AN IMPLEMENTATION OF SEPARATION LOGIC IN COQ

YI WANG

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St.Catharines, Ontario

ABSTRACT

For certain applications, the correctness of software involved is crucial, particularly if human life is in danger. In order to achieve correctness, common practice is to gather evidence for program correctness by testing the system. Even though testing may find certain errors in the code, it cannot guarantee that the program is error-free. The program of formal verification is the act of proving or disproving the correctness of the system with respect to a formal specification. A logic for program verification is the so-called Hoare Logic. Hoare Logic can deal with programs that do not utilize pointers, i.e., it allows reasoning about programs that do not use shared mutable data structures. Separation Logic extends Hoare logic that allows pointers, including pointer arithmetic, in the programming language. It has four-pointer manipulating commands which perform the heap operations such as lookup, allocation, deallocation, and mutation. We introduce an implementation of separation logic in the interactive proof system Coq. Besides verifying that separation logic is correct, we will provide several examples of programs and their correctness proof.

TABLE OF CONTENTS

1 INTRODUCTION

This thesis introduces Separation Logic and the implementation of separation logic in *Coq*, where *CoqIDE* is a formal proof management system [1]. In the introduction for *Coq* Proof Assistant, This system provides "a formal language to write mathematical definitions, executable algorithms." According to John Done, those theorems together with an environment "for semi-interactive development of machine-checked proofs." If such logic is used in the system, it will help avoid human errors. People can always make mistakes whether it is separation logic or any other kind of proof on a piece of paper. A formal proof management system will not let it happened.

The implementation of separation logic in *Coq* is the extension of Hoare logic. However, the Separation Logic extended the predicate calculus with the separation operators. Separation Logic is a Program Logic that can help develop logically correct programs without the need for debugging. For the implementation, the Separation Logic is verified formally for correctness. In this thesis, we presented some proof rules in separation logic and correctness proof [2]. In Chapter two defines the syntax and semantics of the programming language and introduce the Hoare Logic [3]. Chapter three introduces the new forms of assertions and their inference rules in Separation Logic and the specifications and inference rules. The final chapter describes the idea of annotated proofs and presents the proof of Rules [4]. The whole logic is written in the style of the software foundations series, where every definition, lemma, and example is formalized using the *Coq* proof system.

For the language of *Coq*, it can be sorted into two categories: the *Prop* and *Type*. *Prop* is the sort for propositions which are the type of *Prop*, and the new predicates can be defined inductively called *inductive* type. By abstracting over other existing propositions we can have its *Definistion*. *Type* is for the datatypes and mathematical structures. Types can be inductive structures, for tuples or a form of subset types. *Coq* implements a functional programming language supporting these types. Then, functions over inductive types are expressed using a case analysis called *Fixpoint*. Proof development in *Coq* is done through a language of tactics that allows a user-guided proof process to finish the goal.

2 BACKGROUND

Programming verification uses formal, mathematical techniques to debug software and software specifications to avoid any serious consequences. In any application, the program always act in a conscionable manner. We will show that the program behaves correctly in all possible situations. Hoare Logic can be used but does not allow pointers. By using separation logic which can be a compelling program logic for proving programs that involve pointers. The basic idea of Hoare Logic and Separation Logic will be introduced in this chapter.

2.1 The Programming Language

In this chapter, we first define the syntax and a partial function on states that specify the operational semantics of programs. Since the separation logic has been built upon it, we will review the basis of Hoare Logic. In Hoare Logic, we have a rule for reasoning about the different syntactic forms of commands. Those are skip, assignment, sequencing, and conditionals.

In separation logic, it is extended with new commands for the manipulation of mutable shared data structures. It will be able to prove programs correct using these proof rules.

2.1.1 Syntax

An arithmetic expression and Boolean expression is needed to provide a suitable language. Starting with the arithmetic expression, we have variables, and certain operations apply to already existing expressions. Suppose X is a set of variables, then the set of the AExp arithmetic expression is recursively defined by the following rules:

1. Every variable is an arithmetic expression, i.e., X$\subseteq$ AExp.
2. Every integer constant is an arithmetic expression.
3. If $\mathrm{t}_1, t_2 \in$ AExpr, then $\mathrm{t}_1 + t_2 \in$ AExp
4. If $\mathrm{t}_1, t_2 \in$ AExpr, then $\mathrm{t}_1 - t_2 \in$ AExp
5. If $\mathrm{t}_1, t_2 \in$ AExpr, then $\mathrm{t}_1 * t_2 \in$ AExp

For expression, we have the basic datatype for variables and constant numbers. Then, numerical expression is included by introducing notation for addition, multiplication, and subtraction. Here x is a variable in *Var*, n an integer and an operator in (Val $\times$ Val) $\rightarrow$ Val. Such as *Plus*: (Int $\times$ Int) $\rightarrow$ Int, etc,.

Boolean Expression could be basically **True** or **False**. Then there are several comparison operations apply to arithmetic expression like:

1. Boolean Expression for simply true or false
2. If $\mathrm{t}_1, t_2 \in$ Expr, then $\mathrm{t}_1 =? t_2 \in$ Exp
3. If $\mathrm{t}_1, t_2 \in$ Expr, then $\mathrm{t}_1 <=? t_2 \in$ Exp
4. If b$\in$ BExpr, then $\sim (b) \in$ BExp
5. If $\mathrm{b}_1, b_2 \in$ BExpr, then $\mathrm{b}_1 =? b_2 \in$ BExp
6. If $\mathrm{b}_1, b_2 \in$ BExpr, then $\mathrm{b}_1 <=? b_2 \in$ BExp

The Boolean expression simplifies members **true** and **false**, in which it also has Boolean equal, Boolean less equal, Boolean not, Boolean and and Boolean or. Using & as example (Bool $\times$ Bool) $\rightarrow$ Boolean.

The programming language has the introductory statement with the programming language like Skip, Assignment, Sequence, If, While, Lookup, Mutation, Allocation, and Deallocation. The syntax as data type had defined in the implementation. For arithmetic expression and Boolean expression, we have a command. These are the exact data types and which is why we want to define mathematically first [4]. The set of commands COM is recursively defined by:

1. Skip$\in$ COM.

There will be no effect on the state of computation for the execution. Since the command Skip does not change the state, it will preserve any property P.

2. If x$\in$ COM and t$\in$ AExp, then the assignment statement x:=t$\in$ COM.

Assigning the value of the term t of expression to the variable x (x:=t) will change the state for this command.

3. If x$\in$ COM and t$\in$ AExp, then the lookup statement x:=[t]$\in$ COM.

The value we got is stored at location t in the variable x. For the execution of this command, location t must initialize by the previous command of this program. Otherwise, the execution will abort.

4. If $\mathrm{t}_1, t_2 \in$ AExp, then the mutation statement $[\mathrm{t}_1]$:=$\mathrm{t}_2 \in$ COM.

The command $[\mathrm{t}_1]$:= t_2, stores the value of expression t_2 at the location t_1. The location t_1 must be an active cell of the addressable memory to make it happened.

5. If x$\in$ COM t$\in$ cons(AExp), then the allocation statement x:=cons(t)$\in$ COM.

The command says x:=cons(t), the values of t_1,...t_n for n consecutive cells in the memory will be saved in x. The execution of this command expects that the addressable memory has n consecutive cells and the uninitialized cells available. And then, it will create a new *cons* cell in a heap and places a pointer to it in x.

6. If t$\in$ AExp, then the deallocation statement dispose(t)$\in$ COM.

The instruction *dispose(t)* says that deallocate the cell at the address t. The execution of this command will abort if t is not an active cell location.

7. If c_1, c_2 $\in$ COM, then the sequence statement (c_1;c_2)$\in$ COM.

The commands c_1, c_2 are executed in that order. If the command c_1 takes any state where P holds to a state where Q holds, and if c_2 takes any state where Q holds to one where R holds,then doing c_1 followed by c_2 will take any state where P holds to one where R holds: P c_1 Q; Q c_2 R, we get P c_1;;c_2 R.

8. If b$\in$ BExp, c_1, c_2 $\in$ COM, then the conditional statement if b then c_1 else c_2 fi$\in$ COM.

In this command, if the Boolean expression b evaluates to true statement, then c_1 is executed. Else if b evaluates to false, then c_2 is executed. It is a simple conditional statement.

9. If b$\in$ BExp, c$\in$ COM, then the while statement while b do c end $\in$ COM.

If b is a Boolean expression been evaluated to false then there is nothing done. However, if b evaluates to true, then c is executed, and this while command will be repeated. In other words, the command c will be repeated for execution until b becomes false.

2.1.2 Formal Semantics

The formal semantics can be specified to its commands in programming languages. On the state of the computation, two component is extent: a *store* and a *heap*.

The store is just the primary function from variable to values as in the semantics of the unextended simple imperative language. It contains the values of local variables. Furthermore, the heap is a partial function from the variable to the allocation storage, and (mapping addresses) into values represents the mutable structures. We can say both of them can be viewed as partial functions showing below:

$$\text{Heaps} \triangleq \mathit{Location} \rightarrow \text{Int} \qquad \mathit{Stores} \triangleq \mathit{Variables} \rightarrow \text{Int}$$

For the semantic domains we have:

$$Val = Int \cup Bool \cup Atoms \cup Loc$$

$$S = Var \rightharpoonup Val$$

$$H = Loc \rightharpoonup Val \times \text{Val}$$

The Loc = { l_1,l_2,...} presents an infinite set of locations. An infinite set of variables showing as Var ={x,y,...}. In addition, Atoms = {nil,a,...} is a set of atoms. Furthermore, $\rightharpoonup$ is for partial functions. An element S is referred to s ∈ S which is a store, h ∈ H is referred to as a heap, and then the pair (s,h) ∈ S × H is referred to as a state.

The dom(h) is used to denote the domain of definition of a heap h ∈ H, and *dom(s)* to denote the domain of a store s ∈ S.

For t∈ AExp, the semantics: Store→ Values is recursively defined by:

1) [x] (σ) = σ(x)

2) [c] (σ) = c

3) [t_1 + t_2] (σ) = [t_1] (σ) + [t_2] (σ)

4) [t_1 - t_2] (σ) = [t_1] (σ) - [t_2] (σ)

5) [t_1 * t_2] (σ) = [t_1] (σ) * [t_2] (σ)

We have the type expression and the store and the result values. The constructor for variable x would be σ (store) of x. If it is a constant c, then is value c. And then, we have the Plus expression t_1 and t_2, then would be the value of the expression t_1 with it store plus the value of the t_2 with it store. Same idea for the multiplication and subtraction.

For t∈ BExp, the semantics e: Store→ Bool is recursively defined by:

1) BTrue = true

2) BFalse = false

3) [t_1 , t_2] (σ) =? [t_1] (σ) , [t_2] (σ)

4) [t_1 , t_2] (σ) <=? [t_1] (σ) , [t_2] (σ)

5) [b]=∼(b)

6) [b_1 && b_2] (σ)=[b_1] (σ) && [b_2](σ)

7) [b_1 | | b_2] (σ)=[b_1] (σ) || [b_2] (σ)

We simply have *BTrue* for returning *ture* in Boolean expression, *BFalse* for rerturning false. Same idea we have the Boolean euqal, less equal, not, and and or for the expressions.

Operational Semantics (Commands without pointers):

- The semantics of the commands, from the original environment σ.
- Returning the modified environment σ'.
- The fact will be denoted by <c,σ> → σ'.
- The evaluation relation is called → symbol. This relation is defined by the following rules:

(Skip) <skip, σ> → σ

(Assignment) <x := a,σ> → σ[σ(a)/x]

(Sequencing)

$$\frac{<c_0,\sigma> \to \sigma'' \quad <c_1,\sigma''> \to \sigma'}{<c_0; c_1,\sigma> \to \sigma'}$$

(Conditional 1)

$$\frac{<c_0,\sigma> \to \sigma'}{<\text{if b then } c_0 \text{ else } c_1 \text{ fi}, \sigma> \to \sigma'}$$

iff $\models$ Z b[σ]

(Conditional 2)

$$\frac{<c_1,\sigma> \to \sigma'}{<\text{if b then } c_0 \text{ else } c_1 \text{ fi}, \sigma> \to \sigma'}$$

$$\text{iff} \models Z \;\; \neg\, b[\sigma]$$

(Loop 1)

$$<\text{while b do c od}, \sigma> \to \sigma \qquad \text{iff} \models Z \;\; \neg\, b[\sigma]$$

(Loop 2)

$$\frac{<c,\sigma> \to \sigma'' \quad <\text{while b do c od}, \sigma''> \to \sigma'}{<\text{while b do c od}, \sigma> \to \sigma'}$$

$$\text{iff} \models Z \;\; b[\sigma]$$

Operational Semantics (Commands with pointers):

- The notation, *cons* and [-] which refer to the heap memory.
- f[x : v] represents a function whic maps x to v and all other argument y in the domain of f to fy.

(Allocation)

$$\frac{s \models e_1 \Downarrow v_1,..., s \models e_n \Downarrow v_n \qquad l,..., l + n - 1 \in \text{locations - dom h}}{<x := \text{cons}\ (e_1\ ,..., e_n), (s,h)> \to (s[x :l], h[\ l : v_1\ ,..., l+n -1 : v_n])}$$

(Lookup)

$$\frac{s \models e \Downarrow v \qquad v \in \text{dom h}}{<x := [e], (s,h)> \to (s[x:h(v)],h)}$$

$$\frac{s \models e \Downarrow v \qquad v \notin \text{dom h}}{<x := [e], (s,h)> \to \textbf{abort}}$$

(Mutation)

$$\frac{s \models e \Downarrow v \qquad v \in \text{dom h} \qquad s \models e' \Downarrow v'}{<\ [e] := e', (s,h)> \to (s\ ,\ h[v : v'])}$$

$$\frac{s \models e \Downarrow v \qquad v \notin \text{dom h}}{<[e] := e', (s,h)> \to \textbf{abort}}$$

(Dispose)

$$\frac{s \models e \Downarrow v \qquad v \in \text{dom } h}{<\text{dispose } (e), (s,h)> \rightarrow (s, h \rceil (\text{dom } h - \{v\}))}$$

$$\frac{s \models e \Downarrow v \qquad v \notin \text{dom } h}{< \text{dispose } (e), (s,h)> \rightarrow \textbf{abort}}$$

The new command perform the usual heap operations, which shows the manipulation of mutable shared data structures in separation logic shows below:

$< comm >::= \dots$

$\mid < var >:= cons(< exp >, \dots, < exp >)$	*allocation*
$\mid < var >:= [< exp >]$	*lookup*
$\mid [< exp >] :=< exp >$	*mutation*
$\mid dispose < exp >$	*deallocation*

An essential feature of the language is that any attempt to refer to an unallocated address causes the program execution to abort. The semantics of the following sequence of new commands showing as an example below:

Allocation	x:= cons(1,2);	Store: x:4,y:5 Heap: empty ⇓
Lookup	y:=[x];	Store: x:25,y:5 Heap: 25:1,26:2 ⇓
Mutation	[x+1]:= 3;	Store: x:25,y:1 Heap: 25:1,26:2 ⇓
Deallocation	dispose(x+1)	Store: x:25,y:1 Heap: 25:1,26:3 ⇓

| Store: x:25,y:1 Heap: 25:1

The example says, begin with a state where the store maps the variable x with *3* and y with *4*, and the heap is empty. Then the typical effect of each kind of heap-manipulating command showing above. All operations, including lookup, mutation, and deallocation, would cause memory faults which will **abort** for the execution if an inactive address is deallocated.

2.2 Hoare Logic

Hoare logic originated in the 1960s. It lies "at the core of a multitude of tolls that are being used in academia and industry to specify" and "verifis real software systems [3]." Hoare logic is a formal system with a set of logical rules for reasoning about the correctness of computer programs [3]." In Hoare Logic, first we need to show that a partial correctness statement {P} c {Q} holds. If it holds for all stores and interpretations, the statement will be valid. The set of rules in Hoare logic shows below:

Skip rule:

$$\{Q\}\ SKIP\ \{Q\}$$

Assignment rule:

$$\{\{\ Q\ [t\ /x]\ \}\}\ (x ::= t)\ \{\{\ Q\ \}\}$$

Sequence rule:

$$\frac{\{\text{ P }\}\ c_1\ \{\text{ Q }\} \qquad \{\text{ Q }\}\ c_2\ \{\text{ R }\}}{\{\text{ P }\}\ c_1;c_2\ \{\text{ R }\}}$$

If rule:

$$\frac{\{\text{ P} \wedge \text{b }\}\ \text{c1}\ \{\text{Q}\} \qquad \{\text{ P} \wedge \sim \text{b}\}\ \text{c2}\ \{\text{Q}\}}{\{\text{ P }\}\ \text{if b then}\ c_1\ \text{else}\ c_2\ \text{end}\ \{\text{ Q }\}}$$

While rule:

$$\frac{\{\ P \wedge b\ \}\ \ c\ \ \{\ P\ \}}{\{\ P\ \}\ \ \text{while}\ \ b\ \ \text{do}\ \ c\ \ \text{end}\ \ \{\ P \wedge \sim b\}}$$

Consequence rule:

$$\frac{\{\ P'\ \}\ c\ \{\ Q'\ \} \qquad P \Rightarrow\ P' \qquad Q' \Rightarrow\ Q}{\{\ P\ \}\ \ c\ \ \{\ Q\ \}}$$

2.2.1 Hoare Triples

For every Logic has notation of formulas. The formulas in Hoare Logic are Hoare Triples. In Hoare Triples, the way of making formal claims, in terms of the behavior of commands are needed. Transform one state to another is the behavior of a command. The commands in terms of assertions need to be true before and after the command executes. A standard notation is **{P} c {Q}** meaning:

"If command *c* is started in a state satisfying assertion P, and if *c* eventually terminates in some final state, then this final state will satisfy the assertion Q [3]." This claim is called a Hoare Triple. In other words, if the precondition is true and the program terminates, then the postcondition is true, which is partial correctness. Partial correctness means in Hoare logic that an algorithm never terminates with an incorrect result. However, there is a problem with Hoare logic, and it does not cover if we have pointers. in other words, the regular Hoare Logic does not work if pointers are pointing to the same address. If we have pointers pointing to the same location of heap and assignment will change one of them, the precondition will not work because the modification in the regular assertion rule cannot be sharing different values. In order to verify certain programs, it needs to be able to state that two-pointer structures do not share common parts. Therefore, we want to use separation logic.

2.3 Separation Logic

O'Hearn and Reynolds [5] [6] [7] are the first ones who developed separation Logic, which was introduced in the early 2000s. Since then, it has worked perfectly at program verification. It is an extension of standard Hoare Logic with the help of programming language [8]. It addresses reasoning about programs that shared mutable data structures. It was created to reason about pointer manipulating programs of various types.

Separation Logic extended operations for expressing program specification, they are separating conjunction (*) and implication (-*). In Hoare Logic, assertions describe states, but now states contain heaps as well as stores. **Store** is a function mapping variable to values. **Heap** is a partial function mapping memory addresses to values. The separation condition is true for a heap if there exist two heaps and they are disjoint, which means they do not share a common address. Furthermore, the original heap was made of these two. In the first part of the first heap, if the first property is true, the second property would be true for the second heap. Additional rules from Hoare Logic is **Frame rule** which is very important in Separation logic:

$$\frac{\{\ \mathrm{p}\ \}\ \mathrm{c}\ \{\ \mathrm{q}\ \}}{\{\ \mathrm{p}\ *\ \mathrm{r}\ \}\ \ \mathrm{c}\ \ \{\ \mathrm{q}\ \ *\ \mathrm{r}\ \}}$$

The Frame Rule says if a program executes in a small state that satisfies p, it will also satisfy with p * r in a bigger state, which means that the additional part of the state will not be affected by the execution. Meanwhile, in the postcondition, r will keep in *true.* For more implementation, we will introduce them in the later chapter. The rest of the rules for separation logic are **allocation, lookup, mutation and deallocation:**

Allocation	$\{\mathrm{x} = \mathrm{X} \wedge \mathrm{emp}\}$ x:= cons $(e_1,...,e_k)\{\mathrm{x}\mapsto e_1$ [X/x],...,e_k [X/x] }
Lookup	$\{\ \mathrm{e} \mapsto \mathrm{v} \wedge \mathrm{x} = \mathrm{X}\ \}$ x:= [e] $\{\ \mathrm{x}= \mathrm{v} \wedge \mathrm{e}\ [\mathrm{X/x}] \mapsto \mathrm{v}\ \}$
mutation	$\{(\mathrm{e} \mapsto -\ )\}$ [e] := e'{ e $\mapsto$ e' }
deallocation	$\{(\ \mathrm{e} \mapsto -\ )\}$ dispose e { emp }

The **Allocation** says it will end with k contiguous heap cells from e_1 until e_k to have appropriate values since the execution begins with a store with $x = X$ and an empty heap. The variable X to store the value of x before the execution in **Lookup** same as allocation. It asserts that the content of the heap is unchanged; the only difference is in the store. The new value x modify to the value v at the old location e. **Mutation** says that if e points to something before, it points to e' afterward. This rule matches the natural semantics of Mutation. Finally, **Deallocation** says that there will be no active cells in a final state if e is the only allocated memory cell pointing to something before the execution of the command.

3 Implementation of Separation Logic

All of the described mathematical definitions from previous sections in Coq. The first step we have to implement is the Expressions. The expression contains the arithmetic and Boolean expression, which is one to one translation of the mathematics definition. Next, we implemented the basic data type for Command, and we also defined the syntax of Command. We extend the states to contain two components: the *store* maps variables into values, and the *heap* maps addresses into values. In our implementation, we have a file called **BasicTypes** which models Values, Atoms, and Addresses. Values are integers, Address are positive numbers, and Atoms consists only if nil stands for 0. Also, we add coercions from Atoms and Addresses so that they can automatically see as Value [6]. We assume all values are integers. There are all positive integers in Addresses. The Atoms are integers that not addresses and then heaps map addresses into single values:

$$\text{Values} = \text{Integers}$$

$$\text{Addresses} = \text{Positive Integers}$$

$$\text{Atoms} = \{\ \text{nil}\ \}$$

$$\text{nil seen as an integer nil}=0$$

$$\text{where Atoms and Addresses are disjoint}$$

$$\text{Heaps} = \bigcup_{A \subseteq \text{ Addresses}} \text{fin} \qquad (\text{A} \to \text{ Values}).$$

This will occur if only a finite number of positive integers are not addresses. In this logic we also can get:

$$\text{nil} \in Atoms$$

$$\text{StoreX} = \text{X} \to Values$$

$$\text{StatesX} = \text{StoresX} \times Heaps.$$

In the above function, X is a finite set of variables. The definition of the state is *store * heap*. The store is from variables to values. We also define an empty heap as *emp* in *Coq*, which contains no memory cells.

The implementation of the basic data type, expressions, commands, assertions, finite set, finite partial function, and the proving rules for separation logic will be introduced with more details in the following chapters.

3.1 Commands and Expressions

First, the syntax of the language in *Coq* is defined by reusing **aexp** and **bexp** defined for Imp. It is changed into an inductive definition and redefine the basic data type of commands and expression in the following way:

```
Inductive Command : Type :=
| SKIP : Command
| Ass : Vars -> Expr -> Command
| Lookup : Vars -> Expr -> Command
| Mutation : Expr -> Expr -> Command
| Allocation: Vars -> list Expr -> Command
| Dispose : Expr -> Command
| Seq : Command -> Command -> Command
| If : BExpr -> Command -> Command -> Command
| While : BExpr -> Command -> Command.
```

Next, we defined the operational semantics for commands in Coq. We are changing it into an inductive definition as well called **ceval** semantics. The **ceval** is the relation that gives us a new store and new heap if that command executes successfully in the system. More details can be checked in the implementation files.

The data type of expressions and boolean expressions:

```
Inductive Expr : Type :=
| Var : Vars -> Expr
| Const : Values -> Expr
| Plus : Expr -> Expr -> Expr
| Minus : Expr -> Expr -> Expr
| Mult : Expr -> Expr -> Expr.

Inductive BExpr : Type :=
| BTrue : BExpr
| BFalse : BExpr
| BEq : Expr -> Expr -> BExpr
| BLeq : Expr -> Expr -> BExpr
| BNot : BExpr -> BExpr
| BAnd : BExpr -> BExpr -> BExpr
| BOr : BExpr -> BExpr -> BExpr.
```

3.2 Partial Function

As we already know that in separation logic, states now contain heaps as well as stores. Store is a function mapping variable to values. Heap is a partial function mapping memory addresses to values. The value, and the address exist in pairs. For the heap, we need partial functions, so that we have implemented a finite partial function simply because we are starting with the program with an empty heap and only allocating finite many storage cells in one program. For example, we store something in the partial function, the argument value pairs in the list. In this function, we need to make sure the first argument is mapped to something unique. In other words, a finite partial function F from A to B is a partial function, the set where F is defined is finite. They are implemented by using a list of pairs and proof that a list of arguments contains no duplicates. Those are the operations we use:

PFunc: l: list(A * B), NoDup (map fst l) → PFunc A B.

Before we implemented the partial functions, we have implemented the **FiniteSet** in *Coq*.

For the implementation, we have a **set** which contains a list of A elements that the list does not have any duplicates.

Inductive **set** (A: Type):= mk_set: forall (l : list A), NoDup l → set A.

And then, the global instance for set **Setoid** means a Setoid to have equality on sets is used. Because two lists containing certain elements may represent the same set even if they do not contain duplicates, it can be avoided if the list is just reordered. A set is a list together with proof that there are no duplicates, and we consider two sets to be equal if the two lists contain the same elements. The rest of the implementation for Set are the definition of the **empty_set**, **singleton_set**, **disjoint** and some proven properties in a sets.

Definition **Empty** : set A := mk_set [ ] (NoDup_nil A).

Definition **singleton_set** (x : A) : set A := mk_set [x] (NoDup_singleton x).

Definition **disjoint** (s1 s2 : set A) : Prop := disjoint_list (asList s1) (asList s2).

The singleton list does not contain duplicates and then constructs the set. We also define the singleton heap in assertions as in the next chapter. The implementation of **Finite Partial Function** is very similar to the Finite Set. From the definition below, we have a list of A B pairs *list(A*B)*, but with these properties that the A part cannot occur twice which has no duplicates in the map of the first of the list. Because it has to be a function, meaning that if A is map to B, it can not map to C. This is what the partial function is.

Inductive **PFunc**: (A B : Type) := mk_pfunc: forall (l : list(A * B), NoDup (map fst l) → PFunc A B.

Then we have equivalence on those two partial functions if they are the same as sets. The set definition *asSet* is if this is the list of pairs, and if you have no duplicate in the first component, a partial function can always be seen as a set. Moreover, now we consider two partial functions to be equal if they are equal as sets. That is the Setoid on partial function.

Global Instance pfunc_Setoid: Setoid (A p→ B) := {|

equal := fun f1 f2 ⇒ asSet f1 == asSet f2;

setoid_equiv := pfunc_equiv_prop |}

3.3 Assertions

As in Hoare Logic, assertions describe states, but now states contain heaps as well as stores. The usual operations and quantifiers of predicate logic [9]. First, we defined the data type for assertions in Coq file **Assertions**:

```
Inductive Assertion : Type :=
| AEmp : Assertion
| ATrue : Assertion
| Anot : Assertion → Assertion
| AEqExpr : Expr → Expr → Assertion
| ALeqExpr : Expr → Expr → Assertion
| ASingleBlock_heap : Expr → list Expr → Assertion
| Aand : Assertion → Assertion → Assertion
| Aor : Assertion → Assertion → Assertion
| Aimpl : Assertion → Assertion → Assertion
| Aiff : Assertion → Assertion → Assertion
| Asep : Assertion → Assertion → Assertion
| Asepimp : Assertion → Assertion → Assertion
| AForall : Vars → Assertion → Assertion
| AExists : Vars → Assertion → Assertion.
```

The set of assertion, goes beyond the predicates used in the Hoare Logic. Following is the syntax of the new assertions,

$$
\begin{aligned}
< assert > :== \ldots & \\
& | emp \\
& | < exp > | \rightarrow < exp > \\
& | < assert > * < assert > \\
& | < assert > -* < assert >
\end{aligned}
$$

It is important to note that the meaning of these new assertions depends on both the store and the heap.

1. Empty heap

   The heap is empty.

$$emp$$

2. Singleton heap

   The heap contains one cell, at address *e* with contents *e'*.

$$e| \rightarrow e'$$

3. Separating conjunction

   It is also called star operations, and the heap can split into two disjoint parts that *p1* holds for the first part and *p2* holds for the other. *p1* star *p2* means some memory satisfies *p1* and then separately forms that memory is some heap that satisfies *p2*.

$$p1 * p2$$

4. Separating implication

   For the Separating implication, if the heap is entended with a disjoint part that *p1* holds, *p2* will hold for the extended heap.

$$p1 - *p2$$

- The *emp* asserts that the heap is empty:

$$[emp] \text{ assert } s\ h \text{ iff dom } h\ =\ \{\ \}$$

- $e| \rightarrow e^{'}$ assert that the heap contains one cell, at address *e* with contents *e'*:

$$[e\ | \rightarrow\ e^{'}] \text{ assert } s\ h \text{ iff dom } h\ = [e] \text{ exp } s \text{ and } h\ (\ [e] \text{ exp } s\ ) = [e^{'}] \text{ exp } s$$

In the implementation, the singleton heap is defined as two expressions *e* and *e'*. To make the assertion, such assertion is a function that takes in a store and heap and gets the property. The heap contains exactly one element. And then, the domain of the heap is equal to the singleton set, which contains the value of *e*. If it takes what is stored for the value of *e* for the apply. The singleton set will return value *e* if it is equal to some value *e'*:

Definition **singleton** (e e' : Expr): Assertion := mk_Assertion (fun s h → dom h == sing (value e s) ∧ apply h (value e s) = some (value e' s) pr.

- For the separating conjunction, we also call it Star operation (p1 ∗ p2). One part satisfied the first property, and the second part satisfied the second property. And these two parts are disjoint:

  $$h| = \ p1 * p2 \text{ iff exits } h1, h2 \ : h = \ h1 + h2 \ \bigwedge \ h1 \mid \ = \ p1 \ \bigwedge \ h2 \mid = \ p2$$

  H is a heap, and for that heap $p1 \ * \ p2$ is true if and only if there exist h1 and $h2$ so the following thing is h is equal to $h1 + h2$, The symbol + meaning disjoint union. *h1* and *h2* are heaps partial functions, which do not share the common address, and *h1* makes *p1* true, *h2* makes *p2* true. This star operation allows us to split the heap into smaller parts and reason locally. The overall property $p1 \ * \ p2$ the whole heap you want to reason about them locally, you just looking at one part where *p1* is true, and local other part *p2* is true, so that is why we have this operation to work with pointers.

- Here is the example to explain -* implication. Suppose the property P is defined as:
  P = x ↦ 3, 4 ** y ↦ x
  says that *x* points to a record containing 3,4 and *y* points to a cell containing *x*,

left part	right part
y → | . |	x → | 3 | | 4 |

  Now, x ↦ 3,4 -* P says something about the left part of the heap for *y*. It says that if one part of the heap is like the right part of the heap above, the whole heap (left and right together) is the one above. However, this does not say it is like the one above only if the right part is exist.

Now, x $\mapsto$ 1,2 ** (x $\mapsto$ 3,4 -* P) says that x points 1,2 and the stuff from above. This is equivalent to just x $\mapsto$ 1,2 since the left-hand side of the implication is false. However, as a precondition to the program [x] := 3; [x+1] := 4 it is relevant since that program changes the x part of the heap into one that satisfies the left-hand side of the implication. Therefore, we got P afterward. With other words, { x $\mapsto$ 1,2 ** (x $\mapsto$ 3,4 -* P) } [x] := 3; [x+1] := 4 {P}. This can be read as before the program x points to 1,2 and after the program x points to 3,4 and y points to x.

In general, { x $\mapsto$ _ ** (x $\mapsto$ a -* P) } [x] := a {P} is a valid triple in Separation logic. In fact the left-hand side is the weakest precondition for the program [x] := a and postcondition P, i.e., if {Q} [x]:= 3 {P} is true, then Q $\rightarrow$ (x $\mapsto$ _ ** (x $\mapsto$ a -* P)).

The condition for separation implication is:

Forall (h': Heap), dom h' dom h $\wedge$ Assertion_map s h' a1 $\rightarrow$ Assertion_map s (h+h+h') a2.

3.3.1 New Variable

In assertions, the new variable is needed, which does not appear in certain formulas or terms. For instance, if we want to substitute, it could be that the term is not substitutable for the variables in that formula because the term contains a variable free which would become bound by the quantifier. In order to do so, solving that problem in the implementation we would rename this bounded variable y to z. The new variable z we are using is a new variable. It cannot occur in the term. That is the reason why we need this new variable. Before we do that, we need to do the reading and writing numbers in *Coq*. The definition of strings in the standard library of *Coq* has its definition, but it is defined as an inductive type. Because *Coq* has a good notation for strings and *ASCII*, they are much like built-in notation for numbers. Then we can have a function for string-processing. To read the numbers, we first need to convert *asciiS* to *natS* which means if the character is a digit, we return that number. Otherwise, the whole parsing should fail. It defined as **digitToNat** in *Coq*. Now we can use the function to read numbers.

```
Fixpoint readNatAux (s : string) (acc : nat) : option nat :=
match s with
| "" ⇒ Some acc
| String c s' ⇒
match digitToNat c with
| Some n ⇒ readNatAux s' (10 * acc + n)
| None ⇒ None
end
end.
Definition readNat (s : string) : option nat :=
readNatAux s 0.
```

After we have a function to read a function, now we need a function to converts *natS* to their corresponding digits which means we need to write one for printing. Then we defined **natToDigit** in the system. and then we have our printing function as well.

```
Fixpoint writeNatAux (time n : nat) (acc : string) : string :=
let acc' := String (natToDigit (n mod 10)) acc in
match time with
| 0 ⇒ acc'
| S time' ⇒
match n / 10 with
| 0 ⇒ acc'
| n' ⇒ writeNatAux time' n' acc'
end
end.
Definition writeNat (n : nat) : string :=
writeNatAux n n "".
```

We can finish the rest of the proofs to clear that the **readNat** is indeed the inverse of the **writeNat** since we have those functions. After that we do the converting of *nat* to *string* and *string* to *nat*. And then we can get the new variables we want [10].

The implementation for the new variables we have:

```
Fixpoint newVarRec (s : set Vars) (n : nat) : Vars :=
let v := ("z" ++ writeNat n)%string in
match n with
| 0 ⇒ v
| S m ⇒ if (elem string_dec v s) then newVarRec (rem string_dec s v) m else v
end.
Definition newVar (s : set Vars) : Vars := newVarRec s (size s).
```

First, we need a recursion to get the new variables. Starting with *0* and check if it is in the set, if not returns if it is in there do the recursive call, and we do this until the number of iterations is one bigger than the number of elements in the original set.

In other words, we generate a number of variables $x_0....x_n$, so the number is one bigger than the elements in our set, so among those, there must be one not in the set that's the one we need. First, we have a given set of variables, and we want a distinct variable from those. We need this because when we rename the bounded variable, we usually want a completely new fresh variable that does not occur in the formula or the term anywhere. Then, we can finish the proof for our new variables.

Lemma **newVarRecProp** : forall n s, size s = n → ∼ Elem (newVarRec s n) s.

Lemma **newVarProp** : forall s, ∼ Elem (newVar s) s.

3.3.2 Examples

After we have the given axiom schemata for the predicate symbols like asingleblock heap and some other notations. With those new assertion like conjunction and implication to proof those propertity as examples. At the end of Assertion file we also proved some property from the book [7] for predicate ↦. The three lemmata are:

Lemma Page15Th1 (e1 e1' e2 e2': Expr): AValid (e1 ↦ e1' ∧ e2 ↦ e2' ↔ e1 ↦ e1' ∧ e1 = e2 ∧ e1'=e2')

Lemma Page15Th2 (e1 e1' e2 e2': Expr): AValid (e1 (→ e1' ** e2' → e1 <> e2)

Lemma Page15Th4 (e e': Expr)(a : Assertion): AValid (e (→ e1' ∧ a → e ↦ e' ** (e ↦ e' -* a))

To prove those we first add the data type in Assertion for **ATrue** and **Anot**. We need those in Assertion_map and Assertion_FV. Then we extend the proof of Assertion_map being a parametric morphism and Assertion_coincidence as well. And we add the notation as following:

Notation "True" := (Atrue).

Notation "e1 <> e2" := (ANot (AEqExpr e1 e2)).

Notation "e1 (→ e2" := (Asep (ASingleton_heap e1 e2)True)(at level 40).

Notation " a" := (Anot a).

Furthermore, we also add the Lemmata nonempty_notdisjoint, singleton_nonempty, and pfunc_extensional in FiniteSet and FiniteParialFunction files to finish the proof.

3.4 Assertions and their inference rules

The inference rules for predicate calculus remain sound in this enriched setting. Additional axiom schemata for separating conjunction include empty, commutative and associative laws. And we already prove those properties correct in our system. The notation for separating conjunction is defined as one star <*> in mathematical way, and the implementation part we use the notation <**> in Coq.

Commutative:

$$P1 \, * \, P2 \iff P2 \, * \, P1$$

Associative:

$$(\, P1 \, * \, P2) \, * \, P3 \iff P1 \, * \, (P2 \, * \, P3\,)$$

Emp:

$$P * emp \iff P$$

Separating conjunction is a commutative and associative operator with *emp* as a neutral element.

Distributive_Or:

$$(P1 \vee P2) * Q \iff (P1 * Q) \vee (P2 * Q)$$

Separating conjunction distributes over disjunction.

Subdistributive_And:

$$(P1 \wedge P2) * Q \iff (P1 * Q) \wedge (P2 * Q)$$

Separating conjunction semi-distributes over conjunction but not the other direction in general.

Distributive_Exists:

$$(\exists\, x.\, P) * Q \iff \exists\, x.\, (P * Q)\ where\ x\ not\ free\ in\ Q$$

Distributive_Forall:

$$(\forall\, x.\, P) * Q \iff \forall\, x.\, (P * Q)\ where\ x\ not\ free\ in\ Q$$

There is also an inference rule showing that separating conjunction is monotone concerning to implication.

Separation Monotonicity:

$$\frac{\mathrm{P1} \Rightarrow \mathrm{P2} \quad \mathrm{Q1} \Rightarrow \mathrm{Q2}}{\mathrm{P1} * \mathrm{Q1} \Rightarrow \mathrm{P2} * \mathrm{Q2}}$$

Separating conjunction is monotone to implication. In our implementation:

$$\text{AValid (P1} \to \text{P2)} \to \text{AValid (Q1} \to \text{Q2)} \to \text{AValid (P1 ** Q1} \to \text{P2 ** Q2).}$$

AValid is for s and h, which is store and heap. All of those above and below the line should be valid.

Separation Modusponens:

$$(A1 \ * \ (A1 \ - * \ A2)) \ \to \ A2$$

Furthermore, two rules, currying and decurrying, capture the adjunctive relationship between separating conjunction and separating implication showing below:

Currying:

$$\frac{\text{P1 * P2} \Rightarrow \text{P3}}{\text{P1} \Rightarrow \text{(P2 -}{*}\ P3)}$$

Decurrying:

$$\frac{\text{P1} \Rightarrow \ \text{(P2 -}{*}\ P3)}{\text{P1 * P2} \Rightarrow \text{P3}}$$

3.5 Specifications and their Inference rules

In specification logic, specifications are Hoare triples, and they describe commands. The notion of program speciation with variants for both partial and total correctness [11]. Our project will be focusing on partial correctness.

$\langle\ specification\rangle$ ::=

$$\{\ \langle\ assertion\rangle\ \}\ \ \langle\ command\rangle\ \ \{\ \langle\ assertion\rangle\ \}\quad (partial\ \ correctness)$$

$$|\ [\ \langle\ assertion\rangle\ ]\ |\ \ \langle command\rangle\ [\ \langle\ assertion\rangle\ ]\qquad (total\ correctness)$$

The initial assertion is called the precondition for these two flavors, and the last assertion is called the postcondition. The partial correctness specification {p} c {q} is true iff, starting in any state in which p holds:

- No execution of c abort
- Some execution of c terminates in a final state, q holds in the final state.

3.5.1 Substitution

In our system, the command- specific inference rules of Hoare logic remain sound, so we do such structural rule as Substitution.

$$\frac{\{\ \mathrm{P}\ \}\ \mathrm{c}\ \{\ \mathrm{Q}\ \}}{\{\ \mathrm{P}\ /\ \delta\ \}\ (\ \mathrm{c}\ /\ \delta\ )\ \{\ \mathrm{Q}\ /\ \delta\ \}}$$

Where δ is the substitution $\mathrm{v}_1 \rightarrow \mathrm{e}_1, \ldots, v_n \rightarrow \mathrm{e}_n$, $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

We defined the substitution for expression in the Expression file as well. The substitution in t_1 we want to replace for variable Vars is the term t_2. If y is variable and if x and y are the same, the result will be t_2 or y. If it is the constant n, then the result is constant n. If it is plus, we have expression t_3 and t_4, then we want to plus the *subst* in t_3 t_2 for the variable then same as the second one *subst* in t_4 t_2 for *var*. And we got the same idea in *Mult* and *Minus* as following in Coq:

```
Fixpoint substExpr (t1 t2 : Expr) (x : Vars) : Expr :=
match t1 with | Var y ⟹ if eqb x y then t2 elas Var y
| Const n ⟹ Const n
| Plus t3 t4 ⟹ Plus (substExpr t3 t2 x) (substExpr t4 t2 x)
| Mult t3 t4 ⟹ Mult (substExpr t3 t2 x) (substExpr t4 t2 x)
```

| Miuns t3 t4 $\Longrightarrow$ Minus (substExpr t3 t2 x) (substExpr t4 t2 x)
end.

Then we have the lemma to proof the substitution:

Lemma **Expr_substitution** : forall (t1 t2 : Expr) x s, value (t1[t2/x]) s = value t1 (update s x (value t2 s)).

The value of subst t_1 t_2 have the same value of t_1 to update the value of t_2 for store s and then for the variables of x.
After we have the expressions for substitution which is for a expression to be substitutable for a variable in a formula. we also need the substitution for assertions. For all the properties we need to let them to be substitutiable and for all the variables as well. We have the lemma:

Lemma **SubstitutableVar** : forall x y a, ∼Elem y (Assertion_AllVar a) → Substitutiable (Var y) x a.

After the proof of substitution for assertion for all properties from lemma **Assertion_substitution**, the Assertion_Subst_Rename says that given for every formula expression and variable we have an for every formula an equivalent formula so that the expression is substitutable.

Lemma **Assertion_Subst_Rename** : forall e x a, exists a', Substitutiable e x a' ∧ forall st, Assertion_map st a ↔ Assertion_map st a'.

3.5.2 Frame Rule

$$\frac{\{\ p\ \}\ c\ \{\ q\ \}}{\{\ p\ *\ r\ \}\ \ c\ \ \{\ q\ \ *\ r\ \}}$$

Where no variable occurring free in r is modified by c. The frame rule was named in "homage to the frame problem from artificial intelligence, which concerns axiomatizing state changes without enumerating all the things that do not change [12] [7]." The Frame Rule allows you to place a property into a larger context or heap. If the precondition p is true and the program terminates, then the postcondition q is true. Furthermore, if r says something about the heap outside of what is being affected by c, then c will not change the value of r (or we can say r is invariant for c). Here is a simple example for the frame rule:

$\{\ \mathrm{x} \mapsto \_\ \}\ [x] := 3\ \{\ x \mapsto \_\ \}$

$\{\ y \mapsto 42 * x \mapsto \_\}\ [x] := 3\ \{\ y \mapsto 42 * x \mapsto \_\ \}$

If x has some value, then we have x becomes 3, which x has some value. This statement is obviously true. X has some value; it has been allocated and changed and has some value afterward. Then we can use the frame rule. The precondition is 42, which is y. x has something. The heap is separated into two parts, and the first is only defined for y. The second is for x, they will not affect each other. Then the x becomes 3. The postcondition is the same. So, separating things allows them to say that x and y and they do not share anything.

3.6 List Example

We specify the list-reversal program as example in our thesis. It can be says that " If i is a list before execution, then j will be a list afterwards". On the other words " If i is a list representing the sequence α before execution, then afterwards j will be a list representing the sequence that is the reflection of α". In order to implement that, first is to define the set of abstract values, and with its primitive operations, and then to define predicates on the abstract values by structural induction. For the primitive operations, we have ϵ for the empty sequence. The $\alpha \cdot \beta$ for the composition of α followed by β, $\alpha\ \dagger$ for the reflection of α, and α_i for the ith component of α. And we used *singly-linked* list to represent sequences. The **list** α i when i is a list representing the sequence α. The following example shows that the program for reversing in list:

{ list α_0 i }

{ list α_0 i * (**emp** $\wedge$**nil** = **nil**)}
j:=**nil**;
{ list α_0 i * (**emp** $\wedge$ j = **nil**)}
{ list α_0 i * list ϵ j}
{ $\exists$ α,β. (list a i * list β j) $\wedge$ $\alpha_0^\dagger = \alpha^\dagger \cdot \beta$}
while i $\neq$ **nil do**
({ $\exists$a, α,β. (list (a$\cdot$ α) i * list β j) $\wedge$ $\alpha_0^\dagger$ = (a$\cdot$ α)$^\dagger\cdot$ β}
{ $\exists$a, α,β,k. (i $\mapsto$ a,k * list α k * list β j) $\wedge$ $\alpha_0^\dagger$ = (a$\cdot$ α)$^\dagger\cdot$ β}
k:=[i+1];
{ $\exists$a, α,β. (i $\mapsto$ a,k * list α k * list β j) $\wedge$ $\alpha_0^\dagger$ = (a$\cdot$ α)$^\dagger\cdot$ β}
[$i+1$]:=j;
{ $\exists$a, α,β. (i $\mapsto$ a,j * list α k * list β j) $\wedge$ $\alpha_0^\dagger$ = (a$\cdot$ α)$^\dagger\cdot$ β}
{ $\exists$a, α,β. (list α k * list(a$\cdot$ β) i) $\wedge$ $\alpha_0^\dagger = \alpha^\dagger \cdot$ a $\cdot$ β}
{ $\exists$ α,β. (list α k * listβ i) $\wedge$ $\alpha_0^\dagger = \alpha^\dagger \cdot \beta$}
j:=i ; i:= k
{ $\exists$ α,β. (list α i * listβ j) $\wedge$ $\alpha_0^\dagger = \alpha^\dagger \cdot \beta$})
{ $\exists$ α,β. list β j $\wedge$ $\alpha_0^\dagger = \alpha^\dagger \cdot \beta \wedge \alpha = \epsilon$}
{ list $\alpha_0^\dagger$ j}

This is the program reversing a list annotated with all intermediate assertions. The precondition is that i implements a list α_0. For example, in line six says that there are two sequences α, β, implemented by the pointers i and j on separated sections of the heap, so that the reverse $\alpha_0^\dagger$ of the original sequence α_0 is equal to the reverse of α followed by β. This property is the invariant of the loop and implies the post-condition of the program saying that j implements the reverse of α_0.

3.7 Proof Rules

For the proofing rules, we first define the separation triples and its notation. Similar to Hoare triples except it contains the store and the heap. The definition show below:

Definition **sep_triple** (P : Assertion) (c : Command) (Q : Assertion) : Prop :=
forall st1 st2, Assertion_map st1 P → ∼(c / st1 ⇒ abort) ∧ (c / st1 ⇒ st2 → Assertion_map st2 Q).

Then we have the notation **{{ P }} c {{ Q }}.**

If the precondition is true, the program will not abort. This definition says that if the precondition is true and the program terminates, it will not abort. There will be no error in operations, and what the postcondition says is accurate because it is partially correct. It can avoid the error get into the operations because people can not recognize the terminations. A specific example of deallocation may work on a whole heap because that address is in there and will be deallocation, but it will not work on both parts if you separate these into two parts because one of these two parts will not have that address.

3.7.1 Skip Rule

The Skip rule will preserves any assertion Q since the *skip* does not change the state.

$$\{Q\}\ SKIP\ \{Q\}$$

The Skip rule shows:
Lemma **Skip_rule** : forall Q, {{Q}} SKIP {{Q}}.

3.7.2 Assignment Rule

Sometimes, the postcondition can be any assertion Q; the precondition would be Q while the x is replaced by t. The notation Q [t / x] means Q where t is substituted in place of x. By using the substitution, we can give the proof for assignment rule. Because every time we always assume that the terms are substitutable and we have the **Assertion_Subst_Rename** so we can first replace the formula by the equivalent one and then apply the assignment rule.

$$\{\{\ Q\ [t\ /x]\ \}\}\ (x ::= t)\ \{\{\ Q\ \}\}$$

The Assignment rule shows:
Lemma **Assignment_rule** : forall Q x t , Substitutiable t x Q → {{Q[t/x]}} (x::=t) {{Q}}.

3.7.3 Sequence Rule

$$\frac{\{\ \mathrm{P}\ \}\ \ c_1\ \ \{\ \mathrm{Q}\ \} \qquad \{\ \mathrm{Q}\ \}\ \ c_2\ \ \{\ \mathrm{R}\ \}}{\{\ \mathrm{P}\ \}\ \ c_1;c_2\ \ \{\ \mathrm{R}\ \}}$$

The line above says the percondition P holds by command c_1 takes any state where Q holds, and then the command c_2 takes any state where R holds. In the sequence rule, the premises are given in backwards order with command c_2 before the command c_1. It will push the postconditions backwards through commands until we reach the beginning to get the Sequence Rule.The Sequence rule shows:
Lemma **seq_rule** : forall P Q R c1 c2, {{Q}} c2 {{R}} → {{P}} c1 {{Q}} → {{P}} (c1;c2){{R}}.

3.7.4 If Rule

$$\frac{\{\ \mathrm{P} \wedge \mathrm{b}\ \}\ \ \mathrm{c1}\ \ \{\mathrm{Q}\} \qquad \{\ \mathrm{P} \wedge \sim \mathrm{b}\}\ \ \mathrm{c2}\ \ \{\mathrm{Q}\}}{\{\ \mathrm{P}\ \}\ \mathrm{if}\ \ \mathrm{b}\ \ \mathrm{then}\ \ c_1\ \ \mathrm{else}\ \ c_2\ \ \mathrm{end}\ \{\ \mathrm{Q}\ \}}$$

The Boolean expression called b evaluates to True at the beginning of the if conditions else will be evaluated as False. In this rule the line above tells us the premises of the rule to work with the behavior of c_1 and c_2. Now we can formalize the proof rule for conditionals and prove it correct. The If rule shows:

Lemma **if_rule**: forall P Q b c1 c2,{{P ∧ convertBExpr b}} c1 {{Q}} → {{P ∧ convertBExpr b}} c2 {{Q}} → {{P}} (IFB b THEN c1 ELSE c2 FI) {{Q}}.

3.7.5 While Rule

$$\frac{\{\ P \wedge b\ \}\ \ c\ \ \{\ P\ \}}{\{\ P\ \}\ \ \text{while}\ \ b\ \ \text{do}\ \ c\ \ \text{end}\ \ \{\ P \wedge \sim b\}}$$

The While rule is based on the idea of invariant, an assertion whose truth can sure that before and after executing a command. Assertion P is an invariant of c, for this invariant property, if p is true, before one execution of the body, it will also be true afterward. So, no matter how many times the loop body executes, p will be true when the loop finally finishes. To get the complete statement by adding the loop terminates when b becomes false. And then add the loop body will be executed only if b is true. So, we need to strengthen the postcondition in the conclusion and the precondition in the premise. Then we can get the final version of the rule and proof it corrects. The While rule shows:

Lemma **while_rule**: forall P b c, {{P ∧ convertBExpr b}} c {{P}} → {{P}} (WHILE b DO c OD) {{P ∧ convertBExpr b}}.

3.7.6 Consequence Rule

$$\frac{\begin{array}{c}\{\ P'\ \}\ c\ \{\ Q'\ \}\\ P \Rightarrow\ P'\\ Q' \Rightarrow\ Q\end{array}}{\{\ P\ \}\ \ c\ \ \{\ Q\ \}}$$

We can see the strengthening the precondition or weakening the postcondition of a valid triple. This observation is captured by two rules of consequence: consequence_pre (P⇒ P') and consequence_post (Q'⇒ Q). The combined rule of consequence allows us to vary both the precondition and the postcondition, which showing proofing correct. The Consequence rule shows:

Lemma **consequence_rule** : forall P P' Q Q' c, AValid (P → P') → AValid (Q' → Q) → {{P'}} c {{Q'}} → {{P}}c{{Q}}.

3.7.7 Frame Rule

$$\frac{\{\ P\ \}\ \ c\ \ \{\ Q\ \}}{\{\ P * r\ \}\ \ c\ \ \{\ Q * r\ \}}$$

The frame rule says that one can extend local specifications to include any arbitrary claims about variables and heap segments that are not modified or mutated by *c*. The frame rule can be thought of as a replacement for the constancy rule when pointers are involved. Where c is a program, P, Q and r are separation logic formulas, and symbol * is the separating conjunction in separation logic. Intuitively, $P{*}r$ states that P and r hold in disjoint heaps. This conjunction allows the frame rule to guarantee that r is unchanged under the action of *c*. This feature of separation logic is essential for scalability as it allows the proof of a program to be decomposed into smaller ones.
The last rule among the structural rules, called a frame rule, says that one can extend local specifications to include any arbitrary claims about variables and heap segments that are not modified or mutated by *c*. The Frame Rule can be thought of as a replacement for to constancy rule when pointers are involved. The notation <#> means disjoint.The Frame Rule shows:

Lemma **frame_rule** : forall c P Q R, Assertion_FV R # EV c → {{P}} c {{Q}} → {{P ** R}} c {{Q ** R}}.

As mentioned, the **sep_triple** has to use as an assumption that the program does not abort. There are two properties needed to prove the frame rule. The restricted execution will abort if the restriction removes an address that is dereferenced by the command. The property $h_0 \perp h_1$ says these two heaps h_0 and h_1 have disjoint domains. The union of such heaps indicate as $h_0 \cdot h_1$.

- If <c,(s,h)> $\rightarrow$ * **abort**, then <c,(s,h_0)> $\rightarrow$ * **abort**

- If <c,(s,h)> $\rightarrow$ * (s',h') then <c,(s,h_0)> $\rightarrow$ * **abort** or <c,(s,h_0)> $\rightarrow$ * (s',h_0'), where h_0' $\perp$ h_1 and h'= h_0' $\cdot$ h_1.

These two properties used to prove the frame rule in Coq showing as:

Lemma **Prop1** : forall c s h1 h2, c / (s,h1 +h+ h2) $\Rightarrow$ abort $\rightarrow$ c / (s,h1) $\Rightarrow$ abort.

Lemma **Prop2** : forall c s s' h1 h2 h', c / (s,h1 +h+ h2) $\Rightarrow$ (s',h') $\rightarrow$ c / (s,h1) $\Rightarrow$ abort $\vee$ exists h1', c / (s,h1) $\Rightarrow$ (s',h1') $\wedge$ dom h1' # dom h2 $\wedge$ h' == h1' +h+ h2.

3.7.8 Rule for Mutation

By use the frame rule, one can move from local versions of inference rules for the primitive heap-manipulating commands to equivalent global version. For mutation shows below

- Mutation(local)

$$\{(e \mapsto -)\}[e] := e'\{e \mapsto e'\}$$

- Mutation(global)

$$\{(e \mapsto -) * r)\}[e] := e'\{(e \mapsto e') * r\}$$

- Mutation(backwards reasoning)

$$\{(e \mapsto -) * ((e \mapsto e') - *p)\}[e] := e'\{p\}$$

The mutation says that if e points to something before, then it points to *e'* afterward by local reasoning. This resembles the natural semantics of Mutation. Taking r from global to be (e $\mapsto$ e') - $*$ p and using the valid implication q $*$ (q -$*$ p) $\Rightarrow$ p, the third rule for Mutation is suitable for backward reasoning, as one can substitute any assertion for the postcondition p. The Mutation shows:

Lemma **mutation**: forall e e' P, {{(e $\mapsto$ -) ** ((e $\mapsto$ e') -* P) }} ([e] ::= e'){{P}}.

3.7.9 Rule for deallocation

- Deallocation(local)

$$\{(e \mapsto -)\}\ \textit{dispose}\ e\{\textit{emp}\}$$

- deallocation(global, backwards reasoning)

$$\{(e \mapsto -) * r\}\ \textit{dispose}\ e\{r\}$$

A similar development from mutation works for Deallocation. One of these differences is that the global form is itself suitable for backward reasoning as well. By local reasoning, the singleton heap assertion is necessary for the precondition to assure *emp* in the postcondition. It says that if e is the only allocated memory cell before execution of the command, there will be no active cell in the resulting state for postcondition. The Deallocation shows:

Lemma **deallocation**: forall e P, {{e ↦ - ** P }} (Dispose e) {{P}}.

3.7.10 Rule for Allocation

- Allocation(local)

$$\{v = v' \wedge \textit{emp}\}v := \textit{cons}(\overline{e})\{v \mapsto \overline{e}'\}$$

where *v'* is distinct from *v*.

- Allocation(global)

$$\{r\}v := \textit{cons}(\overline{e})\{\exists v'.(v \mapsto \overline{e}') * r'\}$$

where *v'* is distinct from *v*, and is not free in $\overline{e}$ or *r*.

- Allocation(backwards reasoning)

$$\{\forall v'.(v' \mapsto \overline{e}) - *p'\}v := \textit{cons}(\overline{e})\{p\}$$

where *v'* is distinct from *v*, and is not free in $\overline{e}$ or *p*.

In the rule of allocation we indicate substitution by priming metavariables denoting expressions and assertions. The abbreviate e_1,...,e_n is by $\overline{e}$ same as $\overline{e'}$. e_i' for e_i/v $\mapsto$ v' and r' for r/v $\mapsto$ v'. The Allocation shows:

Lemma **allocation** :forall x y e p, Substitutiable (Var x) y p → x <> y → ~Elem x (Expr_FV e) → ~Elem x (Assertion_FV p) → {{ All x, (Var x ↦ e) -* p[(Var x)/y] }} (Allocation y (e::List.nil)) {{ p }}.

This lemma says that the first properity has to be substitutiable and then we havs x is not equal to y which is *v'* distinct from *v*. And next we have the property x does not appear free in e and p. And finally we have the precondition forall x, implication *p'* for new variable x place for y.

3.7.11 Rule for Lookup

- Lookup(local)

$$\{v = v' \wedge (e \mapsto v'')\}v := [e]\{v = v'' \wedge (e' \mapsto v'')\}$$

  where *v*, *v'* and *v"* are distinct.

- Lookup(global)

$$\{\exists v''.(e \mapsto v'') * (r/v' \to v)\}v := [e]\{\exists v'.(e' \mapsto v) * (r/v'' \to v)\}$$

  where *v*,*v'* and *v"* are distinct, *v'* and *v"* do not occur free in *e*, and *v* is not free in *r*.

- Lookup(backwards reasoning)

$$\{\exists v'(e \mapsto v') * ((e \mapsto v') - *p')\}v := [e]\{p\}$$

  where *v'* is not free in *e*, nor free in *p* unless it is *v*.

In Lookup we uses *Var x* to refer to the value of x before execution. It asserts that the content of the heap is unchanged. The only change is in the store where the new value of x is modified to the value at the old location *e*. The lemma for Lookup is very similar from allocation shows below:

Lemma **lookup** :forall x y e p, Substitutiable (Var x) y p $\rightarrow$ ~Elem x (Expr_FV e) $\rightarrow$ ~Elem x (Assertion_FV p) $\vee$ x=y $\rightarrow$ {{ Any x, (e $\mapsto$ Var x) ** (e $\mapsto$ Var x -* p[(Var x)/y]) }} (Lookup y e) {{ p }}.

4 Discussion

The objective of this project is to implement the Separation Logic in the Coq system. Furthermore, building up a library of examples of a simple program that has been verified. As discussed previously, the goal of the logic was to facilitate reasoning about shared mutable data structures. The implementation of separation logic in Coq system can be used to verify the program; Coq system will not allow people to apply the wrong program. Our system works much better than the other existing implementation of separation logic in Coq. It has cleaner logic and definitions, fewer lemmas, and more precise proofs. There are a few things can be improved for the system. The syntax and semantics can be improved in the future. By using the system or logic, for some examples can be verified programs. That will be the next thing which can be done. In those packages, the implementation of FiniteSet and FinitePartialFunction can be improved as well.

The purpose of this package only covers the needs described in this thesis. Further work is needed when considering repurpose the package.

APPENDIX

Appendix I: Packages

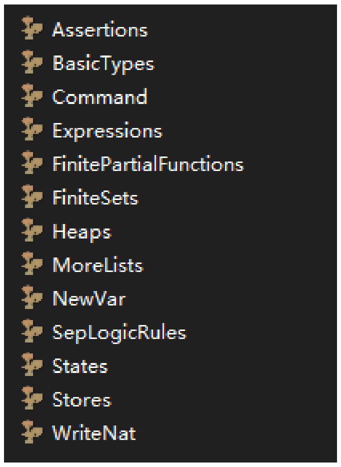
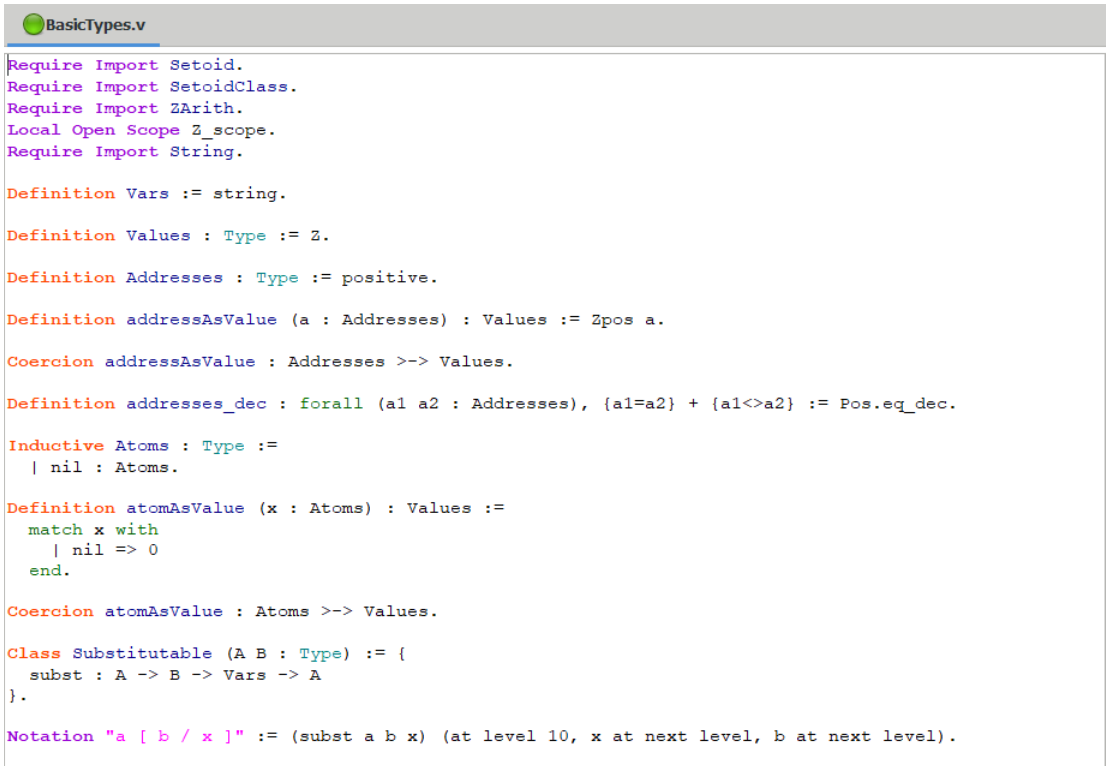

Assertions
BasicTypes
Command
Expressions
FinitePartialFunctions
FiniteSets
Heaps
MoreLists
NewVar
SepLogicRules
States
Stores
WriteNat

Figure 1 Packages for Separation Logic



```
Require Import Setoid.
Require Import SetoidClass.
Require Import ZArith.
Local Open Scope Z_scope.
Require Import String.

Definition Vars := string.

Definition Values : Type := Z.

Definition Addresses : Type := positive.

Definition addressAsValue (a : Addresses) : Values := Zpos a.

Coercion addressAsValue : Addresses >-> Values.

Definition addresses_dec : forall (a1 a2 : Addresses), {a1=a2} + {a1<>a2} := Pos.eq_dec.

Inductive Atoms : Type :=
  | nil : Atoms.

Definition atomAsValue (x : Atoms) : Values :=
  match x with
    | nil => 0
  end.

Coercion atomAsValue : Atoms >-> Values.

Class Substitutable (A B : Type) := {
  subst : A -> B -> Vars -> A
}.

Notation "a [ b / x ]" := (subst a b x) (at level 10, x at next level, b at next level).
```

Figure 2 The Basictypes for Separation Logic

REFERENCES

[1] "The coq proof assistant," 2004. [Online]. Available: https://coq.inria.fr/.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, October 1969.

[3] A. T. Cătălin Hriţcu Vilhelm Sjöberg and B. Yorgey, "Programming language foundation, hoare logic, part i," vol. 2, no. 5.8, 2020. [Online]. Available: https://docs.omnetpp.org/.

[4] S. Abhishek Kr and R. Natrajan, "An outline of separation logic," *School of Technology and Computer Science*, pp. 1–18, [Online]. Available: http://www.tcs.tifr.res.in/~abhishek/separation.pdf.

[5] J. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[6] J. C. R. Peter W. O'Hearn and H. Yang, "Local reasoning about programs that alter data structure," *Queen Mary, University of London, Carnegie Mellon University, University of Birmingham and University of Illinois at Urbana-Champaign*, vol. 2142 of Lecture Notes in Computer Science, pp. 1–19, Berlin, 2001. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/localreasoning.pdf.

[7] P. O'Hear, "Separation logic," *Communications of the ACM*, vol. 62, no. 2, pp. 86–95, February 2019. [Online]. Available: https://cacm.acm.org/magazines/2019/2/234356-separation-logic/fulltext?mobile=true?mobile=false.

[8] T. Tuerk, "A separation logic framework in hol," pp. 1–7, [Online]. Available: http://users.encs.concordia.ca/~tphols08/TPHOLs2008/ET/116-122.pdf.

[9] S. Ishtiaq and P. W. O'Hearn, "Bi as an assertion language for mutable data structures," *In Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Sym-posium on Principles of Programming Languages*, pp. 14–26, New York, 2001. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.224&rep=rep1&type=pdf.

[10] Poleiro, "Reading and writing numbers in coq," March 31, 2013. [Online]. Available: http://poleiro.info/posts/2013-03-31-reading-and-writing-numbers-in-coq.html.

[11] H. Yang and P. W. O'Hearn, "A semantic basis for local reasoning," *oundations of Software Science and Computation Structures*, vol. 2303 of Lecture Notes in Computer Science, pp. 402–416, Berlin,2002. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/basis.pdf.

[12] J. C. Reynolds, "An introduction to separation logic," *Computer Science Department Carnegie Mellon University*, October20–22, 2008. [Online]. Available: https://www.cs.cmu.edu/~jcr/copenhagen08.pdf.