Hoare logic

Lecture 5: Verifying abstract data types in separation logic

Christopher Pulte cp526
University of Cambridge

CST Part II – 2023/2024

Recap

Last time, we introduced separation logic, a reinterpretation of Hoare logic that makes reasoning about pointers tractable. Separation logic is based on the notions of separation and ownership of resources.

A separation logic partial correctness triple ensures that the execution of the command (1) does not fault in a heap matching exactly its precondition, which ensures that it asserts ownership of all the parts of the heap it accesses, and (2) preserves the part of the heap disjoint from that matching the precondition.

In this lecture, we will look at a proof system for separation logic, and put separation logic into practice.

A proof system for separation logic

A proof system for separation logic

Separation logic inherits all the partial correctness rules from Hoare logic from the first lecture, and extends them with

- rules for each new heap-manipulating command;
- structural rules, including the frame rule.

We now want the rule of consequence to be able to manipulate our extended assertion language, with our new assertions $P * Q$, $t_1 \mapsto t_2$, and emp, and not just first-order logic anymore.

Recap: The frame rule

The frame rule is the core of separation logic.
It expresses that separation logic triples always preserve any assertion disjoint from the precondition:

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad mod(C) \cap FV(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The second hypothesis ensures that the frame R does not refer to any program variables modified by the command C.

This builds in modularity.

Other structural rules

Given the rules that we are going to consider for the heap-manipulating commands, we are going to need to include structural rules like the following:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{\exists x.\, P\}\ C\ \{\exists x.\, Q\}}$$

$$\vdots$$

Rules like these were admissible in Hoare logic.

We will represent uses of structural rules by indentation in proof outlines.

The heap assignment rule

Separation logic triples must assert ownership of any heap cells modified by the command. The heap assignment rule thus asserts ownership of the heap location being assigned:

$$\frac{}{\vdash \{E_1 \mapsto t\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}}$$

If expressions were allowed to fault, we would need a more complex rule.

The heap dereference rule

Separation logic triples must ensure the command does not fault.
The heap dereference rule thus asserts ownership of the given heap location to ensure the location is allocated in the heap:

$$\frac{}{\vdash \{E \mapsto v \wedge X = x\}\ X := [E]\ \{E[x/X] \mapsto v \wedge X = v\}}$$

Here, v and x are auxiliary variables; v is used to refer to the value of the dereferenced location, and x is used to refer to the initial value of program variable X in the postcondition.

Allocation and deallocation

The allocation rule introduces a new points-to assertion for each newly allocated location:

$$\frac{}{\vdash \{X = x \wedge emp\}\ X := \mathbf{alloc}(E_0, ..., E_n)\ \{X \mapsto E_0[x/X], ..., E_n[x/X]\}}$$

The deallocation rule destroys the points-to assertion for the location to not be available anymore:

$$\frac{}{\vdash \{E \mapsto t\}\ \mathbf{dispose}(E)\ \{emp\}}$$

Swap example

Specification of swap

To illustrate these rules, consider the following code snippet:

$$C_{swap} \equiv A := [X]; B := [Y]; [X] := B; [Y] := A;$$

We want to show that it swaps the values in the locations referenced by X and Y, when X and Y do not alias:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ C_{swap}\ \{X \mapsto n_2 * Y \mapsto n_1\}$$

Proof outline for swap

$$\begin{array}{l}
\{X \mapsto n_1 * Y \mapsto n_2\} \\
A := [X]; \\
\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\} \\
B := [Y]; \\
\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1 \wedge B = n_2\} \\
[X] := B; \\
\{(X \mapsto B * Y \mapsto n_2) \wedge A = n_1 \wedge B = n_2\} \\
[Y] := A; \\
\{(X \mapsto B * Y \mapsto A) \wedge A = n_1 \wedge B = n_2\} \\
\{X \mapsto n_2 * Y \mapsto n_1\}
\end{array}$$

Justifying these individual steps is now considerably more involved than in Hoare logic.

Detailed proof outline for the first triple of swap

$$
\begin{array}{l}
\{X \mapsto n_1 * Y \mapsto n_2\} \\
\{\exists a.\,((X \mapsto n_1 * Y \mapsto n_2) \wedge A = a)\} \\
\quad \{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = a\} \\
\quad \{(X \mapsto n_1 \wedge A = a) * Y \mapsto n_2\} \\
\qquad \{X \mapsto n_1 \wedge A = a\} \\
A := [X] \\
\qquad \{X[a/A] \mapsto n_1 \wedge A = n_1\} \\
\qquad \{X \mapsto n_1 \wedge A = n_1\} \\
\quad \{(X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2\} \\
\quad \{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\} \\
\{\exists a.\,((X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1)\} \\
\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}
\end{array}
$$

For reference: proof of the first triple of swap

To prove this first triple, we use the heap dereference rule to derive:

$$\{X \mapsto n_1 \wedge A = a\}\ A := [X]\ \{X[a/A] \mapsto n_1 \wedge A = n_1\}$$

Applying the rule of consequence, we obtain:

$$\{X \mapsto n_1 \wedge A = a\}\ A := [X]\ \{X \mapsto n_1 \wedge A = n_1\}$$

Then we use the frame rule:

$$\{(X \mapsto n_1 \wedge A = a) * Y \mapsto n_2\}\ A := [X]\ \{(X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2\}$$

(continued) For reference: proof of the first triple of swap

Using the rule of consequnce to rephrase its pre and post conditions (using separation logic properties we will see later in this lecture):

$$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = a\}\ A := [X]\ \{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$$

Then we existentially quantify the auxiliary variable a:

$$\{\exists a.\,(X \mapsto n_1 * Y \mapsto n_2) \wedge A = a\}\ A := [X]\ \{\exists a.\,(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$$

And finally, we use the rule of consequence to match the intended pre and post condition.

Proof of the first triple of swap (continued)

We relied on many properties of our assertion logic.

For example, to justify the first application of consequence, we need to show that

$$P \Rightarrow \exists a.\,(P \wedge A = a)$$

and to justify the last application of the rule of consequence, we need to show that:

$$((X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2) \Rightarrow ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1)$$

Properties of separation logic assertions

Syntax of assertions in separation logic

We now have an extended language of assertions, with a new connective, the separating conjunction $*$:

$$\begin{array}{rcl} P, Q & ::= & \bot \mid \top \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\ & \mid & P * Q \mid emp \\ & \mid & \forall x.\, P \mid \exists x.\, P \mid t_1 = t_2 \mid p(t_1, \ldots, t_n) \quad n \geq 0 \end{array}$$

$\mapsto$ is a predicate symbol of arity 2.

This is not just usual first-order logic anymore: this is an instance of the classical first-order logic of bunched implication (which is related to linear logic).
We will also require inductive predicates later.

We will take an informal look at what kind of properties hold and do not hold in this logic. Using the semantics, we can prove the properties we need as we go.

Properties of separating conjunction

Separating conjunction is a commutative and associative operator with *emp* as a neutral element (like $\wedge$ was with $\top$):

$$\vdash_{BI} P * Q \Leftrightarrow Q * P$$

$$\vdash_{BI} (P * Q) * R \Leftrightarrow P * (Q * R)$$

$$\vdash_{BI} P * emp \Leftrightarrow P$$

Separating conjunction is monotone with respect to implication:

$$\frac{\vdash_{BI} P_1 \Rightarrow Q_1 \qquad \vdash_{BI} P_2 \Rightarrow Q_2}{\vdash_{BI} P_1 * P_2 \Rightarrow Q_1 * Q_2}$$

Separating conjunction distributes over disjunction:

$$\vdash_{BI} (P \vee Q) * R \Leftrightarrow (P * R) \vee (Q * R)$$

Properties of separating conjunction (continued)

Assertions in separation logic are not freely duplicable in general:

$$\nvdash_{BI} P \Rightarrow P * P$$

in general.

For example, we want

$$\nvdash_{BI} t_1 \mapsto t_2 \Rightarrow (t_1 \mapsto t_2) * (t_1 \mapsto t_2)$$

This is the sense in which assertions in separation logic are resources: we cannot just duplicate them, we have to account for them.

Properties of separating conjunction (continued)

In linear separation logic, $\top$ is not a neutral element for the separating conjunction: we only have

$$\vdash_{BI} P \Rightarrow P * \top$$

but $\not\vdash_{BI} P * \top \Rightarrow P$ in general.

This means that we cannot "forget" about allocated locations: we have $\vdash_{BI} P * Q \Rightarrow P * \top$, but $\not\vdash_{BI} P * Q \Rightarrow P$ in general.

To actually get rid of Q, we have to deallocate the corresponding locations.

Properties of pure assertions

An assertion is **pure** when it does not talk about the heap. Syntactically, this means it does not contain *emp* or $\mapsto$.

Separating conjunction and conjunction become more similar when they involve pure assertions:

$\vdash_{BI} P \wedge Q \Rightarrow P * Q$	when P or Q is pure
$\vdash_{BI} P * Q \Rightarrow P \wedge Q$	when P and Q are pure
$\vdash_{BI} (P \wedge Q) * R \Leftrightarrow P \wedge (Q * R)$	when P is pure

Separating conjunction semi-distributes over conjunction (but not the other direction in general):

$$\vdash_{BI} (P \wedge Q) * R \Rightarrow (P * R) \wedge (Q * R)$$

Axioms for the points-to assertion

We also need some axioms about $\mapsto$:

null cannot point to anything:

$$\vdash_{BI} \forall t_1, t_2.\ t_1 \mapsto t_2 \Rightarrow (t_1 \mapsto t_2 \wedge t_1 \neq \textbf{null})$$

locations combined by $*$ are disjoint:

$$\vdash_{BI} \forall t_1, t_2, t_3, t_4.\ (t_1 \mapsto t_2 * t_3 \mapsto t_4) \Rightarrow ((t_1 \mapsto t_2 * t_3 \mapsto t_4) \wedge t_1 \neq t_3)$$

$\vdots$

We need to repeat the non-duplicable assertions on the right-hand side of the implication to not "lose" them.

Verifying abstract data types

Verifying ADTs

Separation logic is very well-suited for specifying and reasoning about mutable data structures typically found in standard libraries such as lists, queues, stacks, etc.

To illustrate this, we will specify and verify a library for working with lists, implemented using null-terminated singly-linked lists, using separation logic.

A list library implemented using singly-linked lists

First, we need to define a memory representation for our lists.

We will use null-terminated singly-linked list, starting from some designated *HEAD* program variable that refers to the first element of the linked list.
(We have to make do with this unique head in WHILE_p.)

For instance, we will represent the mathematical list $[12, 99, 37]$ as we did in the previous lecture:

Representation predicates

To formalise the memory representation, separation logic uses **representation predicates** that relate an abstract description of the state of the data structure with its concrete memory representations.

For our example, we want a predicate $\mathit{list}(t, \alpha)$ that relates a mathematical list, α, with its memory representation starting at location t (here, $\alpha, \beta, \ldots$ are just terms, but we write them differently to clarify the fact that they refer to mathematical lists).

To define such a predicate formally, we need to extend the assertion logic to reason about inductively defined predicates. We probably also want to extend it to reason about mathematical lists directly rather than through encodings. We will elide these details.

Representation predicates

We are going to define the $\mathit{list}(t, \alpha)$ predicate by induction on the list α:

- The empty list $[]$ is represented as a **null** pointer:

$$\mathit{list}(t, []) \stackrel{\mathit{def}}{=} (t = \mathbf{null}) \wedge \mathit{emp}$$

- The list $h :: \alpha$ (again, h is just a term) is represented by a pointer to two consecutive heap cells that contain the head h of the list and the location of the representation of the tail α of the list, respectively:

$$\mathit{list}(t, h :: \alpha) \stackrel{\mathit{def}}{=} \exists y. (t \mapsto h) * ((t+1) \mapsto y) * \mathit{list}(y, \alpha)$$

(recall that $t \mapsto h \Rightarrow ((t \mapsto h) \wedge t \neq \mathbf{null})$)

Representation predicates

The representation predicate allows us to specify the behaviour of the list operations by their effect on the abstract state of the list.

For example, assuming that we represent the mathematical list α at location $HEAD$, we can specify a push operation C_{push} that pushes the value of program variable X onto the list in terms of its behaviour on the abstract state of the list as follows:

$$\{list(HEAD, \alpha) \land X = x\} \; C_{push} \; \{list(HEAD, x :: \alpha)\}$$

Representation predicates

We can specify all the operations of the library in a similar manner:

$$\{emp\} \;\; C_{new} \;\; \{list(HEAD, [])\}$$

$$\left\{ \begin{array}{l} list(HEAD, \alpha) \land \\ X = x \end{array} \right\} \; C_{push} \; \{list(HEAD, x :: \alpha)\}$$

$$\{list(HEAD, \alpha)\} \;\; C_{pop} \;\; \left\{ \begin{array}{l} \left(\begin{array}{l} list(HEAD, []) \land \\ \alpha = [] \land ERR = 1 \end{array} \right) \lor \\ \left(\exists h, \beta. \left(\begin{array}{l} \alpha = h :: \beta \land \\ list(HEAD, \beta) \land \\ RET = h \land ERR = 0 \end{array} \right) \right) \end{array} \right\}$$

$$\{list(HEAD, \alpha)\} \; C_{delete} \; \{emp\}$$

$$\vdots$$

The emp in the postcondition of C_{delete} ensures that the locations of the precondition have been deallocated.

Implementation of *push*

The *push* operation stores the $HEAD$ pointer into a temporary variable Y before allocating two consecutive locations for the new list element, storing the start-of-block location to $HEAD$:

$$C_{push} \;\equiv\; Y := HEAD; HEAD := \mathbf{alloc}(X, Y)$$

We wish to prove that C_{push} satisfies its intended specification:

$$\{list(HEAD, \alpha) \land X = x\} \; C_{push} \; \{list(HEAD, x :: \alpha)\}$$

(We could use $HEAD := \mathbf{alloc}(X, HEAD)$ instead.)

Proof outline for *push*

Here is a proof outline for the *push* operation:

$$\begin{array}{l}
\{list(HEAD, \alpha) \land X = x\} \\
Y := HEAD; \\
\{list(Y, \alpha) \land X = x\} \\
HEAD := \mathbf{alloc}(X, Y) \\
\{(list(Y, \alpha) * HEAD \mapsto X, Y) \land X = x\} \\
\{list(HEAD, X :: \alpha) \land X = x\} \\
\{list(HEAD, x :: \alpha)\}
\end{array}$$

For the **alloc** step, we frame off $list(Y, \alpha) \land X = x$.

For reference: detailed proof outline for the allocation

$\{list(Y, \alpha) \land X = x\}$

$\{\exists z. \, (list(Y, \alpha) \land X = x) \land HEAD = z\}$

$\{(list(Y, \alpha) \land X = x) \land HEAD = z\}$

$\{(list(Y, \alpha) \land X = x) * (HEAD = z \land emp)\}$

$\{HEAD = z \land emp\}$

$HEAD := \textbf{alloc}(X, Y)$

$\{HEAD \mapsto X[z/HEAD], Y[z/HEAD]\}$

$\{HEAD \mapsto X, Y\}$

$\{(list(Y, \alpha) \land X = x) * HEAD \mapsto X, Y\}$

$\{(list(Y, \alpha) * HEAD \mapsto X, Y) \land X = x)\}$

$\{\exists z. \, (list(Y, \alpha) * HEAD \mapsto X, Y) \land X = x)\}$

$\{(list(Y, \alpha) * HEAD \mapsto X, Y) \land X = x\}$

Implementation of *delete*

The *delete* operation iterates down over the list, deallocating nodes until it reaches the end of the list.

$$C_{delete} \equiv X := HEAD;$$
$$\textbf{while } X \neq \textbf{null do}$$
$$(Y := [X + 1]; \textbf{dispose}(X); \textbf{dispose}(X + 1); X := Y)$$

We wish to prove that C_{delete} satisfies its intended specification:

$$\{list(HEAD, \alpha)\} \; C_{delete} \; \{emp\}$$

For that, we need a suitable loop invariant.
To execute safely, X effectively needs to point to **a** list (which is α only at the start).

Proof outline for *delete*

We can pick the invariant that we own the rest of the list:

$\{list(HEAD, \alpha)\}$

$X := HEAD;$

$\{list(X, \alpha)\}$

$\{\exists \beta. \, list(X, \beta)\}$

$\textbf{while } X \neq \textbf{null do}$

$\{\exists \beta. \, list(X, \beta) \land X \neq \textbf{null}\}$

$(Y := [X + 1]; \textbf{dispose}(X); \textbf{dispose}(X + 1); X := Y)$

$\{\exists \beta. \, list(X, \beta)\}$

$\{\exists \beta. \, list(X, \beta) \land \neg(X \neq \textbf{null})\}$

$\{emp\}$

We need to complete the proof outline for the body of the loop.

Proof outline for the loop body of *delete*

To verify the loop body, we need a lemma to unfold the list representation predicate in the non-null case:

$\{\exists \beta. \, list(X, \beta) \land X \neq \textbf{null}\}$

$\{\exists h, y, \gamma. \, X \mapsto h, y * list(y, \gamma)\}$

$Y := [X + 1];$

$\{\exists h, \gamma. \, X \mapsto h, Y * list(Y, \gamma)\}$

$\textbf{dispose}(X); \textbf{dispose}(X + 1);$

$\{\exists \gamma. \, list(Y, \gamma)\}$

$X := Y$

$\{\exists \gamma. \, list(X, \gamma)\}$

$\{\exists \beta. \, list(X, \beta)\}$

Linear separation logic and deallocation

If we did not have the two deallocations in the body of the loop, we would have to do something with

$$(X \mapsto h) * (X + 1 \mapsto Y)$$

We can weaken that assertion to $\top$, but not fully eliminate it.

We could weaken our loop invariant to $\exists \beta.\, list(X, \beta) * \top$:
the $\top$ would indicate the memory leak.

Linear separation logic forces us to deallocate.

Reasoning about the abstract state

To specify that a command computes the maximum element of a non-empty list, we do not need to change our representation predicate: we can just define a *maxl* predicate on the mathematical list to specify our C_{max} command:

$$maxl([x]) \stackrel{def}{=} x$$
$$maxl(x :: y :: \alpha) \stackrel{def}{=} max(x, maxl(y :: \alpha))$$

where *max* is the maximum function on integers,
and then have the following specification:

$$\{list(HEAD, h :: \alpha)\}\ C_{max}\ \{list(HEAD, h :: \alpha) \wedge M = maxl(h :: \alpha)\}$$

Implementation of *max*

The *max* operation iterates over a non-empty list, computing its maximum element:

$C_{max}\ \equiv$
$\quad X := [HEAD + 1]; M := [HEAD];$
$\quad \textbf{while}\ X \neq \textbf{null}\ \textbf{do}$
$\quad\quad (E := [X]; (\textbf{if}\ E > M\ \textbf{then}\ M := E\ \textbf{else skip}); X := [X + 1])$

We wish to prove that C_{max} satisfies its intended specification:

$$\{list(HEAD, h :: \alpha)\}\ C_{max}\ \{list(HEAD, h :: \alpha) \wedge M = maxl(h :: \alpha)\}$$

For that, we need a suitable loop invariant. However, the lists represented starting at *HEAD* and X are not disjoint.

Representation predicate for partial lists

To talk about partial lists, we can define a representation predicate for partial lists, $plist(t_1, \alpha, t_2)$, inductively:

$$plist(t_1, [], t_2) \stackrel{def}{=} (t_1 = t_2) \wedge emp$$
$$plist(t_1, h :: \alpha, t_2) \stackrel{def}{=} (\exists y.\, t_1 \mapsto h, y * plist(y, \alpha, t_2))$$

In particular, we can split lists in the middle:

$$\vdash_{BI} list(t_1, \alpha +\!\!+ \beta) \Leftrightarrow (\exists y.\, plist(t_1, \alpha, y) * list(y, \beta))$$

Proof outline for *max*

We can use *plist* to express our invariant:

$$\{list(HEAD, h :: \alpha)\}$$
$$X := [HEAD + 1]; M := [HEAD];$$
$$\{(plist(HEAD, [h], X) * list(X, \alpha)) \wedge M = max([h])\}$$
$$\{\exists \beta, \gamma.\ h :: \alpha = \beta +\!\!+ \gamma \wedge (plist(HEAD, \beta, X) * list(X, \gamma)) \wedge M = maxl(\beta)\}$$
$$\textbf{while } X \neq \textbf{null do}$$
$$\quad (E := [X]; (\textbf{if } E > M \textbf{ then } M := E \textbf{ else skip}); X := [X + 1])$$
$$\{list(HEAD, h :: \alpha) \wedge M = maxl(h :: \alpha)\}$$

We only use *plist* in the proof, not in the specification.

Implementation of *merge* (of merge sort)

$$\{list(X, \alpha) * list(Y, \beta) \wedge sorted(\alpha) \wedge sorted(\beta)\}$$
$$Z := \textbf{alloc}(0, \textbf{null}); P := Z;$$
$$\textbf{while } X \neq \textbf{null and } Y \neq \textbf{null do}$$
$$\left(\begin{array}{l} U := [X]; V := [Y]; \\ \textbf{if } U \leq V \textbf{ then } ([P + 1] := X; X := [X + 1]) \\ \textbf{else } ([P + 1] := Y; Y := [Y + 1]); \\ P := [P + 1] \end{array}\right);$$
$$\textbf{if } X = \textbf{null then } ([P + 1] := Y; Y := \textbf{null})$$
$$\textbf{else } ([P + 1] := X; X := \textbf{null});$$
$$P := [Z + 1]; \textbf{dispose}(Z); \textbf{dispose}(Z + 1); Z := P$$
$$\{\exists \gamma.\ list(Z, \gamma) \wedge sorted(\gamma) \wedge permutation(\gamma, \alpha +\!\!+ \beta)\}$$

We need to find a suitable invariant

Specification of *merge*

Again, we did not need to change our representation predicate: we only need to state that the mathematical list that is represented is sorted:

$$sorted([]) \stackrel{def}{=} \top$$
$$sorted([x]) \stackrel{def}{=} \top$$
$$sorted(x :: y :: \alpha) \stackrel{def}{=} x \leq y \wedge sorted(y :: \alpha)$$

and that a list is a permutation of another:

$$\begin{array}{l} permutation(\alpha, \beta) \stackrel{def}{=} \\ \quad (\alpha = \beta = []) \vee \\ \quad (\exists a, \alpha', \beta'.\ \alpha = [a] :: \alpha' \wedge \beta = [a] :: \beta' \wedge permutation(\alpha', \beta')) \vee \\ \quad (\exists a, b, \gamma.\ \alpha = [a] :: [b] :: \gamma \wedge \beta = [b] :::: [a]\gamma) \vee \\ \quad (\exists \gamma.\ permutation(\alpha, \gamma) \wedge permutation(\gamma, \beta)) \end{array}$$

Invariant of *merge*

We can now express our invariant:

$$\begin{array}{l} \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma, \gamma_1, a. \\ \quad \alpha = \alpha_1 +\!\!+ \alpha_2 \wedge \beta = \beta_1 +\!\!+ \beta_2 \wedge \\ \quad sorted(\alpha) \wedge sorted(\beta) \wedge \\ \quad sorted(\gamma) \wedge \gamma_1 +\!\!+ [a] = 0 :: \gamma \wedge \\ \quad permutation(\gamma, \alpha_1 +\!\!+ \beta_1) \wedge \\ \quad list(X, \alpha_2) * list(Y, \beta_2) * \\ \quad plist(Z, \gamma_1, P) * plist(P, [a], q) \end{array}$$

It is a rather readable — albeit detailed — description of why the program is correct.

Summary

We can specify abstract data types using representation predicates which relate an abstract model of the state of the data structure with a concrete memory representation.

We only need to know what the representation predicate is when we implement and verify our library, not when we use it. This gives us abstraction and modularity.

Justification of individual steps has to be made quite carefully given the unfamiliar interaction of connectives in separation logic, but proof outlines remain very readable.

In the next lecture, we will look at some extensions of Hoare logic.