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 setup 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 datastructure 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

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

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

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.

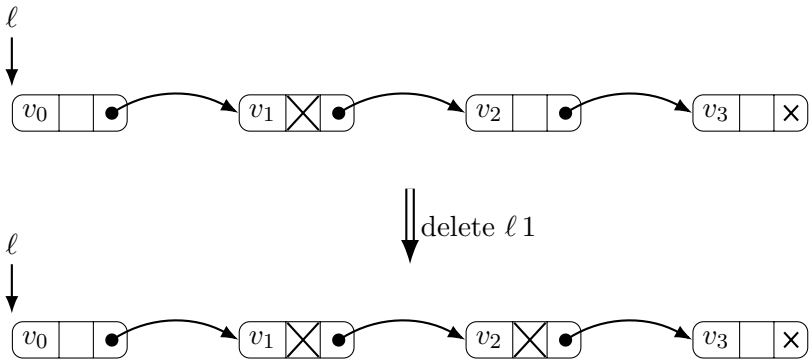
$$
\begin{array}{l}
\text{delete}\ \ell\, i := \mathbf{match}\, \ell\, \mathbf{with} \\
\quad \mathbf{none} \quad \Rightarrow () \\
\quad \mid \mathbf{some}\, \mathit{hd} \Rightarrow \mathbf{let}\, (x, \mathit{mark}, \mathit{tl}) = !\, \mathit{hd}\, \mathbf{in} \\
\qquad\qquad \mathbf{if}\, \mathit{mark}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

The 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 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

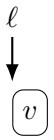
- Separation logic is a logic that allows us to represent the state of memory in a higher order predicate logic

- The syntax is

TODO: What are e and v

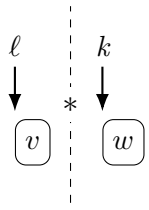
$$P \in \mathit{iProp} ::= \mathsf{False} \mid \mathsf{True} \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \\ \ell \mapsto v \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box P \mid \mathsf{wp}\, e\, [\Phi]$$

- 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 there is 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

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

$$[P]\, e\, [x.\ Q] \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, [x.\ Q])$$

- Explain wand
- Add picture

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

- Explain persistent
- add picture

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

- Explain total wp

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

Question: The ▷ has shown up again

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 made.

- 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

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.

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

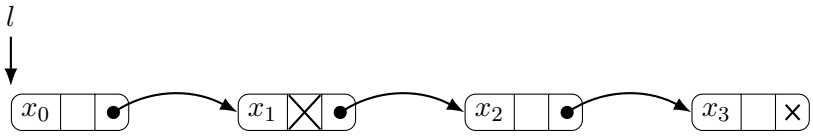
$$\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longrightarrow_{\text{pure}} e_2 \qquad \mathbf{fst}(v_1, v_2) \longrightarrow_{\text{pure}} v_1$$

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

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

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

Figure 1.2: Rules for the weakest precondition assertion.



- 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?

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

- 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

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

- 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 $\mathsf{F} : (A \to iProp) \to (A \to iProp)$ *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 $\Phi, \Psi : A \to \mathit{iProp}$, it holds that

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

- 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 $\mathsf{F} : (A \to B \to \mathit{iProp}) \to (A \to B \to \mathit{iProp})$
- 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 ($\mathsf{lfp}\,\mathsf{F}$ *is the least fixpoint on* F)

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

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

Lemma 1.5 (*least fixpoint strong induction principle*)

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

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

TODO: Maybe on isMLL example

1.5 Proof of delete in MLL