} P$, $\Box\, P$ 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, $P \ast Q$. For $P \ast Q$ 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.



$\ell \mapsto v \ast k \mapsto w$

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.

$$\begin{aligned} \mathsf{True} * P &\dashv\vdash P \\ P * Q &\vdash Q * P \\ (P * Q) * R &\vdash P * (Q * R) \end{aligned} \qquad \begin{array}{c} \scriptstyle *\text{-MONO} \\ \dfrac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \end{array}$$

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.

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

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]

We have two categories of such rules, rules for the language constructs, such as WP-STORE, and rules for reasoning about the structure of the language and the weakest precondition.

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, we can evaluate the postcondition.
- The rule WP-MONO allows for changing the postcondition as long as this change holds for any value.
- The rule WP-FRAME allows for adding any propositions we have as assumption into the postcondition of a weakest precondition we have as assumption.
- The rule WP-BIND allows for extracting the expressions in the head position of a program. This is done by wrapping the head expression in a context as defined at the bottom of figure 2.2. The verification of

the rest of the program is delayed by moving it into the postcondition of the head expression.

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

General rules.

$$\textsc{wp-value}\quad \Phi(v) \vdash \mathsf{wp}\ v\ [\Phi] \qquad\qquad \textsc{wp-mono}\quad \frac{\forall v.\Phi(v) \vdash \Psi(v)}{\mathsf{wp}\ e\ [\Phi] \vdash \mathsf{wp}\ e\ [\Psi]}$$

$$\textsc{wp-frame}\quad Q * \mathsf{wp}\ e\ [x.\ P] \vdash \mathsf{wp}\ e\ [x.\ Q * P] \qquad\qquad \textsc{wp-bind}\quad \mathsf{wp}\ e\ [x.\ \mathsf{wp}\ K[x]\ [\Phi]] \vdash \mathsf{wp}\ K[e]\ [\Phi]$$

Rules for basic language constructs.

$$\textsc{wp-alloc}\quad \frac{}{\forall \ell.\ \ell \mapsto v \mathrel{-\!\!*} \Phi(\ell) \vdash \mathsf{wp}\ \mathbf{ref}(v)\ [\Phi]} \qquad\qquad \textsc{wp-load}\quad \frac{}{\ell \mapsto v * \ell \mapsto v \mathrel{-\!\!*} \Phi(v) \vdash \mathsf{wp}\ !\,\ell\ [\Phi]}$$

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

Pure reductions.

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

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

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

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

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

Context rules

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

Figure 2.2: Rules for the weakest precondition assertion.

2.4 Persistent propositions and nested hoare triples

Question: I am using persistent modality and persistent proposition through each other in the section, is that bad?

In this section we define Hoare triples using the weakest precondition and in the process explain persistent propositions. We end with an example showing why hoare triples are persistent and a verification of this example.

Hoare-def

$$[P]\, e\, [\Phi] \triangleq \square(P \mathrel{-\!\!*} \mathsf{wp}\; e\, [\Phi])$$

This definition is very similar to how we used weakest preconditions with a precondition. However, we wrap our the weakest precondition with precondition in a persistence modality, $\square$.

Persistent propositions A proposition in a persistence modality has the intuitive semantics that once it holds, it will always hold. Thus, a persistent proposition can be duplicated, as can be seen in the rule $\square$-dup below. To prove a statement is persistent, thus that $\square\, P$ holds, we are only allowed to have persistent proposition in our assumptions, as can be seen in the rule $\square$-mono below.

$\square$-dup

$$\square\, P \dashv\vdash \square\, P * \square\, P$$

$\square$-sep

$$\square\,(P * Q) \dashv\vdash \square\, P * \square\, Q$$

$\square$-mono

$$\frac{P \vdash Q}{\square\, P \vdash \square\, Q}$$

$\square$-E

$$\square\, P \vdash P$$

$\square$-conj

$$\square\, P \wedge Q \vdash \square\, P * Q$$

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

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

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

$\square$-I

$$\frac{\square\, P \vdash Q}{\square\, P \vdash \square\, Q}$$

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

Nested Hoare triples In HeapLang we are allowed to store closures on the heap, thus creating a higher order heap. When we store a closure in memory we can use it multiple times and thus might need to duplicate the specification of the closure multiple times. This is the reason Hoare triples are persistent. Take the following example with its specification.

$$\text{refadd} := \lambda n.\, \lambda \ell.\, \ell \leftarrow\, !\, \ell + n$$

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

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.

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

Given that we need to prove a weakest precondition of an expression in a context, and we have 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.

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.

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

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

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

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.

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

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

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

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.

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

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

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

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.

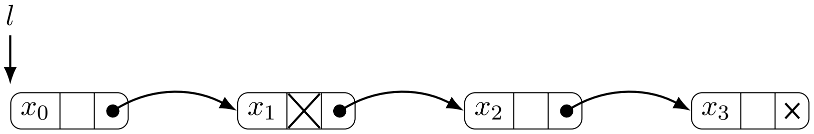
$$\mathsf{wp}\ !\,\ell\, [20.\, \mathbf{true}]$$

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 a logic and reason about it together with the program. However, this does not easily scale to recursive 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 made, 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 $[x_0, x_1, x_2]$. This is expressed as, $\mathsf{isMLL}\,(\texttt{some}\,\ell)\,[x_0, x_2, x_3]$.

To illustrate how isMLL works we give the below inductive predicate. This is not a valid definition for isMLL for the rest of this thesis as is made clear in chapter 3 However, it serves as an explanation in this chapter.

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

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

- The hd is $\texttt{none}$ 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.

$$\textsf{isMLL-}\textsc{ind}$$

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

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

Case Empty MLL: This is the base case, we have to prove Φ with $\texttt{none}$ 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.

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

Lemma 2.2

For any index $i \geq 0$, list $\vec{v}$ of values and $hd \in Val$,

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

Proof. We first use the definition of a Hoare triple, Hoare-def, to create the associated weakest precondition.

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

Since we have only persistent 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:

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

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 $\ell \mapsto (v, \texttt{true}, tl)$ to get our new statement that we need to prove:

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

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

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

Which is the left-hand side of our IH.

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

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

And, we need to prove that:

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

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

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

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

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

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

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

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

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

- We will show how one can apply the Tarski Fixpoint theorem to create an inductive predicate and how we can create the induction principle from it.

3.1 Problem statement

- The logic described here is embedded in Coq.
- We are only allowed to do structural recursion
- The recursive formulation of $\mathsf{isMLL}$is not structurally recursive
- We need another way to define this predicate
- Iris already has a way of defining fixpoints that would be applicable
- Least fixpoints
- Inspired by the Tarski Fixpoint theorem on lattices and ?
-

3.2 Least fixpoint in Iris

Definition 3.1 (*Monotone predicate*)

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

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

- Note that there would have been a similar way we could have written the property of a monotone predicate.

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

- This would be more inline with the way they are written in chapter 2
- However, these rules are a lot more strict in what the context is in which they are used, thus making them a lot harder to use.
- Also, it is the way they are written and used in Iris
- We thus write these like in the definition from now on
- Using this definition of monotone we can define the least fixpoint theorem.

Theorem 3.2 (*Least fixpoint*)

Given a monotone predicate $\mathsf{F}\colon (A \to iProp) \to A \to iProp$, there exists a least fixpoint $\mu\mathsf{F}\colon A \to iProp$ such that

1.

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

2.

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

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

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

We now prove the two properties of the least fixpoint

1. We start with proving this right to left, then using the result, prove left to right.

   R-L We first unfold the definition of $\mu\mathsf{F}\,x$.

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

   Next we introduce $\mathit{\Phi}$ and the wand.

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

   We now apply $\Box(\forall y.\, \mathsf{F}\,\mathit{\Phi}\,y \mathrel{-\!\!*} \mathit{\Phi}\,y)$ to $\mathit{\Phi}\,x$.

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

We can now use the monotonicity of F with the assumption $\mathsf{F}\,\mu\mathsf{F}\,x$

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

After unfolding the definition of $\mu\,x$ and introducing the wand we get

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

This statement holds by application of the first assumption.

L-R We again first unfold the definition of $\mu\mathsf{F}\,x$.

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

We apply the assumption with $\Phi = \mathsf{F}\,\mu\mathsf{F}$ resulting in the following statement after introductions

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

This holds because of monotonicity of F and the above proved property.

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

□

- The second property of the least fixpoint is the normal induction property.
- However, it is often useful to make it stronger

Lemma 3.3 (*least fixpoint strong induction principle*)

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

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

- We now show how this can be applied to create the isMLLpredicate

Example 3.4 (*Iris least fixpoint of* isMLL)

- We want to transform the non-structurally recursive definition of isMLLinto a least fixpoint

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

- We start by ?ing any recursive calls in the definition in order to create a functor?

$$\mathsf{isMLL_F}\,\Phi\,hd\,\vec{v} \triangleq \begin{array}{ll} & hd = \mathbf{none} * \vec{v} = [] \\ \vee & \exists \ell, v', tl.\, hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{true}, tl) * \Phi\, tl\, \vec{v} \\ \vee & \exists \ell, v', \vec{v}'', tl.\, hd = \mathbf{some}\, l * l \mapsto (v', \mathbf{false}, tl) * \\ & \vec{v} = [v'] + \vec{v}'' * \Phi\, tl\, \vec{v}'' \end{array}$$

- Predicate $\mathsf{isMLL_F}$ now has type $(Val \to \overrightarrow{Val} \to iProp) \to Val \to \overrightarrow{Val} \to iProp$

- However, the least fixpoint only works for functors of type $(A \to iProp) \to A \to iProp$

- We solve this by currying $\mathsf{isMLL_F}$ into $\mathsf{isMLL'_F}$: $((Val, \overrightarrow{Val}) \to iProp) \to (Val, \overrightarrow{Val}) \to iProp$

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

- In order to apply the fixpoint theorem, we need $\mathsf{isMLL'_F}$ to be monotone

Proof. To prove $\mathsf{isMLL'_F}$ is monotone, we need the following to hold.

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

We can apply the definition of $\mathsf{isMLL'_F}$, introduce the wands, eliminate the disjunctions on the left and introduce the matching disjunctions on the right in order to get three statements to prove.

Empty MLL: We need to prove

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

This holds trivially

Marked head: We first eliminate any existentials on the left and introduce them using the gained variables on the right. We now need to prove

$$\begin{array}{l} \square(\forall (hd, \vec{v}).\, \Phi\,(hd, \vec{v}) \mathrel{-\!\!*} \Psi\,(hd, \vec{v})) * \\ hd = \mathbf{some}\, \ell * \ell \mapsto (v', \mathbf{true}, tl) * \Phi\, tl\, \vec{v} \end{array} \vdash hd = \mathbf{some}\, \ell * \ell \mapsto (v', \mathbf{true}, tl) * \Psi\, tl\, \vec{v}$$

The propositions that don't include Φ or Ψ cancel each other out, and we are left with the following.

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

This holds trivially using □-E.

Unmarked head: We follow the same prove steps as in the marked head case. □

- Given that $\mathsf{isMLL}'_{\mathsf{F}}$ is monotone, we now know from theorem 3.2 that the least fixpoint exists of $\mathsf{isMLL}'_{\mathsf{F}}$
- We can now define $\mathsf{isMLL}'_{\mathsf{F}}$ as

$$\begin{aligned}\mathsf{isMLL}'\,(hd, \vec{v}) &\triangleq \mu(\mathsf{isMLL}'_{\mathsf{F}})\,(hd, \vec{v})\\ &= \forall \mathit{\Phi}.\ \square(\forall y.\ \mathsf{isMLL}'_{\mathsf{F}}\,\mathit{\Phi}\,y \mathrel{-\!\!*} \mathit{\Phi}\,y) \mathrel{-\!\!*} \mathit{\Phi}\,x\end{aligned}$$

- To finish the definition of isMLLwe uncurry the created fixpoint

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

Question: Also highlight the strong induction already or not?

3.3 Changing arities

- We modify the definitions as described in Iris to allow for multiple arity functors.
- The first step in automating creation of fixpoints is to deal with predicates with more than one argument
- In example 3.4 we solved this by currying the predicate before taking the fixpoint
- When automating the process we solved this somewhat differently
- We change the definitions of and theorems used to match the arity of the predicate we want to take the fixpoint of

Definition 3.5 (*Monotone predicate*)

For any $n \in \mathbb{N}$, predicate $\mathsf{F}\colon (A_1 \to \cdots \to A_n \to \mathit{iProp}) \to A_1 \to \cdots \to A_n \to \mathit{iProp}$ is monotone when for any $\mathit{\Phi}, \Psi\colon A_1 \to \cdots \to A_n \to \mathit{iProp}$, it holds that

$$\vdash \begin{array}{l}\square(\forall x_1, \ldots, x_n.\, \mathit{\Phi}\,x_1\,\ldots\,x_n \mathrel{-\!\!*} \Psi\,x_1\,\ldots\,x_n) \mathrel{-\!\!*}\\ \forall x_1, \ldots, x_n.\,\mathsf{F}\,\mathit{\Phi}\,x_1\,\ldots\,x_n \mathrel{-\!\!*} \mathsf{F}\,\Psi\,x_1\,\ldots\,x_n\end{array}$$

- This definition also applies for $n = 0$

- For example, we can prove the separating conjunction monotone in both its arguments

Lemma 3.6 (*Seperation conjuction is monotone*)

The separation conjunction is monotone in its left and right argument.

Proof. We only prove monotonicity in its left argument, the proof for the right side is identical. We thus need to prove $\Phi_R P = P * R$ is monotone. expanding the definition of monotone for arity one we get the following statement.

$$\vdash \Box(P \mathrel{-\!\!*} Q) \mathrel{-\!\!*} P * R \mathrel{-\!\!*} Q * R$$

We introduce the wands and persistence modalities giving us the assumptions, $P \mathrel{-\!\!*} Q$, P and R. We then use $*$-MONO using the first two assumptions for proving P and using the last assumption for proving R. That $P \mathrel{-\!\!*} Q * P \vdash Q$ holds follows from $\mathrel{-\!\!*}$I-E, and $R \vdash R$ holds directly. □

- In the same way we also modify the least fixpoint theorem

Theorem 3.7 (*Least fixpoint*)

Given an $n \in \mathbb{N}$ and a monotone predicate $\mathsf{F} \colon (A_1 \to \cdots \to A_n \to iProp) \to A_1 \to \cdots \to A_n \to iProp$, there exists a least fixpoint $\mu\mathsf{F} \colon A_1 \to \cdots \to A_n \to iProp$ such that

1.

$$\mu\mathsf{F}\, x_1 \,\ldots\, x_n \dashv\vdash \mathsf{F}\,(\mu\mathsf{F})\, x_1 \,\ldots\, x_n$$

2.

$$\vdash \begin{array}{l} \Box(\forall y_1, \ldots, y_n.\, \mathsf{F}\,\Phi\, y_1 \,\ldots\, y_n \mathrel{-\!\!*} \Phi\, y_1 \,\ldots\, y_n) \mathrel{-\!\!*} \\ \forall y_1, \ldots, y_n.\, \mu\mathsf{F}\, x_1 \,\ldots\, x_n \mathrel{-\!\!*} \Phi\, x_1 \,\ldots\, x_n \end{array}$$

- The proof follows the same steps as the proof for theorem 3.2

3.4 Monotone proof search

- We create a system for syntactically finding proofs of monotonicity
- Based on generalized rewriting system in coq by Sozeau [Soz09].
- Define monotonicity of connectives in separation logic using proper elements of relations

TODO: This is not sufficient but stuck on it

Definition 3.8 (*Proper element of a relation*)

Given a relation $R \colon A \to A \to iProp$ and an element $x \in A$, x is a proper element of R if $R\,x\,x$

- When the relation is reflexive, all possible elements are Proper
- For example if we take the magic wand as relation, all propositions are proper.
-

Definition 3.9 (***Respectful relation***)

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

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

Definition 3.10 (***Persistent relation***)

The persistent relation $\boldsymbol{\Box} R \colon A \to A \to \mathit{iProp}$ for a relation $R \colon A \to A \to \mathit{iProp}$ is defined as

$$\boldsymbol{\Box} R \triangleq \lambda x, y.\ \Box(R\,x\,y)$$

- We can rewrite lemma 3.6 using the relations we described above

Lemma 3.11 (***Separating conjuction monotone***)

The separating conjunction is a proper element of the relation

$$\boldsymbol{\Box}{\mathrel{-\!\!*}} \Longrightarrow \boldsymbol{\Box}{\mathrel{-\!\!*}} \Longrightarrow {\mathrel{-\!\!*}}$$

- Writing out the above statement gives

$$\vdash \forall P, Q.\ \Box(P \mathrel{-\!\!*} Q) \mathrel{-\!\!*} \forall P', Q'.\ \Box(P' \mathrel{-\!\!*} Q') \mathrel{-\!\!*} P * Q \mathrel{-\!\!*} P' * Q'$$

- This is monotonicity on the left and right side of the separating conjunction at the same time

Definition 3.12 (***Pointwise relation***)

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

$$\gtrdot R \triangleq (= \Longrightarrow R)$$

Lemma 3.13 (***Existential quantification monotone***)

The existential quantification is a proper element of the relation

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

Example 3.14 ($\mathsf{isMLL_F}$ *is monotone*)

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

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

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

Proof. We assume $\Box\,(\forall hd\,\vec{v}.\,\mathit{\Phi}\,hd\,\vec{v} \mathrel{-\!\!*} \Psi\,hd\,\vec{v})$ holds and for arbitrary hd and $\vec{v}$, $\mathsf{isMLL_F}\,\mathit{\Phi}\,hd\,\vec{v}$ holds. After applying the definition of $\mathsf{isMLL_F}$ we need to prove

$$\mathsf{isMLL_F}\,\Psi\,hd\,\vec{v}$$

□

Chapter 4

Implementing an Iris tactic in Elpi

In this chapter we will show how Elpi together with Coq-Elpi can be used to create new tactics. We will do this by giving a tutorial on how to implement the `iIntros` tactic from Iris.

4.1 `iIntros` example

The tactic `iIntros` is based on the Coq **`intros`** tactic. The Coq **`intros`** tactic makes use of a domain specific language (DSL) for quickly introducing different logical connective. In Iris this concept was adopted for the `iIntros` tactic, but modified to the Iris contexts. Also, a few expansions, as inspired by ssreflect [HKP97; GMT16], were added to perform other common initial proof steps such as **`simpl`**, **`done`** and others. We will show a few examples of how `iIntros` can be used to help prove lemmas.

We have seen in chapter 2 how we often have two types of propositions as our assumptions during a proof. There are persistent and non-persistent (also called spatial from now on) proposition. In Coq assumption management is a very important part of writing proofs. Thus, in Coq implementation of the separation logic Iris, theses two types of assumptions have been made into two contexts, the persistent and the spatial context. Together with the Coq context, we thus have three context. As an example given we have the separation logic statement.

$$\Box P * Q \vdash R$$

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

```
P, Q, R: iProp
============
"HP" : P
```

```
------------□
"HR" : Q
------------∗
R
```

Above the double lined line we have the types of all our proof variables and any other statements in the Coq logic. Next we have a section of persistent proposition we have as assumptions, each one named. The assumption P is thus named `"HP"`. Following the persistent context we have the spatial context, where again each assumption is named. At the bottom we have the statement we want to prove. We will now show how the `iIntros` tactic modifies these contexts. Given the below proof state, we would want to introduce P and Q.

```
P, Q: iProp
============
------------∗
P -∗ Q -∗ P
```

We can use `iIntros "HP HQ"`, this will intelligently apply −∗I-E twice.

```
P, Q: iProp
============
"HP" : P
"HQ" : Q
------------∗
P
```

We have introduced the two separation logic propositions into the spatial context. This does not only work on the magic wand, we can also use this to introduce more complicated statements. Take the following proof state,

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

It consists of a universal quantification, an existential quantification, a seperating conjunction and a disjunction. We can again use one application of `iIntros` to introduce and eliminate the premise.

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

When applied we get two proof states, one for each side of the disjunction elimination. These different proof states are shown with the `(1/2)` and `(2/2)` prefixes.

```
(1/2)
P: nat → iProp
x, y: nat
==================
"Hx" : P x
"Hy" : P y
------------------*
P 1

(2/2)
P: nat → iProp
x: nat
==================
"H0" : P 0
------------------*
P 1
```

The intro pattern consists of multiple sub intro patterns. Each sub intro pattern starts with a forall introduction or wand introduction. We then interpret the intro pattern for the introduced hypothesis. A few of the possible intro patterns are:

- `"H"` represents renaming a hypothesis. The name given is used as the name of the hypothesis in the spatial context.
- `"%H"` represents pure elimination. The introduced hypothesis is interpreted as a Coq hypothesis, and added to the Coq context.
- `"[IPL | IPR]"` represents disjunction elimination. We perform a disjunction elimination on the introduced hypothesis. Then, we apply the two included intro patterns two the two cases created by the disjunction elimination.
- `"[IPL IPR]"` represents separating conjunction elimination. We perform a separating conjunction elimination. Then, we apply the two included intro patterns two the two hypotheses by the separating conjunction elimination.
- `"[%x IP]"` represents existential elimination. If first element of a separating conjunction pattern is a pure elimination we first try to eliminate an exists in the hypothesis and apply the included intro pattern on the resulting hypothesis. If that does not succeed we do a conjunction elimination.

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

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

4.2 Contexts

- Iris uses a named context instead of the entailment
- `env A` is a list of pairs from identifiers to `A`.

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

Record envs (PROP : bi) := Envs {
  env_persistent : env PROP;
  env_spatial : env PROP;
  env_counter : positive;
}.
```

Question: Should I use *iProp* or `PROP` here?

Question: I am simplifying the environments here, should I do that?

- Identifiers are either anonymous or named
- Environments are maps from identifiers to values
- The final context is two environments of propositions and a counter
- The first environment is the persistent context
- The second environment is the spatial context
- The two environments can't have overlapping identifiers
- The counter is used to always be able to generate a fresh anonymous identifier
- Semantics of the contexts is

```
Definition of_envs {PROP : bi}
    (Γp Γs : env PROP) : PROP :=
  (□ [∧] Γp ∧ [∗] Γs)%I.
```

Question: Should I differentiate between □ and <pers>

- The persistent environment is combined with ∧ and surrounded by a □.
- The spatial environment is combined with ∗
- We can now write our entailment as

```
Definition envs_entails {PROP : bi}
    (Δ : envs PROP) (Q : PROP) : Prop :=
  of_envs (env_intuitionistic Δ) (env_spatial Δ) ⊢ Q.
```

- This is represented in the proof state as

```
P, Q, R: iProp
============
"HP" : P
------------□
"HR" : Q
------------∗
R
```

- P is a persistent proposition
- Q is a spatial proposition
- We need to proof R

4.3 Tactics

- The proof rules in chapter 2 are hard to use with the context
- Define Lemma's that work with the context
- We have already seen one, wp-apply
- These allow us to define our tactics easily

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

- Introduces a magic wand
- Add introduced proposition to the spatial context
- The condition we need to satisfy is a Coq function that resolves to Q with P added to the context
- If `i` already exists in the context, we have to proof False
- Tactics now just process the arguments and call necessary Lemma's

The tactics the IPM adds are build to replicate many of the behaviors of the Coq tactics while manipulating the Iris contexts. In the next section we will show how the Iris variant of the **`intros`** tactic works.

4.4 Elpi

We implement our tactic in the λProlog language Elpi [Dun+15; GCT19]. Elpi implements λprolog [MN86; Mil+91; BBR99; MN12] and adds constraint handling rules to it [Mon11]. constraint handling will be explained in Section ?.

TODO: Defer constraint handling to later

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

Our Elpi implementation `eiIntros` consists of three parts as seen in figure 4.1. The first two parts will interpret the DSL used to describe what we want to introduce. Then, the last part will apply the interpreted DSL. In section 4.5 we describe how a string is tokenized by the tokenizer. In section 4.6 we describe how a list of tokens is parsed into a list of intro patterns. In section 4.7 we describe how we use an intro pattern to introduce and eliminate the needed connectives. In every section we describe more parts of the Elpi programming language and the Coq-Elpi connector starting with the base concepts of the language and working up to the mayor concepts of Elpi and Coq-Elpi.

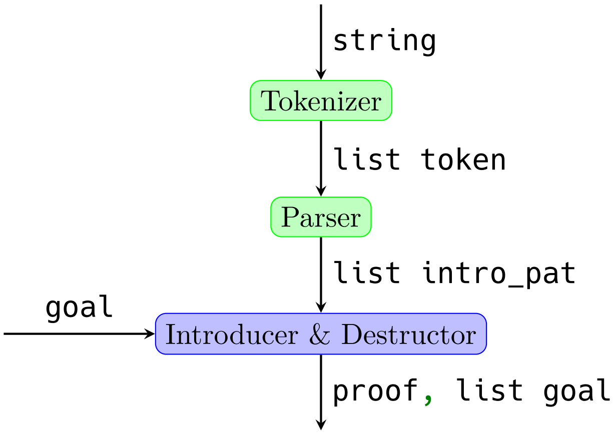


Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

4.5 Tokenizer

The tokenizer takes as input a string. We will interpret every symbol in the string and produce a list of tokens from this string. Thus, the first step is to define our tokens. Next we show how to define a predicate that transform our string into the tokens we defined.

4.5.1 Data types

We have separated the introduction patterns into several distinct tokens. Most tokens just represent one or two characters, but some tokens also contain some data associated with that token. For example `"H1"` is tokenized as the name token containing the string ”H1”.

```
kind token type.

type tAnon, tFrame, tBar, tBracketL, tBracketR, tAmp,
     tParenL, tParenR, tBraceL, tBraceR, tSimpl,
     tDone, tForall, tAll token.
type tName string -> token.
type tNat int -> token.
type tPure option string -> token.
type tArrow direction -> token.

kind direction type.
type left, right direction.
```

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

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

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

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

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

4.5.2 Predicates

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

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

Line 1 describes the type of the predicate. The keyword `pred` starts the definition of a predicate. Next we give the name of the predicate, ”tokenize”. Lastly, we give a list of arguments of our predicate. Each argument is marked as either `i:`, they act as an input or `o:`, they act as an output, in section 4.5.3 a more precise definition is given. In the only rule of our predicate, defined on line 2, we assign a variable to both of the arguments. `S` has type `string` and is bound to the first argument. `O` has type `list token` and is bound to the second argument. By calling predicates after the `:-` symbol we can define the relation between the arguments. The first predicate we call, `rex.split`, has the following type:

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

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

The next line, line 4, calls the recursive tokenizer, tokenizer.rec[1], on the list of split string and assigns the output to the output variable O.

The reason predicates in Elpi are called predicates and not functions, is that they don't always have to take an input and give an output. They can sometimes better be seen as predicates defining for which values of their arguments they hold. Each rule defines a list of predicates that need to hold for their premise to hold. Thus, a predicate can have multiple values for its output, as long as they hold for all contained rules. These multiple possible values can be reached by backtracking, which we will discuss in section 4.5.5. To execute a predicate, we thus find the first rule for which its premise is sufficient for the arguments we supply. We then check if each of the predicates in the conclusion hold starting at the top. If they hold, and we get a value for every output argument, we are done executing our predicate. How we determine when arguments are sufficient and what happens when a rule does not hold, we will discuss in the next two sections.

4.5.3 Matching and unification

The arguments of a predicate can be more than just a variable. We can supply a value containing variables and depending on the argument mode, input or output, we match or unify the input with the premise respectively.

tokenize.rec uses matching and unification to solve most cases.

```
pred tokenize.rec i:list string, o:list token.
tokenize.rec [] [] :- !.
tokenize.rec [" " | SL] TS :- !, tokenize.rec SL TS.
tokenize.rec ["$" | SL] [tFrame | TS] :- !,
  tokenize.rec SL TS.
tokenize.rec ["/", "/", "=" | SL] [tSimpl, tDone | TS] :- !,
  tokenize.rec SL TS.
tokenize.rec ["/", "/" | SL] [tDone | 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.

This predicate has several rules, we chose a few to highlight here. The first rule, on line 2, has a premise and a cut as its conclusion, we will discuss cuts in section 4.5.5, for now they can be ignored. This rule can be used when the first argument matches `[]` and if the second argument unifies with `[]`. The difference is that, for two values to match they must have the exact same constructors and can only contain variables in the same places in the value. Thus, the only valid value for the first argument of the first rule is `[]`. When unifying two values we allow a variable to be unified with a constructor, when this happens the variable will get assigned the value of the constructor. Thus, we can either pass `[]` to the second argument, or some variable `V`. After the execution of the rule the variable `V` will have the value `[]`.

The next four rules use the same principle. They use the list pattern `[E1, ..., En | TL]`, where `E1` to `En` are the first n values and `TL` is the rest of the list, to match on the first few elements of the list. We unify the output with a list starting with the token that corresponds to the string we match on. The tails of the input and output we pass to the recursive call of the predicate to solve.

When we encounter multiple rules that all match the arguments of a rule we try the first one first. The rules on line 6 and 8 would both match the value `["/", "/", "="]` as first argument. But, we interpret this use the rule on line 6 since it is before the rule on line 8. This results in our list of strings being tokenized as `[tSimpl, tDone]`.

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

Question: Don't know what to do with this, but is an interesting fact and shows the versatility, we might use it later.

4.5.4 Functional programming in Elpi

While our language is based on predicates we still often defer to a functional style of programming. The first language feature that is very useful for this goal is spilling. Spilling allows us to write the entry point of the tokenizer as defined in section 4.5.2 without the need of the temporary variable to pass the list of strings around.

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

We spill the output of a predicate into the input of another predicate by using the `{ }` syntax. We don't specify the last argument of the predicate and only the last argument of a predicate can be spilled. It is mostly equal to the previous version, but just written shorter. There is one caveat, but it will be discussed in ?.

TODO: Refer to relevant section

The second useful feature is how lambda expressions are first class citizens of the language. A pred statement is a wrapper around a constructor definition using the keyword type, where all arguments are in output mode. The following predicate is equal to the type definition below it.

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

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

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

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

4.5.5 Backtracking

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

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

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

The predicate contains two rules. The first rule, defined on lines 2 and 3, recurses as long as the input predicate, Pred holds for the input list, [X|XS]. The second rule returns the last part of the list as soon as Pred no longer holds.

The first rule destructs the input in its head X and its tail XS. It then checks if Pred holds for X, if it does, we continue the rule and call take-while-split on the tail while assigning X as the first element of the first output list and the output of the recursive call as the tail of the first output and the second output. However, if `Pred X` does not succeed we backtrack to the previous rule in our conclusion. Since there is no previous rule in the conclusion we instead undo any unification that has happened and try the next possible rule. This will be the rule on line 4 and returns the input as the second output of the predicate.

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

```
1  type tName string -> token.
2
3  tokenize.rec SL [tName S | TS] :-
4    take-while-split SL is-identifier S' SL',
5    { std.length S' } > 0, !,
6    std.string.concat "" S' S,
7    tokenize.rec SL' TS.
```

To tokenize a name we first call `take-while-split` with as predicate `is-identifier`, which checks if a string is valid identifier character, whether it is either a letter or one of a few symbols allowed in identifiers. It thus splits up the input string list into a list of string that is a valid identifier and the rest of the input. On line 5 we check if the length of the identifier is larger than 0. We do this by spilling the length of `S'` into the `>` predicate. Next, on line 6, we concatenate the list of strings into one string, which will be our name. And on line 7, we call the tokenizer on the rest of the input, to create the rest of our tokens.

If our length check does not succeed we backtrack to next rule that matches, which is

```
tokenize.rec XS _ :- !,
  coq.say "unrecognized tokens" XS, fail.
```

It prints an error messages saying that the input was not recognized as a valid token, after which it fails. The predicate thus does not succeed. There is one problem, if line 6 or 7 fails for some reason in the `tName` rule of the tokenizer, the current input starting at `X` is not unrecognized as we managed to find a token for the name at the start of the input. Thus, we don't want to backtrack to another rule of `tokenize.rec` when we have found a valid name token. This is where the cut symbol, `!`, comes in. It cuts the backtracking and makes certain that if we fail beyond that point we don't backtrack in this predicate.

If we take the following example

```
1 tokenize.rec ["H","^"] TS
2                  ⇓ calls
3 tokenize.rec ["^"] TS'
```

When evaluating this predicate we would first apply the name rule of the `tokenize.rec` predicate. This would unify `TS` with `[tName "H" | TS']` and call line 3, `tokenize.rec ["^"] TS'`. Every rule of `tokenize.rec` fails including the last fail rule. This rule does first print `"unrecognized tokens ^"` but then also fails. Now when executing the rule of line 1, we have failed on the last predicate of the rule. If there was no cut before it, we would backtrack to the fail rule and also print `"unrecognized tokens [H, ^]"`. But, because there is a cut we don't print the faulty error message. Thus, we only print meaningful error message when we fail to tokenize an input.

4.6 Parser

Question: Ik twijfel over dit hele stuk aangezien er niet zo veel nieuws in wordt uitgelegt dat nuttig is voor later, behalve pas op met backtracking.

- Describe sections of parser

Alternative for this section

- Parser uses many of the same techniques as tokenizer for parsing
- Not much to explain
- Implements a reductive descent parsing
- Minimize backtracking
- Look at code for full details

4.6.1 Data structure

- structured to be easily read to apply the intro pattern.

```
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.
```

- Tree structure?
- iList is combination of existential, disjunction and conjunction pattern

4.6.2 Reductive descent parsing

- We can translate a grammar directly into a parser
- Below, partial grammar for the intro patterns

⟨*intropattern_list*⟩ ::= ϵ
| ⟨*intropattern*⟩ ⟨*intropattern_list*⟩

⟨*intropattern*⟩ ::= ⟨*ident*⟩
| ‘`?`’ | ‘`/=`’ | ‘`//`’
| ‘`[`’ ⟨*intropattern_list*⟩ ‘`]`’
| ‘`(`’ ⟨*intropattern_conj_list*⟩ ‘`)`’

⟨*intropattern_list*⟩ ::= ϵ
| ⟨*intropattern*⟩ ‘`|`’ ⟨*intropattern_list*⟩
| ⟨*intropattern*⟩ ⟨*intropattern_list*⟩

⟨*intropattern_conj_list*⟩ ::= ϵ
| ⟨*intropattern*⟩ ‘&’ ⟨*intropattern_conj_list*⟩

- Explain structure of parser
- give example of anon, simpl and done
- Using tokenizer name has become the same

```
pred parse_ip i:list token, o:list token, o:intro_pat.
parse_ip [tAnon | TS] TS (iFresh) :- !.
parse_ip [tSimpl | TS] TS (iSimpl) :- !.
parse_ip [tDone | TS] TS (iDone) :- !.
parse_ip [tName X | TS] TS (iIdent (iNamed X)) :- !.
```

- Check after calling new parser that conditions for values hold
- Post process conj parser result

```
parse_ip [tBracketL | TS] TS' (iList L) :- !,
parse_ilist TS [tBracketR | TS'] L.
parse_ip [tParenL | TS] TS' IP :- !,
parse_conj_ilist TS [tParenR | TS'] L',
{std.length L'} >= 2,
foldr {std.drop-last 2 L'} (iList [{std.take-last 2 L'}]) (x\ a\ r\ r =
```

- Recursive parser

```
pred parse_ilist i:list token, o:list token, o:list (list intro_pat).
parse_ilist [tBracketR | TS] [tBracketR | TS] [[]].
parse_ilist TS R [[IP] | LL'] :-
  parse_ip TS [tBar | RT] IP,
  parse_ilist RT R LL'.
parse_ilist TS R [[IP | L] | LL'] :-
  parse_ip TS RT IP,
  parse_ilist RT R [L | LL'].

pred parse_conj_ilist i:list token, o:list token, o:list intro_pat.
parse_conj_ilist TS [tParenR | R] [IP] :-
  parse_ip TS [tParenR | R] IP.
parse_conj_ilist TS R [IP | L'] :-
  parse_ip TS [tAmp | RT] IP,
  parse_conj_ilist RT R L'.
```

4.6.3 Danger of backtracking

- Show timing of current `parse_ilist` code on larger inputs
- Change backtracking
- Show new timings
- Explain why it is better

```
pred parse_ilist i:list token, o:list token, o:list (list intro_pat).
parse_ilist [tBracketR | TS] [tBracketR | TS] [[]].
parse_ilist TS R [IPS | LL'] :-
  parse_ip TS RT IP,
  (
    (
      RT = [tBar | RT'],
      parse_ilist RT' R LL',
      IPS = [IP]
    );
    (
      parse_ilist RT R [L | LL'],
      IPS = [IP | L]
    )
  ).
```

4.7 Applier

- Only used standard Elpi
- Now use Coq-Elpi
- What Coq-Elpi adds
- Section overview

4.7.1 Elpi coq HOAS

- First step, represent Coq terms in Elpi
- Names and function application are just constructors

```
1+1
```

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

- Explain app, global, const, indc and «»
- Coq-Elpi uses higher-order abstract syntax (HOAS)
- functions in Coq are functions that produce terms in Coq-Elpi

```
fun (n: nat), n + 1
```

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

- fun constructor taking name, type and function producing term
- footnote about names all being convertible

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

- prod, let, fix work the same

4.7.2 Coq context in Elpi

- Looking at terms in functions becomes hard as we need to give the function an input to get the term
- introduce fresh constant using `pi x\`

```
FUN = fun _ _ F,
pi x\ F x = app [_, _, P],
P = app [global (indc «S»), global (indc «O»)]
```

- Take function out of constructor
- Fill in function with existential variable to inspect contents
- Take out number we add
- We lose type and name information about x

```
pred decl i:term, o:name, o:term.
decl x `n` (global (indt «nat»)).
```

- decl rule describes types and names of variables
- Lookup type using `decl x N T`
- We have to add the rule when we define x

```
pi x\ decl x `n` (global (indt «nat»))
        => coq.typecheck (F x) Type ok.
```

- We add a rule to the top of the rules for the execution of the code after the `=>`
- During typechecking, `decl x N T` is executed resulting in ...
- `Type` becomes (global (indt «nat»))
- `=>` has many more uses later on

4.7.3 Quotation and anti-quotation

- Writing terms is a lot of work
- Coq-Elpi allows us to write Coq code that is translated immediately using imports in current file

```
{{ λ (n: nat), n + 1 }} =
  fun `n` (global (indt «nat»)) c0 \
      app [global (indt «sum»),
          c0,
          app [global (indc «S»), global (indc «O»)]]
```

- Coq-Elpi also allows putting Elpi vars in Coq terms (anti quotation)

```
{{ @envs_entails lp:PROP (@Envs lp:PROPE lp:CI lp:CS lp:N) lp:P }}
```

- Extract values from term
- Insert values in term, useful in proofs

```
{{ as_emp_valid_2 lp:Type _ (tac_start _ _) }}
```

- Lemma useful in next section
- Type is type of goal we want to proof
- Term becomes lemma we can apply to goal

4.7.4 Proofs in Elpi

- Proofs in Elpi built up proof term step by step
- Pass around Type of goal and variable to assign proof term to
- This is hole

```
kind hole type.
type hole term -> term -> hole. % hole Type Proof
```

- Proofs take a hole and often produce new holes
- Following proof step applies the ex-Falso proof step

- Replace type with False

```
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.
```

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

- Elaborate Lemma against type to generate proof term will be Lemma filled in with necessary values
- Next, extract New proof variable
- Get type of new proof variable

Iris context counter

- Iris can have anonymous hypotheses in context
- Keep track of number to assign to anon hypothesis
- Normally in Type
- Since we derive the type from the proof term we have to apply increases in this number in the proof term
- Instead we keep track of it separately

```
pred do-iStartProof i:hole, o:ihole.
do-iStartProof (hole Type Proof) (ihole N (hole NType NProof)) :-
  coq.elaborate-skeleton {{ as_emp_valid_2 lp:Type _ (tac_start _ _) }} T
  Proof = {{ as_emp_valid_2 _ _ (tac_start _ lp:NProof) }},
  coq.typecheck NProof NType ok,
  NType = {{ envs_entails (Envs _ _ lp:N) _}}.
```

- Start proof applies start proof lemma
- Next extracts current anon hypotheses count
- Stores it in hole using new type ihole

```
kind ihole type.
type ihole term -> hole -> ihole. % ihole iris hyp counter, (hole type p
```

- Counter is Coq positive since increasing it is fairly easy

```
pred increase-ctx-count i:term, o:term.
increase-ctx-count N NS :-
  coq.reduction.vm.norm {{ Pos.succ lp:N }} _ NS.
```

- We can increase counter and put it in the resulting `ihole` when necessary.

4.7.5 Continuation Passing Style

- When introducing a forall we need to add the variable to our context
- Next steps in the proof thus need the new value in the context
- We have to use continuation passing style

```
pred do-intro-anon i:hole, i:(hole -> prop).
do-intro-anon (hole Type Proof) C :-
  coq.ltac.fresh-id "a" {{ False }} ID,
  coq.id->name ID N,
  coq.elaborate-skeleton (fun N _ _) Type Proof ok,
  Proof = (fun _ T IntroFProof),
  @pi-decl N T x\
    coq.typecheck (IntroFProof x) (F x) ok,
    C (hole (F x) (IntroFProof x)).
```

- This introduces a variable without needing a name
- first two steps create the name of the variable
- Next we use a function as the proof term
- We extract the (term -> term) proof variable and the type
- Add the new variable to the context with the name
- Get the type of the new hole
- Call the continuation function on the hole in the context

- In our eiIntros tactic we will be calling predicates like `do-intro-anon` and thus we get a similar type

```
pred do-iIntros i:(list intro_pat), i:ihole, i:(ihole -> prop).
do-iIntros [] IH C :- !, C IH.
do-iIntros [iPure (none) | IPS] (ihole N (hole Type Proof)) C :-
  coq.elaborate-skeleton {{ tac_forall_intro_nameless _ _ _ _ _ _ }} Typ
  Proof = {{ tac_forall_intro_nameless _ _ _ _ _ lp:IProof }},
  coq.typecheck IProof IType ok, !,
  do-intro-anon (hole IType IProof) (h\ sigma IntroProof\ sigma IntroTyp
    h = hole IntroType IntroProof,
    coq.reduction.lazy.bi-norm IntroType NormType, !,
    do-iIntros IPS (ihole N (hole NormType IntroProof)) C
  ).
```

- The predicate `do-iIntros` gets a list of intro patterns, an ihole and the continuation function
- Base case calls the cont. predicate
- Pure intro case
- First transform goal to put forall at the top of goal
- Then use `do-intro-anon` to introduce that variable
- Lastly normalize the type and call iIntros on the new hole
- No anon Iris hypotheses introduced thus counter stays the same

4.7.6 Backtracking in proofs

Question: We don't actually need to backtrack here, we can just look at the type and see which case we need

```
pred do-iIntro-ident i:ident, i:ihole, o:ihole.
do-iIntro-ident ID (ihole N (hole Type Proof))
                   (ihole N (hole IType IProof)) :-
  ident->term ID _ T,
  coq.elaborate-skeleton
    {{ tac_impl_intro _ lp:T _ _ _ _ _ _ _ }}
    Type Proof ok, !,
  Proof =
    {{ tac_impl_intro _ _ _ _ _ _ _ _ lp:IProof }},
  coq.typecheck IProof IType' ok,
  pm-reduce IType' IType,
  if (IType = {{ False }})
```

```
     (coq.error "eiIntro: " X " not fresh")
     (true).
do-iIntro-ident ID (ihole N (hole Type Proof))
                   (ihole N (hole IType IProof)) :-
  ident->term ID _ T,
  coq.elaborate-skeleton
    {{ tac_wand_intro _ lp:T _ _ _ _ _ }}
    Type Proof ok, !,
  Proof = {{ tac_wand_intro _ _ _ _ _ _ lp:IProof }},
  coq.typecheck IProof IType' ok,
  pm-reduce IType' IType,
  if (IType = {{ False }})
     (coq.error "eiIntro: " X " not fresh")
     (true).
do-iIntro-ident ID _ _ :-
  ident->term ID X _,
  coq.error "eiIntro: " X " could not introduce".
```

4.7.7 Starting the tactic

- Solve is the entry point
- Gets a goal with type proof and the arguments

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

- First we parse the arguments
- Ten start proof and get the ihole
- Then start do-iIntros where at the end we put the context counter in the proof
- ...
- ...

4.8 Writing commands

Chapter 5

Elpi implementation of Inductive

5.1 Functor

- We can also make commands
- What do we get as input for our commands
- What do we need to turn it in to
- Show example for isMLL

5.2 Monotone

5.2.1 Proper

- Write tactic for solving IProper proofs
- We write small tactics for different possible steps
- Simple steps, for respectful, point-wise, persistent
- Finishing steps for assumption and reflexive implication
- Apply other proper instance
- Find how many arguments to add to connective
- Lemma to get IProper instance from IProperTop instance
- Apply Lemma IProper
- Compose till all goals proven

5.2.2 Induction for proper

- Create Proper Type for fix-point
- Add point-wise for every constructor using fold-map
- Add this to left and right of respectful with a persistent around left-hand side
- Apply proper solver

5.3 Least fix-point

- The basic structure is this ...
- We recurse over the type of the fix-point to introduce lambda's and existential quantification
- As the last step we add lambda's for any parameters we have

Chapter 6

Conclusion

6.1 Application

6.2 Evaluation of Elpi

6.3 Related work

6.4 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.

[Mon11] Eric Monfroy. “Constraint Handling Rules by Thom Frühwirth, Cambridge University Press, 2009. Hard Cover: ISBN 978-0-521-87776-3.” In: *Theory Pract. Log. Program.* 11.1 (Jan. 2011), pp. 125–126. DOI: 10.1017/S1471068410000074.

[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.

[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.