Chapter 1

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 **??**. First we will set up the example we will discuss in this chapter in section 1.1. Next, we will be looking at separation logic as we will use it in the rest of this thesis in section 1.2. Then, we show how to give specifications using Hoare triples and weakest preconditions in section 1.3. Next, we will show how we can create a predicate used to represent a data structure for our example in section 1.4. Lastly, we will finish the specification and proof of a program manipulating marked linked lists in section 1.5.

1.1 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 together with any syntactic sugar can be found in figure 1.1. For more information about HeapLang one can reference the Iris technical reference [Iri23].

The program we will be using as an example will delete an index out of

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

$$
\begin{aligned}
\mathbf{let}\, x = e\, \mathbf{in}\, e' &\triangleq (\lambda x.\, e')(e) \\
\mathbf{none} &\triangleq \mathbf{inl}_{\mathbf{v}}(()) \\
\mathbf{some}\, v &\triangleq \mathbf{inr}_{\mathbf{v}}(v) \\
e; e' &\triangleq \mathbf{let}\, \_ = e\, \mathbf{in}\, e'
\end{aligned}
$$

Figure 1.1: Fragment of the syntax of HeapLang as used in the examples with at the bottom syntactic sugar being used

the list by marking that node, thus logically deleting it.

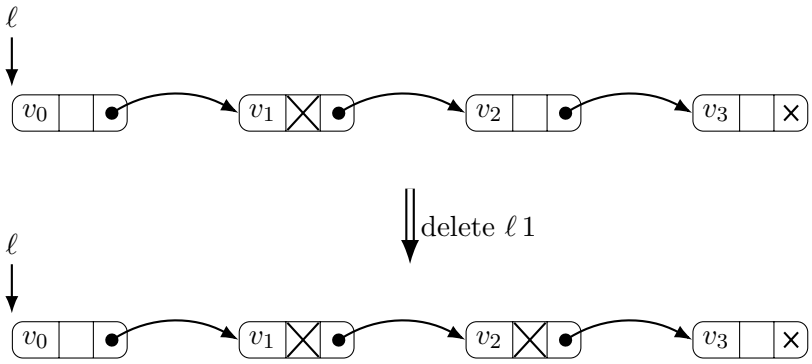
$$
\begin{aligned}
\text{delete}\ \mathit{hd}\, i := {} & \mathbf{match}\, \mathit{hd}\, \mathbf{with} \\
& \quad \mathbf{none} \Rightarrow () \\
& \mid \mathbf{some}\, \ell \Rightarrow \mathbf{let}\, (x, \mathit{mark}, \mathit{tl}) = !\, \ell\, \mathbf{in} \\
& \qquad\qquad \mathbf{if}\, \mathit{mark}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

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.

1.2 Separation logic

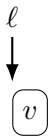
- Separation logic is a logic that allows us to represent the state of memory in a higher order predicate logic

- The syntax is

TODO: What are e and v

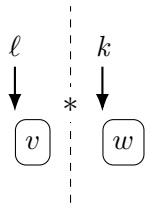
$$P \in \mathit{iProp} ::= \mathsf{False} \mid \mathsf{True} \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \\ \ell \mapsto v \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \mathsf{wp}\, e\, [\Phi]$$

- We will sometimes write $\mathsf{wp}\, e\, [\Phi]$ as $\mathsf{wp}\, e\, [v.\, P]$ where Φ is a predicate that takes a value
- It contains the normal higher order predicate logic connectives on the first line
- The first two connectives on the second line will be discussed in this section
- The last three connectives on the second line will be discussed in section 1.3
- We start with points to, $\mapsto$



$\ell \mapsto v$

- Picture of memory with $\ell \mapsto v$ next to it
- $\ell \mapsto v$ means we own a location in memory, l, and it has value v
- $\wedge$ now no longer works as expected
- introduce $*$



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

- Describe rules of $*$

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

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

- This does not include $P \vdash P * P$

1.3 Writing specifications of programs

- We will discuss how to specify the actions of a program
- delete will be the example
- The goal will be total correctness
- Guarantee that given some preconditions in separation logic hold, the program will terminate and some postconditions in separation logic hold and e is safe
- Typically use Hoare triples

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

- Given that P holds
- e terminates
- returns v
- $\Phi(v)$ now holds
- We often write $[P]\, e\, [v.\, Q]$, thus leaving out a λ
- We can also write a value for v to express that the returned value is that value
- Thus the specification of $\ell \leftarrow w$ becomes

$$[\ell \mapsto v]\, \ell \leftarrow w\, [().\ \ell \mapsto w]$$

- The precondition of our specification is that there is a location ℓ that has value v
- Then, $\ell \leftarrow w$ return unit
- New state of memory is $\ell \mapsto w$
- The Hoare triple of delete is

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

- This uses a predicate we will talk more about in section 1.4
- It tells is that the MLL in memory at hd is represented by the list of value $\vec{v}$
- remove is the function on mathematical lists that removes the element at index i from the list $\vec{v}$

WP

TODO: better title

- There is a second way to specify programs
- We use weakest preconditions, $\mathsf{wp}\ e\ [\Phi]$
- The weakest precondition states that expression e is safe to execute, terminates with value v and afterwards $\Phi(v)$ holds.
- We use the same way of writing predicates in the weakest precondition as with Hoare triples
- There is precondition in the weakest preconditions, instead we add that using the magic wand
- We add that using the magic wand, $P \mathrel{-\!\!*} \mathsf{wp}\ e\ [\Phi]$
- Magic wand is implication that reasons about resources
- $Q \mathrel{-\!\!*} R$ describes resources where if we add Q we get R
- as can bee seen in the below law

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

- This law works both ways.
- Thus, using the magic wand we add that if we have P then $\mathsf{wp}\ e\ [\Phi]$ holds.
- Thus, if we want to specify $\ell \leftarrow v$ using weakest precondition we use the rule WP-STORE in figure 1.2
- To prove, $\mathsf{wp}\ (\ell \leftarrow w)\ [\Phi]$, we have to prove that $\ell \mapsto v$ holds and $\ell \mapsto w \mathrel{-\!\!*} \Phi()$. In other words, if we add that $\ell \mapsto w$ holds, $\Phi()$ should hold.
- Besides rules about specific expression we also have general rules about the weakest precondition
- WP-VALUE is used when a program is finished
- WP-MONO allows us to transform the post condition of wp
- WP-FRAME allows us to ???

Question: what does this again

- WP-BIND can extract the expression that is to be executed inside the whole expression using the possible contexts

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
$$\frac{e \longrightarrow_{\text{pure}} e' \qquad \mathsf{wp}\ e'\ [\Phi]}{\mathsf{wp}\ e\ [\Phi]}$$

Pure reductions.

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

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

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

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

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

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 1.2: Rules for the weakest precondition assertion.

TODO: Explain why Hoare does not work, but wp does

- Thus we define the hoare triple as a weakest precondition
- Only use weakest pre-conditions in our proofs

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

- We make use of new connective, $\Box$
- Our weakest precondition with its precondition are wrapped in a box, making the proposition persistent
- Any persistent proposition has the property that once we know it holds, it always holds
- As can be seen by the rule $\Box$-DUP below
- We are allowed to duplicate any persistent proposition
- We do have to prove that it is persistent
- To prove a proposition persistent we can only use persistent proposition in our assumptions as can be seen in the rule $\Box$-MONO below
- Other rules about persistent proposition can be seen below

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

$\Box$-SEP
$$\Box P * Q \dashv\vdash \Box P * \Box Q$$

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

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

$\Box$-DISTR
$$\Box P \wedge Q \vdash \Box P * Q$$

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

- From the definition of a hoare triple we now know it is persistent
- This is needed since we have a higher order heap, we can store closures
- Take the following function with its specification below

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

- The 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
- We can now use this function in a program like below

$$\begin{aligned}
&\textbf{let}\ f = \text{refadd}\ 10\ \textbf{in}\\
&\textbf{let}\ \ell = \textbf{ref}\, 0\ \textbf{in}\\
&f\,\ell; f\,\ell\\
&!\,\ell
\end{aligned}$$

- Now since we can't duplicate resources and resources are used up once use, once we gain from the specification of addref that f has specification $[\ell \mapsto m]\, f\, \ell\, [().\, \ell \mapsto n]$
- If a hoare triple was not persistent, we could only use this specification once
- Thus we could not verify what would happen the second time we call f.
- But since they are we can use $\square$-DUP to duplicate the specification of f and use it twice
- Thus we can prove that the above program returns 20

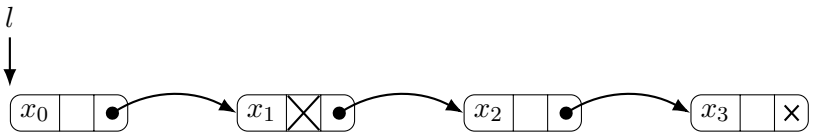
1.4 Representation predicates

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 proves and by linking the program world to the mathematical world we can prove properties of the program.

TODO: Maybe move the first part of this section to an earlier section

We have shown in the previous two 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 datatypes. One such datatype is the MLL. We want to connect a MLL in memory to a mathematical list. In section 1.3 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 this section we will show how such a predicate can be used.

- We need an inductive predicate to reason about a recursive structure
- For $\mathsf{isMLL}\, (\mathbf{some}\, \ell)\, [x_0, x_2, x_3]$ look below



- Our end goal should work like below
- this does not work because it is not necessarily finite?

Question: This is correct right, and is Coq the reason why it has to be finite?

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

- We first turn our desired predicate into a functor
- It transforms a predicate Φ into a predicate that applies Φ to the tail of the MLL if it exists

TODO: write more correct

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

- This gets rid of the possible infinite nature of the statement
- but not strong enough
- we want to find a Φ such that

$$\forall hd\, \vec{v}.\, \mathsf{isMLLPre}\,\Phi\,hd\,\vec{v} \mathrel{-\!\!*\!\!*} \Phi\,hd\,\vec{v}$$

- This is the fixpoint of isMLLPre
- Use Knaster-Tarski Fixpoint Theorem to find this fixpoint [Tar55]
- Specialized to the lattice on predicates

Theorem 1.1 (*Knaster-Tarski Fixpoint Theorem*)

Let $\mathsf{F}\colon (A \to iProp) \to (A \to iProp)$ *be a monotone predicate, then*

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

defines the least fixpoint of F

Question: Where to introduce *iProp*?

- Monotone is defined as

Definition 1.2 (*Monotone predicate*)

Any F is monotone when for any $\Phi, \Psi\colon A \to iProp$, it holds that

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

- In general F is monotone if all occurrences of its Φ are positive
- This is the case for isMLL
- We can expand theorem 1.1 to predicates of type $\mathsf{F}\colon (A \to B \to iProp) \to (A \to B \to iProp)$
- Thus the fixpoint exists and is

$$\mathsf{lfp}\,\mathsf{isMLLPre}\,hd\,\vec{v} = \forall \Phi.\, (\forall hd'\ \vec{v}'.\, \mathsf{isMLLPre}\,\Phi\,hd'\ \vec{v}' \mathrel{-\!\!*} \Phi\,hd'\ \vec{v}') \mathrel{-\!\!*} \Phi\,hd\,\vec{v}$$

- We can now redefine isMLL as

$$\mathsf{isMLL}\,hd\,\vec{v} \triangleq \mathsf{lfp}\,\mathsf{isMLLPre}\,hd\,\vec{v}$$

- Using the least fixpoint we can now define some additional lemmas

Lemma 1.3 (lfp F *is the least fixpoint on* F)

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

$$\forall x.\, \mathsf{F}\,(\mathsf{lfp}\,\mathsf{F})\,x \mathrel{-\!\!*\!\!*} \mathsf{lfp}\,\mathsf{F}\,x$$

Lemma 1.4 (*least fixpoint induction principle*)

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

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

Lemma 1.5 (*least fixpoint strong induction principle*)

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

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

TODO: Maybe on isMLL example

1.5 Proof of delete in MLL

In this section we will proof 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 1.6

For any 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 proof that

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

We can use $\square$-MONO and $-\!\!*$I-E to assume $\mathsf{isMLL}\ hd\ \vec{v}$, and we now have to proof

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

And we can do induction on $\mathsf{isMLL}\ hd\ \vec{v}$ for any i using the induction principle defined in **??**, and get three cases.

Case Empty MLL: We know that $hd = \mathbf{none}$ and $\vec{v} = []$, thus 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 and WP-BIND rules 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 $hd = \mathbf{some}\,\ell$ and $\ell \mapsto (v', \mathbf{true}, tl)$ with disjointed as IH the following.

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

Thus, we need to prove that

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

By repeatedly using the WP-PURE and WP-BIND rules, we get that we need to prove

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

Recall that let statements are defined in terms of lambda expressions, we can thus use WP-BIND with as context

$$K = \left(\begin{array}{l} \textbf{let}\,(x, mark, tl) = \bullet\,\textbf{in} \\ \textbf{if}\; mark = \textbf{false}\;\&\&\; i = 0\,\textbf{then} \\ \quad \ell \leftarrow (x, \textbf{true}, tl) \\ \textbf{else if}\; mark = \textbf{false}\,\textbf{then} \\ \quad \text{delete}\; tl\; (i-1) \\ \textbf{else} \\ \quad \text{delete}\; tl\; i \end{array}\right)$$

We get the following statement we need to prove

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

And we can use WP-LOAD with $\ell \mapsto (v, \textbf{true}, tl)$ and $-\!\!*$I-E to get our new statement to prove:

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

We again repeatedly use WP-PURE with WP-BIND to reach the following.

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

Which is the left-hand side of our IH.

Case Unmarked head: We know that $hd = \textbf{some}\,\ell$, $\vec{v} = [v'] + \vec{v}''$ and $\ell \mapsto (v', \textbf{false}, tl)$ with disjointed as IH the following.

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

Thus, we need to prove that

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

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

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

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 with WP-BIND until we reach:

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

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

$$\mathsf{wp}\ ()\ \left[(). \ \mathsf{isMLL}\ (\textbf{some}\ \ell)\ (\mathsf{remove}\ 0\ ([v'] + \vec{v}'')) \right]$$

We use WP-VALUE to reach:

$$\mathsf{isMLL}\ (\textbf{some}\ \ell)\ (\mathsf{remove}\ 0\ ([v'] + \vec{v}''))$$

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

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

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

We use WP-MONO with as assumption our the left-hand side of the IH, since it is not persistent we now lose the IH, but we don't need it any more. We now need to prove the following:

$$\mathsf{isMLL}\ tl\ (\mathsf{remove}\ i\ \vec{v}'') \vdash \mathsf{isMLL}\ (\textbf{some}\ \ell)\ (\mathsf{remove}\ (i-1)\ ([v'] + \vec{v}''))$$

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

□