Master thesis
Computing Science

Radboud University

Extending Iris with Inductive predicates using Elpi

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

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

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

March 26, 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]. Iris has been added to Coq in what is called the Iris Proof Mode (IPM) [KTB17; Kre+18]. Iris the seperation logic and the IPM will be used interchangeably.

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 will set up the example we will discuss in this chapter in section 2.1. Next, we will be looking at separation logic as we will use it in the rest of this thesis 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 will show how Hoare triples and weakest preconditions relate to one another and in the progress explain persistent propositions. Next, we will show how we can create a predicate used to represent a data structure for our example in section 2.5. Lastly, we will finish the specification and proof of a program manipulating marked linked lists in section 2.6.

2.1 Setup

We will be defining a program that deletes an element at an index in a MLL as our example for this chapter. 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 won't need any of these features. It can thus 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 make use of a few pieces of syntactic sugar to simplify notation. We write let statements, $\texttt{let}\ x = e\ \texttt{in}\ e'$, using rec expressions $(\lambda x.\, e')(e)$. To sequence expressions we write $e; e'$ which is defined using a let where we ignore the result of the first expression. The keywords **none** and **some** are just **inl** and **inr** respectively, both in values and in the match statement. We define the short circuit and $e_1 \&\& e_2$ using the following if statement,

if e_1 **then** e_2 **else false**. Lastly we omit the

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

Figure 2.1: Fragment of the syntax of HeapLang as used in the examples

The program we will be using as an example will delete an index out of the list by marking that node, thus logically deleting it.

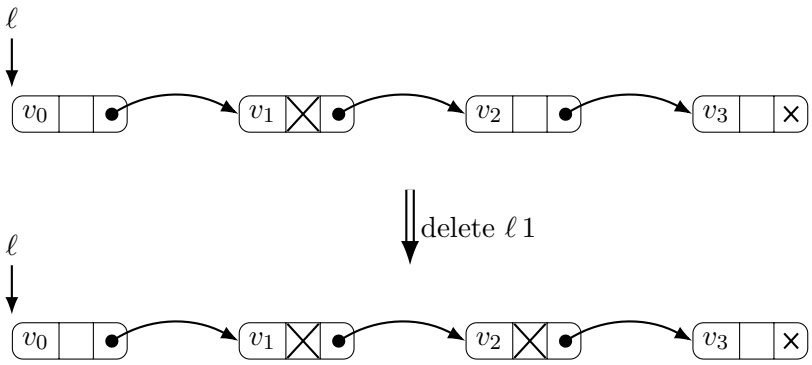
```
delete hd i := match hd with
                 none  ⇒ ()
               | some ℓ ⇒ let (x, mark, tl) = !ℓ in
                           if mark = false && i = 0 then
                             ℓ ← (x, true, tl)
                           else if mark = false then
                             delete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

The program is a function called delete, the function has two arguments. The first argument ℓ is either **none**, for the empty list, or **some** hd where hd is a pointer to a MLL. HeapLang has no null pointers, thus we use **none** as the null pointer. The second argument is the index in the MLL to delete. The first step this recursive function does in check whether the list we are deleting from is empty or not. We thus match ℓ on either **none**, the MLL is empty, or on **some** hd, where hd becomes the pointer to the MLL and the MLL contains some nodes. If the list is empty, we are done and return unit. If 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 options for our list.

- If our index is zero and the element is not marked, thus logically deleted, we want to delete it. We write to the *hd* pointer our node, 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.
- Lastly 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.

delete $\ell\, 1$ will thus apply 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 thinking about it in terms of lists, delete $\ell\, 1$ deletes from the list $[v_0, v_2, v_3]$ the element v_2, thus resulting in the list $[v_0, v_3]$. In the next section we will show how separation logic can be used to reason about sections of memory, such as shown above.

2.2 Separation logic

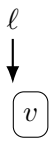
Separation logic, [IO01; Rey02], is a logic that allows us to represent the state of memory in a higher order predicate logic. We also make use of recent additions to separation logic as seen in ?. We take a subset of features from these separation logics and present them below, starting with the syntax.

$$P \in \mathit{iProp} ::= \mathsf{False} \mid \mathsf{True} \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid$$

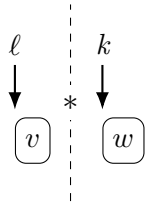
$$\lceil \phi \rceil \mid \ell \mapsto v \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box\, P \mid \mathsf{wp}\, e\, [\Phi]$$

Separation logic contains all the usual higher order predicate logic connectives as seen on the first line, where τ is any type we have seen. The second row contains separation logic specific connectives. The first connective 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 from context it is clear a statement is pure, we may omit the pure brackets. The next two connectives will be discussed in this section. The last three connectives will be discussed when they become relevant in section 2.3 and section 2.4. We will sometimes contract the $\mathsf{wp}\, e\, [\Phi]$ statement, and we will also discuss these contractions in section 2.3.

The first connective to discuss is the points to, $\ell \mapsto v$. The statement $\ell \mapsto v$ holds for any state of memory in which we own a location ℓ and at this location ℓ we have the value v. We represent this state of memory using the below diagram.



To describe two values in memory we could try to write $\ell \mapsto v \land k \mapsto w$. 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 $\ell \mapsto v \land k \mapsto w$. Thus, we introduce a second form of conjunction, the separating conjunction, $P * Q$. For $P * Q$ to hold we have to split the memory in two disjunct parts, P should hold for one part and Q should hold for the other part.



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

The separating conjunction is formally defined by a set of rules.

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

$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}\ {}^{*\text{-MONO}}$$

These rules mirror the rules for the normal conjunction, however, there is one omission. We can not duplicate a proposition using the separating

conjunction. Thus, the following rule is missing $P \vdash P * P$. This makes sense intuitively 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 disjunct sections of memory where ℓ resides in both.

2.3 Writing specifications of programs

The goal in specifying programs is to connect the world in which the program lives to the mathematical world. In the mathematical world we are able to create proofs and by linking the world of the program to the mathematical world we can prove properties of the program.

In this section we will discuss how to specify actions of a program, we will do so using two different methods, the Hoare triple and the weakest precondition. In the next section, section 2.4, we will discuss how they are related. We will use delete as defined in section 2.1 as an example throughout this section.

Hoare triples Our goal when we specify a program will be total correctness. Thus, given some precondition holds, the program does not crash and 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 terminates with a return value v and $\Phi(v)$ holds afterwards. 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 will often happen as quite a few programs return (). We can 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. Our specification states that as a precondition, $\ell \mapsto v$, thus, there we own a location ℓ, and it has value v. Next, we can execute $\ell \leftarrow w$, and it won't 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.

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

We make use of a predicate we will explain in section 2.5. The predicate $\mathsf{isMLL}\, hd\, \vec{v}$ holds if the MLL starting at hd contains the mathematical list $\vec{v}$. The 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 can be harder to work with in conjuction with predicates like isMLL. Instead, we introduce the total weakest precondition, $\mathsf{wp}\, e\, [\Phi]$, also 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 terminates with a return value v. Afterwards, $\Phi(v)$ holds. We make use of the same contractions when writing the predicate of the weakest precondition as with the Hoare triple.

We still need a concept of a precondition when working with the specification of a program, but 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 distribution of sections of memory. 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.

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

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

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

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

To prove that this specification holds we use the rules for the weakest precondition in figure 2.2. We can use the WP-STORE rule to prove that the specification holds. We have two categories of rules, rules for the language constructs, such as WP-STORE, and rules for reasoning about the structure of the language.

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 that for $\mathsf{wp}\ \mathtt{ref}(v)\ [\Phi]$ to hold, $\Phi(\ell)$ should hold for a new ℓ for which we know that $\ell \mapsto v$.

- 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 $\ell \mapsto v$. Thus, 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 itselef 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 that is in the head position of a program. This is done by using contexts 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

Structural rules.

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

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

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

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

Rules for basic language constructs.

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

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

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

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

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

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

Figure 2.2: Rules for the weakest precondition assertion.

2.4 Persistent propositions and nested hoare triples

In this section we will show Hoare triples are defined 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.

$$\textsc{Hoare-def}\quad [P]\, e\, [\Phi] \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, [\Phi])$$

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, $\Box$.

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 $\Box$-DUP below. To prove a statement is persistent, thus that $\Box\, P$ holds, we are only allowed to have persistent proposition in our assumptions, as can be seen in the rule $\Box$-MONO below.

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

$$\Box\text{-E}\quad \Box\, P \vdash P \qquad \Box\text{-CONJ}\quad \Box\, P \wedge Q \vdash \Box\, P * Q \qquad \lceil \phi \rceil \vdash \Box \lceil \phi \rceil \quad \mathsf{True} \vdash \Box\, \mathsf{True}$$

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

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

$$\Box\text{-I}\quad \frac{\Box\, P \vdash Q}{\Box\, P \vdash \Box\, Q}$$

We keep the fact that the assumption is persistent and thus still allow for duplicating the assumption while still removing the persistence modality around the conclusion.

Nested Hoare triples From the definition of the Hoare triple, we know that Hoare triples are persistent. This is needed since we have a higher order heap, in other words, we can store closures in memory. 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. 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 it's postcondition another Hoare triple for the returned closure. We just need one more derived rule before we can apply this specification of refadd in a proof.

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

Given 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

The below Hoare triple holds given that

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

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

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

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

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.

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

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

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

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.

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

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

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

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.

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

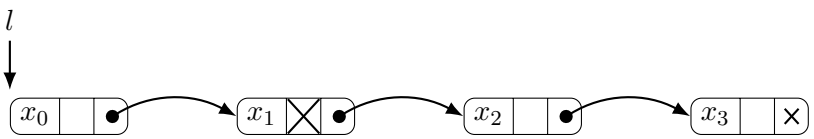
With as assumption that $\ell \mapsto 20$ holds. We can now use the WP-LOAD rule to prove the statement.

□

2.5 Representation predicates

We have shown in the previous three sections how one can represent simple states of memory in a logic and reason about it together with the program. However, this does not easily scale to more complicated data types, especially recursive data types. One such data type is the MLL. We want to connect a MLL in memory to a mathematical list. In section 2.3 we used the predicate $\mathsf{isMLL}\ hd\ \vec{v}$, which tells us that the in the memory starting at hd we can find a MLL that represents the list $\vec{v}$. In the next chapter we will show how such a predicate can be made, in this section we will 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, $\mathsf{isMLL}\,(\mathbf{some}\,\ell)\,[x_0, x_2, x_3]$.

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

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

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 disjunct by use of the separating conjunction.
- The value hd contains a pointer to some node, and this node is not marked as deleted. The list $\vec{v}$ now starts with the value v' and ends in the list $\vec{v}''$. Lastly, the value tl is a MLL represented by this mathematical list $\vec{v}''$

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

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

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

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

Case Marked Head: This is the first inductive case, we have to prove Φ for a head containing a pointer ℓ and the list $\vec{v}$. We get as assumption that ℓ points to a node that is marked as deleted and contains a possible null pointer tl. We also get 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 get as assumption that ℓ points to a node that is marked as not deleted and the node contains a possible null pointer *tl*. We also get 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 two 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 will see this in practice in the next section, section 2.6.

2.6 Proof of delete in MLL

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

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

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. We thus need to prove that

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

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

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

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

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

And, as a result we get three cases we need to prove:

Case 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

Case 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.

Case 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

3.1 Finite predicates and functors

3.2 Monotone functors

3.3 Fixpoints of functors

3.4 Induction principle

Chapter 4

Implementing an Iris tactic in Elpi

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

4.1 `iIntros` example

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

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

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

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

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

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

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

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

We can use `iIntros "HP HQ"`, this will intelligently apply $-\!\!*$I-E twice.

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

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

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

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

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

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

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

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

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

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

Thus, we can break down `iIntros "%x [[%y [Hx Hy]] | H0]"` into its components. We first forall introduce or first sub intro pattern `"%x"`

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

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

4.2 Contexts

4.3 Tactics

4.4 Elpi

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

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

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

4.5 Tokenizer

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

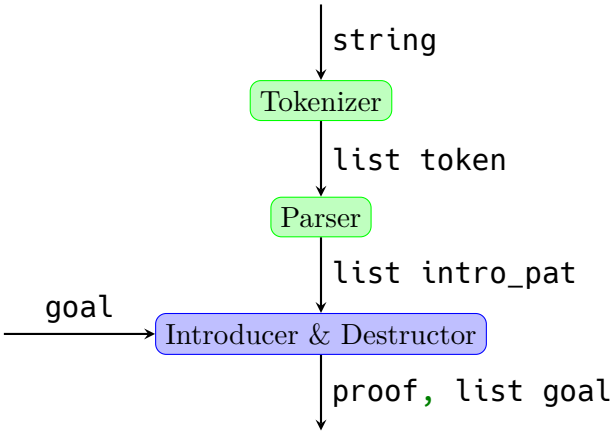


Figure 4.1: Structure of `eiIntros` with the input and output types on the edges.

4.5.1 Data types

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

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

We first define a new type called token using the `kind` keyword, where `type` specifies the kind of our new type. Then we define several constructors for the token type. These constructors are defined using the `type` keyword, we specify a list of names for the constructors and then the type of those constructors. The first set of constructors do not take any arguments, thus have type `token`, and just represent one or more constant characters. The next few constructors take an argument and produce a token, thus allowing us to

store data in the tokens. For example, `tName` has type `string -> token`, thus containing a string. Besides `string`, there are a few more basic types in Elpi such as `int`, `float` and `bool`. We also have higher order types, like `option A`, and later on `list A`.

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

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

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

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

4.5.2 Predicates

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

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

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

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

When we call it, we assign the empty string to its first argument, the string we want to tokenize to the second argument, and we store the output list of string in the new variable `SS`. This predicate allows us to split a string at a certain delimiter. We take as delimiter the empty string, thus splitting the string up in a list of strings of one character each. Strings in Elpi are

based on OCaml strings and are not lists of characters. Since Elpi does not support pattern matching on partial strings, we need this workaround.

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

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

4.5.3 Matching and unification

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

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

```
1 pred tokenize.rec i:list string, o:list token.
2 tokenize.rec [] [] :- !.
3 tokenize.rec [" " | SL] TS :- !, tokenize.rec SL TS.
4 tokenize.rec ["$" | SL] [tFrame | TS] :- !,
5   tokenize.rec SL TS.
6 tokenize.rec ["/", "/", "=" | SL] [tSimpl, tDone | TS] :- !,
7   tokenize.rec SL TS.
8 tokenize.rec ["/", "/" | SL] [tDone | TS] :- !,
9   tokenize.rec SL TS.
```

This predicate has several rules, we chose a few to highlight here. The first rule, on line 2, has a premise and a cut as its conclusion, we will discuss cuts in section 4.5.5, for now they can be ignored. This rule can be used when the first argument matches `[]` and if the second argument unifies with `[]`. The difference is that, for two values to match they must have the exact same constructors and can only contain variables in the same places in the

[1] Names in Elpi can have special characters in them like `.`, `-` and `>`, thus, `tokenize` and `tokenize.rec` are fully separate predicates. It is just a convention that when creating a helper predicate we name it by adding a dot and a short name for the helper.

value. Thus, the only valid value for the first argument of the first rule is `[]`. When unifying two values we allow a variable to be unified with a constructor, when this happens the variable will get assigned the value of the constructor. Thus, we can either pass `[]` to the second argument, or some variable `V`. After the execution of the rule the variable `V` will have the value `[]`.

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

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

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

4.5.4 Functional programming in Elpi

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

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

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

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

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

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

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

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

4.5.5 Backtracking

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

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

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

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

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

and try the next possible rule. This will be the rule on line 4 and returns the input as the second output of the predicate.

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

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

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

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

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

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

If we take the following example

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

When evaluating this predicate we would first apply the name rule of the `tokenize.rec` predicate. This would unify `TS` with `[tName "H" | TS']`

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

4.6 Parser

4.6.1 Data structure

4.6.2 Reductive descent parsing

4.6.3 Danger of backtracking

4.7 Applier

4.7.1 Elpi coq HOAS

4.7.2 Coq context in Elpi

4.7.3 Quotation and anti-quotation

4.7.4 Proofs in Elpi

Iris context counter

4.7.5 Continuation Passing Style

4.7.6 Backtracking in proofs

4.7.7 Starting the tactic

4.8 Writing commands

Chapter 5

Elpi implementation of Inductive

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: (Oct. 11, 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.

[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.