Bachelor's Thesis Computing Science

Radboud University Nijmegen

Soundness for program synthesis using separation logic

Author:
Arvid Bonten
s1052608

First supervisor/assessor:
Dr. Robbert Krebbers

Second assessor:
Dr. Engelbert Hubbers

January 16, 2025

Abstract

Separation logic is widely used to reason about pointer programs. Code synthesis through separation logic is therefore useful to generate pointer programs. Watanabe et al. have created an automatic certification next to existing synthesisers to certify a synthesisers capabilities. Soundness is another property which can reason about the correctness of the synthesiser. In this thesis we show that the language from Polikarpova and Sergey [2018] is sound.

Contents

Chapter 1

Introduction

There are many different forms of program synthesis, some are very well known, but not under the name code synthesis. Generative AI can generate code from a text based description of the algorithm. This concept is called code synthesis, generally, code synthesis is deriving code based on a specification. Some different types of specifications include: Generative AI [Chen et al., 2021], testcases [Perelman et al., 2014], pre- and postconditions [Polikarpova and Sergey, 2018, Watanabe et al., 2021].

Each type of program synthesis has its own up- and downsides. As most of you probably know, for generative AI, there is no formal testing used to ensure that the code it generates is correct. For test-driven program synthesis, this is also not the case, as the induction problem indicates that there could be a test it will still fail.

However, using pre- and postconditions from Hoare logic [Hoare, 1969], we can formally prove that the generated code holds for its specification. This would have the weakness of not being able to generate pointer programs and would only work with state-based programs.

Luckily, there exists an extension of Hoare logic, called separation logic [Reynolds, 2002]. This extension introduces operators for heap manipulation and disjunctive properties. The research of Polikarpova and Sergey [2018], Watanabe et al. [2021] uses this concept to generate a proof script which can be run in the Coq proof assistant.

However, for each time the program outputs code, you would need to start the Coq proof assistant and check whether the proof holds. Another property in Hoare logic is called soundness, which can be used to prove that when the synthesiser gives an output program, that it automatically holds that the code is valid for the given specification [Hoare, 1969, Reynolds, 2002].

Formal verification tools such as the Coq proof assistant are necessary to show that the proof is complete without any doubt. The research done by Watanabe et al. [2021], Polikarpova and Sergey [2018] does not include a

proof of soundness. In this thesis, we will prove that this language is sound. However, we do not implement the synthesiser in Coq, which leaves some degree of uncertainty that the code generated will be correct.

The solution found uses the Coq proof assistant to prove the soundness property for a simplified version of the language used by Watanabe et al. [2021], Polikarpova and Sergey [2018]. The simplifications are that we remove negative integers, loops, scopes, functions and thus recursion. The proof script Bonten [2025] proves soundness for all the cases. However, the frame rule is replaced with a frame lemma. This does not change the soundness of the language as having a proven frame lemma has the same functionality as a proven frame rule.

In this thesis, we will first discuss the prerequisite knowledge on separation logic in chapter 2 using the language from Polikarpova and Sergey [2018] and the separation logic style of Reynolds [2002]. We will then introduce the necessary changes required for code synthesis to separation logic in chapter 3 as defined in Polikarpova and Sergey [2018]. After which we will then walk through some of my proof script in chapter 4 to introduce the concepts which were required to prove soundness in Coq. We will then continue to the related work in chapter 5 and continue to future work in chapter 6.

Chapter 2

Separation logic

This chapter will start with an introduction into separation logic in the style of Reynolds [2002] for the language by Polikarpova and Sergey [2018], the modifications needed for synthesis will be done in the next chapter. First, we will introduce the basics of Hoare logic in section 2.1. Followed by section 2.2, which will start with an example of why separation logic is useful. Then, section 2.3 will explain the syntax and big-step of the language used in this thesis. Then, section 2.4 will explain the assertions and their semantics of the version of separation logic we use in this research. We continue with the inference system in section 2.5 and validity in section 2.6. Lastly, we will show the soundness lemma in section 2.7 and revisit the example using the previous definitions in section 2.8. This chapter can be skimmed if the reader is knowledgeable about separation logic.

2.1 Introduction to Hoare logic

Hoare logic is an inference based logic system Hoare [1969], this system is designed s.t. we can see what is present on the state. Hoare logic uses syntax called triples, these are denoted as follows:

$$\{P\}\, c\, \{Q\}$$

Here, the pre-condition is denoted as P. The post-condition is denoted with Q and the commands run in between denoted with c. A Hoare triple can be used to check the property valid. A Hoare triple is considered valid in total correctness when the program is both correct and terminates in a finite amount of steps. Formally, a program is correct if the precondition holds for all states, the post condition there exists a state s.t. it holds. Let us take a look at the following example of code which changes its post condition based on specific pre-conditions:

$$(1)\ \texttt{if}\ y < x\ \texttt{then}\ z \coloneqq y\ \texttt{else}\ z \coloneqq x$$

In this case, we can decide which branch will be calculated by the program by using the state at the start. We can also include a precondition to include extra information about the relation.

$$\{w > x \wedge w > y\}$$
$$\texttt{if}\ y < x\ \texttt{then}\ z := x\ \texttt{else}\ z := y$$
$$\{x \leq z \wedge y \leq z \wedge w > z\}$$

As we can see, depending on whether we pick x or y to be larger the answer changes, z is the larger of the 2. So this if statement calculates which variable is the maximum. If we started with any number bigger than x and y then we know that any number we picked is bigger than the maximum.

2.2 Introduction example

Separation logic [Reynolds, 2002] is an extension to Hoare logic [Hoare, 1969]. Separation logic adds the separating conjunction operator, denoted as *. The separating conjunction is based on the simple property of separation on the heap. Separation is used to denote what is present on the heap, similar to Hoare logic, which denotes what is present in the state. The $\mapsto$ operator specifically is used to indicate that a pointer points to either a value or an array of values. In separation logic, we can create the following property with the separating conjunction which does not hold for any heap, $l \mapsto n * l \mapsto n'$. The same property with the normal boolean conjunction would give: $l \mapsto n \wedge l \mapsto n'$ which implies that $n = n'$.

The separating conjunction allows us to define properties which need exclusion to properly work. An example of this is the function `memcpy`(x,y,n) of the standard C library. The 'man' page states" The `memcpy`(x,y,n) function copies n bytes from memory area src to memory area dest. The memory areas must not overlap. Use memmove(3) if the memory areas do overlap." Here, x is the area src and y is the area dest. We can use the * operator to denote the property 'the memory areas must not overlap':

Example 1.

$$\{x \mapsto \langle x_1..x_n\rangle * y \mapsto \langle y_1..y_n\rangle\}\ \texttt{memcpy}(x,y,n)\ \{x \mapsto \langle x_1..x_n\rangle * y \mapsto \langle x_1..x_n\rangle\}$$

To start with, x is a pointer, which points to values on the heap. The values in the array from x_1 to x_n is what x points to. We then get the * operator, meaning that whatever comes after this is disjoint from what came before. For the pointer y and its values, it works in the same way as the previously explained pointer x and its values. The command which is run is the `memcpy`(x,y,n) function.

Now that we know what the variables are, we can interpret the example:"before the execution of `memcpy`(x,y,n) , there are 2 pointers x and y

which point to an array of length n, where x and y are disjoint (not overlapping). After execution of the code, the pointers will point to the same content.".

2.3 Language and big-step

In this section, we will start by explaining the language from Polikarpova and Sergey [2018]. We will start with its syntax, then we will introduce heaps and states and lastly, we will introduce its big-step rules.

2.3.1 Syntax

The language used is a simplified version from Polikarpova and Sergey [2018]. The expressions have been simplified by taking away complexity. This was done by changing `malloc()`: to not take the size of the allocation, taking away the function calls and thus recursion. Lastly, the expressions will contain only non-negative integers.

$$
\begin{aligned}
a, a' \in \mathbf{Aexp} &::= n \mid a + a' \mid a - a' \mid a \cdot a' \mid x \\
b, b' \in \mathbf{Bexp} &::= tt \mid ff \mid a = a' \mid a \mathrel{!}= a' \mid a < a' \mid a > a \mid \neg b \mid b \wedge b' \\
c \in \mathbf{Com} &::= x = *(x') \mid *(x) = a \mid x := a \mid \texttt{skip} \mid \texttt{error} \mid \\
&\qquad c_1; c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid x = \texttt{malloc()} \mid \texttt{free}(x) \\
\texttt{Here we let}\ & x \in \mathbf{Variable}, n \in \mathbf{Nat}
\end{aligned}
$$

In this language, we have arithmetic expressions (**Aexp**), which can either be a natural number, a binary operation on two arithmetic expressions or a variable. There are also Boolean expressions (**Bexp**), containing *tt* as boolean true and *ff* as false. Then there are basic comparisons, negations and a boolean conjunction ($\wedge$).

There are also commands. The $x = *(x')$ command (load), which loads the value of the heap at location x' into the variable x. The $*(x) = a$ command (store), which stores the evaluation of a into the location at value x in the heap. After which we have the assignment rule, this assigns a value to a variable in the state. Then we have `skip` and `error`, error stops the program abruptly when called. We also have seq and if commands. Lastly, we have `malloc()` and `free()`. To make our proof of concept easier, the command `malloc()`, allocates 1 location to the pointer x which can hold any size data without requiring the bit or byte size of the data. The next example will show how to calculate the square of 5 while using the syntax of this language.

Example 2.

(1) $x = \texttt{malloc()};$
(2) $*(x) = 5;$
(3) $y = *(x);$
(4) $y = y \cdot y;$
(5) $\texttt{if}\ y > *(x)\ \texttt{then}$
(6) $\quad \texttt{skip}$
(7) $\texttt{else}$
(8) $\quad \texttt{error};$
(9) $\texttt{free}(x)$

In this example, we can see that we first allocate memory for the pointer x. Then, on line 2 we initialize the value x points to 5. On line 3, we store the value of the pointer x into y. Then through lines 5 to 8 we check whether the value at the pointer y is bigger than the pointer x. If this is the case, we skip and otherwise we throw error. Of course, we know that x^2 is always bigger than x so this should always be the case. Lastly, we free the variable x.

2.3.2 Heaps and states

Our heaps and states are both functions where the heap is on a finite domain and the state is a function from string to natural numbers:

$$st \in \mathbf{State} \triangleq \mathbf{String} \to \mathbf{Nat}$$
$$h \in \mathbf{Heap} \triangleq \mathbf{Nat} \xrightarrow{\mathsf{fin}} \mathbf{Nat}$$

Where $\xrightarrow{\mathsf{fin}}$ is defined as follows:

$$\mathbf{A} \xrightarrow{\mathsf{fin}} \mathbf{B} \triangleq \{f : \mathbf{A} \to \mathbf{Option}\,\mathbf{B} \mid \mathit{dom}(f)\ \mathit{is\ finite}\}$$

In the rule [malloc$_{NS}$] in section 2.3.4, we need to get the next free location to return for `malloc()`. To be able to do this, we will need the heaps to be finite. This is due to the edge case which happens with an infinite heap which maps all locations to some value. This map exists, but is unreachable through code. The problem this creates is that we cannot request the next free location any more, as this does not exist. We can also see from this syntax, that we use natural numbers for locations on the heap as well as their values.

The **Option** keyword can either be **Some** followed by a value or it can be **None**, meaning it has no value. We can use the option keyword for heaps, heaps have values which are either pointing to nothing or to something. On

the heap, we can then create pointers which point to pointers. We can do this due to the return type of the heap being an optional **Nat**.

To see how the heap grows, we will check the heap in the previous example. In this example, we first start with an empty heap. Then, we call `malloc()`: once for x, thus, there is a pointer on the heap. Later, we call `free(`x`)`, returning to a heap with 0 values. Which means that for each command we run, the most the heap can grow or shrink by is 1 per command. Which means that if we want to reach an infinite sized heap where all locations point to some value, we would need to have infinitely many commands. We can assume that this will not be the case and thus we will not lose any important functionality by having finite heaps.

Update functions

We will also need the update function on the state and heap, which will define how the state and heap can be updated. This is used in the big-step semantics.

$$(st[x \mapsto n])\ y \triangleq \texttt{if}\ (x = y)\ \texttt{then}\ n\ \texttt{else}\ st\ y$$
$$(h[l \mapsto n?])\ k \triangleq \texttt{if}\ (l = k)\ \texttt{then}\ n?\ \texttt{else}\ h\ l$$

Here, $n?$ is an element of **Option Nat**.

2.3.3 Evaluation functions

We will define the evaluation functions, which can be evaluated from **Aexp** or **Bexp** to **Nat** and **Bool**, respectively.

$$\begin{aligned}
\mathcal{A}[\![_]\!]_ &: \mathbf{Aexp} \to \mathbf{State} \to \mathbf{Nat} \\
\mathcal{A}[\![n]\!]_{st} &\triangleq n \\
\mathcal{A}[\![a + a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} + \mathcal{A}[\![a']\!]_{st} \\
\mathcal{A}[\![a - a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} - \mathcal{A}[\![a']\!]_{st} \\
\mathcal{A}[\![a \cdot a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} \cdot \mathcal{A}[\![a']\!]_{st} \\
\mathcal{A}[\![x]\!]_{st} &\triangleq st\ x
\end{aligned}$$

In this evaluation function, we can see that the expression reduces each time until reaching either a variable or a natural number, at which point it evaluates to the value of this expression.

$$
\begin{aligned}
\mathcal{B}[\![\_]\!]\_ &: \mathbf{Bexp} \to \mathbf{State} \to \mathbf{Bool} \\
\mathcal{B}[\![tt]\!]_{st} &\triangleq \texttt{true} \\
\mathcal{B}[\![ff]\!]_{st} &\triangleq \texttt{false} \\
\mathcal{B}[\![a = a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} = \mathcal{A}[\![a']\!]_{st} \\
\mathcal{B}[\![a \mathrel{!}= a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} \neq \mathcal{A}[\![a']\!]_{st} \\
\mathcal{B}[\![a < a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} < \mathcal{A}[\![a']\!]_{st} \\
\mathcal{B}[\![a > a']\!]_{st} &\triangleq \mathcal{A}[\![a]\!]_{st} > \mathcal{A}[\![a']\!]_{st} \\
\mathcal{B}[\![\neg b]\!]_{st} &\triangleq \neg \mathcal{B}[\![b]\!]_{st} \\
\mathcal{B}[\![b \wedge b']\!]_{st} &\triangleq \mathcal{B}[\![b]\!]_{st} \wedge \mathcal{B}[\![b']\!]_{st}
\end{aligned}
$$

The boolean evaluation function, much like the arithmetic evaluation function, recursively evaluates the expressions from values in syntax to values we can use in semantics.

2.3.4 Big-step semantics

This thesis uses natural semantics, we will define a judgement using the following syntax:

$$\langle c, st, h \rangle \to \langle st', h' \rangle$$

In this judgement, we have a c which is a command, a starting st, state and h, heap. We imply that the command terminates, which then will reach the final state and heap. By choosing the smallest relation which satisfies the inference rules, we can define how the commands mutate the state and heap implying that they terminate.

$$\text{LOAD}_{NS}\ \frac{h\ (st\ x') = \textbf{Some}\ n}{\langle x = *(x'), st, h\rangle \to \langle st[x \mapsto n], h\rangle}$$

$$\text{STORE}_{NS}\ \frac{h\ (st\ x) = \textbf{Some}\ n}{\langle *(x) = a, st, h\rangle \to \langle st, h[st\ x \mapsto \textbf{Some}\ \mathcal{A}[\![a]\!]_{st}]\rangle}$$

$$\text{ASSIGN}_{NS}\ \langle x \coloneqq a, st, h\rangle \to \langle st[x \mapsto \mathcal{A}[\![a]\!]_{st}], h]\rangle \qquad \text{SKIP}_{NS}\ \langle \texttt{skip}, st, h\rangle \to \langle st, h\rangle$$

$$\text{SEQ}_{NS}\ \frac{\langle c_1, st, h\rangle \to \langle st', h'\rangle \quad \langle c_2, st', h'\rangle \to \langle st'', h''\rangle}{\langle c_1; c_2, st, h\rangle \to \langle st'', h''\rangle}$$

$$\text{IF}^{tt}_{NS}\ \frac{\langle c_1, st, h\rangle \to \langle st', h'\rangle \quad \mathcal{B}[\![b]\!]_{st} = \texttt{true}}{\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, st, h\rangle \to \langle st', h'\rangle}$$

$$\text{IF}^{ff}_{NS}\ \frac{\langle c_2, st, h\rangle \to \langle st', h'\rangle \quad \mathcal{B}[\![b]\!]_{st} = \texttt{false}}{\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, st, h\rangle \to \langle st', h'\rangle}$$

$$\text{MALLOC}_{NS}\ \frac{h\ l = \textbf{None}}{\langle x = \texttt{malloc()}, st, h\rangle \to \langle st[x \mapsto l], h[l \mapsto \textbf{Some}\ 0]\rangle}$$

$$\text{FREE}_{NS}\ \frac{h\ (st\ x) = \textbf{Some}\ n}{\langle \texttt{free(}x\texttt{)}, st, h\rangle \to \langle st, h[st\ x \mapsto \textbf{None}]\rangle}$$

Note that there is not an error rule, this is because when error is seen, the program terminates abruptly.

In the load rule, the command updates the state to contain the value of x' at the variable x. For store, this happens on the heap, at the location of the evaluation of x, instead of the string x. The skip rule does not edit the heap or state. The seq and if rules are similar to other languages, seq uses a semicolon to separate two consecutive commands. The if rule will branch to c_1 if the boolean evaluation of b is true. While it will branch to c_2 if the boolean evaluation of b is false.

The malloc rule does not contain any bit or bytesize as its argument, due to a simplification. In this thesis we will assume that malloc will allocate enough space automatically as this has a negligible influence on the code

generated by the synthesizer. Here, we will also initialize the value of the location that was given by malloc to be 1. The free rule will free the heap of the value stored at the location 'st x'.

2.4 Semantics of separation logic assertions

This section will introduce the separation logic assertions without any of the necessary modifications for synthesis.

2.4.1 Syntax extended

The following syntax combines the state system from standard Hoare logic with a new type of heap predicates such that it can express the state of the heap and the boolean expressions at the same time. This is not a complete version of separation logic as this does not contain the magic wand operator as this is not necessary for this thesis.

$$P, Q \in \textbf{SymbolicHeap} ::= b \mid emp \mid a \mapsto a' \mid P * Q$$

A separation logic triple in this syntax would then be defined as:

$$\{P\}\, c\, \{Q\}$$

Here, we have a symbolic heap (P, Q), which can either be a boolean expression or an expression signifying what is present in the heap. The heap expression can be empty. It can be an assignment, stating that at the location which is at the value of the evaluation of the arithmetic expression a, contains the value of the evaluation of the arithmetic expression a'. Lastly, it can be a separating conjunction between two symbolic heaps. The assertion is currently only syntactic sugar. In the next chapter we will split the symbolic heap into parts and recombine the parts through the assertion datatype.

2.4.2 Evaluation functions for separation logic

To be able to define the evaluation function for the symbolic heap, we will now also use the heap in the evaluation call.

$$\mathcal{S}[\![\_]\!]_{\_,\_} : \textbf{SymbolicHeap} \to \textbf{State} \to \textbf{Heap} \to \textbf{Prop} \tag{2.1}$$

$$\mathcal{S}[\![b]\!]_{st,h} \triangleq \mathcal{B}[\![b]\!]_{st} = \texttt{true} \wedge h = \emptyset \tag{2.2}$$

$$\mathcal{S}[\![emp]\!]_{st,h} \triangleq h = \emptyset \tag{2.3}$$

$$\mathcal{S}[\![a \mapsto a']\!]_{st,h} \triangleq h = (\mathcal{A}[\![a]\!]_{st}, \mathcal{A}[\![a']\!]_{st}) \tag{2.4}$$

$$\mathcal{S}[\![P * Q]\!]_{st,h} \triangleq \exists_{h_1,h_2}, \mathcal{S}[\![P]\!]_{st,h_1} \wedge \mathcal{S}[\![Q]\!]_{st,h_2} \wedge h = h_1 \cup h_2 \wedge h_1 \cap h_2 = \emptyset \tag{2.5}$$

Here, we can see how the separating conjunction works. In the base cases, we see that the heap should either be a boolean expression, empty or contain a singleton. However, when we look at the recursive case, we see that the recursive call contains two new sub heaps of the original heap. These heaps are disjoint and their union is our original heap.

We can also note that we can now construct a symbolic heap while keeping the functionality of the underlying Hoare logic. This is due to us writing that the heap has to be empty in the case of b. If h was not empty in b, we could not construct the following valid symbolic heap: $\{x = n * y \mapsto a\}$ This symbolic heap has a separating conjunction between the Hoare expressions and the separation logic expressions. Due to (2.2) in the evaluation of the symbolic heap, the conjunction transforms into the normal boolean conjunction when we evaluate the separating conjunction where the heap is empty in h_1 and thus $h_2 = h$.

2.5 Inference system

2.5.1 Substitution functions

To be able to define the inference system, we need to be able to substitute variables into our **SymbolicHeap**. Which means that we need to be able to substitute into **Aexp** and **Bexp**. We will first define the syntax for a substitution and then we will define the substitution function for each of the evaluation functions.

$$a[x \mapsto a']$$

In this evaluation function, we see that we want to substitute x to be the value of the expression a'. Next, we will define how this will work semantically.

$$\_[\_ \mapsto \_] : \mathbf{Aexp} \to \mathbf{Variable} \to \mathbf{Aexp} \to \mathbf{Aexp}$$

$$n[x \mapsto a] \triangleq n$$

$$(a + a')[x \mapsto a''] \triangleq a[x \mapsto a''] + a'[x \mapsto a'']$$

$$(a - a')[x \mapsto a''] \triangleq a[x \mapsto a''] - a'[x \mapsto a'']$$

$$(a \cdot a')[x \mapsto a''] \triangleq a[x \mapsto a''] \cdot a'[x \mapsto a'']$$

$$x[y \mapsto a] \triangleq \begin{cases} a & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

Next, the Boolean Expression substitution function:

$$\begin{aligned}
_[_ \mapsto _] &: \mathbf{Bexp} \to \mathbf{Variable} \to \mathbf{Aexp} \to \mathbf{Bexp} \\
tt[x \mapsto a'] &\triangleq tt \\
ff[x \mapsto a'] &\triangleq ff \\
(a = a')[x \mapsto a''] &\triangleq a[x \mapsto a''] = a'[x \mapsto a''] \\
(a \mathrel{!}= a')[x \mapsto a''] &\triangleq a[x \mapsto a''] \mathrel{!}= a'[x \mapsto a''] \\
(a < a')[x \mapsto a''] &\triangleq a[x \mapsto a''] < a'[x \mapsto a''] \\
(a > a')[x \mapsto a''] &\triangleq a[x \mapsto a''] > a'[x \mapsto a''] \\
(\neg b)[x \mapsto a] &\triangleq \neg b[x \mapsto a] \\
(b \wedge b')[x \mapsto a] &\triangleq b[x \mapsto a] \wedge b'[x \mapsto a]
\end{aligned}$$

Lastly, the substitution function for **SymbolicHeap**:

$$\begin{aligned}
_[_ \mapsto _] &: \mathbf{SymbolicHeap} \to \mathbf{Variable} \to \\
&\qquad \mathbf{Aexp} \to \mathbf{SymbolicHeap} \\
b[x \mapsto a] &\triangleq b[x \mapsto a] \\
emp[x \mapsto a] &\triangleq emp \\
(a \mapsto a')[x \mapsto a''] &\triangleq a[x \mapsto a''] \mapsto a'[x \mapsto a''] \\
(P * Q)[x \mapsto a] &\triangleq P[x \mapsto a] * Q[x \mapsto a]
\end{aligned}$$

We can see that this function trickles down until it gets to the variables, where it will substitute the value of x if it is the same variable name.

2.5.2 Entailment under assertions

We will also need to be able to define entailment on assertions, which we can define as follows:

$$\begin{aligned}
_ \implies _ &: \mathbf{SymbolicHeap} \to \mathbf{SymbolicHeap} \to \mathbf{Prop} \\
P \implies Q &\triangleq \forall_{st,h}, \mathcal{S}[\![P]\!]_{st,h} \implies \mathcal{S}[\![Q]\!]_{st,h}
\end{aligned}$$

2.5.3 Derivable

To define the inference system that states whether a triple is derivable, we need to be able to get the variables in a symbolic heap to check if a variable is being used. We will define a function `vars` and `EV`. The `vars` function will return all the variables in a **SymbolicHeap** into a set. The `EV` function, which stands for environment variables, will do the same but for an **Assertion**, which will be used next chapter.

LOAD

$$\vdash \{(y \mapsto n * P)[x \mapsto n]\}\ x = *(y)\ \{y \mapsto n * P\}$$

STORE

$$\vdash \{x \mapsto a * P\}\ *(x) = a'\ \{x \mapsto a' * P\}$$

ASSIGN

$$\vdash \{P[x \mapsto a]\}\ x := a\ \{P\}$$

SKIP

$$\vdash \{P\}\ \texttt{skip}\ \{P\}$$

INCONSISTENCY

$$\frac{P \implies \mathit{ff}}{\vdash \{P\}\ \texttt{error}\ \{Q\}}$$

IF

$$\frac{\vdash \{P * b\}\ c_1\ \{Q\} \quad \vdash \{P * \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}$$

SEQ

$$\frac{\vdash \{P\}\ c_1\ \{Q\} \quad \vdash \{Q\}\ c_2\ \{R\}}{\vdash \{P\}\ c_1; c_2\ \{R\}}$$

MALLOC

$$\frac{x \notin \texttt{vars}(P)}{\vdash \{P\}\ x = \texttt{malloc()}\ \{P * x \mapsto 0\}}$$

FREE

$$\vdash \{P * x \mapsto n\}\ \texttt{free}(x)\ \{P\}$$

CONSEQUENCE

$$\frac{P \implies P' \quad \vdash \{P'\}\ c\ \{Q'\} \quad Q' \implies Q}{\vdash \{P\}\ c\ \{Q\}}$$

FRAME

$$\frac{\vdash \{P\}\ c\ \{Q\} \quad \texttt{vars}(c) \cap \texttt{vars}(R) = \emptyset}{\vdash \{P * R\}\ c\ \{Q * R\}}$$

The rules assign, skip, inconsistency, if, seq and consequence are standard Hoare logic rules Hoare [1969], while the others are standard separation logic rules Reynolds [2002]. We will first explain the Hoare logic rules where necessary and then all the separation logic rules.

The assign rule checks whether the precondition holds when the variable is substituted by the command, it is standard in Hoare logic to do the substitution on the precondition Hoare [1969]. The skip rule indicates that the precondition does not change. The inconsistency rule ensures that the precondition is entailed by ff, meaning the program terminates abruptly. Note that the if rule only needs one rule as for both branches of the if rule we get a derivable conclusion. This is due to either b or $\neg b$ evaluating to ff. The seq rule indicates that when we run 2 commands consecutively, we need the postcondition of the first command to be the precondition of the second command. Lastly, we have the consequence rule in the standard Hoare logic rules. This rule is special as it does not require any specific command to be present. This rule strengthens the precondition and weakens the postcondition, which is useful to transform boolean expressions in the pre- and postcondition.

The separation logic rules also work with the same substitution into the precondition as in Hoare logic [Hoare, 1969, Reynolds, 2002]. Which we can see in the load rule, which if we recall from the big step semantics, loads a value from a location on the heap into the state. In the store rule, we see that the value of the heap changes from a to a'. For the malloc rule, we need to know that the location is not yet used in the condition P, if this is the case, then we will create this value on the heap and initialize it to 0. For the free rule we do the opposite, we need to know in the precondition that there is a value at location x and then we erase it. Lastly, we have the frame rule, this rule states that if a condition is present in the pre- and postcondition and the condition does not have any variables which are in the command, we can safely remove this condition from both pre- and postcondition.

2.6 Validity and evaluation of a separation logic triple

We will now introduce the valid function; it ensures that the rules and reasoning methods you use in Hoare or separation logic accurately reflect the behaviour of the program. We will go into this function in more detail in the next chapter.

$$\models \_\_\_ : \mathbf{SymbolicHeap} \to \mathbf{Com} \to \mathbf{SymbolicHeap} \to \mathbf{Prop}$$

$$\begin{aligned} \models \{P\}\, c\, \{Q\} \triangleq \forall_{st \in \mathbf{State}, h \in \mathbf{Heap}}, \mathcal{S}[\![P]\!]_{st,h} \implies \\ \exists_{st' \in \mathbf{State}, h' \in \mathbf{Heap}}, \langle c, st, h\rangle \to \langle st', h'\rangle \wedge \mathcal{S}[\![Q]\!]_{st',h'} \end{aligned}$$

2.7 Soundness

Completeness and soundness are two possible properties that relate derivability and validity. Completeness is defined as: valid implies derivable. However, soundness is its implication in the opposite direction. We will prove that our inference system is sound, which can be used later to show that our synthesizer is always correct when giving an output program. Soundness in Hoare or separation logic means that if a triple is provably true within the logic, then it is also true in the actual execution of the program.

Theorem 1 (Soundness)**.** For all commands (c), pre- and postconditions (P, Q), if a separation logic is derivable, then it is valid.

$$\vdash \{P\}\, c\, \{Q\} \implies \models \{P\}\, c\, \{Q\}$$

This property is very useful for synthesis, as this is a theorem required to prove that we get valid code when the synthesizer gives code as output. We will not prove soundness for this base language, as in the next chapter, we will prove soundness for the full language.

2.8 Example

We will now evaluate a variant of the program used in the Hoare introduction.

Example 3.

$$
\begin{aligned}
&(1)\ x = \texttt{malloc()}\\
&(2)\ *(x) = 17\\
&(3)\ x_1 = *(x)\\
&(4)\ x_2 = *(y)\\
&(5)\ \texttt{if}\ x_1 < x_2\ \texttt{then}\ z \coloneqq x_2\ \texttt{else}\ z \coloneqq x_1\\
&(6)\ \texttt{free}(x)
\end{aligned}
$$

When we decorate this program, starting with the condition $\{y \mapsto n\}$ we can see the following behaviour:

$$
\begin{aligned}
&\{y \mapsto n\}\\
&(1)\ x = \texttt{malloc()}\\
&\{y \mapsto n * x \mapsto 0\}\\
&(2)\ *(x) = 17\\
&\{y \mapsto n * x \mapsto 17\}\\
&(3)\ x_1 = *(x)\\
&\{x = 17 * y \mapsto n * x \mapsto 17\}\\
&(4)\ x_2 = *(y)\\
&\{x_1 = 17 * x_2 = n * y \mapsto n * x \mapsto 17\}\\
&(5)\ \texttt{if}\ x_1 < x_2\ \texttt{then}\ z \coloneqq x_2\ \texttt{else}\ z \coloneqq x_1\\
&\{z \leq n * z \leq 17 * y \mapsto n * x \mapsto 17\}\\
&(6)\ \texttt{free}(x)\\
&\{z \leq n * z \leq 17 * y \mapsto n\}
\end{aligned}
$$

A decorated program is much like a tree, but for Hoare or Separation logic. We will use these to prove the validity of a program. Due to soundness, when a triple is derivable, it is valid.

Chapter 3

Separation logic for synthesis

In this chapter, we will change the syntax of assertions using the language and a simplified version of the semantics of Polikarpova and Sergey [2018]. We do these modifications by separating the boolean expressions from the symbolic heap in section 3.1. We will then change the required semantics in section 3.2 and the inference system in section 3.3. We then show validity and the soundness lemma again in section 3.4 after which we revisit the example again with the new syntax in section 3.5.

3.1 Syntax

This syntax combines the state system from standard Hoare logic with a different symbolic heap that can express the state of the heap and the boolean expressions separately. This is useful for synthesis as this splits the complex expression from standard separation logic into the separate types.

$$P, Q \in \textbf{SymbolicHeap} ::= emp \mid a \mapsto \langle a' \rangle \mid P * Q$$
$$\mathcal{P}, \mathcal{Q} \in \textbf{Assertion} ::= b, P$$

A separation logic triple in the new syntax would then be defined as:

$$\{\mathcal{P}\}\, c\, \{\mathcal{Q}\}$$

In this definition, we first have the precondition ($\mathcal{P}$), then we have the command to be run (c) and then we have the postcondition ($\mathcal{Q}$).

3.2 Semantics of separation logic assertions

In the semantics, we need to evaluate the assertions, as assertions are a new data type. Which means we also change the evaluation for a **SymbolicHeap** to not pattern match the boolean expression any more.

We start by changing the evaluation function of a **SymbolicHeap**:

$$\mathcal{S}[\![\_]\!]_{\_,\_} : \textbf{SymbolicHeap} \to \textbf{State} \to \textbf{Heap} \to \textbf{Prop}$$
$$\mathcal{S}[\![emp]\!]_{st,h} \triangleq h = \emptyset$$
$$\mathcal{S}[\![a \mapsto \langle a' \rangle]\!]_{st,h} \triangleq h = (\mathcal{A}[\![a]\!]_{st}, \mathcal{A}[\![a']\!]_{st})$$
$$\mathcal{S}[\![P * Q]\!]_{st,h} \triangleq \exists_{h_1,h_2}, \mathcal{S}[\![P]\!]_{st,h_1} \wedge \mathcal{S}[\![Q]\!]_{st,h_2} \wedge h = h_1 \cup h_2 \wedge h_1 \cap h_2 = \emptyset$$

We can see that the only difference from this and the previous chapter is that the boolean expression is now handled outside of the symbolic heap.

$$\mathcal{AS}[\![\_]\!]_{\_,\_} : \textbf{Assertion} \to \textbf{State} \to \textbf{Heap} \to \textbf{Prop}$$
$$\mathcal{AS}[\![b, P]\!]_{st,h} \triangleq \mathcal{B}[\![b]\!]_{st} = \texttt{true} \wedge \mathcal{S}[\![P]\!]_{st,h}$$

In the evaluation of **Assertion**, we now split off to the other evaluation functions as this definition is here to split up the previous **SymbolicHeap**.

3.3 Inference system

For the inference system, we need to change entailment and derivable as these both use the new syntax. We also need to add a substitution function for the new **Assertion** type.

$$\_[\_ \mapsto \_] : \textbf{Assertion} \to \textbf{Variable} \to \textbf{Aexp} \to \textbf{Assertion}$$
$$(b, P)[x \mapsto a] \triangleq b[x \mapsto a], P[x \mapsto a]$$

Just like the assertion evaluation function, this function calls the function for its arguments.

Entailment also needs to be on assertions instead of just the symbolic heaps.

$$\_ \implies \_ : \textbf{Assertion} \to \textbf{Assertion} \to \textbf{Prop}$$
$$\mathcal{P} \implies \mathcal{Q} \triangleq \forall_{st,h}, \mathcal{AS}[\![\mathcal{P}]\!]_{st,h} \implies \mathcal{AS}[\![\mathcal{Q}]\!]_{st,h}$$

The previous definition of section 2.4.2 for **SymbolicHeap** is now also introduced for **Assertion**. The only difference here is that we use **Assertion** instead of **SymbolicHeap**.

LOAD

$$\vdash \{(b, y \mapsto n * P)[x \mapsto n]\}\ x = *(y)\ \{b, y \mapsto n * P\}$$

STORE

$$\vdash \{b, x \mapsto a * P\}\ *(x) = a'\ \{b, x \mapsto a' * P\}$$

ASSIGN

$$\vdash \{\mathcal{P}[x \mapsto a]\}\ x \coloneqq a\ \{\mathcal{P}\}$$

SKIP

$$\vdash \{\mathcal{P}\}\ \texttt{skip}\ \{\mathcal{P}\}$$

INCONSISTENCY

$$\frac{\{b, emp\} \implies \{\mathit{ff}, emp\}}{\vdash \{b, P\}\ \texttt{error}\ \{b, Q\}}$$

IF

$$\frac{\vdash \{b \wedge b', P\}\ c_1\ \{b'', Q\} \quad \vdash \{b \wedge \neg b', P\}\ c_2\ \{b'', Q\}}{\vdash \{b, P\}\ \texttt{if}\ b'\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{b'', Q\}}$$

SEQ

$$\frac{\vdash \{\mathcal{P}\}\ c_1\ \{\mathcal{Q}\} \quad \vdash \{\mathcal{Q}\}\ c_2\ \{\mathcal{R}\}}{\vdash \{\mathcal{P}\}\ c_1; c_2\ \{\mathcal{R}\}}$$

MALLOC

$$\frac{x \notin \texttt{vars}(P)}{\vdash \{b, P\}\ x = \texttt{malloc()}\ \{b, P * x \mapsto 0\}}$$

FREE

$$\vdash \{b, P * x \mapsto n\}\ \texttt{free}(x)\ \{b, P\}$$

CONSEQUENCE

$$\frac{\mathcal{P} \implies \mathcal{P}' \quad \vdash \{\mathcal{P}'\}\ c\ \{\mathcal{Q}'\} \quad \mathcal{Q}' \implies \mathcal{Q}}{\vdash \{\mathcal{P}\}\ c\ \{\mathcal{Q}\}}$$

FRAME

$$\frac{\vdash \{b, P\}\ c\ \{b', Q\} \quad \texttt{vars}(c) \cap \texttt{vars}(R) = \emptyset}{\vdash \{b, P * R\}\ c\ \{b', Q * R\}}$$

Most rules are understandable from the previous explanation, there is only the inconsistency rule which is significantly different. The inconsistency rule can no longer use entailment on the entire **SymbolicHeap** like before, now it creates a new **Assertion** by using b and emp. These are combined into an **Assertion** and then we use the entailment function on this to get the same functionality.

3.4 Valid and evaluation of separation logic triples

To be able to define when a triple is valid, we will need to evaluate the assertion to a Prop. We can do this with all the evaluation functions we have defined in previous sections and preliminaries. The separation logic triple will be valid iff:

$$\models \_\_\_ : \textbf{Assertion} \to \textbf{Com} \to \textbf{Assertion} \to \textbf{Prop}$$
$$\models \{\mathcal{P}\}\, c\, \{\mathcal{Q}\} \triangleq \forall_{st \in \textbf{State}, h \in \textbf{Heap}}, \mathcal{AS}[\![\mathcal{P}]\!]_{st,h} \implies$$
$$\exists_{st' \in \textbf{State}, h' \in \textbf{Heap}}, \langle c, st, h\rangle \to \langle st', h'\rangle \wedge \mathcal{AS}[\![\mathcal{Q}]\!]_{st',h'}$$

3.4.1 Total vs. Partial example

There are two different ways to define valid for a triple, we call these either partial or total correctness. We have defined total correctness. The reason this is important for synthesis is due to termination. If we chose partial correctness, the judgement would be allowed not to terminate. If we then use a synthesizer for any pre- and postcondition, we can generate the loop `while true do skip`. This is not possible in my language due to the simplifications, but we should still account for it.

3.4.2 Soundness with new syntax

Due to the new definitions, we also need to restate our soundness with the new syntax. This will be proven through Coq in the next chapter and we will show parts of the code.

Theorem 2 (Soundness)**.** For all commands (c), pre- and postconditions ($\mathcal{P}, \mathcal{Q}$), if a separation logic is derivable, then it is valid.

$$\vdash \{\mathcal{P}\}\, c\, \{\mathcal{Q}\} \implies \models \{\mathcal{P}\}\, c\, \{\mathcal{Q}\}$$

3.5 Example with new syntax

(1) $x =$ `malloc()`
(2) $*(x) = 17$
(3) $x_1 = *(x)$
(4) $x_2 = *(y)$
(5) `if` $x_1 < x_2$ `then` $z \coloneqq x_2$ `else` $z \coloneqq x_1$
(6) `free(`x`)`

In this program, we first use malloc for a variable x, we store a value on the heap. We then assign 2 variables to be the value of 2 different locations on the heap, after which we calculate the max of these 2 variables and store the value in the variable z.

$\{y \mapsto n\}$
(1) $x =$ `malloc()`
$\{y \mapsto n * x \mapsto 0\}$
(2) $*(x) = 17$
$\{y \mapsto n * x \mapsto 17\}$
(3) $x_1 = *(x)$
$\{x = 17, y \mapsto n * x \mapsto 17\}$
(4) $x_2 = *(y)$
$\{x_1 = 17 \wedge x_2 = n, y \mapsto n * x \mapsto 17\}$
(5) `if` $x_1 < x_2$ `then` $z \coloneqq x_2$ `else` $z \coloneqq x_1$
$\{z \leq n \wedge z \leq 17, y \mapsto n * x \mapsto 17\}$
(6) `free(`x`)`
$\{z \leq n \wedge z \leq 17, y \mapsto n\}$

This example applies all the inference rules for each step, we also use the consequence rule between line 5 and 6, these steps are not shown. Note that we leave out the boolean expression in the upper three conditions, this means that the expression only consists of `true`.

Chapter 4

Soundness for synthesis in Coq

In this chapter, we will walk through the code written in Coq and explain the concepts needed to understand the code. We will have a section for the syntax, evaluation functions, semantics, the inference system, soundness and our special frame rule. We will choose parts of the code which are using new concepts and leave out the parts which repeat definitions.

4.1 functional

Coq is a functional language, like Haskell and Scala. But unlike Haskell and Scala, Coq is a proof assistant. This is useful to us as this allows for easy algebraic data types and pattern matching. A proof assistant can introduce lemmas, inductive properties and prove lemmas, which Haskell and Scala cannot Coq Development Team 1998.

4.2 Implementation

We will show our implementation of the language in Polikarpova and Sergey [2018] and show parts of the proof for soundness to introduce the concepts used in Coq.

4.2.1 Syntax

In Coq, you can introduce types and syntax definitions with an inductive type. We use these for all the syntax and we will show an example of how we used this to define **Com**. The required **Aexp** and **Bexp** are defined similarly, while string is used to represent the variables.

```
Inductive Com :=
 | Load : string -> string -> com
```

```
 | Store : string -> aexp -> com
 | Assign : string -> aexp -> com
 | Skip : com
 | Error : com
 | Seq : com -> com -> com
 | If : bexp -> com -> com -> com
 | Malloc : string -> com
 | Free : string -> com.
```

As can be seen in the example above, the inductive type **Com** contains all the commands as previously given in section 2.3.1. Next, we will use the Definition command to define the types of the state and heap.

```
Definition state := StringMap.finmap nat.
Definition heap := NatMap.finmap nat.
```

Notably, we use a library of Krebbers and Jacobs [2024] for the finite maps used for state and heap. These can be compared with the collection of finite functions used for the heap in section 2.3.2. Due to also using the library for the state, the state will also be returning an optional value. Which is why we normalise the state's return value and we create Notation for both heaps and states s.t. this will not be an issue.

```
Notation "st '//' x" :=
 match (StringMap.lookup st x) with
 | Some y => y
 | None => 0
 end (at level 80).
Notation "h '///' x" := (NatMap.lookup h x) (at level 80).
```

The Notation command can create a notation for the user, for which the definition under the notation can be unfolded during proofs same as before. In this example, we have the notation for state and heap lookup. In the state lookup, when we receive a **None**, we will map this to 0.

4.2.2 Evaluation

In Coq, for any non-recursive function, you have to use Definition. If the function is recursive, then you have to use Fixpoint. One such example is the evaluation of the arithmetic expression.

```
Fixpoint aeval (st : state) (a : aexp) : nat :=
 match a with
 | ANum n => n
 | APlus a1 a2 => (aeval st a1) + (aeval st a2)
 | AMinus a1 a2 => (aeval st a1) - (aeval st a2)
 | AMult a1 a2 => (aeval st a1) * (aeval st a2)
 | AVar s => st // s
 end.

Definition bexp_true (b : bexp) : Prop :=
  forall st, Is_true (beval st b).
```

In the recursive definition aeval, we can see that we get the same evaluation rules as in section 2.3.3. We have a type of expression denoted with the names in the front and we match the expression a based on its form.

The non-recursive bexp_true uses a similar definition for beval. It denotes that for every state the boolean the function Is_true holds. This means that b evaluates to true for every state.

4.2.3 Semantics

There is another way of using inductive types, which is using them to denote derivation rules. We use this construction to show both the bigstep rules and the inference rules, derivable.

```
Inductive big_step : com -> heap -> state -> heap -> state ->
    Prop :=
 | E_Load :   forall n st h x y ,
              (h /// (st // y)) = Some n ->
              big_step (Load x y) h st h (x !-> n;st)
 | E_Store :  forall n st h h' x a,
              (h /// (st // x)) = Some n ->
              h' = ((st // x) !->> (aeval st a);h) ->
              big_step (Store x a) h st h' st
 | E_Assign : forall st st' h x a,
              st' = (x !-> (aeval st a);st) ->
              big_step (Assign x a) h st h st'
 | E_Skip :   forall st h,
              big_step Skip h st h st
 | E_IfTrue : forall st st' h h' b c1 c2,
              beval st b = true ->
              big_step c1 h st h' st' ->
              big_step (If b c1 c2) h st h' st'
 | E_IfFalse :forall st st' h h' b c1 c2,
              beval st b = false ->
              big_step c2 h st h' st' ->
              big_step (If b c1 c2) h st h' st'
 | E_Seq :    forall st st' st'' h h' h'' c1 c2,
              big_step c1 h st h' st' ->
              big_step c2 h' st' h'' st'' ->
              big_step (Seq c1 c2) h st h'' st''
 | E_Malloc : forall st st' h h' l x,
              l = NatMap.fresh h ->
              h' = (l !->> 0;h) ->
              st' = (x !-> l;st) ->
              big_step (Malloc x) h st h' st'
 | E_Free :   forall n st h h' x,
              (h /// (st // x)) = Some n ->
              h' = NatMap.delete (st // x) h ->
              big_step (Free x) h st h' st.

Notation "'(' st ',' h ')' '=[' c ']=>' '(' st' ',' h' ')'" :=
    (big_step c h st h' st').
```

In this definition, we have translated the definitions from section 2.3.4. We will look at the Malloc rule, in the rule it states that the variable x has to not be present in the current heap. In the code, this can be seen as the location assigned to x has to be fresh. The implementation of freshness is taken from the library and ensures that l is the next free location.

4.2.4 Inference system

We then implement helper functions such as vars and substitution recursively for each datatype. These are not shown as these are the same as in section 2.5.1. Thus, we will show derivable and valid.

```
Inductive derivable : assertion -> com -> assertion -> Prop :=
 | S_Load :   forall x y b sh a,
              (x =? y)%string = false ->
              ~elem_of x (vars_aexp a) ->
              derivable
                  (subst
                    (Assert b (Separate (Points_to (AVar y) a)
                        sh))
                    x a)
                  (Load x y)
                  (Assert b (Separate (Points_to (AVar y) a) sh
                     ))
 | S_Store:   forall x a a' b sh,
              derivable
                  (Assert b (Separate (Points_to (AVar x) a) sh
                     ))
                  (Store x a')
                  (Assert b (Separate (Points_to (AVar x) a')
                     sh))
 | S_Assign : forall b sh x a,
              derivable
                  (subst (Assert b sh) x a)
                  (Assign x a)
                  (Assert b sh)
 | S_Skip :   forall b b' sh,
              bexp_imp b b' ->
              derivable (Assert b sh) Skip (Assert b' sh)
 | S_Error :  forall P b sh,
              bexp_true (BNot b) ->
              derivable (Assert b sh) Error P
 | S_If   :   forall Q b sh c1 c2 b',
              derivable (Assert (BAnd b b') sh) c1 Q ->
              derivable (Assert (BAnd (BNot b) b') sh) c2 Q ->
              derivable (Assert b' sh) (If b c1 c2) Q
 | S_Seq  :   forall P c1 Q c2 R,
              derivable P c1 Q ->
              derivable Q c2 R ->
              derivable P (Seq c1 c2) R
 | S_Malloc:  forall b x sh,
              is_free x (Assert b sh) ->
```

```
                derivable
                    (Assert b sh)
                    (Malloc x)
                    (Assert b (Separate (Points_to (AVar x) (ANum
                        0)) sh))
 | S_Free :    forall b x a sh,
                derivable
                    (Assert b (Separate (Points_to (AVar x) a) sh
                       ))
                    (Free x)
                    (Assert b sh)
 | S_Conseq : forall (P Q P' Q' :assertion) c,
                derivable P' c Q' ->
                P ->> P' ->
                Q' ->> Q ->
                derivable P c Q.
```

Note that we do not have a frame rule. This is due to the observation that the frame rule can be proven as a lemma. This lemma will be shown in section 4.3.1. Next to this, to be able to prove soundness, we needed to ensure that the load function does not change heap while doing its substitution into the state, which is why we have the two extra predicates. The other rules are implemented in the same way as shown in section 2.5.3.

4.3 Soundness

We will need the valid function again. Which will be the exact definition of valid as shown in section 3.4.

```
Definition valid (P : assertion) (c : com) (Q : assertion) :
                                                          Prop :=
  forall st h,
    interp st  P h ->
    exists st' h', (st,h) =[ c ]=> (st',h') /\ interp st' Q h'.
```

To prove soundness, we have created definitions for each sub-goal created by the induction step in the soundness proof. This makes the proof a lot more readable and we will show one of the sub lemmas as an explanation of its correctness.

```
Theorem soundness : forall P c Q,
  derivable P c Q ->
  valid P c Q.
Proof.
  intros. induction H.
  - eapply sep_load.
  - eapply sep_store.
    (*More of the same*)
Qed.
```

This shows that for each case of the resulting goals, we have a definition

sep_command which proves its own case. We will show sep_store and the required lemma, interp_heap_sub, to prove it.

```
Lemma interp_heap_sub : forall st x a a' h1 h2 sh,
  interp_heap st (Points_to (AVar x) a) h1 ->
  interp_heap st sh h2 ->
  NatMap.disjoint h1 h2 ->
  interp_heap st (Separate (Points_to (AVar x) a') sh)
  (st // x !->> (aeval st a');NatMap.union h1 h2).

Theorem sep_store : forall x a a' b sh,
     valid
       (Assert b (Separate (Points_to (AVar x) a) sh))
       (Store x a')
       (Assert b (Separate (Points_to (AVar x) a') sh)).
```

The lemma, interp_heap_sub, is required to add **SymbolicHeap** to **Assertion** while keeping intact the command interp. The command interp is the evaluation of an assertion, which is required to stay consistent with this change. This holds because the value is separate from the original heap, and thus does not change the evaluation.

4.3.1 Frame rule

```
Lemma S_Frame_help :  forall sh c P Q,
  vars_overlap sh c ->
  derivable P c Q ->
  derivable (add_sh P sh) c (add_sh Q sh).

Lemma S_Frame :  forall b b' sh sh' sh'' c,
              vars_overlap sh'' c ->
              derivable (Assert b sh) c (Assert b' sh') ->
              derivable (Assert b (Separate sh sh'')) c (Assert
                  b' (Separate sh' sh'')).
```

The frame rule has been defined as a lemma instead. The consequence of this is that this is not part of the soundness proof and thus does not require the traditional lemmas for its proof. The frame lemma, as seen above, cleverly leverages the structure of our derivability rules. These rules are very generalised. Thus, if a shared pre- and postcondition does not contain variables used in the command, it can be taken out of the frame.

We prove the frame lemma by using a helper lemma, this helper lemma was solved by using induction. Similar to the soundness proof, this proof is quite large. However, this proof would be easier to follow than the proof if this were the standard frame rule.

The helper lemma is using an extra definition add_sh which adds a symbolic heap to an assertion. The reason we needed the helper lemma is due to the induction step. The induction hypothesis would be too weak in the S_Frame lemma.

Chapter 5

Related Work

Separation logic

The seminal work by Reynolds [2002] has introduced the first framework of separation logic. This framework was made to introduce a logic system which handles shared mutable data structures. Other key insights given in this framework is the magic wand operator (separating implication) which can be used to generate the weakest precondition. The key differences in our implementation and the implementation in Reynolds [2002] are the frame rule and the looping constructions. The frame rule is proven through the use of 4 different lemmas about the semantics and judgements. In our implementation the frame rule is not a rule, but a lemma by itself. The looping constructions are taken out in our implementation whilst this is present in the work by Reynolds [2002].

Separation logic has been used for many different use cases. In Appel and Blazy [2007] separation logic is used to formalize the language C minor. C minor is a mid-level imperative programming language below C. What is interesting in this paper is that the formalised semantics are shown to be sound while containing the functions: `break`, `continue` and `return`. These functions are not present in our implementation nor in what our implementation is based on.

Another implementation difference happens in Brookes [2007], Jung et al. [2018], where these both implement concurrency in their implementation of separation logic. In both of these papers, soundness is proven, while being capable of concurrency. These logic systems could be useful for code synthesis as they introduce handling of shared variables.

Instead of proving soundness, it is also possible to prove completeness of separation logic as was done in de Boer et al. [2023]. Completeness is a property which would be useful, as this means that for every valid program specification given to the synthesiser, a derivation can be made. These derivations are sequences of commands. Which means that with both soundness and completeness of the inference system, we know that every

valid specification can derive a valid program and every valid derivation has to be valid for the specification given.

Separation logic in Coq

Many different versions of separation logic have also been implemented in the Coq proof assistant, for example Appel and Blazy [2007], Charguéraud [2020], Chlipala [2011], de Boer et al. [2023], Jung et al. [2018]. These versions all differ from our implementation. In Charguéraud [2020], one of the key feature differences is that it focusses on sequential execution, unlike in Brookes [2007]Jung et al. [2018]. These implementations all prove soundness and there are some key differences in each implementation. One of the big differences are the different types of correctness and a concept called frame baking Birkedal et al. [2008][Schwinghammer et al., 2009][p. 12]. In our implementation, we have implemented total correctness because we would need this for synthesis. Frame baking is a new technique which bakes in the frame rule into each normal rule, which means it does not have to be proven in the traditional way as done in Reynolds [2002]. All these versions are very different from our own framework, these works all focus one or a few different concepts to model. We have chosen the direction of code synthesis, while most of these works did not have that in mind.

Another use for separation logic in coq is the automatic verification of programs Chlipala [2011]. This paper mostly automates the verification of low level programs. A similar tactic could be used to verify programs produced by a synthesiser.

There are also tools made in Coq to introduce a standardised model for Separation logic proofs in Jung et al. [2018], Krebbers et al. [2018, 2017]. These systems are very different from the implementation of separation logic we have. The Iris implementation contains all the separation logic rules, it can reason about higher order logic as well as concurrency. Another implementation difference is that it cannot only do partial correctness, but supports other types of reasoning as well Krebbers et al. [2017].

Synthesis through separation logic

Our research was based on the work in Polikarpova and Sergey [2018]. We have made many simplifications to the language used in Polikarpova and Sergey [2018]. We have removed function calls, scopes and negative integers from the syntax. These simplifications have made it easier to model in Coq, but are possible to be implemented in the future. Both Polikarpova and Sergey [2018], Watanabe et al. [2021] use separation logic for synthesis and Watanabe et al. [2021] then verifies that the synthesiser gives correct code by giving a Coq proof script which has to be run afterwards by the user.

Verification and certification are then highly important to research for

synthesisers. A synthesiser cannot be verified to work for every specification given beforehand as there are specifications which exists that are mathematically correct but impossible to derive. Instead, we can certify our synthesiser and develop a metric for how good the synthesiser has worked as done in Watanabe et al. [2021].

Chapter 6

Conclusion and Future work

In this thesis, we have implemented a simplified version of the language in Polikarpova and Sergey [2018] and have proven this to be sound in the Coq proof assistant.

In the future, we would like to implement a synthesiser into Coq and certify the synthesiser for some number of general pointer programs. We will have to do this by also implementing an SMT solver in Coq as this does not exist in the standard library of Coq. We would like to implement an SMT solver and synthesiser without all the simplifications made in this thesis. If a synthesiser is implemented, it would be necessary to test over a large set of pre- and postconditions to ensure functionality.

Bibliography

Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007. doi: 10.1007/978-3-540-74591-4_3. URL https://doi.org/10.1007/978-3-540-74591-4_3.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. A simple model of separation logic for higher-order store. pages 348–360. Springer, 2008. Serie: Lecture Notes in Computer Science, Springer, 0302-9743, 1611-3349 Volumne: 5126/2008; ICALP 2008 35th International Colloquium on Automata, Languages and Programming ; Conference date: 06-07-2008 Through 13-07-2008.

Arvid Bonten. Soundness proof script in coq, 2025. URL https://gitlab.science.ru.nl/abonten/coq-repo-thesis/-/tree/main?ref_type=heads.

Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007. doi: 10.1016/J.TCS.2006.12.034. URL https://doi.org/10.1016/j.tcs.2006.12.034.

Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP):116:1–116:34, 2020. doi: 10.1145/3408998. URL https://doi.org/10.1145/3408998.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu

Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL `https://arxiv.org/abs/2107.03374`.

Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 234–245. ACM, 2011. doi: 10.1145/1993498.1993526. URL `https://doi.org/10.1145/1993498.1993526`.

Frank S. de Boer, Hans-Dieter A. Hiep, and Stijn de Gouw. Dynamic separation logic. In Marie Kerjean and Paul Blain Levy, editors, *Proceedings of the 39th Conference on the Mathematical Foundations of Programming Semantics, MFPS XXXIX, Indiana University, Bloomington, IN, USA, June 21-23, 2023*, volume 3 of *EPTICS*. EpiSciences, 2023. doi: 10.46298/ENTICS.12297. URL `https://doi.org/10.46298/entics.12297`.

C A R Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969. doi: 10.1145/363235.363259. URL `https://doi.org/10.1145/363235.363259`.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi: 10.1017/S0956796818000151. URL `https://doi.org/10.1017/S0956796818000151`.

Robbert Krebbers and Jules Jacobs. A Coq library as part of the course "Program verification with types and logic" at radboud universiteit nijmegen. available at `https://gitlab.science.ru.nl/program-verification`, 2024.

Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi: 10.1145/3009837.3009855. URL `https://doi.org/10.1145/3009837.3009855`.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in

separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30, 2018. doi: 10.1145/3236772. URL https://doi.org/10.1145/3236772.

Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *SIGPLAN Not.*, 49(6):408–418, June 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594297. URL https://doi.org/10.1145/2666356.2594297.

Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *CoRR*, abs/1807.07022, 2018. URL http://arxiv.org/abs/1807.07022.

John C Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029817. URL https://doi.org/10.1109/LICS.2002.1029817.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested hoare triples and frame rules for higher-order store. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 2009. doi: 10.1007/978-3-642-04027-6_32. URL https://doi.org/10.1007/978-3-642-04027-6_32.

The Coq Development Team. *The Coq proof assistant reference manual (version 18.0)*, 1998. http://pauillac.inria.fr/coq/doc/main.html.

Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. Certifying the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 5:1–29, 2021. doi: 10.1145/3473589. URL https://doi.org/10.1145/3473589.