Separation Logic

Farzaneh Derakhshan
based on material by John C. Reynolds

CS 536: Science of Programming, Fall 2023
Lecture 20

In the previous lecture, we introduced addressable memory and pointers in our programming language. The goal of this lecture is to provide a framework for reasoning about the extended language. We will see that Hoare logic is not enough for this goal, and we will need to use an extension of it called separation logic. The reason is that in the new language, we are dealing with shared mutable data structures. These are structures where an updatable field can be referenced from more than one point. To understand why shared mutable data structures can be problematic, we first revisit the *append* program from the previous lecture, which appends a linked list of length three to another.

$$\begin{aligned} \mathit{append} := \quad & x_0 := !(\mathit{head_1} + \overline{1}); \\ & x_1 := !(x_0 + \overline{1}); \\ & !(x_1 + \overline{1}) := \mathit{head}_2; \\ & \mathit{head} := \mathit{head}_1 \end{aligned}$$

We want to prove that if, in the initial state, we start with two linked lists L_1 and L_2, where L_1 has a length of 3, and its head is stored in the store variable head_1, and L_2 has an arbitrary length with its head stored in the store variable head_2, then, the program will result in a final state with a single linked list L with its head stored in store variable *head*. Let's define a logical predicate[1] $\mathit{list}(L_1, i, n)$ to describe *L is a list of size n with its head being the address* `i`:

$$\mathtt{list}(\epsilon, i, 0) \triangleq i = \mathit{nil} \quad \mathit{and} \quad \mathtt{list}(a; L, i, n+1) \triangleq i \hookrightarrow a, j \wedge \mathtt{list}(L, j, n),$$

where ϵ refers an empty list, and $\hookrightarrow$ means *points to*, i.e., $i \hookrightarrow a, j$ means i and $i+1$ are the addresses of memory cells that contain a and j, respectively. $\mathtt{list}(\epsilon, i, 0) \triangleq i = \mathit{nil}$ states that the empty set of length 0,

We are interested in proving the partial correctness triple

$$\{\mathtt{list}(L_1, \mathit{head}_1, 3) \wedge \exists n.\mathtt{list}(L_2, \mathit{head}_2, n)\} \ \mathit{append} \ \{\exists m.\mathtt{list}(L_1; L_2, \mathit{head}, m), \}$$

where $L_1; L_2$ is simply the result of merging the lists L_1 and L_2. For example, if $L_1 = 10; 5; 3$ and $L_2 = 30; 5; 3$ then $L_1; L_2 = 10; 5; 3; 30; 5; 3$.

At first glance, this partial triple looks correct. However, on a closer look, we realize that the postcondition only holds if, in the initial state, the two lists L_1 and L_2 are entirely separate. Let's take, for example, the initial state of the heap as shown in Figure 1. If we execute the *append* program in this initial state, we get the final state seen in Figure 2. But the structure in Figure 2 is not even a linked list! In particular, there is no length m for which we have $\mathtt{list}(L, \mathit{head}, m)$. The problem is that in the initial states we have shared nodes, i.e., the second node is referenced by first nodes of both L_1 and L_2.

So far, we learned that the *append* program only works correctly if L_1 and L_2 do not have any shared nodes in the heap, i.e., they have to be in separate parts of the heap. But how can we specify this separation in the precondition? The key is to introduce a new logical operation $p * q$ called separating conjunction, which was discovered independently by Reynolds, and Ishtiaq and O'Hearn. The separating conjunction $p*q$ asserts that the heap can be divided into two (disjoint) parts, such that the first part has property p and the second part has property q.

[1] We will modify this definition later.

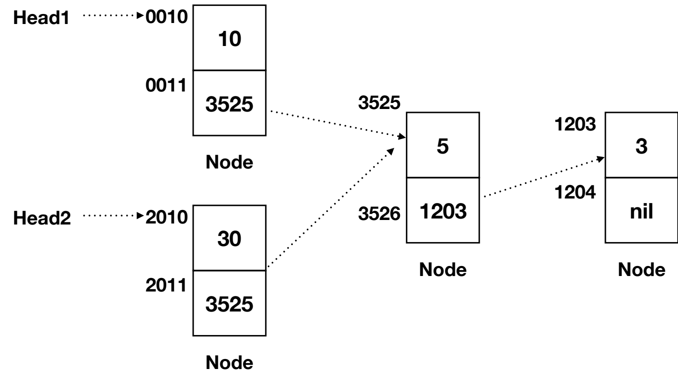

Figure 1: Append - initial state

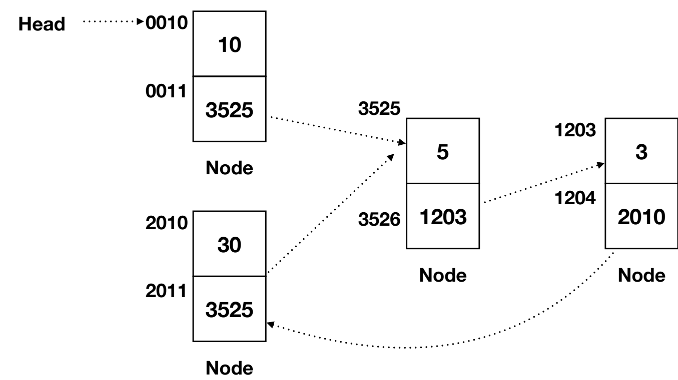

Figure 2: Append - final state

Using this new operator, we can rewrite the precondition as

$$\texttt{list}(L_1, head_1, 3) * \exists n.\texttt{list}(L_2, head_2, n).$$

It asserts that the heap can be separated into two disjoint parts: the first part satisfies $\texttt{list}(L_1, head_1, 3)$ and the second part satisfies $\exists n.\texttt{list}(L_2, head_2, n)$, implying that the two linked lists L_1 and L_2 are disjoint. In particular, the heap showed in Figure 1 does not satisfy the new precondition with the seaparating conjunction.

In the rest of this lecture, we first discuss the logical assertions and rules for reasoning about the heap separation and then build the separation logic by extending the Hoare-triple rules.

1 Assertions

The assertions in this context describe states that now contain heaps as well as stores. In addition to the usual operations and quantifiers of predicate logic, there are four new forms of assertion that describe the heap. All of this is based on the idea of separation.

Assertions	p	$::=$	$\cdots$	
		$\mid$	$\mathbf{emp}$	Empty heap
		$\mid$	$e \mapsto e'$	Singleton heap
		$\mid$	$p_1 * p_2$	Separating conjunction
		$\mid$	$p_1 \mathrel{-\!\!*} p_2$	Separating implication

We will explain each of them in more detail below:

- **Empty heap**: **emp** is a (nullary) predicate that asserts the heap is empty.
- **Singleton heap**: $e \mapsto e'$ states that the heap contains **exactly one cell**: the address of the cell is e and its value is e'.
- **Separating conjunction** $p_1 * p_2$ states that the heap can be divided into two separate heaps, the first part satisfying p_1 and the second one satisfying p_2.
- **Separating implication** $p_1 -\!\!* \; p_2$ states that if we extend the heap by adding a disjoint part that satisfies p_1 holds, then the extended heap satisfies p_2.

In this course, we do not discuss the separating implication beyond this [2]. Let's take a look at some examples involving the singleton heap and separating conjunction:

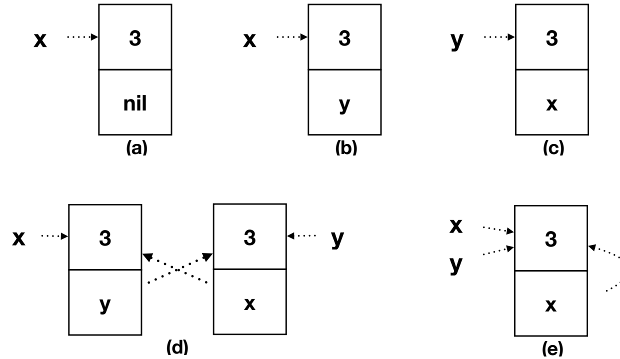

Figure 3: Differnt heap configurations (Examples 1-6)

Example 1. $x \mapsto 3, nil$ asserts that *the heap has exactly two adjacent memory cells*, x points to the first memory cell and the content of the memory cells are 3 and nil. In other words, the store maps x into some value ℓ where ℓ is an address, and the heap maps address ℓ to value 3 and address $\ell + 1$ to nil (See Figure 3(a)).
Example 2. $x \mapsto 3, y$ asserts that the heap has exactly two adjacent memory cells, x points to the first memory cell and the content of the memory cells are 3 and y; the store maps y into some value β (See Figure 3(b)).
Example 3. $y \mapsto 3, x$ asserts that the heap has exactly two adjacent memory cells, y points to the first memory cell and the content of the memory cells are 3 and x. (See Figure 3(c)).
Example 4. $x \mapsto 3, y * y \mapsto 3, x$ asserts that *the heap has exactly four memory cells* as in Figure 3(d), i.e., the heap can be divided into two parts, for the first part $x \mapsto 3, y$ holds and for the second part $y \mapsto 3, x$ holds.
Example 5. $x \mapsto 3, y \wedge y \mapsto 3, x$, on the other hand, descibes *a heap with exactly two cells* as in Figure 3(e). It asserts that the heap satisfies both $y \mapsto 3, x$ and $y \mapsto 3, x$, meaning the heap is a singleton heap in which y points to $3, x$ and x points to $3, y$. The only possible topology for such a heap is the one we show in Figure 3(e) with x being equal to y.

1.1 Precise and Imprecise Assertions

We learned that the assertions built using $e \mapsto e'$ and separating conjunction provide a precise description of the heap's domain. For instance, Example 1-3 asserted that the heap contains precisely two locations, while Example 4 asserted that it has exactly four locations.

Sometimes, we may want to indicate that somewhere in the heap, e points to e' without being precise about the heap's exact domain. In this case, we use the notation $e \hookrightarrow e'$. We can define $e \hookrightarrow e'$ as $e \mapsto e' * T$: the heap can be divided into one singleton in which e maps to e' and another part for which T holds. Since

[2] If you wish to learn more about it, please do come and talk to us.

every heap satisfies T, this gives us that *somewhere in the heap* $e \mapsto e'$ is correct. The assertion $e \mapsto e'$ is called a precise assertion, while $e \hookrightarrow e'$ is called an imprecise assertion.
Example 6.$x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$ describes that the heap is either as in Figure 3(d) or Figure 3(e).

1.2 Some Abbreviations

There are a couple of helpful abbreviations that we can use:

- We write $e \mapsto _$ when the heap has exactly one location e but its value is not important for us. It is defined as $\exists x.e \mapsto x$, where x is not free in e.
- We write $e \mapsto e_1, \cdots e_n$ when pointing to several consecutive fields. It is defined as $e \mapsto e_1 * e + 1 \mapsto e_2 * \cdots * e + n - 1 \mapsto e_n$. We already used this notation for $n = 2$ in Examples 1-5.
- Similarly, we write $e \hookrightarrow e_1, \cdots e_n$, when we mean $e \hookrightarrow e_1 * e + 1 \hookrightarrow e_2 * \cdots * e + n - 1 \hookrightarrow e_n$ also equivalent to $e \mapsto e_1, \cdots e_n * T$. We already used this notation for $n = 2$ in Example 6.

1.3 Example - the List Predicate

Now that we are familiar with the concept of separation and precise assertions, we can revisit the definition of the list provided at the beginning of this lecture. That definition is not quite correct, in particular, because it doesn't enforce that different nodes of a list are distinct, thus allowing a circular list (which we don't want). So we modify the definition of the predicate *list*(L_1, i, n) using a separating conjunction and precise description as

$$\texttt{list}(\epsilon, i, 0) \triangleq i = nil \quad and \quad \texttt{list}(a; L, i, n+1) \triangleq i \mapsto a, j * \texttt{list}(L, j, n),$$

2 Logic of Resources

The predicate logic is not enough to reason about the new assertions we introduced in the previous section. To address this, in this section, we extend predicate logic to the logic of resources. This new logic accounts for memory cells, which can be viewed as resources.

The inference rules for predicate logic, which don't involve the new operators we introduced, still hold in the new logic. We add new axioms and an inference rule to reason about separating conjunction. These include commutative and associative laws, the recognition that **emp** is a neutral element, and (semi-) distributive laws.

Axioms:

$$\begin{array}{lcll}
p_1 * p_2 & \Leftrightarrow & p_2 * p_1 & \\
(p_1 * p_2) * p_3 & \Leftrightarrow & p_1 * (p_2 * p_3) & \\
p * \textbf{emp} & \Leftrightarrow & p & \\
(p_1 \vee p_2) * q & \Leftrightarrow & (p_1 * q) \vee (p_2 * q) & \\
(p_1 \wedge p_2) * q & \Leftrightarrow & (p_1 * q) \wedge (p_2 * q) & \\
(\exists x.p_1) * p_2 & \Leftrightarrow & \exists x.(p_1 * p_2) & \textit{when } x \textit{ is not free in } p_2. \\
(\forall x.p_1) * p_2 & \Leftrightarrow & \forall x.(p_1 * p_2) & \textit{when } x \textit{ is not free in } p_2.
\end{array}$$

Inference rule:

$$\frac{p_1 \Rightarrow p_2 \qquad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2}\ \textsc{Monotonicity}$$

Note that the separating conjunction behaves differently than regular conjunction. For example, we can prove $p \Rightarrow p \wedge p$ and $p \wedge q \Rightarrow p$, but $p \Rightarrow p * p$ and $p * q \Rightarrow p$ are both unsound.

Exercise 1. Can you provide a counterexample for $p \Rightarrow p * p$ and $p * q \Rightarrow p$?

3 Separation Logic

Finally, we get to introduce the rules of separation logic. Beyond the rules of Hoare logic, we need inference rules for each of the new heap-manipulating commands and also a rule called the frame rule which is the key to "local reasoning" about the heap.

3.1 Frame Rule

The frame rule is designed based on the idea that reasoning and specification about a program should be confined to the cells that the program actually accesses. The value of any other cell will remain unchanged automatically.

Let's consider a program S that works on an initial heap with a property of $p * r$. Here, the assertion $p * r$ divides the heap into two parts: the first part has property p, while the second part has property r. If we know that the program S doesn't read or change any cells that occur in the second part, then we can ignore it while reasoning and only focus on the first part that satisfies property p.

Now, suppose we have $\{p\}\, S\, \{q\}$, meaning that given the initial state p, if the program S terminates, then it creates the final state q. Since we know that the heap locations in r remain untouched by S, we can conclude that if we start with the initial state $p * r$ and run the program S, the final state will be $q * r$, hence $\{p * r\}\, S\, \{q * r\}$.

With this reasoning, we can write the following rule when no variable occurring free in r is read/modified by S:

$$\frac{\{p\}\, S\, \{q\}}{\{p * r\}\, S\, \{q * r\}}\ \textsc{Frame}$$

By applying the frame rule, we can extend the local specification of S by adding arbitrary predicates r about heap cells that are not read or mutated by S. This is possible as long as no variable occurring free in r is read/modified by S.

3.2 Heap-manipulating Commands

Next, we provide the rules for each of the new heap-manipulating commands. Since we already have the frame rule, we can design these rules to be local. That is, we only need to consider the parts of the heap that the program actually accesses. The global version of these rules can be obtained by applying the frame rule.

Mutation. The local rule for mutation specifies the effect of mutation on the pre-state with a single cell with address e_1. In the post-state, the content of this cell is mutated to be e_2:

$$\frac{}{\{e_1 \mapsto _\}\ !e_1 := e_2\ \{e_1 \mapsto e_2\}}\ \textsc{Mutation}$$

If we apply the frame rule on $\{e_1 \mapsto _\}\ !e_1 := e_2\ \{e_1 \mapsto e_2\}$, we get $\{e_1 \mapsto _ * r\}\ !e_1 := e_2\ \{e_1 \mapsto e_2 * r\}$ for any r:

$$\frac{\dfrac{}{\{e_1 \mapsto _\}\ !e_1 := e_2\ \{e_1 \mapsto e_2\}}\ \textsc{Mutation}}{\{e_1 \mapsto _ * r\}\ !e_1 := e_2\ \{e_1 \mapsto e_2 * r\}}\ \textsc{Frame}$$

It means that anything in the heap beyond the cell being mutated (e_1) is left unchanged.

Deallocation The local rule for deallocation specifies the effect of deallocation of e on the pre-state with a single cell with address e. Since e is the only cell in the pre-state, in the post-state, the heap becomes empty, hence **emp**:

$$\frac{}{\{e \mapsto _\}\ \mathbf{dispose}(e)\ \{\mathbf{emp}\}}\ \textsc{Deallocation}$$

Again, with an application of the frame rule we get $\{e_1 \mapsto _ * r\}\ \mathbf{dispose}(e)\ \{\mathbf{emp} * r\}$ for any r. Moreover, by the rules of Section 2, we know that $\mathbf{emp} * r \Rightarrow r$, so we can apply the consequence rule to get $\{e_1 \mapsto _ * r\}\ \mathbf{dispose}(e)\ \{r\}$:

$$\frac{\dfrac{\dfrac{}{\{e \mapsto _\}\ \mathbf{dispose}(e)\ \{\mathbf{emp}\}}\ \textsc{Deallocation}}{\{e_1 \mapsto _ * r\}\ \mathbf{dispose}(e)\ \{\mathbf{emp} * r\}}\ \textsc{Frame} \qquad \mathbf{emp} * r \Rightarrow r}{\{e_1 \mapsto _ * r\}\ \mathbf{dispose}(e)\ \{r\}}\ \textsc{Consequence}$$

Exercise 2. Prove $\mathbf{emp} * r \Rightarrow r$ using the rules in Section 2.

Allocation For allocation, we first start with a simple but resticted rule and then generalize it to lift the restriction. In the pre-state we assume an empty heap, and in the post state, we allocate the address stored in x to e_1 and $x + i - 1$ to e_i for $1 \leq i \leq n$, hence $x \mapsto e_1, \cdots, e_n$.

$$\frac{x \notin FV(e_1, \cdots, e_n)}{\{\mathbf{emp}\}\ x := \mathbf{cons}(e_1, \cdots, e_n)\ \{x \mapsto e_1, \cdots, e_n\}}\ \textsc{Allocation-r}$$

The restriction on this rule is that x cannot occur as a free variable in any of $e_1, \cdots e_n$. But why do we need this restriction? The reason is similar to what we saw in previous lectures when designing the forward assign rule. For example, consider the program $x := cons(x)$, working on an initial store $\sigma_1 \triangleq \{x = 2\}$ and an initial empty heap. The program allocates a fresh address, let's say 5000 and stores the original value of x which is 2 in that address. Then, it updates the value of x to the fresh allocated address. In the final state, we have $\sigma_1 \triangleq \{x = 5000\}$ and the heap $5000 \mapsto 2$. However, the Allocation-r rule above, states that the post state is of the form $x \mapsto x$ which is not true.

When the original value of the variable being modified plays a role in the program, we remember the original value in the pre-state using a ghost variable x_0, and then substitute the original value x_0 for x in every e_i.

$$\frac{\forall 1 \leq i \leq n.\, e'_i = [x_0/x]e_i}{\{x = x_0 \wedge \mathbf{emp}\}\ x := \mathbf{cons}(e_1, \cdots, e_n)\ \{x \mapsto e'_1, \cdots, e'_n\}}\ \textsc{Allocation}$$

We can apply a frame rule on both Allocation-R and Allocation rules, to build their global versions.

Lookup For lookup, similar to allocation, we first describe a simple but restricted rule, where the original value of the variable being modified plays no role in the program:

$$\frac{x \notin FV(e)}{\{e \mapsto v\}\ x :=!e\ \{x = v \wedge e \mapsto x\}}\ \textsc{Lookup-r}$$

Given a pre-state in which address e contains value v, the lookup assigns the value v to x and keeps $e \mapsto v$ intact. Note that in the post state, we use a regular conjunction and not a separating conjunction since $x = v$ is not on a separate part of the heap. Rather, it's on the stack that coexists with $e \mapsto x$.

Again, the restriction on the rule is that x cannot occur as a free variable in any of e. If it does, then we have to use a ghost variable x_0 to remember the original value of x in the prestate and then substitute the original value for x in e in the post state:

$$\frac{}{\{x = x_0 \wedge e \mapsto v\}\ x :=!e\ \{x = v \wedge [x_0/x]e \mapsto x\}}\ \textsc{Lookup}$$

3.3 Summary of the New Rules

$$\frac{}{\{e_1 \mapsto _\}\ !e_1 := e_2\ \{e_1 \mapsto e_2\}}\ \textsc{Mutation} \qquad \frac{}{\{e_1 \mapsto _\}\ \mathbf{dispose}(e)\ \{\mathbf{emp}\}}\ \textsc{Deallocation}$$

$$\frac{x \notin FV(e_1, \cdots, e_n)}{\{\mathbf{emp}\}\ x := \mathbf{cons}(e_1, \cdots, e_n)\ \{x \mapsto e_1, \cdots, e_n\}}\ \textsc{Allocation-r}$$

$$\frac{\forall 1 \le i \le n.\, e'_i = [x'/x]e_i}{\{x = x' \wedge \mathbf{emp}\}\ x := \mathbf{cons}(e_1, \cdots, e_n)\ \{x \mapsto e'_1, \cdots, e'_n\}}\ \textsc{Allocation}$$

$$\frac{x \notin FV(e)}{\{e \mapsto v\}\ x :=!e\ \{x = v \wedge e \mapsto x\}}\ \textsc{Lookup-r} \qquad \frac{}{\{x = x_0 \wedge e \mapsto v\}\ x :=!e\ \{x = v \wedge [x_0/x]e \mapsto x\}}\ \textsc{Lookup}$$

$$\frac{\{p\}\ S\ \{q\}}{\{p * r\}\ S\ \{q * r\}}\ \textsc{Frame}$$

3.4 Example - Append

We use the rules of separation logic to prove that our append program is correct, i.e., our goal is to prove:

$$\{\texttt{list}(L_1, head_1, 3) * \texttt{list}(L_2, head_2, n)\}\ \mathit{append}\ \{\texttt{list}(L_1; L_2, head, m), \}$$

Recall the definition of the list predicate from Section 1.3:

$$\texttt{list}(\epsilon, i, 0) \triangleq i = nil \quad \mathit{and} \quad \texttt{list}(a; L, i, n+1) \triangleq i \mapsto a, j\ *\ \texttt{list}(L, j, n),$$

By the definition, we can rewrite the precondition as

$$\texttt{list}(a_1; a_2; a_3, head_1, 3) * \texttt{list}(L_2, head_2, n),$$

knowing that L_1 of length 3 is equal to $a_1; a_2; a_3$.

Next, we build the proof outline for this program. The blue lines are the pre/post conditions and the black lines are the program code. For each line we apply the corresponding local rule combined with the frame rule.

$\{\texttt{list}(a_1; a_2; a_3, head_1, 3) * \texttt{list}(L_2, head_2, n)\} \quad \Rightarrow^{(1)}$
$\{head_1 \mapsto a_1, i_1\ * i_1 \mapsto a_2, i_2\ * i_2 \mapsto a_2, i_3\ * \texttt{list}(L_2, head_2, n)\}$
$x_0 :=!(head_1 + \bar{1});$
$\{(x_0 = i_1 \wedge head_1 \mapsto a_1, i_1)\ * i_1 \mapsto a_2, i_2\ * i_2 \mapsto a_2, i_3\ * \texttt{list}(L_2, head_2, n)\} \Rightarrow^{(2)}$
$\{head_1 \mapsto a_1, x_0\ * x_0 \mapsto a_2, i_2\ * i_2 \mapsto a_2, i_3\ * \texttt{list}(L_2, head_2, n)\}$
$x_1 :=!(x_0 + \bar{1});$
$\{head_1 \mapsto a_1, x_0\ * (x_1 = i_2 \wedge x_0 \mapsto a_2, i_2)\ * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n)\} \Rightarrow^{(3)}$
$\{head_1 \mapsto a_1, x_0\ * x_0 \mapsto a_2, x_1\ * x_1 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n)\}$
$!(x_1 + \bar{1}) := head_2;$
$\{head_1 \mapsto a_1, x_0\ * x_0 \mapsto a_2, x_1\ * x_1 \mapsto a_2, head_2 * \texttt{list}(L_2, head_2, n)\}$
$head := head_1$
$\{head \mapsto a_1, x_0\ *\ x_0 \mapsto a_1, x_1\ *\ x_1 \mapsto a_1, head_2 * \texttt{list}(L_2, head_2, n)\} \Rightarrow^{(4)}$
$\{\texttt{list}(L_1; L_2, head, n+3)\}$

In the proof outline above, there are also four applications of the consequence rule, which are numbered 1-4. We need to prove each logical implication using the definition of the predicate list and the rules of the resource logic in Section 2.

The 1st consequence rule: By expanding the definition further and **emp** being the nuetral element we get

$$\begin{array}{ll}
(x_0 = i_1 \wedge head_1 \mapsto a_1, i_1) * i_1 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
head_1 \mapsto a_1, x_0 * x_0 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) & \\
head_1 \mapsto a_1, i_1 * i_1 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3 * \texttt{list}(\epsilon, i_3, 0) * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
head_1 \mapsto a_1, i_1 * i_1 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3 * \mathbf{emp} * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
head_1 \mapsto a_1, i_1 * i_1 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) &
\end{array}$$

The 2nd consequence rule: To prove the second and third logical implication, we need a few more rules in addition to those in Section 2. These rules consider those situations in which one or both of the predicates are completely independent of the heap. Here, for example, $x_0 = i_1$ only refers to the store and does not refer to any location in the heap.

1. $p \wedge q \Rightarrow p * q$ when p or q is pure
2. $p * q \Rightarrow p \wedge q$ when p and q are both pure
3. $(p \wedge q) * r \Rightarrow (p * r) \wedge q$ when q is pure.

By rule (3) above and the rules of predicate logic, we get

$$\begin{array}{ll}
(x_0 = i_1 \wedge head_1 \mapsto a_1, i_1) * i_1 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
(x_0 = i_1 \wedge (head_1 \mapsto a_1, i_1 * i_1 \mapsto a_2, i_2)) * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
(x_0 = i_1 \wedge (head_1 \mapsto a_1, x_0 * x_0 \mapsto a_2, i_2)) * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
(head_1 \mapsto a_1, x_0 * x_0 \mapsto a_2, i_2) * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) &
\end{array}$$

The 3rd consequence rule: Similarly by rule (3) above and the rules of predicate logic, we get

$$\begin{array}{ll}
head_1 \mapsto a_1, x_0 * (x_1 = i_2 \wedge x_0 \mapsto a_2, i_2) * i_2 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
head_1 \mapsto a_1, x_0 * (x_1 = i_2 \wedge (x_0 \mapsto a_2, i_2 * i_2 \mapsto a_2, i_3)) * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
head_1 \mapsto a_1, x_0 * (x_1 = i_2 \wedge (x_0 \mapsto a_2, x_1 * x_1 \mapsto a_2, i_3)) * \texttt{list}(L_2, head_2, n) & \Rightarrow \\
head_1 \mapsto a_1, x_0 * x_0 \mapsto a_2, x_1 * x_1 \mapsto a_2, i_3 * \texttt{list}(L_2, head_2, n) &
\end{array}$$

The 4th consequence rule: This is straightforward and by the definiton of the list predicate.