Introduction to Separation Logic

Ana Nora Evans

October 25, 2016

1 Motivation

We would like to be able to reason about memory safety and correctness of programs. For example for the program below,

$$
\begin{aligned}
&int\ *x = malloc(sizeof(int));\\
&int\ *y = malloc(sizeof(int));\\
&*x = 3;\\
&*y = *x + 1;
\end{aligned}
$$

we would like to be able to prove that the two allocated memory locations hold values 3, and 4 respectively. For this to be true, we need $malloc$ to return a fresh memory address each time. In addition, we would like to be able to say that all the other heap locations are unchanged.

Separation logic, allows us to reason about portions of the heap independently. With some effort, we are able to prove assertions like:

- if the program terminates then the variable z has value 3
- the program will access only previously allocated memory
- the command will change exactly one valid memory location, and leave the rest unchanged
- allows us assert that two point do not alias

Separation logic is an extension of the Hoare logic for reasoning about imperative languages that use a shared mutable data structure. A shared mutable data structure is one that can be updated and accessed from more than one point. The heap is an example of such a structure.

Recall, from grad PL, in axiomatic semantics or Hoare Logic, we have assertions or Hoare triples:

$$\{A\}c\{B\}$$

which mean: if A holds in state σ and if $< c, \sigma > \Downarrow \sigma'$ the B holds in σ'. Given

1. a language for assertions (first order predicate logic on top of IMP expressions)
2. a programing language specification (the famous IMP)
3. derivation rules for Hoare triples

one can prove correctness given that she finds the magical preconditions and postconditions.

The state $\sigma : V \to \mathbb{Z}$ is a map from a finite set V of program variables to integers. If one wants to reason about dynamically allocated data structures or concurrency, then this logic must be extended to deal with the heap.

Previously, one modeled the heap as a symbolic mapping $\mu : Addresses \to \mathbb{Z}$. The content of a memory μ is given by the expression $sel(\mu, a)$ where a is an address. An update of an address a, produces a new memory state $upd(\mu, a, e)$. To reason about the memory expressions, one would use inference rules like McCarthy's axioms:

$$sel(upd(\mu, a_1, v), a_2) = \begin{cases} v \text{ if } a_1 = a_2 \\ sel(\mu, a_2) \text{ if } a_1 \neq a_2 \end{cases}$$

Separation logic treats the heap as a partial function, defined only on a subset of the address space, and allows us to separate the heap into parts and reason about each part independently.

Our motivating example is the deletion of the head of a list:

$$\begin{aligned} &struct\ node = \{ \\ &\quad int\ data; \\ &\quad struct\ node\ *next; \\ &\} \\ &\ldots \\ &struct\ node\ *aux = head \to next; \\ &free(head); \\ &head = aux; \end{aligned}$$

What would one like to prove?

- if the list represents a sequence $a_1, \ldots, a_n$, the result of the command is $a_2, \ldots, a_n$
- no other heap locations are modified by the command

What preconditions do we need?

- the list is not empty

What do we need to define?

- the heap
- extend IMP with new commands for heap manipulations
- extend the language of assertions to include reasoning about the heap
- give derivation rules for the new commands

The sections 2,3, 4,5,6 are adapted from Reynold [1].

2 Heap

In axiomatic semantics, the state $\sigma : V \to \mathbb{Z}$ is a map from a finite set V of program variables to integers. If one wants to reason about dynamically allocated data structures or concurrent access to a shared memory, then the program state has to be extended.

In separation logic, the program state is composed of the variable store (the old σ) and a **heap** (or memory) which is just a partial function $h : \mathbb{N} \rightharpoonup Integers$. The domain of the heap is the subset of $\mathbb{N}$ for which the function h is defined.

3 Programming Language

The programming language contains instructions for allocation, deallocation, access and update of a memory location.

$$
\begin{array}{ll}
c ::= \ldots & \\
\quad x := cons(e_1, \ldots, e_n) & \text{allocation} \\
\quad x := [e] & \text{select} \\
\quad [e_1] := e_2 & \text{update} \\
\quad dispose\ e & \text{deallocation}
\end{array}
$$

The analogue in C commands is

$int\ *p = malloc(sizeof(int));$	allocation
$int\ x = *p$	select
$*p = 2;$	update
$free(p);$	deallocation

In Ocaml,

$let\ x : int\ ref = ref\ 3\ in$	allocation
$let\ y : int = !x\ in$	select
$x := 5$	update

In a state (σ, h), the *allocation command* $x := cons(e_1, \ldots, e_n)$ adds n consecutive heap addresses to the heap's domain and stores the value of expression e_i. More precisely, the variable x of the store σ is updated with some integer l such that $l + i \notin dom(h)$, for all $0 \leq i \leq n-1$, the expressions e_i are evaluated for the store σ to values v_i, the domain of the heap h is extended with definitions $h(l+i-1) = v_i$ for all $1 \leq i \leq n$.

$$\frac{< e_i, \sigma > \Downarrow v_i, 1 \leq i \leq n}{< x := cons(e_1, \ldots, e_n), (\sigma, h) > \Downarrow (\sigma[x := l], h[l := v_1, l+1 := v_2, \ldots, l+n-1 := v_n])}$$

where $\sigma[x := l]$ is a new state with

$$\sigma[x := l](y) = \begin{cases} l, \text{if } x = y \\ \sigma(y), \text{if } x \neq y \end{cases}$$

and, similarly, the new heap $h[l := v_1, l+1 := v_2, \ldots, l+n-1 := v_n]$ satisfies the following conditions $(l+i) \notin dom(h)$ for all i, and

$$h[l := v_1, l+1 := v_2, \ldots, l+n-1 := v_n] = \begin{cases} h(a), \text{if } a \in dom(h) \\ v_i, \text{if } a = l+i-1 \end{cases}$$

The *select command*, $x := [e]$, in state (σ, h), evaluates the expression e for the store σ to a value l, and if $l \in dom(h)$ then updates the store variable x with the value $h(l)$.

$$\frac{< e, \sigma > \Downarrow l \qquad l \in dom(h)}{< x := [e], (\sigma, h) > \Downarrow (\sigma[x := h(l)], h)}$$

$$\frac{< e, \sigma > \Downarrow l \qquad l \notin dom(h)}{< x := [e], (\sigma, h) > \Downarrow abort}$$

The *update command*, $[e] := e'$, evaluates the expression e for the store σ to a value l. If $l \in dom(h)$, then the expression e' is evaluated for the store σ to value v.

Finally, the new heap will have exactly the same domain as h, but it will differ at exactly one location l where it will store the value v.

$$\frac{< e, \sigma > \Downarrow l \qquad l \in dom(h) \qquad < e', \sigma > \Downarrow v}{< [e] := e', (\sigma, h) > \Downarrow (\sigma, h[l := v])}$$

$$\frac{< e, \sigma > \Downarrow l \qquad l \notin dom(h)}{< [e] := e', (\sigma, h) > \Downarrow abort}$$

The ***deallocation*** command $dispose\ e$ evaluates the expression e for the store σ to a value l. If $l \in dom(h)$, then the new heap will be a restriction of the heap h, with domain $dom(h) \setminus \{l\}$. Note that it is possible to free only one of the locations that have been allocated as a group.

$$\frac{< e, \sigma > \Downarrow l \qquad l \in dom(h)}{< dispose\ e, (\sigma, h) > \Downarrow (s, h|_{dom(h)-\{l\}})}$$

$$\frac{< e, \sigma > \Downarrow l \qquad l \notin dom(h)}{< dispose\ e, (\sigma, h) > \Downarrow abort}$$

In our new language, the command that deletes the first element of a list is:

$$\begin{aligned} &h' := [h + 1]; \\ &dispose\ h; \\ &dispose\ h + 1; \\ &h := h'; \end{aligned}$$

4 Assertion Language

We already defined the heap and we extended the language with commands for manipulating it. Next step is to extend the **assertion language** to allow us to state facts about our new program state, including the heap.

$$\begin{array}{ll} A ::= \ldots & \\ \quad emp & \text{empty heap} \\ \quad e \mapsto e' & \text{singleton heap} \\ \quad A * B & \text{separating conjuction} \\ \quad A \mathrel{-\!\!*} B & \text{separating implication} \end{array}$$

The ***empty heap assertion,*** emp is true for a state (σ, h) iff the domain of the heap is the empty set:

$$(\sigma, h) \vDash emp \text{ iff } dom(h) = \emptyset$$

If the expression e with store σ evaluates to a value l and the the expression e' with store σ to a value v, then the *singleton heap assertion,* is true iff h is only defined for l and $h(l) = v$.

$$(\sigma, h) \vDash e \mapsto e' \text{ iff } dom(h) =< e, \sigma >\Downarrow \text{ and } h(< e, \sigma >\Downarrow) =< e', \sigma >\Downarrow$$

I will introduce next, two heap operators: the disjoint heaps $\perp$ and union of disjoint heaps $\cdot$. Let h_1 and h_2 be heaps. $h_1 \perp h_2$ is $true$ iff $dom(h_1) \cap dom(h_2) = \emptyset$. The union of disjoint heaps h_1 and h_2 is the heap h defined by:

$$h(l) = \begin{cases} h_1(l), \text{ where } l \in dom(h_1) \\ h_2(l), \text{ where } l \in dom(h_2) \end{cases}$$

The semantics of the *separating conjunction* are:

$(\sigma, h) \vDash A * B$ iff $\exists h_1, h_2$ such that $h_1 \perp h_2$ and $h_1 \cdot h_2 = h$ and $(\sigma, h_1) \vDash A$ and $(\sigma, h_2) \vDash B$

I will define next a few shorthand notations.

$$e \mapsto - \stackrel{def}{=} \exists x'.e \mapsto x' \text{ where } x' \text{ is not free in } e$$

$$e \mapsto e_1, e_2, \ldots, e_n \stackrel{def}{=} e \mapsto e_1 * (e+1) \mapsto e_2 * \cdots * (e+n-1) \mapsto e_n$$

$$e \hookrightarrow e' \stackrel{def}{=} e \mapsto e' * true$$

$$e \hookrightarrow e_1, e_2, \ldots, e_n \stackrel{def}{=} e \hookrightarrow e_1 * (e+1) \hookrightarrow e_2 * \cdots * (e+n-1) \hookrightarrow e_n$$

We will look at a few examples of assertions:

1. $(\sigma, h) \vDash x \mapsto 3$.

   x is a store variable whose value $l = \sigma(x)$ is the entire domain of the heap, $dom(h) = \{l\}$, and $h(l) = 3$

2. $(\sigma, h) \vDash x \mapsto 3, 4$.

   x is a store variable with value $l = \sigma(x)$. The domain of the heap is $dom(h) = \{l, l+1\}$, and $h(l) = 3$, $h(l+1) = 4$.

3. $(\sigma, h) \vDash x \mapsto 3, y$.

   x is a store variable with value $l = \sigma(x)$. y is a store variable with value v. The domain of the heap is $dom(h) = \{l, l+1\}$, and $h(l) = 3$, $h(l+1) = v$.

4. $x \mapsto 3, y * y \mapsto 3, x$

   There exist disjoint heaps $h_1 \perp h_2$ such that:

   $$(\sigma, h_1) \models x \mapsto 3, y$$
   $$(\sigma, h_2) \models y \mapsto 3, x$$
   $$h = h_1 \cdot h_2$$

From the previous case we know x and y are variables in the store σ with values l_x and l_y and:

$$dom(h_1) = \{l_x, l_{x+1}\}$$
$$h_1(l_x) = 3, h_1(l_x + 1) = l_y$$
$$dom(h_2) = \{l_y, l_{y+1}\}$$
$$h_2(l_y) = 3, h_2(l_y + 1) = l_x$$

Since h_1 and h_2 are disjoint, we know that $l_x \neq l_y$.

In conclusion, the assertion states that there are two store variables x, y with values l_x, l_y, respectively, the heap contains four locations $l_x, l_x + 1, l_y, l_y + 1$ with values $3, l_y, 3, l_x$.

5. $x \mapsto 3, y \wedge y \mapsto 3, x$

   Let l_x, l_y be the values of store variables x, y. The logical and $\wedge$ means that the assertions $x \mapsto 3, y$ and $y \mapsto 3, x$ hold for the same state (σ, h). Using the notations from above, we have

$$dom(h) = \{l_x, l_x + 1\} = \{l_y, l_y + 1\}$$
$$h(l_x) = 3, h(l_x + 1) = l_y$$
$$h(l_y) = 3, h(l_y + 1) = l_x$$

   We conclude that $x = y$ and the heap contains two locations $l, l + 1$ and $h(l) = 3, h(l + 1) = l$

6. $x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$ asserts that either of the previous two cases holds and that the heap may contain other locations

5 Inference Rules

The command-specific inference rules for Hoare logic remain sound, and so do structural rules like:

- rule of consequence

$$\frac{\vdash A \Rightarrow A' \qquad \vdash \{A'\}c\{B'\} \qquad \vdash B' \Rightarrow B}{\vdash \{A\}c\{B\}}$$

- rule of substitution

$$\frac{\{A\}c\{B\}}{(\{A\}c\{B\})/v_1 \to e_1, \ldots, v_n \to e_n}$$

  where $v_1, \ldots, v_n$ are variables occurring free in A, c, or B, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

The rule of constancy:

$$\frac{\{A\}c\{B\}}{\{A \wedge D\}c\{B \wedge D\}} \text{where no variable occurring free in } D \text{ is modified by } c$$

is not sound anymore as shown by the example

$$\frac{\{x \mapsto -\}[x] := 4\{x \mapsto 4\}}{\{x \mapsto - \ \wedge y \mapsto 3\}[x] := 4\{x \mapsto 4 \wedge y \mapsto 3\}}$$

If $x = y$, the postcondition is false, when the precondition is true.

The rule of constancy is replaced by the **frame rule**

$$\frac{\{A\}c\{B\}}{\{A * D\}c\{B * D\}} \text{where no variable occurring free in } D \text{ is modified by } c$$

This rule allows us to extend any local reasoning about the command c, involving only parts of the heap actually used or modified by c, to variables and parts of the heap that are not modified by c.

If we replace the logical and with the separating conjunction in the counterexample for the rule of constancy, then aliasing $x = y$ is precluded by the precondition $x \mapsto - * y \mapsto 3$.

The derivation rules for the new commands are:

- allocation

  $$\overline{\{emp\}x := cons(\bar{e})\{x \mapsto \bar{e}\}}$$

  where x is not free in e.

- lookup

  $$\overline{\{x = x' \wedge e \mapsto x''\}x := [e]\{x = x'' \wedge (e/x \to x') \mapsto x''\}}$$

  where x,x', and x'' are distinct.

- lookup (global)

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

  where x,x', and x'' are distinct; x', and x'' are not free in e; x is not free in r.

- mutation

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

- deallocation

  $$\overline{\{e \mapsto -\}dispose\ e\{emp\}}$$

O'Hearn called the local inference rules, small rules, because each assumes entire heap is only the command footprint. In [2] there is a detailed description derivation of the global rules from local ones.

6 Example

Let's look at the delete the list head example:

$$h' := [h + 1];$$
$$dispose\ h;$$
$$dispose\ h + 1$$
$$h := h';$$

First, we will notice that there is an assumption about the structure of the list. If h contains the address of a list cell, then the address pointed by h stores the data, and the consecutive address points to the next element in the list. We also must start with a non-empty list. If we start with a list representing a sequence $a_1, \ldots, a_n$ we would like to end up with a list representing the sequence $a_2, \ldots, a_n$.

We will start by defining a $list$ predicate recursively. Let α be a sequence $\alpha_1, \ldots, \alpha_n$. Let (h, t) denote the list segment with head h and with last next pointer t. By definition, the list segment (h, t) represents the empty sequence if

$$list\ \epsilon\ (h, t) = emp \wedge h = t.$$

This assertion is true for the state (σ, h) with $\sigma(h) = \sigma(t)$ and empty heap h.

For a non-empty sequence, denoted by $a \cdot \alpha$, where a is the first element of the sequence and α the rest of the sequence,

$$list\ (a \cdot \alpha)\ (h, t) = \exists h'.h \mapsto a, h' * list\ \alpha\ (h', t)$$

The use of the separating conjunction guarantees that there are no cycles in our list segment. The definition also ensures the desired list structure with consecutive addresses for data and next pointer.

Here is how we use this predicate. The precondition

$$\{list\ (a \cdot \alpha)\ (h, t)\}$$

assures us we are not starting with an empty list and the list has the expected structure.

By the definition of $list$ predicate, we have

$$\{\exists h_1.h \mapsto a, h_1 * list\ \alpha\ (h_1, t)\}$$

Since $h \mapsto a, h_1$ is by definition $h \mapsto a * (h + 1) \mapsto h_1$, we have

$$\{\exists h_1.h \mapsto a * (h + 1) \mapsto h_1 * list\ \alpha\ (h_1, t)\}$$

Since h_1 is not free in $h \mapsto a$, we have

$$\{h \mapsto a * \exists h_1.(h+1) \mapsto h_1 * list\ \alpha\ (h_1, t)\}$$

We apply the global lookup rule

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

with $x'' = h_1$, $e = h+1$, $x = h'$, $r = list\ \alpha\ (h_1, t) = list\ \alpha\ (x'', t)$

$$\begin{array}{l}\{\exists h_1.((h+1) \mapsto h_1) * ((list\ \alpha\ (h_1, t))/x' \to h')\} \\ \quad h' := [h+1] \\ \{\exists x'.((h+1)/h' \to x') \mapsto h' * (list\ \alpha\ (h_1, t)/h_1 \to h')\}\end{array}$$

We apply the substitutions in the postcondition

$$\{\exists x'.(h+1) \mapsto h' * (list\ \alpha\ (h', t)\}$$

We can eliminate the existential quantifier since x' is not used in the predicate

$$\{(h+1) \mapsto h' * (list\ \alpha\ (h', t)\}$$

We use the frame rule and the above application of the global lookup rule to obtain the postcondition

$$\{h \mapsto a * (h+1) \mapsto h' * list\ \alpha\ (h', t)\}$$

which is

$$\{h \mapsto a, h' * list\ \alpha\ (h', t)\}$$

An easy application of the deallocation rule with frame rule finishes the example.

$$\begin{array}{l}\{list\ (a \cdot \alpha)\ (h, t)\} \\ \{\exists h'.h \mapsto a, h' * list\ \alpha\ (h', t)\} \\ h' := [h+1]; \\ \{h \mapsto a, h' * list\ \alpha\ (h', t)\} \\ dispose\ h; \\ \{h+1 \mapsto h' * list\ \alpha\ (h', t)\} \\ dispose\ h+1 \\ \{list\ \alpha\ (h', t)\} \\ h := h'; \\ \{list\ \alpha\ (h, t)\}\end{array}$$

After all this work we proved the correctness of the following four line program:

$$h' := [h + 1];$$
$$dispose\ h;$$
$$dispose\ h + 1;$$
$$h := h';$$

For a proof of an in place list reversal see [1].

7 Brief History

Separation logic was developed in early 2000's by: John C. Reynolds (CMU) [1], Peter O'Hearn (UCL), Hongseok Yang (Oxford) [2], Samin Ishtiaq (MSR).

I think the delete head of the list example convinced you it is not easy to write these types of proofs by hand. If we ignore the heap, this is just axiomatic semantics. As we saw from grad PL, symbolic execution and theorem provers made it practical.

In 2005, Berdine et al. [3] , defined *symbolic heaps*.

In 2006, Smallfoot [4] a tool for checking separation logic specifications uses the previous work.

Since 2006, number of academic tools have been proposed to search for memory specific bugs.

SLayer [5] is a tool developed by Microsoft designed to **automatically** prove memory safety of industrial systems code.

Infer [6] is a tool developed by Facebook to find resource leaks, null dereference and tainted value reaching sensitive function.

References

[1] John C. Reynolds. Separation logic: A logic for shared mutable data structures, 2002.

[2] Hongseok Yang Peter OHearn, John Reynolds. A local reasoning about programs that alter data structures, 2001.

[3] Peter W. OHearn Josh Berdine, Cristiano Calcagno. Symbolic execution with separation logic, 2005.

[4] Peter W. OHearn Josh Berdine, Cristiano Calcagno. Smallfoot: modular automatic assertion checking with separation logic, 2005.

[5] Samin Ishtiaq Josh Berdine, Byron Cook. Slayer: Memory safety for systems-level code, 2011.

[6] Facebook infer.