Separation Logic Introduction

Stefan Muller
(Based on material by John Reynolds and Peter O'Hearn)

CS 536: Science of Programming, Spring 2022
Lecture 26

1 Pointers and Aliasing

So far, we've only dealt with "local variables": we have a state σ that maps variables to values. In a program, these correspond to variables stored on the stack or in registers. Programming languages like C also allow you to access values on the *heap*, through a "pointer" to a location in memory.

We'll add this to our IMP language. We now have a type of "locations", which represents a pointer to a location in the heap (which contains a value, which may be an integer, a Boolean, or another location). We can "dereference" locations with the syntax $[e]$ (this corresponds to `*e` in C).

This leads to a lot of new problems. Consider the following triple. The predicates don't use exactly the syntax we'll end up using, but it'll work for now.

$$\not\models \{[y] = 0\}\ [x] := 5\ \{[y] = 0\}$$

It seems fairly obvious that this triple should be valid, and yet it isn't! The reason is something we talked about briefly when discussing array assignments: *aliasing*. It's the same reason this triple isn't valid:

$$\not\models \{a[i] = 0\}\ a[j] := 5\ \{a[i] = 0\}$$

At runtime, x and y may point to the same location in the heap, so setting $[x]$ to 5 would also set $[y]$ to 5.

With arrays, we solved this problem by adding conditions to the predicates about whether i and j were equal at runtime. Something like that would likely work here too, but researchers in the early 2000s developed a more powerful extension of Hoare logic that's designed to handle exactly this by adding explicit assumptions about (non)-aliasing: *separation logic*.

We'll make the following additions to our language:

$$s ::= \ldots \mid [e] := e \mid x := [e] \mid \mathsf{alloc}\ e \mid \mathsf{free}\ e$$

The statement $[e_1] := e_2$ evaluates e_1 and e_2 and sets the heap location pointed to by e_1 to e_2. The statement $x := [e]$ sets x (which is still a normal variable in the state, as before) to the value in the heap pointed to by e. The statement $\mathsf{alloc}\ e$ allocates a new location in the heap initialized to the value of e, and $\mathsf{free}\ e$ evaluates e to a location and frees it; these are like `malloc` and `free` in C.

The state now has two parts: a *store*, which we'll continue to denote σ, and a heap, which we'll denote h. This impacts our whole operational semantics: the small-step judgment is now $\langle s, (\sigma, h)\rangle \to \langle s', (\sigma', h')\rangle$ and the big-step judgment is $M(s, \sigma, h)$. Even evaluation of expressions needs to use both the store and heap: $(\sigma, h)(e)$. Finally, we need to consider satisfaction of triples and predicates under both a store and a heap:

$$\begin{array}{rcl} \sigma, h & \models & p \\ \sigma, h & \models & \{p\}\ s\ \{q\} \end{array}$$

1.1 Semantics

The semantics of the new statements use and update the heap. We'll use the notation ℓ to represent heap locations.

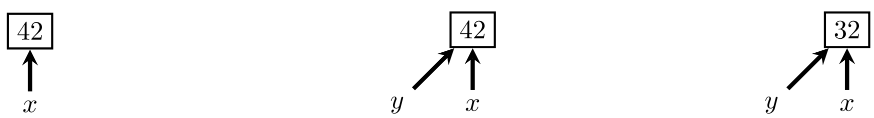
$$\frac{(\sigma, h)(e) = \ell \qquad h(\ell) = e'}{\langle x := [e], (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma[x \mapsto e'], h)\rangle} \qquad \frac{(\sigma, h)(e_1) = \ell \qquad \ell \in Dom(h)}{\langle [e_1] := e_2, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma, h[\ell \mapsto (\sigma, h)(e_2)])\rangle}$$

$$\frac{\ell\ \mathsf{fresh}}{\langle x := \mathsf{alloc}\ e, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma[x \mapsto \ell], h[\ell \mapsto (\sigma, h)(e)])\rangle} \qquad \frac{(\sigma, h)(e) = \ell \qquad h' = h \setminus \ell}{\langle \mathsf{free}\ e, (\sigma, h)\rangle \to \langle \mathsf{skip}, (\sigma, h')\rangle}$$

As an example, let $s = x := \mathsf{alloc}\ 42; y := x; [y] := 32$ and $\sigma = \{\}$ and $h = \{\}$.

$$\begin{aligned} &\langle s, (\sigma, h)\rangle \\ &\quad \to^2 \quad \langle y := x; [y] := 32, (\{x = \ell\}, \{\ell \mapsto 42\})\rangle \\ &\quad \to^2 \quad \langle [y] := 32, (\{x = \ell, y = \ell\}, \{\ell \mapsto 42\})\rangle \\ &\quad \to^2 \quad \langle \mathsf{skip}, (\{x = \ell, y = \ell\}, \{\ell \mapsto 32\})\rangle \end{aligned}$$

We usually draw heaps with box-and-arrow diagrams like this:



Heap cells can also contain pointers; this even allows us to create cycles.

$$\begin{aligned} &x := \mathsf{alloc}\ 42; \\ &[x] := x \end{aligned}$$

2 Separation Logic

2.1 Predicates

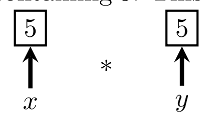
We'll introduce new types of predicates we can use in conditions:

- emp means the heap is empty (this generally isn't useful on its own, but will be useful in conjunction with the other predicates).
- $x \mapsto e$ means that the heap consists of *exactly* a location x containing e. Again, requiring exactly one heap location is a big restriction, but this will turn out to be useful.
- $p_1 * p_2$ means that the heap can be divided into two *disjoint* heaps h_1 and h_2 (disjoint meaning they contain different locations) and h_1 satisfies p_1 and h_2 satisfies p_2.

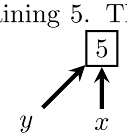
We'll also use $x \hookrightarrow e$ as shorthand for $x \mapsto e * T$; this says that the heap has a binding of x to e and possibly others (if there are others, we split them into the disjoint heap about which we only assert T, which always holds). Finally, we'll use $x \mapsto —$ to say that x is in the heap, but we don't care what value it points to; we could write this formally as $\exists e. x \mapsto e$.

We can use these together with the usual connectives $\wedge, \vee, \to, \leftrightarrow, \neg$, with their usual meanings. These interact in interesting ways.

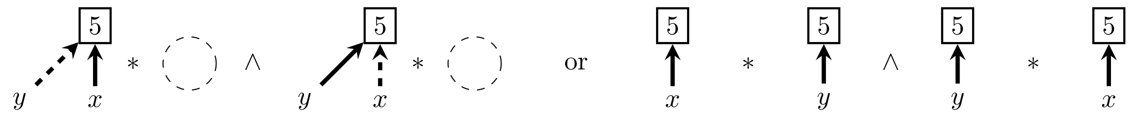
- $\sigma, h \vDash x \mapsto 5 * y \mapsto 5$ means that h can be divided into disjoint heaps h_1 and h_2, $\sigma(x)$ is a location in h_1 containing 5 and $\sigma(y)$ is a location in h_2 also containing 5. This means that x and y **cannot be aliases**!



- $\sigma, h \vDash x \mapsto 5 \wedge y \mapsto 5$ means that h is a heap that has exactly one location x containing 5 and is also a heap with exactly one location y containing 5. This means that x and y **must be aliases**!



- $x \mapsto 5 \wedge y \mapsto 10$ is a contradiction.
- $\sigma, h \vDash x \hookrightarrow 5 \wedge y \hookrightarrow 5$ means that x and y may or may not be aliases.



2.2 Rules

The rules for allocation and lookup both also perform an assignment to a variable, and thus have a feature that should be familiar from the forward assignment rule: because e can use x, we have to make a new variable x_0 to store the old value of x and substitute x_0 for x in e. Otherwise, the rule for allocation says that if we have an empty heap before, the new heap maps x to e. If e maps to v in the heap, the lookup rule says that x is now bound to v. The update rule is fairly straightforward, but also requires that e is in the heap before the update. It turns out this is actually necessary to enforce partial correctness, not just total correctness, as we'll see. The deallocate rule takes a heap with a binding for e to an empty heap.

Of course, all of these rules require pretty strong conditions. Just like with standard triples, we'll want a way to weaken them; for example, we want to be able to update the value of a location in a heap with more than one location. The rule that allows us to do this is called the Frame Rule: it says that if there's a disjoint part of the heap r that isn't affected by the program and if $\{p\}\ s\ \{q\}$, then the same program runs on a heap that satisfies $p * r$ and leaves r unchanged.

$$\frac{}{\vdash \{x_0 = x \wedge \mathsf{emp}\}\ x := \mathsf{alloc}\ e\ \{x \mapsto [x_0/x]e\}}\ (\textsc{Alloc})$$

$$\frac{}{\vdash \{x_0 = x \wedge e \mapsto v\}\ x := [e]\ \{x = v \wedge [x_0/x]e \mapsto v\}}\ (\textsc{Lookup}) \qquad \frac{}{\vdash \{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}}\ (\textsc{Update})$$

$$\frac{}{\vdash \{e \mapsto -\}\ \mathsf{free}\ e\ \{\mathsf{emp}\}}\ (\textsc{Deallocate}) \qquad \frac{\vdash \{p\}\ s\ \{q\} \qquad \mathsf{FV}(r) \cap \mathit{Change}(s) = \emptyset}{\vdash \{p * r\}\ s\ \{q * r\}}\ (\textsc{Frame})$$

2.3 Examples

Let's take a look back at the program from the beginning and see what goes wrong when we try to prove it:

$$\{y \mapsto 0\}\ [x] := 5\ \{y \mapsto 0\}$$

Wee can't prove this using the Update rule because the precondition doesn't say anything about x, so we need to add this to the precondition. There are (at least) three ways we can do this, corresponding to the cases we saw above:

$$\begin{array}{rcl} p_1 & \equiv & y \mapsto 0 \wedge x \mapsto — \\ p_2 & \equiv & y \mapsto 0 * x \mapsto — \\ p_3 & \equiv & y \hookrightarrow 0 \wedge x \hookrightarrow — \end{array}$$

All of these are a bit stronger than our original precondition (they need to be): p_1 and p_3 just say that x is in the context; p_2, on the other hand, requires that x is not an alias of y. Furthermore, p_1 and p_3 still don't match what we need to apply the rule. On the other hand, if we use p_2, we can prove a modified version of the triple with the Frame Rule:

$$\dfrac{\dfrac{}{\vdash \{x \mapsto —\}\ [x] := 5\ \{x \mapsto 5\}}\ (\textsc{Update})}{\vdash \{x \mapsto — * y \mapsto 0\}\ [x] := 5\ \{x \mapsto 5 * y \mapsto 0\}}\ (\textsc{Frame})$$

This makes sense: the program was incorrect because x and y could be aliases; if we assume in the precondition that they're not, then we can prove the program correct.

3 References

These notes were based heavily on the following two resources, which are good places to look for more information:

1. John C. Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures.* LICS '02

2. Peter W. O'Hearn, "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". *Software Safety and Securuty; Tools for Analysis and Verification.* NATO Science for Peace and Security Series, vol. 33, pp. 286–318, 2012.