University of Birmingham
School of Computer Science
MSc Advanced Computer Science

First Semester Mini Project

Separation Logic

Natalie Ravenhill

Supervisor : Uday Reddy

Date: January 14, 2016

Abstract

In this report, we describe and explain Separation Logic and give examples of its use in current research. We also prove the correctness of programs that manipulate simple mutable data structures, including singly and doubly linked lists and trees. We implement our programs in a selection of automatic verification tools based on Separation Logic and the tools are compared.

Keywords

Separation Logic, Program Logic, Program Verification

Contents

‘

Chapter 1

Introduction

Separation Logic is a tool for expressing proofs of correctness on programs involving shared mutable data structures. It is an extension of Hoare logic [Hoare, 1969], and allows us to reason about programs with sharing more easily than with Hoare logic. (see 2.1).

The logic itself was devised mostly by John Reynolds [Reynolds, 2002]. This paper specifies the rules of separation logic on a simple imperative programming language and gives examples of its usage in research.

We study the foundations of the separation logic (see 2) and write proofs on basic data structures that use sharing, including singly linked lists, doubly linked lists and trees. (see 3).

Many tools, including theorem provers and static analysis tools [Calcagno et al., 2015] have been created, that are based on Separation Logic. We focus on two specific tools, Smallfoot [Berdine et al., 2006] and HIP [Chin et al., 2007], but there are many more available. For a good list, see `http://www0.cs.ucl.ac.uk/staff/p.ohearn/SeparationLogic/Separation_Logic/Tools.html`.

Our proofs are implemented in these tools, to ensure their correctness and a comparison between the two tools are made. (see 4).

Separation Logic has been applied to many areas, including concurrency. [Reddy and Reynolds, 2012] and the logic has been adjusted over time to include more appropriate rules for certain applications, such as the addition of a overlapping conjunction to specify data structures that share the same pointers, such as graphs and directed acyclic graphs [Hobor and Villard, 2013]. This connective has also been used in specifying a program logic for Javascript [Gardner et al., 2012], which will be used for creating a verification tool for Javascript programs [Gardner, 2015].

Chapter 2

Background

2.1 Hoare Logic

A statement of Hoare logic is used to reason about a program at a certain stage of its execution. It contains assertions that must hold in the state given at that point of the program.

A **precondition** is a statement that must hold in the state before a program is executed and a **postcondition** is a statement that must hold in the state after a program has been executed.

We can use these statements to form a **specification**:

$$\{P\}\ \mathrm{C}\ \{Q\}$$

The precondition P is true before the command C is executed and the postcondition Q is true after C is executed. The curly brackets denote a *partial correctness* specification, which means that the conditions hold *if* the command C terminates. We can also write *total correctness* specifications, where the command C must terminate. (we replace the curly brackets with square brackets in this case). Generally we prove correctness and termination separately, so we just consider partial correctness in this report.

An example of a specification is the following:

$$\{\mathrm{X} = 5\}$$
$$\mathrm{X} := \mathrm{X}{*}2;$$
$$\{\mathrm{X} = 10\}$$

The precondition is $X = 5$ and the postcondition is $X = 10$. The command C calculates $5 * 2$ and assigns X this value. The postcondition $X = 10$ will be true if this command terminates. Therefore our specification is correct.

2.1.1 Rules of Hoare Logic

The program above is very simple, so we use some more inference rules to help us prove specifications for larger programs:

Composition

A program usually has more than one command, so we use the following inference rule for writing proofs for programs on multiple lines:

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

Given two commands, if the precondition of the second is identical to the postcondition of the first, then we can compose them to form a larger program, where the precondition of the whole program is that of the first command, and the postcondition is that of our second command. For example, given the following program:

$$\begin{gathered}\{\text{X} = 10\} \\ \text{X := X-1;} \\ \{\text{X} = 9\}\end{gathered}$$

If we compose it with the first program, we get the following larger program, where P is $X = 5$, Q is $X = 10$ and R is $X = 9$:

$$\begin{gathered}\{\text{X} = 5\} \\ \text{X := X}*2; \\ \text{X := X-1;} \\ \{\text{X} = 9\}\end{gathered}$$

There is also another composition rule we can use, when there are no commands:

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}$$

Assignment

We should also consider assignment in more detail. Its inference rule is the following:

$$\frac{}{\{P[E/V]\}\ V := E\ \{P\}}$$

Which says that we replace all instances of the variable V in P with the expression E. Because we are putting the value of E into V, we must know that P holds for E before we do the assignment. This is called *backwards propagation*.

For example, in our original program, if the value of X was used by another variable, we would need to change the other variables in the precondition too:

$$\begin{gathered}\{X = 5 \ \wedge \ Y = 30\} \\ \{X * 2 = 10 \ \wedge \ Y = (X * 2) * 3\} \\ X := X * 2; \\ \{X = 10 \ \wedge \ Y = X * 3\}\end{gathered}$$

If $Y = X * 3$ in the postcondition, it must be equal to the multiplication of the original X value with 2 and 3 in the precondition, so our proof must state this, as we cannot partially apply the assignment rule.

Conditional Branching

$$\frac{\{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

To use an if statement in a program, there are two possible commands, C_1 for when the condition B is true and C_2 for when it is false. Therefore, if we have a postcondition Q that can be obtained by executing C_1 in the precondition and true condition, and by executing C_2 in the precondition and false condition, then we can use P and Q as the pre and postconditions for the whole if statement.

Here is an example of its usage:

$\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge \ (X \leq$
$Y \Rightarrow Y = max(X, Y))\}$
if $X > Y$ **then**
  $Z := X;$
else
  $Z := Y;$
end if
$\{Z = max(X, Y)\}$

where max is a binary function that returns the maximum of the two arguments. If the arguments are equal, we assume the result of max is the second one.

The above specification is true because for the true branch we have:

$$\begin{gathered}\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge \ (X \leq Y \Rightarrow Y = max(X, Y)) \ \wedge \ X > Y\} \\ Z := X; \\ \{Z = max(X, Y)\}\end{gathered}$$

which is true because the precondition reduces to $max(X, Y) = X$, and Z is assigned X. The value $max(X, Y)$ is a logical value, so we do not change the X and Y inside it when we use the assignment rule.

And for the false branch we have:

$$\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge \ (X \leq Y \Rightarrow Y = max(X, Y)) \ \wedge \ \neg(X > Y)\}$$
$$Z := Y;$$
$$\{Z = max(X, Y)\}$$

which is true because the precondition reduces to $Y = max(X, Y)$, and Z is assigned Y.

We can prove this by using the proof rule for the assignment operator:

$\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge (X \leq$
$Y \Rightarrow Y = max(X, Y)\}$
if $X > Y$ **then**
  $\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge (X \leq$
  $Y \Rightarrow Y = max(X, Y)) \ \wedge \ X > Y \ \}$
  $\{X = max(X, Y)\}$
  $Z := X;$
  $\{Z = max(X, Y)\}$
else
  $\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge (X \leq$
  $Y \Rightarrow Y = max(X, Y)) \ \wedge \ \neg(X > Y) \ \}$
  $\{(X > Y \Rightarrow X = max(X, Y)) \ \wedge (X \leq$
  $Y \Rightarrow Y = max(X, Y)) \ \wedge \ X \leq Y \ \}$
  $\{Y = max(X, Y)\}$
  $Z := Y;$
  $\{Z = max(X, Y)\}$
end if
$\{Z = max(X, Y)\}$

In the true case we use the assignment rule to show that when the condition is true, assigning Z to X gives the maximum, as we use $X > Y$ to reduce the precondition to this, and when the condition is false we get the opposite.

Precondition Strengthening

We can weaken proofs, in Hoare Logic, so that they are true for less specific states. One way we do this is by using a precondition strengthening inference rule, to show that if we have a specification for a weak precondition, P and there is a stronger assertion P' such that $P' \Rightarrow P$, then we can have the triple $\{P'\}C\{Q\}$, where we replace P with P':

$$\frac{P' \Rightarrow P \quad \{P\} \ C \ \{Q\}}{\{P'\} \ C \ \{Q\}}$$

For example, if we have the program $I := I + 1$ with precondition $\{I > 0\}$ and postcondition $\{I > 0\}$, then backwards reasoning gives us $\{I \geq 0\}$. This means that we have a proof for all values of I more than or equal to 0, but we didn't want to include 0. We know that $I > 0 \Rightarrow I \geq 0$, so we can use the rule above to get our specification $\{I > 0\}I := I + 1\{I > 0\}$.

Using the inference rule, this gives:

$$\frac{I > 0 \Rightarrow I \geq 0 \quad \{I \geq 0\}\ C\ \{I > 0\}}{\{I > 0\}\ C\ \{I > 0\}}$$

So the proof will be:

$\{I > 0\}$ //use rule here
$\{I \geq 0\}$
$\{I + 1 > 0\}$
$I := I + 1$
$\{I > 0\}$

We can give a more detailed example, using the following program:

$\{x > y\ \wedge\ x > z\}$
if $(y < z)$ **then**
  $x := x - z;$
else
  $x := x - y;$
end if
$\{x > 0\}$

The proof of its correctness is as follows:

$\{x > y\ \wedge\ x > z\}$
if $(y < z)$ **then**
  $\{x > y\ \wedge\ x > z\ \wedge\ y < z\}$
  $\{x > z\}$
  $\{x - z = 0\}$
  $x := x - z;$
  $\{x > 0\}$
else
  $\{x > y\ \wedge\ x > z\ \neg(y < z)\}$
  $\{x > y\}$
  $\{x - y > 0\}$
  $x := x - y;$
  $\{x > 0\}$
end if
$\{x > 0\}$

Else part

Working back from the postcondition, we must first prove the else condition, by proving that the precondition and false if condition give the postcondition, so we must prove $\{x > y \ \wedge \ x > z \ \neg(y < z)\} else\_code \{x > 0\}$.

First we apply the assignment rule, so we get the following:

$$\begin{array}{l} \{x > y\} \\ \{x - y > 0\} \\ x := x - y; \\ \{x > 0\} \end{array}$$

Now we have to add the other conditions from the precondition to $\{x > y\}$. Using the precondition strengthening rule, we have the following:

$$\frac{x > y \ \wedge \ x > z \ \wedge \ \neg(y < z) \Rightarrow (x > y)' \quad \{(x > y)\} \ x := x - y \ \{x > 0\}}{\{x > y \ \wedge \ x > z \ \wedge \ \neg(y < z) \Rightarrow (x > y)\} \ x; = x - y; \ \{x > 0\}}$$

We already have $\{x > y\} \ x := x - y \ \{x > 0\}$ so we just need to prove that $x > y \ \wedge \ x > z \ \wedge \ \neg(y < z) \Rightarrow (x > y)$. As $x > y$ is on the left hand side, $x > y$ will alwys be true when the left hand side is true.

Therefore we have the following:

$$\begin{array}{l} \{x > y \ \wedge \ x > z \ \neg(y < z)\} \\ \{x > y\} \end{array}$$

Therefore, by using the precondition strengthening rule, we can prove that the precondition and false if condition gives us the postcondition.

The true if statement will be proved in exactly the same way, but with z instead of y.

Postcondition Weakening

We can also make the postcondition weaker, using the following inference rule:

$$\frac{Q \Rightarrow Q' \quad \{P\} \ C \ \{Q\}}{\{P\} \ C \ \{Q'\}}$$

If we have a stronger condition Q that we have a triple $\{P\}C\{Q\}$ for and we know that $Q \Rightarrow Q'$, then we can have $\{P\}C\{Q'\}$.

For example, given the triple $\{I \geq 0\}I := I + 1\{I > 0\}$, we can use this rule as $I > 0 \Rightarrow I \geq 0$, to get $\{I \geq 0\}I := I + 1\{I \geq 0\}$. This is only true if we already know $I \geq 0$, so we would have the following proof:

$\{I \geq 0\}$
$\{I + 1 > 0\}$
$I := I + 1;$
$\{I > 0\}$ //use rule here
$\{I \geq 0\}$

Then we can also combine the two rules, to give us the following rule:

$$\frac{P' \Rightarrow P \quad \{P\}C\{Q\} \quad Q \Rightarrow Q'}{\{P'\}C\{Q'\}}$$

Iteration

We can also use a proof rule for while loops:

$$\frac{\{P \wedge B\}C\{P\} \quad P \wedge \neg B \ \Rightarrow Q}{\{P\} \textbf{ while } B \textbf{ do } C\{Q\}}$$

This rule needs two things; first that when P and the condition of the while loop, B, are true, executing the commands inside the loop, C, will give us a postcondition that is the same as the precondition. Second, if B is false and P is true, then the postcondition Q must be true. When we have both of these statements, then P is the precondition for the entire while loop and Q is its postcondition.

The statement given by P is usually called the *loop invariant*

Here is an example program using a while loop:

$I := 1$
$A := 1$
while $I < N$ **do**
  $I := I + 1$
  $A := A * I$
end while

This program creates a factorial of a number N and stores the result in A. Therefore we can have $P = \{A = I! \wedge I \geq 1\}$ and postcondition $Q = \{A = N!\}$.

Then to prove the correctness of this loop, we must prove the loop iteration:

while $I < N$ **do**
$\{A = I! \ \wedge \ I \geq 1 \ \ \wedge \ I \leq N\}$
$\{A = I! \ \wedge \ I \geq 0\}$
$\{A * (I + 1) = I! * (I + 1) \ \wedge \ I \geq 0\}$
$\{A * (I + 1) = (I + 1)! \ \wedge \ (I + 1) \geq 1\}$
$I := I + 1$
$A := A * I$
$\{A = I! \ \wedge \ I \geq 1\}$
end while

We use the assignment rule to show that changing the values of A and I still gets us the precondition and loop, as when $I \geq 1$, then $I \geq 0$ will always be true, so we can use precondition strengthening here.

Then the loop exit:

$(A = I! \wedge I \geq 1 \wedge \neg(I < N)) \Rightarrow A = N!\}$
$(A = I! \wedge I \geq 1 \wedge (I = N \vee I > N)) \Rightarrow$
$A = N!\}$
$(A = I! \ \wedge \ I \geq 1 \ \wedge \ I = N) \Rightarrow A = N!\}$
$\{A = N!\}$

The value of I must be $\geq N$ in the precondition. The only possible value for $A = N!$ to hold is $I = N$, so we use this to get the postcondition.

Therefore our while loop in this program is correct, using the while loop rule.

Frame Rule

Another rule is the frame rule, which is the following:

$$\frac{\{P\} \ C \ \{Q\}}{\{P \ \wedge \ R\} \ C \ \{Q \ \wedge \ R\}}$$

If C does not modify the free variables of R.

This says that if we have some precondition and postcondition for a program, we can add more information to those conditions and they will still hold, as long as the command does not change anything that would make the new information change. This is similar to how in animation, the background of frames stays the same, but we just edit a small part of the image. Here the new information we are adding is like adding more detail to the background. For example, in our first program we can add $\{A = 5 \ \wedge B = 7\}$ without changing anything else, as our program does not change those values:

$$\{X = 5 \ \wedge \ A = 5 \ \wedge \ B = 7\}$$
$$X := X * 2;$$
$$\{X = 10 \ \wedge \ A = 5 \ \wedge \ B = 7\}$$

2.2 Separation Logic

This frame rule is the basis of Separation Logic. The idea of separation logic is that we want to reason about different parts of the heap in isolation. We could try this using the original frame rule, but we want to reason about *disjoint* areas of the heap. For example if we had the following:

$$\{list\ \alpha\ (i,j)\ \wedge list\ \beta\ (m,n)\}$$

Then the lists may share elements and we would have to give extra predicates to explain they are separate, making our proofs more complicated. Therefore we use a new connective, called the *separating conjunction*, which is written using $*$.When we have two predicates, p_0 and p_1, $p_0 * p_1$ means that p_0 holds in one part of the heap and p_1 holds in the other, and these partial heaps cannot overlap.

So now we have:

$$\{list\ \alpha\ (i,j)\ *\ list\ \beta\ (m,n)\}$$

meaning that one separate part of the heap contains a list α and the other contains a list β.

The frame rule we used for Hoare logic is adjusted to include the separating conjunction, giving us the following rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P\ *\ R\}\ C\ \{Q\ *\ R\}}$$

Assertion Language

The assertions of separation logic include the usual logical connectives ($\neg, \wedge, \vee, \rightarrow, \exists, \forall$), all the rules of Hoare logic discussed and some new rules. The assertions describe states that are a combination of a *store* and a *heap*, which are two functions:

1. $Store : Var \rightharpoonup Val$, maps a variable to the value currently stored in it
2. $Heap : Addr \rightharpoonup Val$, maps an address to the value contained at that address

All values in Val and all addresses in $Addr$ are integers, and we assume the sets Var and $Addr$ are finite.

The meanings of all the assertions depend on the entire state, so must be described on both the store, s, and the heap h, as in [Reynolds, 2002]. Therefore the semantics of the assertion are described by the following:

$$[\![p \in \langle assert \rangle]\!]_{assrt} \in Stores \to Heaps \to \{true, false\}$$

which says that any assertion p, when evaluated, is a function that maps a store in $Stores$ and a heap in $Heaps$ to true or false.

Note that $[\![e]\!]_{asrt}$ is an evaluation in the assertion language and that $[\![e]\!]_{exp}$ is an evaluation in the program language

The rules we add to our logic for assertions are the following:

- *empty heap:* the empty heap is described by **emp** and asserts that the heap is empty:

$$[\![\mathbf{emp}]\!]_{asrt}\ s\ h\ \Leftrightarrow\ dom\ h = \emptyset$$

- *singleton heap:* given two expressions e and e', $e \mapsto e'$ asserts that there is one cell in the heap, at address e and containing e':

$$[\![e \mapsto e']\!]_{asrt}\ s\ h\ \Leftrightarrow\ ((dom\ h = \{[\![e]\!]_{exp}\ s\})\ \wedge\ (h([\![e]\!]_{exp} s) = [\![e']\!]_{exp}\ s))$$

  This says that the heap must contain the value obtained by evaluating the expression e in the store, and that the value in the address given by the result of e in the heap must be the value obtained by evaluating e' on the store (as expressions must be evaluated on variables).

- *separating conjunction:* as described before, combines two distinct parts of the state. Given two assertions, p_0 and p_1 and the heap is $h = h_0 \cdot h_1$ such that $h_0 \perp h_1$, then p_0 holds for h_0 and p_1 holds for h_1 :

$$[\![p_0 * p_1]\!]_{asrt}\ s\ h\ \Leftrightarrow (\exists h_0, h_1.\ (h_0 \perp h_1) \wedge (h_0 \cdot h_1 = h) \wedge ([\![p_0]\!]_{asrt}\ s\ h_0) \wedge ([\![p_1]\!]_{asrt}\ s\ h_1))$$

- *separating implication:* this connective is slightly more complicated. It says that given a heap h, if we extend this heap with a disjoint heap h', for which p_0 is true, then p_1 will hold in the heap $h \cdot h'$:

$$[\![p_0 -\!\!* p_1]\!]_{asrt}\ s\ h\ \Leftrightarrow \forall h'.\ (h' \perp h\ \wedge\ [\![p_0]\!]_{asrt}\ s\ h') \Rightarrow [\![p_0]\!]_{asrt}\ s\ (h \cdot h')$$

We can also write some abbreviations to make our proofs easier, as in [Reynolds, 2002]:

- $e \mapsto - = \exists x.\ e \mapsto x$: e maps to *any* single element, x, that is not free in e
- $e \hookrightarrow e' = e \mapsto e' * \mathbf{true}$: e maps to a value e' and there are other values in the heap too.
- $e \mapsto e_1, \ldots, e_n = e \mapsto e_1 * \ldots * e + n - 1 \mapsto e_n$: e maps to *only* a list of values
- $e \hookrightarrow e_1, \ldots, e_n = e \hookrightarrow e_1 * \ldots * e + n - 1 \hookrightarrow e_n \Leftrightarrow e \mapsto e_1, \ldots, e_n * \mathbf{true}$: e maps to a list of values and there are other values in the heap too

Now we have everything we need to prove assertions. Finally we define a small language, as in [Reynolds, 2002] to use our assertions on:

Program Language

The language used is an imperative language, using assignment, variables, while loops and if statements, as defined in [Hoare, 1969]. Its semantics can be described using a relation $\rightsquigarrow$, which is a transition relation between different configurations that are one of the following:

- $\langle c, (s, h)\rangle$, a pair of a command, c, and a state (s, h)

when a program has not finished its execution, or:

- A state (s, h)
- **abort**, for when the program crashes

When the program has finished its execution.

As the language contains assignment, if statements and while loops, the rules of Hoare logic can be used, and the following new commands:

- **Allocation** is $\langle var\rangle := \mathbf{cons}(\langle exp\rangle, \ldots \langle exp\rangle)$. It initialises a group of consecutive cells in the heap, the values in the heap being the result of the expressions given. It does not say where in the heap the values are stored, so they are allocated beginning at a random free location l. Its semantics is:

  $$\frac{}{\langle v := \mathbf{cons}(e_1, \ldots e_n), (s, h)\rangle \rightsquigarrow ([s|v : l], [h|l : [\![e_1]\!]_{exp}\ s \mid \ldots |l + n - 1 : [\![e_n]\!]_{exp}\ s])}$$

  where $l \ldots l + n - 1$ are address that are not already in the heap. The variable v is updated in the store and its value is now the heap location l.

- **Lookup** is $\langle var \rangle := [\langle exp \rangle]$. When the result of the expression is an address of a heap value, it assigns the variable that heap value in the store, otherwise it aborts. The heap is unchanged. Its semantics when $[\![exp]\!]\ s \in dom\ h$ is :

$$\overline{\langle v := [e], (s, h)\rangle \rightsquigarrow ([s|v : h([\![e]\!]_{exp}\ s)], h)}$$

  Its semantics when the expression's result is not in *dom* is:

$$\overline{\langle v := [e], (s, h)\rangle \rightsquigarrow\ \ e, (s, h)\rangle \rightsquigarrow \mathbf{abort}}$$

- **Mutation** is $[\langle exp \rangle] := \langle exp \rangle$. When the result of the expression on the left hand side is an address in the heap, that address now contains the expression on the right hand side, otherwise it aborts. When the left hand expression's result is in $dom\ h$, its semantics are:

$$\overline{\langle [e] := e', (s, h)\rangle \rightsquigarrow (s, [h|[\![e]\!]_{exp}\ s : [\![e']\!]_{asrt}\ s])}$$

  Its semantics when the expression's result is not in *dom* is:

$$\overline{\langle [e] := e', (s, h)\rangle \rightsquigarrow\ \ e, (s, h)\rangle \rightsquigarrow \mathbf{abort}\}}$$

- **Dipsose** is $\mathbf{dispose}\langle exp \rangle$. When the result of the expression is an address in the heap, the values at that address are removed from the heap. The store is unchanged. Otherwise the program aborts. The semantics when the expression's result is in $dom\ h$ is:

$$\overline{\langle \mathbf{dispose}\ e, (s, h)\rangle \rightsquigarrow (s, h\rceil(dom\ h - \{[\![e]\!]_{exp}\ s\}))}$$

  where $h\rceil(dom\ h - \{[\![e]\!]_{exp}\ s\})$ is the restriction of the heap to the heap where the address that is the result of e has been removed.

  Its semantics when the expression's result is not in *dom* is:

$$\overline{\langle \mathbf{dispose}\ e, (s, h)\rangle \rightsquigarrow \mathbf{abort}}$$

Note that $[f|x : a]$ *is a function that maps* x *to* a *and any other arguments* $y \in dom\ f$ *into* $f\ y$

In the new rules, where we use :=, this is not the same as the assignment rule of Hoare logic, as they refer to the heap, and Hoare logic does not, so we would not use the assignment rule. Instead we use the following inference rules, the global rules, as in [Reynolds, 2002] :

- Allocation:

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

  Here we add the information that the location described by v must contain e'

- Lookup :

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

- Mutation :

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

  Here we say that the location described by e must contain e'

- Deallocation:

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

  Here we just remove everything at the address e from the heap

Chapter 3

Proofs in Separation Logic

3.1 Linked Lists

We define linked lists in the same way as [Reynolds, 2002].

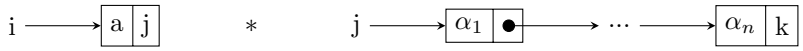
A non-empty linked list is specified by the following predicate:

$$list\ a \cdot \alpha\ (i,k) = \exists j.i \mapsto a, j *\ list\ \alpha\ (j,k)$$

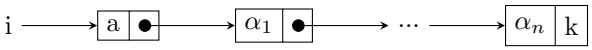
This gives us a list representing the sequence $a \cdot \alpha$. This is a single element a concatenated with a sequence α.

Each element in the list fills two consecutive cells in memory; the first contains the value (of the sequence) we are at currently, and the second cell contains the address of the next value. Therefore i contains the address of the first value in the sequence and k is the contents of the cell after the final value in the list.

In a diagram, the list predicate gives us:



where $\alpha = \alpha_1 \ldots \alpha_n$. This forms the list:



as described above.

Note that because we use the separating conjunction between every element of the list, list segments do not have to have all of their elements in consecutive locations.

An empty list predicate is the following:

$$list\ \epsilon\ (i,j) = \mathbf{emp} \wedge i = j$$

where ϵ is the empty sequence. **emp** represents an empty heap, as we do not refer to any elements.

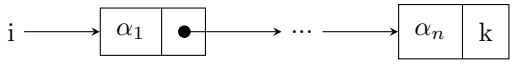
3.1.1 Insert an element at the start of a list

A program to prepend an element to a list can be written as follows, where we start with $list\ \alpha\ (i, k)$ and end with $list\ a \cdot \alpha\ (i, k)$:

$$x := \mathbf{cons}(a, i);$$
$$i := x;$$

We create a list cell at x, containing a new element a and the address i. using the allocation operation, **cons**. The address that is the value of i is that of the first element of the list we are prepending our a element to, so now we have list $a \cdot \alpha\ (x, k)$. Then we change the address at i to be the address in x, so that i is now the address of the first value of the new list.

Pictorially, we start with:

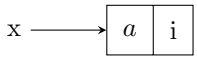


If we assume i points to the address 10 in memory:

Store	$i : 10, x : -, a : a$
Heap	$10 : \alpha_1$, $11 :$ (address of α_2) ...

We have the above values in the store and heap. We include x because the store is a function from variables to values, so currently the store function has this element in its domain, but has no mapping to a value yet, (hence why it is a partial function). The heap maps addresses to values, so currently maps 10 to the first element of i, which is α_1. The other elements of the list can be at any address in memory.

Then we allocate the new following new list element, with two cells:



Giving us the following, assuming that the address of a is 40, and that this and 41 do not contain any elements of the previous list::

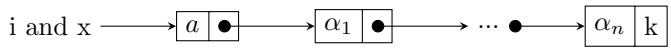
Store	$i : 10, x : 40, a : a$
Heap	$10 : \alpha_1$, $11 :$ (address of α_2), ... , $40 : a$, $41 : 10$

In the store, a evaluates to the logical value a that we want to represent and i is the address of the first value in the original list, so we set 40 and 41 in the heap to these values. In the store, we set x's value to be 40, the address of the first element it is allocating.

Then we change i to be the same value as x, which just changes the value of i in the store, as we are using variable assignment:

Store	$i : 40, x : 40, a : a$
Heap	$10 : \alpha_1$, $11 :$ (address of α_2) , ... , $40 : a$, $41 : 10$

Which now gives us the following picture:



which is a list from i to k (as x and i now point to the same address). We gave specific values for i and x in this demonstration, but in reality, the address x allocates would be some random value each time (as long as it is not already in use in the program).

The logic does not use specific values, so we do not have this problem in our proof:

Precondition	$\{list\ \alpha\ (i,k)\}$
Postcondition	$\{list\ a \cdot \alpha\ (i,k)\}$

$\{list\ \alpha\ (i,k)\}$
$x := \mathbf{cons}(a, i)$
$\{x \mapsto a, i * list\ \alpha\ (i,k)\ \}$
$\{\exists j. x \mapsto a, j * list\ \alpha\ (j,k)\}$
$i := x;$
$\{\exists j. i \mapsto a, j * list\ \alpha\ (j,k)\}$
$\{list\ a \cdot \alpha\ (i,k)\}$

The j we refer to is a placeholder for the old value of i. As we only assume it exists, we do not need the value, so it can be different each time. In our example, $j = 10$.

Starting from the postcondition, we expand it to get the head and tail, then use the assignment rule, to change i to x (in the store). Then we use precondition strengthening to replace the j with the old value of i, so we now have the new element and our original list, which is an expansion of $list\ \alpha\ (i,k)$, giving us the precondition.

3.1.2 Completely deallocate a list

A program to completely deallocate a list can be written as follows, where we start with list $\alpha\ (i,k)$ and end with the empty heap, **emp**:

while $(i \neq k)$ **do**
  $j := [i + 1];$
  dispose $i;$
  dispose $(i + 1);$
  $i := j;$
end while

For example, given the following store and heap, where the first element of the list is at location 10 and the second is at location 23:

Store	i : 10 , j : - , k :20
Heap	10 : α_1, 11 : 23, 23: α_2 ...

we set the variable j to be the contents of $i + 1$, which is 23:

Store	i : 10 , j :23, k: 20
Heap	13 : α_1, 11 : 23, 23: α_2 ...

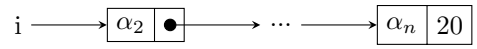
Then we dispose of i and $i + 1$, removing their values from the heap. The store is unchanged:

Store	i : 10, j :23, k :20
Heap	23: α_2 ...

And set the value of i to be the same as the value of j:

Store	i : 23, j :23 , k:20
Heap	23: α_2 ...

So now in memory, we have the following list:



The separation logic proof of its correctness is as follows:

Precondition	$\{\text{list}\ \_\ (i, k)\}$
Loop Invariant	$\{\text{list}\ \_\ (i, k)\}$
Postcondition	$\{\mathbf{emp}\}$.

The underscores represent any possible sequence, as the sequence represented by the list changes in every iteration of the while loop. As our program contains this loop, we now have to give our proof for the program in three parts:

1. Precondition to loop invariant
2. true loop invariant to loop invariant
3. false loop iteration to postcondition

To check up to the loop execution:

$\{\text{list}\ \_\ (i, k)\}$
$\{\text{list}\ \_\ (i, k)\}$

This part is simple, as there is no code before the loop.

To check the loop execution:

while $(i \neq k)$ **do**
$\{\text{list}\ \_\ (i,k) \wedge i \neq k\}$
$\{\exists j.i \mapsto -, j * \text{list}\ \_\ (j,k) \wedge i \neq k\}$
$j := [i+1];$
$\{i \mapsto -, j * \text{list}\ \_\ (j,k) \wedge i \neq k\}$
dispose i;
dispose $(i+1)$;
$\{\text{list}\ \_\ (j,k) \wedge i \neq k\}$
$i := j;$
$\{\text{list}\ \_\ (i,k) \wedge i \neq k\}$
$\{\text{list}\ \_\ (i,k)\}$
end while

Starting from the postcondition, we have a list whose contents we do not specify from i to k. Then as $list\ \_\ (i,k) \wedge i \neq k \Rightarrow list\ \_\ (i,k)$ is true, we can add the $i \neq k$ using postcondition weakening. Then using the assignment rule, replace the i with a j. Replace i and $i+1$ in memory then change the contents of $i+1$ to j, using the assignment rule. Now we can simplify this to get a list from i to k.

To check the loop exit:
$\{\text{list}\ \_\ (i,k) \wedge i = k\}$
$\{\text{list}\ \epsilon\ (i,k)\}$
$\{\mathbf{emp} \wedge i = k\}$
$\{\mathbf{emp}\}$

As $i = k$, the only possible sequence represented by the list is ϵ, giving us $\mathbf{emp} \wedge i = k$, which is the same as **emp**.

3.1.3 Get length of a list

A program to get the length of a list α, which we can refer to using the logical value, $\#\alpha$ can be written as follows:

n := 0;
while $i \neq k$ **do**
j : = [i + 1];
n = n+1;
i := j;
end while

And the proof is as follows:

Precondition	$\{\exists\alpha.\ list\ \alpha\ (m,k)\}$
Loop Invariant	$\{\exists\alpha,\beta,\gamma.\ list\ \beta\ (m,i)\ *\ list\ \gamma\ (i,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma)\}$
Postcondition	$\{\exists\alpha.\ list\ \alpha\ (m,k)\ *\ n=\#\alpha\}$.

Up to loop:

$\{\exists\alpha.\ list\ \alpha\ (m,k)\}$
$\{\exists\alpha.\ \mathbf{emp}\ \wedge m=i\ *\ list\ \alpha\ (m,k)\}$
$\{\exists\alpha.\ \mathbf{emp}\ \wedge m=i\ *\ list\ \alpha\ (m,k)\}$
$\{\exists\alpha,\gamma.\ list\ \epsilon\ (m,i)\ *\ list\ \gamma\ (i,k)\}\ \wedge\alpha=(\epsilon\cdot\gamma)\}$
n : = 0;
$\{\exists\alpha,\gamma.\ list\ \epsilon\ (m,i)\ *\ list\ \gamma\ (i,k)\}\ \wedge\alpha=(\epsilon\cdot\gamma)\wedge n=0)\}$
$\{\exists\alpha,\gamma,\beta.\ list\ \beta\ (m,i)\ *\ list\ \gamma\ (i,k)\}\ \wedge\alpha=(\beta\cdot\gamma)\wedge n=\#\beta)\}$

From the precondition, we can add an empty list ϵ from m to i, where $m=i$, Then as $m=i$, $list\ \alpha(m,k)=list\ \alpha(i,k)$. Renaming α to γ gives us a list from m to k, representing the sequence $(\epsilon\cdot\gamma)=\alpha$.

Now we assign 0 to n. Renaming our empty list to β means that $\#\beta=0=n$, so we can see that $n=\#\beta$. Now we have everything we need for the loop invariant.

Loop :

while $i\neq k$ **do**
$\{\exists\alpha,\beta,\gamma.\ list\ \beta\ (m,i)\ *\ list\ \gamma\ (i,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma)\ \wedge\ i\neq k\}$
$\{\exists\alpha,\beta,\gamma',\gamma''.\ list\ \beta\ (m,i)\ *\ \exists j.i\mapsto\gamma',j\ *\ list\ \gamma''\ (j,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma'\cdot\gamma'')\ \wedge\ i\neq k\}$
j : = [i + 1];
$\{\exists\alpha,\beta,\gamma',\gamma''.list\ \beta\ (m,i)\ *\ i\mapsto\gamma',j\ *\ list\ \gamma''\ (j,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma'\cdot\gamma'')\ \wedge\ i\neq k\}$
$\{\exists\alpha,\beta,\gamma',\gamma''.list\ (\beta\cdot\gamma')\ (m,j)\ *\ list\ \gamma''\ (j,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma'\cdot\gamma'')\ \wedge\ i\neq k\}$
$\{\exists\alpha,\beta,\gamma''.list\ \beta\ (m,j)*list\ \gamma''\ (j,k)*n{+}1=\#\beta\wedge\alpha=(\beta{\cdot}\gamma'')\wedge i\neq k)\}$
n = n+1;
$\{\exists\alpha,\beta,\gamma''.list\ \beta\ (m,j)*list\ \gamma''\ (j,k)*n=\#\beta\wedge\alpha=(\beta{\cdot}\gamma'')\wedge i\neq k\}$
i := j;
$\{\exists\alpha,\beta,\gamma''.list\ \beta\ (m,i)\ *\ list\ \gamma''\ (i,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma'')\}$
$\{\exists\alpha,\beta,\gamma.list\ \beta\ (m,i)\ *\ list\ \gamma\ (i,k)\ *\ n=\#\beta\ \wedge\ \alpha=(\beta\cdot\gamma)\}$
end while

We start with the loop invariant and true while condition. Then we split $list\ \gamma\ (i,k)$ into the head element γ' and tail list γ'', giving us $\exists j.i\mapsto\gamma',j\ *\ list\ \gamma''(j,k)$. Therefore $\alpha=\gamma'\cdot\gamma''$).

Then we set j to be the value in $[i+1]$, to remove that existential quantifier. Then we combine $list\beta(m,i)$ and our γ' element to get a new sequence $*\beta \cdot \gamma')$ which we set to the value of β, giving us $list\ \beta\ (m,j)$. Now we have changed the β to include an extra element, $\#\beta = n+1$. This means we can now apply our assignment rule using the backwards reasoning to get $\#\beta = n$ again for the end of the loop.

Now we have $list\ \beta\ (m,j)\ *\ list\ \gamma''\ (j,k)$, so we rename the γ'' to γ to get back to the loop invariant. So now we have a list β with one more element (the original $(\beta \cdot \gamma')$) and a list γ with one less element (the original γ''), so n is still $\#\beta$.

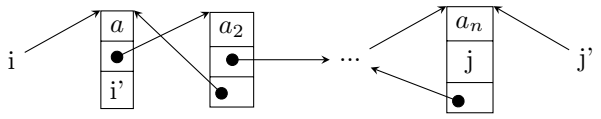
After Loop :

$\{\exists\alpha, \beta, \gamma.\ list\ \beta\ (m,i)\ *\ list\ \gamma\ (i,k)\ *\ n = \#\beta\ \wedge\ \alpha = (\beta \cdot \gamma)\ \wedge\ i = k\}$
$\{\exists\alpha, \beta,\ list\ \beta\ (m,i)\ *\ list\ \epsilon\ (i,k)\ *\ n = \#\beta\ \wedge\ \alpha = (\beta \cdot \epsilon)\}$
$\{\exists\alpha, \beta,\ list\ \beta\ (m,i)\ *\ \mathbf{emp}\ \wedge\ i = k\ *\ n = \#\beta\ \wedge\ \alpha = \beta\}$
$\{\exists\alpha.list\ \alpha\ (m,k)\ * n = \#\alpha\ \wedge\ \alpha = \beta\ \wedge\ i = k\}$
$\{\exists\alpha.\ list\ \alpha\ (m,k)\ * n = \#\alpha\}$

Starting from the loop invariant and false loop condition, we see that we have a list from i to k, but $i = k$, so this must be the empty list and therefore the sequence γ is now the empty sequence, ϵ. Now we have $\#\alpha = \#(\beta \cdot \epsilon) = \#\beta$, so $list\ \beta\ (m,k)$ is the same as $list\ \alpha\ (m,k)$ and $n = \#\alpha$. This is the same as the postcondition for the program.

3.2 Doubly Linked Lists

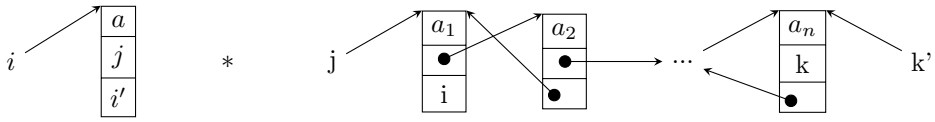
We also define doubly linked lists in the same way as [Reynolds, 2002]. The following is the list *dlist* (i, i', j, j'):



A non-empty doubly linked list can be defined with the following predicate:

$$dlist\ a \cdot \alpha\ (i, i', k, k') = \exists j.i \mapsto a, j, i' *\ dlist\ \alpha\ (j, i, k, k')$$

Which looks like this:



This gives us a doubly linked list representing the sequence $a \cdot \alpha$. This is a single element a concatenated with a sequence α (where $\alpha = \alpha_1 \ldots \alpha_n$).

Each element in the list now fills *three* consecutive cells in memory; the first contains the value (of the sequence) we are at currently, and the second cell contains the address of the next value, and the third cell contains the address of the previous value. Therefore given a doubly linked list *dlist* α (i, i', j, j'), i is the address of the first element of the first cell, i' is the address stored as the third cell of the first element (as the element has no previous address). j is the address of the final element in the sequence α and j' is the address in the second cell of this element, as there is no next value in the sequence. *(so i' and j' can be any values in memory)*

An empty doubly linked list predicate is the following:

$$dlist\ \epsilon\ (i, i'j, j') = \mathbf{emp}\ \wedge\ i = j\ \wedge\ i' = j'$$

where ϵ is the empty sequence. **emp** represents an empty heap, as we do not refer to any elements.

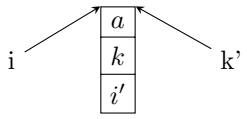
Therefore a single linked list predicate should be the following:

$$dlist\ a\ (i, i'k, k') = \exists j.i \mapsto a, j, i'\ * dlist\ \epsilon\ (j, i, k, k')$$

So let $j = k$, so we have:

$$dlist\ a\ (i, i'k, k') = i \mapsto a, k, i'\ * dlist\ \epsilon\ (k, i, k, k')$$

Which is the following picture:



So it will be the case that $k = k$ and $i = k'$, so we can have an empty list here, giving us:

$$dlist\ a\ (i, i'k, k') = i \mapsto a, k, i'\ *\ \mathbf{emp}\ \wedge\ k = k\ \wedge\ i = k'$$

which can be reduced to:

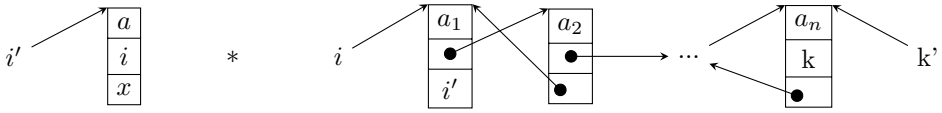
$$dlist\ a\ (i, i'k, k') = i \mapsto a, k, i'\ \wedge\ i = k'$$

3.2.1 Insert an element at the start of a doubly linked list

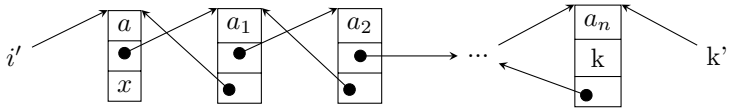
A program to prepend an element to a doubly linked list can be written as follows, where we start with $dlist\ \alpha\ (i, i', k, k')$ and end with $dlist\ a \cdot \alpha\ (i, i', k, k')$:

$$\begin{array}{l} i' := \mathbf{cons}(a, i, x); \\ i := i' \\ i' := x \end{array}$$

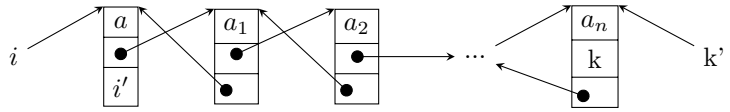
We create a list cell at i', containing a new element a and the addresses i and x. using the allocation operation, **cons**:



Which gives us the following list:



So we have a dlist from i' to k'. Then we just change these values to i and k' to get the postcondition:



Finally we have our list, $dlist\ a \cdot \alpha\ (i, i', k, k')$.

A proof of its correctness is as follows:

Precondition	$\{dlist\ \alpha\ (i, i', k, k')\}$
Postcondition	$\{dlist\ a \cdot \alpha\ (i, i, k, k')\}$

$\{$ dlist $\alpha\ (i, i', k, k')\}$
$i' := \mathbf{cons}(a, i, x);$
$\{i' \mapsto a, i, x\ *\ dlist\ \alpha\ (i, i', k, k')\}$
$\{\exists i.\ i' \mapsto a, i, x\ *\ dlist\ \alpha\ (i, i', k, k')\}$
$\{dlist\ a \cdot \alpha\ (i', x, k, k')\}$
$i := i';$
$i' := x;$
$\{dlist\ a \cdot \alpha\ (i, i', k, k')\}$

Starting with the precondition, we add our value of i' we have just allocated to the heap with **cons**, then use the definition of *dlist* to simplify this heap to a dlist for $(a \cdot \alpha)$ from i' to k'. We then use the assignment rule, to show that we can change the values of i and i, to get the required variables for the postcondition.

3.2.2 Completely deallocate a doubly linked list

A program to completely deallocate a doubly linked list can be written as follows, where we start with $\{\ dlist\ \_\ (i, i', k, k')\}$ and end with $\{\mathbf{emp}\}$, the empty heap:

while $(i \neq k')$ **do**
  $j := [i + 1];$
  dispose $i;$
  dispose $i + 1;$
  dispose $i + 2;$
  $i := j;$
  $i' = [i + 2];$
end while
dispose $i;$
dispose $i + 1;$
dispose $i + 2;$

We set the location of the second value in the dllist to be the contents of j, then deallocate the first element of the list. We then set i to be this value and i' to be the previous value of the second element, so we can keep going round the loop again with a sublist at i, until the list contains only one element and $i = k'$. Then we dispose the last list element, giving us an empty heap.

A proof of its correctness is as follows:

Precondition	$\{dlist\ \_\ (i, i', k, k')\}$
Loop Invariant	$\{dlist\ \_\ (i, i', k, k')\}$
Postcondition	$\{\mathbf{emp}\}$.

There is no code before the loop execution, so up to that point we have:

$\{dlist\ \_\ (i, i', k, k')\}$
$\{dlist\ \_\ (i, i', k, k')\}$

Loop execution:

$\{\text{dlist } \_\ (i, i', k, k')\}$
while $(i \neq k')$ **do**
$\{\text{dlist } \_\ (i, i', k, k') \wedge i \neq k'\}$
$\{\exists j.\ i \mapsto -, j, i' * \text{dlist } \_\ (j, i, k, k') \wedge i \neq k'\}$
$j := [i + 1];$
$\{\ i \mapsto -, j, i' * \text{dlist } \_\ (j, i, k, k') \wedge i \neq k'\}$
dispose i;
dispose $i + 1$;
dispose $i + 2$;
$\{\text{dlist } \_\ (j, nil, k, k')\}$
$i := j;$
$\{\text{dlist } \_\ (i, nil, k, k') \wedge i \neq k'\}$
$\{\exists j.\ i \mapsto -, j, nil * \text{dlist } \_\ (j, i, k, k') \wedge i \neq k'\}$
$i' = [i + 2];$
$\{\exists j.\ i \mapsto -, j, i' * \text{dlist } \_\ (j, i, k, k') \wedge i \neq k'\}$
$\{\ \text{dlist } \_\ (i, i', k, k') \wedge i \neq k'\}$
end while
$\{\text{dlist } \_\ (i, i', k, k')\}$

Starting with a dlist at (i, i', k, k') representing any sequence, and the true loop condition, we expand the definition, to get the locations of the cells of the first element separately. Then we set the variable j to be the j referred to by the dlist definition.

Then we dispose all the cells that make the element at i, giving us $dlist\ \_\ (j, nil, k, k')$. Then we set i to be j, as this is now the first list element, so we just use the assignment rule here. Then set i' to be its new value according to the dlist definition. Simplifying the dlist definition, we will now have $dlist\ \_\ (i, i', k.k')$, which matches the loop invariant.

Loop Exit:

$\{dlist\ \_\ (i, i', k, k')\ \wedge\ i = k'\}$
$\{dlist\ a\ (i, i', k, k')\ \wedge\ i = k'\}$
$\{i \mapsto a, k, i'\ \wedge\ i = k'\ \wedge\ i = k'\}$
$\{i \mapsto a, k, i'\ \wedge\ i = k'\}$
dispose i
dispose $i + 1$
dispose $i + 2$
$\{\mathbf{emp}\}$

For the list to be a dlist it must be one where $i = k'$, which is only possible in a singleton list. As the list is now a singleton list at i, disposing its next three cells disposes the single element and gives us an empty heap. We can see this using the definition of dlist and the dispose rule.

3.3 Trees

We define trees in the same way as [Reynolds, 2002], in which they are defined as *S-expressions*.

3.3.1 S-expressions

The structures represented by trees in [Reynolds, 2002] are called *S-expressions* and are specified in the following way:

$\tau \in S - exps \Leftrightarrow \tau \in Atoms$
$\vee \ \tau = \tau_1, \tau_2 \in S - exps$

τ is an S-expression if it is an atom, or is the concatenation of two other S-expressions. Therefore, an expression is a binary tree.

Atoms

A atom is a tree containing a single element and is defined by the following predicate:

$$tree\ \alpha\ (i) \Leftrightarrow \mathbf{emp}\ \wedge\ i = \alpha$$

This says that i and α are the same variables, which refer to some undefined location, as our heap is empty.

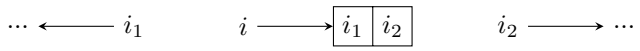
In our programs, the condition $isAtom()$ is only true when the tree given exactly matches the above predicate.

Expressions

An expression τ can be formed by concatenating two other expressions τ_1 and τ_2. It will be a non-empty tree, specified by the following predicate:

$$tree\ (\tau_1 \cdot \tau_2\ (i) \Leftrightarrow \exists i_1, i_2 . i \mapsto i_1, i_2 *\ tree\ \tau_1\ (i_2) *\ tree\ \tau_2(i_2)$$

This gives us a tree where i contains the location of the left subtree, i_1, which defines τ_1. $i + 1$ contains the location of the right subtree, i_2, which defines τ_2. Therefore all nodes of the tree take up two consecutive memory locations, where they represent two subexpressions that are being concatenated, or a combination of that and atoms, or just atoms. When atoms are in the subexpressions, i_1 and/or i_2 will be pointers to an empty heap, as in the atom predicate.



3.3.2 Completely deallocate a tree

A program to completely deallocate a tree can be written as follows, where we start with $\{\exists\tau.\ \text{tree}\ \tau\ (i)\}$ and end with the empty heap, **emp**:

```
disposeTree(i){
if isAtom(i) then
  skip;
else
  l := [i];
  r := [i + 1]
  dispose i;
  dispose i + 1;
  disposeTree(l);
  disposeTree(r);
end if
}
```

We first check if the tree is an atom. If so then we already have an empty heap. Otherwise, we set l to be the location of the left subtree (the contents of $[i]$) and the set r to be the location of the right subtree (the contents of $[i+1]$. Now we can dispose i and $i+1$, so we just have the subtrees. We call our function again to dispose the entire left subtree, then do the same for the right subtree, so we have an empty heap.

For example, if we had the following store and heap:

Store	$i : 10, i_1 : 20, i_2 : 40, l : -, r : -$
Heap	$10 : 20, 11 : 40, 20 : \tau_1, 40 : \tau_2, ...$

We can see the tree is not atomic, becuase i does not point to an empty heap, so we set l to the contents, of i (so contents of 10) and r to the contents, of $i+1$:

Store	$i : 10, i_1 : 20, i_2 : 40, l : 20, r : 40$
Heap	$10 : 20, 11 : 40, 20 : \tau_1, 40 : \tau_2, ...$

Then we dispose of i_1 and i_2, so we remove the values at these locations from the heap:

Store	$i : 10, i_1 : 20, i_2 : 40, l : 20, r : 40$
Heap	$10 : -, 11 : -, 20 : \tau_1, 40 : \tau_2, ...$

Then we do the whole function for the trees at l and r, removing them from the heap too:

Store	$i : 10, i_1 : 20, i_2 : 40, l : 20, r : 40$
Heap	$10 : -, 11 : -, 20 : -, 40 : -, ...$

Now we have no ownership of anything in the heap, so it is empty.

A proof of its correctness is the following:

Precondition	$\{\exists\tau.\ \text{tree}\ \tau\ (i))\}$
Postcondition	$\{\mathbf{emp}\}$

disposeTree$(i)\{$
$\{\exists\tau.\ \text{tree}\ \tau\ (i)\}$
if $isAtom(i)$ **then**
  $\{\exists\tau.\ isAtom(\tau) \wedge \mathbf{emp} \wedge i = \tau\}$
  skip;
  $\{\exists\tau.\ isAtom(\tau) \wedge \mathbf{emp} \wedge i = \tau\}$
  $\{\mathbf{emp}\}$
else
  $\{\ \text{tree}\ (\tau_1 \cdot \tau_2)(i)\}$
  $\{\ \exists i_1, i_2.i \mapsto i_1, i_2 * \text{tree}\ \tau_1(i_1) * \text{tree}$
  $\tau_2(i_2)\}$
  $l := [i]$;
  $r := [i+1]$
  $\{i \mapsto l, r * \text{tree}\ \tau_1(l) * \text{tree}\ \tau_2(r)\}$
  dispose i;
  dispose $i+1$;
  $\{\ \text{tree}\ \tau_1(l) * \text{tree}\ \tau_2(r)\}$
  disposeTree(l);
  $\{\ \text{tree}\ \tau_2(r)\}$
  disposeTree(r);
  $\{\mathbf{emp}\}$
end if
$\}$

Given a tree representing an S-expression τ at i, if it is an atom, by its predicate we can see it is already an empty heap, so we get the empty heap using postcondition weakening.

Otherwise, $\tau = \tau_1 \cdot \tau_2$, so we set the location of l to that of τ_1 and the location of r to that of τ_2 using the assignment rule. Then we use the dispose rule to remove i_1 and i_2 from the heap. Now we can use the pre and post conditions of our function in conjunction with the frame rule for $*$, when we call it again to dispose the subtree at l, to get from $tree\ \tau_1\ (l)\ *\ tree\ \tau_2\ (r)$ to $tree\ \tau_2\ (r)$ and then apply the pre and post condition for the function to get **emp**.

3.3.3 Get number of elements in a tree

A program to get the number of elements of a tree can be written as follows, where we start with $\{\exists\tau.\ tree\ \tau\ (i)\ *n = m\ \wedge\ m \geq 0\}$ and end with $\{\exists\tau.\ tree\ \tau\ (i)\ *n = size(\tau)\}$

$count(i)\{$
if $isAtom(i)$ **then**
  $n := n + 1$
else
  $count([i])$;
  $count([i + 1])$;
end if
$\}$

A proof of its correctness is the following:

Precondition	$\{\exists \tau.\ tree\ \tau\ (i)\ *n = m\ \wedge\ m \geq 0\}$
Postcondition	$\{\exists \tau.\ tree\ \tau\ (i)\ *n = m + size(\tau)\ \wedge\ m \geq 0\}$

We give a precondition involving another value m because the function is recursive, so m represents the size of the part of the tree we have already used before the recursive call to the function, which may or may not be empty.

We also use a logical function, $size$, to compare the number of elements we have calculated with the actual size of the tree:

$$size(\tau) = \tau = \alpha \mapsto 1\ \vee\ \tau = \tau_1 \cdot \tau_2 \mapsto size(\tau_1) + size(\tau_2)$$

Then the proof is the following:

$count(i)\{$
$\{\exists \tau.\ tree\ \tau\ (i)\ *n = m\ \wedge\ m \geq 0\}$
if $isAtom(i)$ **then**
  $\{\exists \tau.\mathbf{emp}\ \wedge\ i = \tau\ *n = m\ \wedge\ m \geq 0\}$
  $\{\exists \tau.\mathbf{emp}\ \wedge\ i = \tau\ *n + 1 = m + 1\ \wedge\ m \geq 0\}$
  $n := n + 1$
  $\{\exists \tau.\mathbf{emp}\ \wedge\ i = \tau\ *n = m + 1\ \wedge\ m \geq 0\}$
  $\{\exists \tau.\mathbf{emp}\ \wedge\ i = \tau\ *n = m + size(\tau)\ \wedge\ m \geq 0\}$
else
  $\{\exists i_1, i_2.i \mapsto i_1, i_2 * \text{tree}\ \tau_1(i_1) * \text{tree}\ \tau_2(i_2)\ *n = m\ \wedge$
  $m \geq 0\}$
  $count([i])$;
  $\{\exists i_1, i_2.i \mapsto i_1, i_2 * \text{tree}\ \tau_1(i_1) * \text{tree}\ \tau_2(i_2)\ *n = m +$
  $size(\tau_1)\ \wedge\ m \geq 0\}$
  $count([i + 1])$;
  $\{\exists i_1, i_2.i \mapsto i_1, i_2 * \text{tree}\ \tau_1(i_1) * \text{tree}\ \tau_2(i_2)\ *n = m +$
  $size(\tau_1) + size(\tau_2)\ \wedge\ m \geq 0\}$
  $\{\exists \tau.\ tree\ \tau\ (i)\ *n = m + size(\tau)\ \wedge\ m \geq 0\}$
end if
$\}$

Given a tree representing an S-expression τ at i, if it is an atom, then we add one to whatever our n value. Because size of an atom is 1, by using the assignment rule and simplfying, we get from $n = m$ to $n = m + size(\tau)$.

Otherwise $\tau = \tau_1 \cdot \tau_2$, so we use the frame rule and the pre and post condition of *count* to change $tree\ \tau_1(i_1)\ *\ n = m\ \wedge\ m \geq 0$ to $tree\tau_1(i_1)\ *\ n = m + size(\tau_1)\ \wedge\ m \geq 0$ and then change $tree\ \tau_2(i_2)\ *\ n = m + size(\tau_1)\ \wedge\ m \geq 0$ to $tree\ \tau_2(i_2)\ *\ n = m + size(\tau_1) + size(\tau_2)\ \wedge\ m \geq 0$. Then by defintition of size we have the postcondition.

Chapter 4

Tools

4.1 Smallfoot

Smallfoot [Berdine et al., 2006] is an automatic verification tool for checking separation logic assertions of sequential and concurrent programs that manipulate shared data structures. For now we just focus on sequential programs. The tool conducts *shape analysis* of the programs, meaning that it checks that the memory is still allocated correctly at the end of a program, but doesn't check that the contents have been changed correctly. (For example, in a program to reverse a list, it would check that we still have a valid list in memory, but not that the list's data is the reverse of the original data). The authors impose this restriction to ensure the tool can be fully automatic, as explained in [Berdine et al., 2006].

The tool focuses on a small subset of possible data structures, including predicates for singly and doubly linked lists, trees and xor-linked lists.

To write a correct program in Smallfoot, we must give pre and post conditions and loop invariants. The tool uses a symbolic execution algorithm, described in [Berdine et al., 2005] to decide Hoare triples that are created for symbolic instructions. These Hoare triples are converted to verification conditions for the symbolic instructions, which the symbolic execution algorithm decides, as described in [Berdine et al., 2006].

Singly Linked Lists

The singly linked list predicate is specified by the following form:

```
""list" "(" (field ";")? form_exp ")".
```

This is the word list followed by brackets containing any or no fields (because of the question mark), which give names to the elements in the lists. Without specifying the fields we have the following:

```
[ list (x)]
```

This is a predicate that says we have a list from x in memory. By default, we refer to members of the list using hd and tl.

We also have a list segment predicate, lseg.

```
[ lseg (x,y)]
```

Which refers to a partial list from x to y in memory.

Now we can give an example program. The following is a program that makes no changes to the memory:

```
list_test(x) [list (x)] {
  local a;
  a = 5;
} [list (x)]
```

The precondition is a list at x. *Note that we use square brackets, but we are still referring to partial correctness specifications, this is just Smallfoot syntax.* We then set a local variable to 5. This does not change our list at x, so the postcondition will be the same as our precondition, as t is a local variable that gets deallocated at the end of the function.

Therefore, we get the following output when we run the program in the terminal:

```
Function list_test

Valid
```

If we run Smallfoot with the option "$-all\_states$", we can see the intermediate states of the proof:

```
Function list_test
File "listTest.sf", line 3, characters 2-8:
Intermediate State:
[listseg(tl; x, 0)]
a=5;
[listseg(tl; x, 0)]

File "listTest.sf", line 4, characters 2-12:
Intermediate State:
[5==a * listseg(tl; x, 0)]
[listseg(tl; x, 0)]
```

```
Valid
```

Before and after setting a to 5, we have the list segment from x to nil and afterwards we still have that. The second intermediate state says that $a = 5$ and we have our list segment from before. Then we get the postcondition from this using the frame rule, as a is not a free variable in the list predicate.

Deallocating a singly linked list

The following is a Smallfoot version of our deallocating a singly linked list :

```
list_deallocate(i) [list (i)] {
  while(i != NULL) [list (i)] {
    j = i->tl;
    dispose i;
    i = j;
  }
} [emp]
```

The precondition is a list from i and the postcondition is the empty heap. We set a variable j to be the remainder of the list then deallocate i. Then set i to be the rest of the list, to maintain the loop invariant that we have a list from i. At the final loop exit, we dispose the last element so $j = NULL$ and our heap is empty, giving us the correct postcondition.

Therefore running this function in Smallfoot gives the following output:

```
Function list_deallocate

Valid
```

The detailed output includes the following:

```
File "listTest.sf", line 3, characters 4-14:
Intermediate State:
[0!=i * listseg(tl; i, 0)]
j=i->tl;
dispose(i);
i=j;
[listseg(tl; i, 0)]
```

When i is not empty, we start with the following in each iteration, [0!=i * listseg(tl; i, 0)]. Then we end with the following [listseg(tl; i, 0)]

When i is empty we have the following:

```
File "listTest.sf", line 7, characters 2-7:
Intermediate State:
[0==i * listseg(tl; i, 0)]
[emp]
```

So we can see that the Hoare triples created give a sensible proof.

Trees

The tree predicate is specified by the following form:

```
"tree" "(" (field ";" field ";")? form_exp ")"
```

This is the word tree followed by brackets containing any or no fields (because of the question mark), which give names to the branches of the tree. Without specifying the fields we have the following:

```
[ tree (t)]
```

Deallocating a singly linked list

The following is a Smallfoot version of our deallocating a tree :

```
tree_deallocate(t) [tree (t)] {
  local l,r;
  if (t != NULL){
    l = t->l;
    r = t->r;
    dispose(t);
    tree_deallocate(l);
    tree_deallocate(r);
  }
} [emp]
```

The precondition is a tree where the root is t and the postcondition is the empty heap. We set local variables l and r to be the left and right subtrees and then dispose the root. Now we deallocate the left and right subtrees by calling the method again. This means that eventually we will get the postcondition of an empty heap.

Therefore running this function in Smallfoot gives the following output:

```
Function tree_deallocate

Valid
```

In the intermediate states we have:

```
File "treeTest.sf", line 11, characters 2-7:
Intermediate State:
[0==t * tree(l,r;t)]
[emp]
```

So when t is an empty heap, we already have the postcondition.

Otherwise we call the function again:

```
File "treeTest.sf", line 5, characters 4-13:
Intermediate State:
[0!=t * tree(l,r;t)]
l=t->l;
r=t->r;
dispose(t);
fcall({},emp, t_0==l);
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]
```

So we can see that the Hoare triples created give a sensible proof.

4.2 HIP

HIP [Chin et al., 2007] is an automatic verification tool that can be used to create specifications for programs in a simple imperative language. Each method and loop in the code has a pre and post condition, in the form of "requires" and "ensures" statements. HIP then creates a set of implication checks between formulae of separation logic and other logical statements (to solve predicates on sets/bags).

The implication checks are converted to pure logic constraints by another tool, SLEEK and sent to a theorem prover. The theorem prover used by default and used here is the Omega Calculator.

HIP has been applied to real world problems such as verifying operating system components [Ferreira et al., 2014], memory usage [He et al., 2009].

4.2.1 Specification Language

Specifiications in HIP are written using a simple imperative language in input files, which contain three parts:

- **Data Declarations**: First create the data types you want to use; int, bool, float and arrays are built-in, but any other data structure (such as lists and trees) must be created by the user in the following way:

  ```
  data struct_name { (base_type field_name;)*}
  ```

  where struct_name is replaced with the name of the data structure and inside the brackets we have any number of fields, with their base type. For example, for a list element we can have "data node int val; node next;".

- **Predicate Definitions**: All predicates are of the form:

  pred_name <v*> $==$ ϕ inv π ;

  where pred_name is replaced with the name of the predicate, v can be replaced with any (or no) locations. On the other side of the ==, we have the actual formula. ϕ is a separation logic formula either in disjunctive normal form, or using a special syntax. We use the disjunctive normal form predicates in our proofs.

  and π is an invariant on the whole heap.

- **Method Declarations**: The actual code we want to verify goes at the end of the file. HIP verifies every method individually, so all the code must be in the following form:

  ```
  return_type name
    requires : pred_1
  ```

```
  ensures : pred2;
{
  method_line;*
}
```

We give a return type (or void if no return type is needed) and the method name. Then before we give the opening bracket for the method, we give a precondition $pred_1$ after the word requires, that is a predicate as described above, and a postcondition $pred_2$ after the word ensures.

4.2.2 Simple Example

Here is a simple example, were we start with a heap containing anything and end with the result of the input multiplied by 2:

```
int example (int x)
 requires true;
 ensures res=x*2;
{
  x = x * 2;
  return x;
}
```

res is used to refer to the result of the method

Running the program in a file in HIP gives the following output:

```
Checking procedure example$int...
Procedure example$int SUCCESS.
Stop Omega... 30 invocations
0 false contexts at: ()
```

So it is correct.

4.2.3 Singly linked lists

To write programs for singly linked lists in HIP, we must first write some code to represent a data structure for a list, as this is not built-in. This can be written as follows:

```
data node {
  int val;
  node next;
}
```

So we have a structure containing an integer value and the next node in the list.

Then we can define the following predicate:

```
ll<> == emp & self=null
  or self::node<_,q> * q::ll<>
  inv true;
```

which says that "ll $\langle\,\rangle$" represents a list in memory if it is an empty heap, or at the location of the value (given by "self"), there is a node where the value can be anything, but the next node is at a location defined by q, and q and the rest of the list is at q.

Insert an element at the start of a list

Now we can write a function using this data structure, for example, to insert another element at the start of a list:

```
void insert(node x, node y)
 requires x::ll<> * y::node<_,z>
 ensures y::ll<>;
{
  y.next = x;
}
```

Our precondition is a predicate that says there is a list at x and some node at y where we don't care what the value is, but there is some next value that may not be x. Then the postcondition says we now only have a list at y, as we have now added x to the end of it to form a list.

Running this in HIP gives the following output:

```
Checking procedure insert$node~node...
Procedure insert$node~node SUCCESS.
Stop Omega... 45 invocations
0 false contexts at: ()
```

So it is correct.

Get length of a list

Here is a program for getting the length of a list:

```
int length (node x, int n)
 requires x::ll<> & x != null
 ensures x::ll<>;
{
  if (x.next != null)
    length(x.next, n);
  return (n+1);
}
```

Our precondition is a predicate that says there is a list at x that is not empty and the postcondition says we have a list at x, as we have not changed the structure.

Running this in HIP gives the following output:

```
Checking procedure length$node~int...
Procedure length$node~int SUCCESS.
Stop Omega... 63 invocations
0 false contexts at: ()
```

This only says that we maintain something that is a list, so we can create another list predicate with more information about the size of the list:

```
lls<n> == emp & self=null & n=0 or self::node<_,q>*q::lls<n-1>
                  inv n>=0;
```

Now we have a predicate *lls* with a parameter n, which is the number of nodes in that list. The new predicate says we either have an empty heap and n =0 or the location the predicate describes contains a node with any value and the next node is a list of size $n-1$.

Now we can rewrite our length program with new predicates:

```
int length (node x, int n)
 requires x::lls<m> $\&$ x != null
 ensures x::lls<m> $\&$ res=n+1;;
{
  if (x.next != null)
    length(x.next, n);
  return (n+1);
}
```

Now we can prove that the function does not change the size of the list, but does add one to the result each time it is called recursively.

Running in HIP gives the following output:

```
Checking procedure length$node~int...
Procedure length$node~int SUCCESS.
Stop Omega... 88 invocations
0 false contexts at: ()
```

So it is correct.

4.2.4 Doubly Linked Lists

To write programs for doubly linked lists in HIP, we need to modify the data structure for a list to include another node value for the previous node:

```
data node2 {
  int val;
  node2 next;
  node2 prev;
}
```

Then we can define the following predicate:

```
dll<p> ==  emp & self=null
     or  self::node2<_,q,p> * q::dll<self>
     inv true;
```

which says that "dll ⟨ p ⟩" represents a doubly linked list in memory if it an empty heap, or at the location of the value (given by "self") there is a node with any value, a previous node at p and the next node is at q, and this node at q represents a doubly linked list with the previous node being this location we are currently at.

Insert an element at the start of a doubly linked list

Now we can write a function using this data structure, for example, to insert another element at the start of a doubly linked list:

```
void insert(node2 x, node2 y)
 requires x::dll<p> * y::node2<_,r,s>
 ensures y::dll<s>;
{
  y.next = x;
  if (x != null)
   x.prev = y;
}
```

Our precondition is a predicate that says there is a doubly linked list at x, where the previous node is p and some node at y where the previous node is s and the next node is r. Then the postcondition is a doubly linked list from y, with the previous node now being s, the previous node of y.

Running this in HIP gives the following output:

```
Checking procedure insert$node2~node2...
Procedure insert$node2~node2 SUCCESS.
Stop Omega... 68 invocations
0 false contexts at: ()
```

So it is correct.

Get length of a doubly linked list

As with our list predicate, we can also modify the doubly linked list predicate to include length, by adding a length parameter n to it:

```
dlls<p,n> == emp & self=null & n=0 or self::node2<_,q,p>*q::dlls<self,n-1>
                 inv n>=0;
```

Now the rest of the list at q must have length $n - 1$.

Here is a program for length of lists, using our new doubly linked list predicate:

```
int length (node2 x, int n)
 requires x::dlls<p,m> & x != null
 ensures x::dlls<p,m> & res=n+1;
{
  if (x.next != null)
    length(x.next, n);
  return (n+1);
}
```

The program is exactly the same as that for singly linked lists, but the predicates are changed to include our new doubly linked list predicate.

Running in HIP gives the following output:

```
Checking procedure length$node2~int...
Procedure length$node2~int SUCCESS.
Stop Omega... 91 invocations
0 false contexts at: ()
```

4.2.5 Trees

We can use our data structure for doubly linked lists and replace the previous and next nodes with left and right subtrees:

```
data node2 {
  int val;
  node2 left;
  node2 right;
}
```

Then we can define the following predicate:

```
tree<> == emp & self=null
 or self::node2<_,l,r> * l::tree<> * r::tree<>
  inv true;
```

which says that "tree ⟨⟩" represents a binary tree in memory if it is an empty heap, or at the location of the value (given by "self"), there is a node where the value can be anything, the left value defines another tree at l and the right value defines another tree at r.

Number of elements in a tree

Now we can write a function using this data structure, for example, to count the number of elements in our tree:

```
int size (node2 t)
  requires t::tree<> & t!=null
  ensures t::tree<>;
{
 if((t.left == null) && (t.right == null))
   return 1;
 else{
   if(t.left == null)
     return (1 + size(t.right));
   else
     return (1 + size(t.left));
 }
}
```

Our precondition is a non null tree at t and our postcondition is that we still have a tree at t.

Running this in HIP gives us:

```
Checking procedure size$node2...
Procedure size$node2 SUCCESS.
```

```
Stop Omega... 181 invocations
0 false contexts at: ()
```

As our predicate only specifies the shape of the tree, we want a predicate that also gives information about the size of the tree:

```
trees<h> == emp & self=null & h=0
  or self::node2<_,l,r>*l::trees<h1>*r::trees<h2> & h=1+h1+h2
  inv h>=0;
```

Now we have a tree with a size value h, where either we have an empty heap and $h = 0$, or we have a node at the location given by "self" , which contains any value , a subtree at l and a subtree at r and the h is equal to the total of the size of the left subtree ($h1$), the size of the right subtree ($h2$) and 1.

Running our function in HIP gives us:

```
Procedure size\$node2 FAIL.(2)

Exception Failure("Post condition cannot be derived.") Occurred!

Error(s) detected when checking procedure size\$node2
Stop Omega... 215 invocations
0 false contexts at: ()
```

So our function we wrote previously was actually wrong! Using shape only predicates can make mistakes like this easy to reach, so by having more specific pre and post conditions, we are less likely to write incorrect programs. In this case, we have forgotten to add a case when left and right subtrees are both not null, so our end result is < the size of the tree.

WE can confirm this by changing the postcondition to say the result must be $\leq$ h:

```
int size (node2 t)
  requires t::trees<h> & t!=null
  ensures t::trees<h> & res<=h;
{
 if((t.left == null) && (t.right == null))
   return 1;
 else{
   if(t.left == null)
     return (1 + size(t.right));
   else {
       return (1 + size(t.left));
    }
  }
```

Runnig this in HIP gives us the following:

```
Checking procedure size$node2...
Procedure size$node2 SUCCESS.
Stop Omega... 223 invocations
0 false contexts at: ()
```

So now we have a program that matches our "incorrect" specification. Therefore we add the case when neither left nor right subtrees are null:

```
int size (node2 t)
  requires t::trees<h> & t!=null
  ensures t::trees<h> & res = h;
{
 if((t.left == null) && (t.right == null))
   return 1;
 else{
   if(t.left == null)
     return (1 + size(t.right));
   else {
     if (t.right == null){
       return (1 + size(t.left));
     }
     else {
       return (1 + size(t.left) + size (t.right));
     }
    }
  }
}
```

So now when we run HIP we get the following:

```
Checking procedure size$node2...
Procedure size$node2 SUCCESS.
Stop Omega... 234 invocations
0 false contexts at: ()
```

4.3 Comparison

4.3.1 Expressiveness

Smallfoot only allows proofs on the shape of the memory (i.e. what data structures they contain). It also contains preformed predicates for lists and trees, although we can give names to the fields contained in the structure (eg. change the default hd and tl in list to other names).

The language of Smallfoot is more similar to that of [Reynolds, 2002] allowing us to dispose data directly by calling a "dispose" method.

In contrast, HIP requires us to write our own predicates. This means we can form our own version of a list predicate and change it to include information about its size, if the program uses this too (like in the length function). Although this is more challenging for the user, it gives them more control over their specifications.

HIP also lets the user prove properties about the contents of the structures, when another theorem prover Isabelle is used to prove the statements HIP outputs [Chin et al., 2007]. However this requires the user to install yet another theorem prover, Isabelle, that does not come with HIP, so it would be even more expressive if the tool could do this itself.

4.3.2 Usability

Smallfoot has the advantage of having built in predicates, but it is harder to use these predicates because it is less obvious how they work.

The program language of HIP doesn't have a dispose or free method, so it is harder to write proofs to dispose structures because the user has to create a new method to do this.

HIP tells the user how long it took to complete the verification, but Smallfoot does not. However, HIP's error messages are not always very helpful. For example, for our incorrect tree size program we had the following full output:

```
Checking procedure size$node2...
Post condition cannot be derived:
  (may) cause: OrL[ t'!=null & r_1467!=null & l_1465!=null
& (((l_1465=1 & r_1467=2 & t'=3 &
1<=h2_1468 & 1<=h1_1466) | (r_1467=1 & t'=2 & 1<=h2_1468
 & l_1465=null &
h1_1466=0) | (l_1465=1 & t'=2 & r_1467=null
& h2_1468=0 & 1<=h1_1466) |
(t'=1 & r_1467=null & h2_1468=0 & l_1465=null
& h1_1466=0))) & 0<=h1_1466 &
0<=h1_1466 & res=h1_1466+1 &
h=h2_1468+1+h1_1466 |-  res=h. LOCS:[12;11;15;18;1;7;9;0;21;8;13]
 (must-bug),valid]

Context of Verification Failure: 1 File "",Line:0,Col:0

ERROR: at _0:0_0:0
Message: Post condition cannot be derived.
```

```
Procedure size$node2 FAIL.(2)

Exception Failure("Post condition cannot be derived.")
 Occurred!

Error(s) detected when checking procedure size$node2
Stop Omega... 204 invocations
0 false contexts at: ()
```

which does not explain much about where the error is or how to fix it.

A good feature of Smallfoot that HIP does not have is that by adding the flag "-all-states", the user can see every state in the proof. See 6.1.1 for an example when we deallocated the tree.

Overall HIP was easier to use as it was more expressive and we could write size preconditions as well as shape ones, but programs in Smallfoot were easier to debug.

Chapter 5

Conclusion

In this project I have built on my knowledge of Hoare Logic from the "Principles of Programming Languages" module, to learn a logic that is based on it but more complex. I studied the rules of the logic as stated by [Reynolds, 2002] and proved the correctness of programs on simple data structures. Some of these proofs were difficult, as some operations that seemed simple, such as swapping two values, did not work with the assignment rule, so I had to change my programs to make this work. I succeed in proving programs on lists, doubly linked lists and trees. Although small programs, they form the building blocks of larger programs, so be proving these we can make a step towards proving larger programs that contain these functions.

I then studied two automatic verification tools that are based on Separation Logic, Smallfoot and HIP. As the program languages of these tools were quite different to that specified in Reynold's paper, I did not replicate my programs exactly, as I said I would in my declaration, but applied the same algorithms and proved similar pre and post conditions.

Using the tools proved to be difficult, as they were not very well documented, particularly in the case of Smallfoot, and Smallfoot was much more restrictive than HIP, as I was only able to use it to prove properties about the shape of the memory. I used Smallfoot for proving programs disposing data structures, as only having predicates about shape gives all the information needed to have a decent specification. Where we needed information about the size of the data structures, HIP came in much more useful. Therefore we could see that the tool being used can have a great effect on how easy it is to verify a program, so the significance of this is that although there are many automatic verification tools for Separation Logic, they may not all be good at proving the same programs .

I did not manage to study the concurrent separation logic in detail, as there was not enough time in the mini project. I studied [Reddy and Reynolds, 2012] and learnt about using permissions to share resources between concurrent processes.

It would have been interesting to study this in more detail and write some proofs on programs using this form of Separation Logic.

I also studied [Hobor and Villard, 2013], to understand a correct way of writing proofs on directed acyclic graphs, using a new rule called Ramification However the application of this rule proved to be complex, so if I had more time I could have understood this and been able to prove programs on more complex data structures such as directed acyclic graphs.

In my declaration I discussed the study of current research, including concurrent abstract predicates, as defined in [Dinsdale-Young et al., 2010], which are used to prove concurrent programs by developing an abstraction of a module to produce a specification for it, and then using separation logic to prove these specifications.

Chapter 6

Appendices

6.1 Hoare Logic Rules

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}} \; Composition$$

$$\frac{}{\{P\} \textbf{ skip } \{P\}} \; Skip$$

$$\frac{}{\{P[E/V]\} \; V := E \; \{P\}} \; Assignment$$

$$\frac{P' \Rightarrow P \quad \{P\} \; C \; \{Q\}}{\{P'\} \; C \; \{Q\}} \; StrengthenPrecondition$$

$$\frac{Q \Rightarrow Q' \quad \{P\} \; C \; \{Q\}}{\{P\} \; C \; \{Q'\}} \; PostconditionWeakening$$

$$\frac{I > 0 \Rightarrow I \geq 0 \quad \{I \geq 0\} \; C \; \{I > 0\}}{\{I > 0\} \; C \; \{I > 0\}} \; Consequence$$

$$\frac{\{P \wedge B\}C\{P\} \quad P \wedge \neg B \; \Rightarrow Q}{\{P\} \textbf{ while } B \textbf{ do } C\{Q\}} \; WhileLoop$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P\ \wedge\ R\}\ C\ \{Q\ \wedge\ R\}}\ FrameRule$$

If C does not modify the free variables of R.

6.1.1 All states of deallocating a tree in Smallfoot

```
Function tree_deallocate
File "treeTest.sf", line 3, characters 2-123:
Intermediate State:
[tree(l,r;t)]
if 0==t then  else l=t->l;
r=t->r;
dispose(t);
fcall({},emp, t_0==l);
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
;
[emp]

File "treeTest.sf", line 11, characters 2-7:
Intermediate State:
[0==t * tree(l,r;t)]
[emp]

File "treeTest.sf", line 5, characters 4-13:
Intermediate State:
[0!=t * tree(l,r;t)]
l=t->l;
r=t->r;
dispose(t);
fcall({},emp, t_0==l);
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 6, characters 4-13:
Intermediate State:
[split_2==l * 0!=t * t |-> l:split_2,r:split_3 *
 tree(l,r;split_2) * tree(l,r;split_3)]
r=t->r;
dispose(t);
fcall({},emp, t_0==l);
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 7, characters 4-15:
```

```
Intermediate State:
[split_3==r * split_2==l * 0!=t * split_3!=t * l!=t *
 t |-> l:l,r:split_3 * tree(l,r;l) * tree(l,r;split_3)]
dispose(t);
fcall({},emp, t_0==l);
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 8, characters 4-23:
Intermediate State:
[split_3==r * split_2==l * 0!=t * l!=t * r!=t *
 tree(l,r;l) * tree(l,r;r)]
fcall({},emp, t_0==l);
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 8, characters 4-23:
Intermediate State:
[split_3==r * split_2==l * t_0==l * 0!=t * l!=t * r!=t *
 tree(l,r;l) * tree(l,r;r)]
fcall({},tree(l,r;t_0), emp);
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 9, characters 4-23:
Intermediate State:
[split_2==l * split_3==r * t_0==l * 0!=t * l!=t * r!=t
* tree(l,r;r)]
fcall({},emp, t_1==r);
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 9, characters 4-23:
Intermediate State:
[split_2==l * split_3==r * t_0==l * t_1==r * 0!=t * l!=t * r!=t
* tree(l,r;r)]
fcall({},tree(l,r;t_1), emp);
[emp]

File "treeTest.sf", line 11, characters 2-7:
Intermediate State:
```

[t_0==l * split_3==r * split_2==l * t_1==r * 0!=t * l!=t * r!=t]
[emp]

Valid

6.2 Mini Project Declaration

The University of Birmingham

School of Computer Science

First Semester Mini-Project: Declaration

This form is to be used to declare your choice of mini-project. Please complete all three sections and upload an electronic copy of the form to Canvas: https://canvas.bham.ac.uk/courses/16010

Deadline: 12 noon, 15 October 2015

1. Project Details

Name: Natalie Ravenhill

Student number: 1249790

Mini-project title: Separation Logic

Mini-project supervisor: Uday Reddy

---

2. Project Description

The following questions should be answered in conjunction with a reading of your programme handbook.

Aim of mini-project	To understand Separation Logic and apply it to examples involving lists and concurrency, by using tools to implement my own proofs.

Objectives to be achieved	• Study the foundations of Separation Logic and basic proofs on programs that manipulate lists • Use tools such as Smallfoot, Infer and Ynot to implement programs and proofs in Separation Logic • Study current research in Separation Logic, including its application to concurrency; particularly concurrent abstract predicates. • If time, or it seems appropriate, explore the possibility of implementing my own solver in Separation Logic.

Project management skills **Briefly explain how you will devise a management plan to allow your supervisor to evaluate your progress**	• *First I will study the initial papers I have been given to understand the logic itself and apply what I have learned to simple proofs on lists.* • *Then I will try to replicate these proofs using the tools for Separation Logic, while studying applications of the Logic to other areas* • *Then I will study the more advanced topics, including concurrent abstract predicates.*

Systematic literature skills **Briefly explain how you will find previous relevant work**	• *Initial papers on subject as introduction to the field obtained from supervisor* • *Supplementary papers will be found by references in papers given and from the other work of researchers active in the field of Separation Logic.*

Communication skills **What communication skills will you practise during this mini-project?**	• *I will meet regularly with my supervisor (at least once a week) to evaluate my progress and to go through material that I do not fully understand.* • *Presentation on project to other students, who are unfamiliar with the topic, will be given in the Research Skills module*

3. Project Ethics Self-Assessment Form

Does the research involve contact with NHS staff or patients?
No

Does the research involve animals?
No

Will any of the research be conducted overseas?
No

Will any of the data cross international boarders?
No

Will the research project involve humans as subjects of the research, including as participants in a requirements gathering or evaluation exercise?
No

Are the results of the research project likely to expose any person to physical or psychological harm?
No

Will you have access to personal information that allows you to identify individuals, or to corporate or company confidential information (that is not covered by confidentiality terms within an agreement or by a separate confidentiality agreement)?
No

Does the research project present a significant risk to the environment or society?
No

Are there any ethical issues raised by this research project that in the opinion of the investgator require further ethical review?
No

DECLARATION

By submitting this form, I declare that the questions above have been answered truthfully and to the best of my knowledge and belief, and that I take full responsibility for these responses. I undertake to observe ethical principles throughout the research project and to report any changes that affect the ethics of the project to the University Ethical Review Committee for review.

6.3 Information Searching

6.3.1 Parameters of literature search

Forms of literature to be retrieved

The important forms are firstly conference papers and journal articles, as many of the important developments in the field have been communicated in this way, particularly at conferences in programming languages research such as POPL. The proceedings have later been released as a special journal issue.

I will also consider PhD theses, as there are many separation logic researchers whose students also work in this area and produce notable original work in their theses. They are often supported with extensive background material that helps give a overview of the field. As Separation Logic is a young field ($\tilde{1}$0 years old), this replaces the absent books in the area.

Geographical/language coverage

Important work will be from many areas of the world. There are research groups working in the UK, USA and Singapore, which is where most of my background reading has originated, as the Smallfoot tool was created in England, the HIP tool was created in Singapore and the logic itself originated in the work of academics in the US and the UK. The only languages of the papers is English, as I have not come across any work written in other languages in my literature search.

Retrospective coverage and currency

It is sufficient to search retrospectively for 15 years for Separation Logic, as it is a young research topic. The paper that gives the most often referenced overview of the subject was published in 2002, so 15 years gives us a couple extra years of leeway.

For Hoare Logic we can search right back to the original paper [Hoare, 1969] , so this would be 50 years.

I use machine based services to do my literature search as they are suitable for my needs, as every paper I have needed to access is available from online databases and university library online resources.

6.3.2 Appropriate search tools

ACM Digital Library

ACM Digital Library is a database of all publications by the Association for Computing Machinery (ACM) including articles, magazines and conference proceedings. It does not include PhD theses. I used ACM Digital Library `http://dl.acm.org` to search for conference and journal papers. It has the retrospective coverage and currency required.

Google Scholar

Google Scholar is part of the search engine that is exclusively used for searching academic documents. I used to search for conference and journal papers.

6.3.3 Search statements

The search statements used will be based on: ‘

“separation logic” “separation logic” AND “HIP” “separation logic” AND “Smallfoot”

This may need to be refined in the number of recalled items is too large. Keywords from items that have been found will be reviewed in order to refine the set of keywords used.

6.3.4 Brief evaluation of the search

ACM Digital Library

Searching for the phrase “Separation Logic” on ACM DL, for papers published since 200, gives us 131 relevant results, of which 113 were conference proceedings, 73 were newsletter articles and 14 were journal articles.The conference articles are generally republished in the newsletters, so this is why we have more results by splitting them into their different forms.

As there are quite a number of results, we can look at just the first 10, as these will be the ones with the most citations.

Searching for “Separation Logic” and “Smallfoot” gives 2 results, [O’Hearn, 2007], a conference talk on a Smallfoot, and [Stewart et al., 2012], a conference paper on a tool that is built on Samllfoot.

Searching for “Separation Logic” and “HIP” gives 1 result, which is [Chin et al., 2011], a conference paper demonstrating the use of HIP.

Google Scholar

Searching for the phrase "Separation Logic" in Google Scholar gives 368 results. We can take the most cited results. As Google Scholar does not give much information about the origin of articles, it is harder to tell their forms. Therefore, we assume the ones with the most citations are from peer reviewed conferences and journals, unless stated otherwise.

Searching for "Separation Logic" and "Smallfoot" gives 365 results,

Searching for "Separation Logic" and "HIP" gives 81 results.

Bibliography

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005. ISBN 3-540-29735-9. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=64175`.

Josh Berdine, Cristiano Calcagno, and Peter W Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, pages 115–137. Springer, 2006.

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter OHearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, pages 3–11. Springer, 2015.

Wei-Ngan Chin, C. David, Him Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 307–320, July 2007. doi: 10.1109/ICECCS.2007.17.

Wei-Ngan Chin, Cristina David, and Cristian Gherghina. A hip and sleek verification system. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 9–10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048152. URL `http://doi.acm.org/10.1145/2048147.2048152`.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5.

Joao F Ferreira, Cristian Gherghina, Guanhua He, Shengchao Qin, and Wei-Ngan Chin. Automated verification of the freertos scheduler in hip/sleek.

International Journal on Software Tools for Technology Transfer, 16(4):381–397, 2014.

Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. *ACM SIGPLAN Notices*, 47(1):31–44, 2012.

Phillipa Gardner. A trusted mechanised specification of javascript: One year on. pages 43 – 46. Presented at CAV15, San Francisco, 2015. http://i-cav.org/2015/wp-content/uploads/2015/07/CAV15-talk.pdf.

Guanhua He, Shengchao Qin, Chenguang Luo, and Wei-Ngan Chin. Memory usage verification using hip/sleek. In Zhiming Liu and AndersP. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 166–181. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04760-2. doi: 10.1007/978-3-642-04761-9_14. URL `http://dx.doi.org/10.1007/978-3-642-04761-9_14`.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN 0001-0782.

Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. *ACM SIGPLAN Notices*, 48(1):523–536, 2013.

Peter O'Hearn. Separation logic and concurrent resource management. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 1–1, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: 10.1145/1296907.1296908. URL `http://doi.acm.org/10.1145/1296907.1296908`.

Uday S. Reddy and John C. Reynolds. Syntactic control of interference for separation logic. *SIGPLAN Not.*, 47(1):323–336, January 2012. ISSN 0362-1340.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.

Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364531. URL `http://doi.acm.org/10.1145/2364527.2364531`.