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 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 use several pieces of syntactic sugar to simplify notation. Lambda expressions, $\lambda x.\, e$, are defined using rec expressions. We write let statements, $\texttt{let}\ x = e\ \texttt{in}\ e'$, using lambda expressions $(\lambda x.\, e')(e)$. Expression sequencing is written as $e; e'$, this 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, $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{false}$.

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

Figure 2.1: Relevant fragment of the syntax of HeapLang

Lastly, we often omit the **rec** keyword whenever we define a recursive function.

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

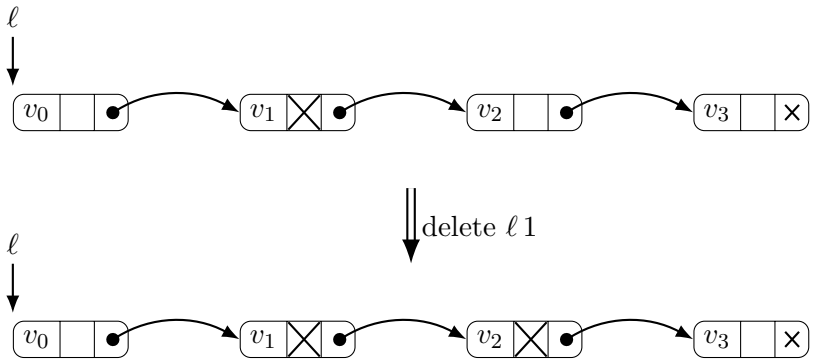
```
delete hd i = match hd with
                none  ⇒ ()
              | some ℓ ⇒ let (x, mark, tl) = !ℓ in
                         if mark = false && i = 0 then
                           ℓ ← (x, true, tl)
                         else if mark = false then
                           delete tl (i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

The example is a recursive function called delete, the function has two arguments. 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 taken is checking whether we are deleting from the empty list. To do this, we perform a match on *hd*. Either *hd* is **none**, the MLL is empty. Or, *hd* is **some** *hd*, *hd* becomes the pointer to the MLL and the MLL contains some nodes. If the list is empty,

the function is done, and it returns 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 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

Separation logic, [IO01; Rey02], is a logic that is used to represent the state of memory in a predicate logic. This is the basis for the logic we use. We also

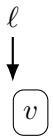
make use of several additions to separation logic taken from Iris [Kre+17]. This logic is presented below, starting with the syntax.

$$P \in \mathit{iProp} ::= \mathsf{False} \mid \mathsf{True} \mid P \land P \mid P \lor P \mid P \Rightarrow P \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \\ \lceil \phi \rceil \mid \ell \mapsto v \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box\, P \mid \mathsf{wp}\; e\; [\Phi]$$

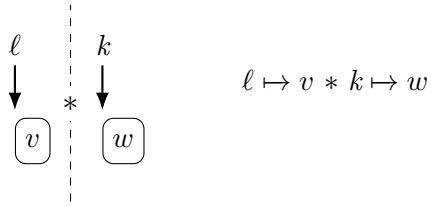
Separation logic contains all the usual higher order predicate logic connectives as seen on the first line. The symbol τ, represents is any type we have seen, including *iProp* itself. The second row contains separation logic specific connectives. The first connective embeds any Coq proposition, also called a pure proposition, into separation logic. Coq propositions include common connectives like equality, list manipulations and set manipulations. Whenever it is clear from context that a statement is pure, we may omit the pure brackets. The next two connectives are discussed in this section. The last three connectives 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 and its value. Using this semantic model of separation logic we can reason about its connectives.

The statement $\ell \mapsto v$, 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 $\ell \mapsto v \land k \mapsto w$. 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 for a heap we have to split it in two disjunct parts, P should hold while owning only location in the first part and Q should hold with only the second part.



The separating conjunction has the following set of rules.

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

The separating conjunction is commutative, associative and respects True as identity element. Instead of an introduction and elimination rule, like the normal conjunction has, there is the $*$-MONO rule. The separating conjunction is not duplicable. 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 a heap where ℓ resides in both.

2.3 Writing specifications of programs

The goal when 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 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 with a return value v, with $\Phi(v)$ holding 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 often happens 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. 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 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.

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

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 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 conjunction with predicates like isMLL. 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 with a return value v. Afterwards, the postcondition $\Phi(v)$ 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 precondition when working with the specification of a program, thus we embed this in the logic using the magic wand.

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

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

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

If we have as assumption P and need to prove $Q \mathrel{-\!\!*} R$, We can add Q to our assumptions in order to prove R. Thus, if we add ownership of the heap described by Q we can prove R. Note that this rule works both ways, as signified by the double lined rule. It is both the introduction and the elimination rule.

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

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

This specification holds from WP-STORE in figure 2.2. 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.

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 `wp` **`ref`**(v) $[\Phi]$ to hold, $\Phi(\ell)$ should hold for a new ℓ with $\ell \mapsto v$.
- The rule WP-LOAD defines that for `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 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 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

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

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

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

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 define Hoare triples using the weakest precondition and in the process explain persistent propositions. We end with an example

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

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 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.

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

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

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

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.

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

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

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

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.

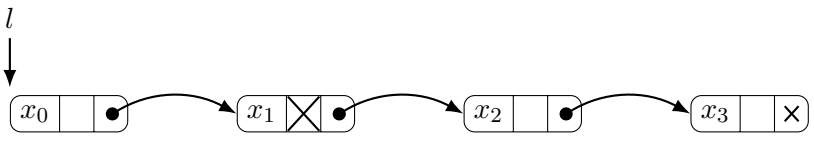
$$\mathsf{wp}\ !\, \ell\, [20.\, \textbf{true}]$$

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 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, isMLL (**some** ℓ) $[x_0, x_2, x_3]$.

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.

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

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.

isMLL-IND

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

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

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

Case Marked Head: This is the first inductive case, we have to prove Φ for a head containing a pointer ℓ and the list $\vec{v}$. We have the assumption that ℓ points to a node that is marked as deleted and

contains a possible null pointer *tl*. We also have the following induction hypothesis: the tail, *tl*, is a MLL represented by $\vec{v}$, and Φ holds for *tl* and $\vec{v}$.

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

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

2.6 Proof of delete in MLL

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

```
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
```

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.

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

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

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

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

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

We need to prove three cases:

Empty MLL: We need to prove the following

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

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

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

This follows from the definition of isMLL

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

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

And, we need to prove that:

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

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

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

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

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

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

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

Which is the left-hand side of our IH.

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

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

And, we need to prove that:

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

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

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

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

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

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

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

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

We use WP-VALUE to reach:

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

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

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

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

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

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

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