Programming Languages: Imperative Program Construction 11. Separation Logic I

Shin-Cheng Mu

Autumn Term, 2021

Separation Matters

- Our reasoning so far is based on an important assumption: variables, having different names, are independent from each other.
- With $\mathbf{var}\ a, b$, for example, mutating a does not change the value of b.
- Otherwise most of our reasoning would fail.

Remark: Procedure Calls

- Problem with procedures with call-by-reference variables.

$$\begin{array}{l}\mathbf{proc}\ swap\ (ref\ x, y : Int) = \\ \quad x := x - y; y := x + y; x := y - x\end{array}$$

- $swap\ (a, b)$ should swap the values of a and b — we have proved so before, haven't we?
- However, $swap\ (a, a)$ sets a to 0.
- Extra care is needed to handle function/procedure calls, which we unfortunately won't cover in this course.

Most materials of this lecture are adapted from Reynold's course in CMU [Rey11].

Other suggested reading materials: [Rey02] [O'H19], [O'H12].

1 Dynamic Memory Management

- Another source of possible violation is the heap memory model.
- Recall: variables declared are supposed to be located in *stacks*. (Also called a *store*).
- In the heap model, programmers can allocate blocks of memories in *heaps*.
- We can store addresses of heap cells in variables, lookup the content of a heap given the address, or deallocate a cell.

Pointer Manipulation

- A pointer is a variable that stores a memory address.
- In our setting we let $Addr = Int$, and let $\mathbf{nil}$ be a unique address.
- $p := \mathbf{cons}\ (1, 2)$ — allocate two consecutive heap cells, set their values to 1 and 2, and store the address of the first cell in p.
  - One has no control what the address will be, other than that it won't be $\mathbf{nil}$.
- $x := {}^*e$ — look up the value stored in the cell with address e, and copy the value to variable x.
- ${}^*e := f$ — let the value stored in cell with address e be updated to f.
- $\mathbf{free}\ e$ — free the cell having address
- In the last three cases the address e must have been allocated.

Example

program	store and heap
	$s : x = 3 \wedge y = 4; h : \mathbf{emp}$
$x := \mathbf{cons}\ (1, 2)$	$s : x = 34 \wedge y = 4$ $h : 34 \mapsto 1, 35 \mapsto 2$
$y := {}^*x$	$s : x = 34 \wedge y = 1$ $h : 34 \mapsto 1, 35 \mapsto 2$
${}^*(x + 1) := 3$	$s : x = 34 \wedge y = 1$ $h : 34 \mapsto 1, 35 \mapsto 3$
$\mathbf{free}\ (x + 1)$	$s : x = 34 \wedge y = 1$ $h : 34 \mapsto 1$

Notes:

- Apart from that **cons** does not return **nil**, the program cannot predict what address (e.g. 34) **cons** would return.
- Reading from, writing to, or deallocating an address that is not yet allocated aborts the program.
- We do not have an operator that gives you the address of variables in store (like $\&$ in C).

Linked Lists

- We abbreviate $i \mapsto 1$ and $i + 1 \mapsto 2$ to $i \mapsto 1, 2$.
- Assume that we represent lists in heap by linked lists.
- E.g $[1, 2, 3]$ is represented in the following heap, starting from address 34:

$$34 \mapsto 1, 92$$
$$60 \mapsto 3, \mathbf{nil}$$
$$92 \mapsto 2, 60\ .$$

- (We will present a more formal definition later.)

In-Place List Reversal

- If the address i represents a list XS, after executing the following program, i points to **nil** and j represents the *reverse* of XS.

$$\{\, i \text{ represents } XS \,\}$$
$$j := \mathbf{nil}$$
$$\mathbf{do}\ i \neq \mathbf{nil} \to k := {}^*(i + 1)$$
$$\qquad\qquad {}^*(i + 1) := j$$
$$\qquad\qquad j, i := i, k$$
$$\mathbf{od}$$
$$\{\ j \text{ represents } reverse\ XS \,\}$$

- That is, the program reverts a linked list without using additional space.
- Can we prove that it is correct?
- Not that easy..! The loop only works if i and j do not share any nodes. The loop invariant would be something like:

i represents $xs \wedge$
j represents $ys \wedge$ ...
i and j share only **nil**.

- Furthermore, we want to ensure that other data structure in the heap should remain unchanged. Assume that we have another linked-list k, we will need in the invariant:

i represents $xs \wedge$
j represents $ys \wedge$ ...
i and j share only **nil** $\wedge$
k and (i union j) share only **nil**.

- We need to mention every pointer in the invariant. This does not scale well.

2 Separation Logic Basics

- *Separation logic*: a logic for describing and reasoning about heaps, in which sections of heaps are *separated* by default.
- Developed by people including Reynolds and O'Hearn in early 2000's.
- Widely adopted by industry in around 2010's [O'H19].
- Recall: assertions in Hoare logic are predicates on state space (values of variables in the store).
- Assertion in separation logic are predicates on the store *and* the heap.
- We will start with an informal description and give a more formal definition later.

Store and Heap

- A *store* is a (partial) function from variable names to values: $Store = Var \to Val$, where $Val = Int \cup Bool \cup ....$
- A *heap* is a (partial) function from addresses to integers: $Heap = Int \to Int$ — an address is also a Int.

- The domain of a function f is denoted $dom\ f$.
- We denote $dom\ h_0 \cap dom\ h_1 = \emptyset$ by $h_0 \perp h_1$.
- Given functions h_0 and h_1 where $h_0 \perp h_1$, define

$$\begin{aligned}(h_0 \cdot h_1)\ x &= h_0\ x \ \text{ if } x \in dom\ h_0, \\ &= h_1\ x \ \text{ if } x \in dom\ h_1.\end{aligned}$$

Some Primitives

Given a heap h,

- $\mathbf{emp}\ h$ holds if $dom\ h = \emptyset$.
  - **emp** says that nothing is allocated in the heap.
- $e \mapsto e'$ holds of h if $dom\ h = \{\, e \,\}$ and $h\ e = e'$.
  - h is a singleton heap containing only e' in address e.
  - Note that both e and e' are expressions!
- $P * Q$ holds of h if $h = h_0 \cdot h_1$ and $P\ h_0$ and $Q\ h_1$.
  - That $h_0 \cdot h_1$ being defined implies that $h_0 \perp h_1$.
  - h can be decomposed into two *disjoint* heap h_0 and h_1 such that p holds of h_0 and q holds of h_1.
- $True$ holds of any h, while $False$ holds of no h.
- $e \mapsto e_0, e_1, .. e_n \equiv (e \mapsto e_0) * (e{+}1 \mapsto e_1) * .. (e{+}n \mapsto e_n)$.
- $e \mapsto \_ \equiv \langle \exists v :: e \mapsto v \rangle$.
- $e \hookrightarrow e' \equiv (e \mapsto e') * True$.
- *separating implication* $p \ \mathrel{-\!\!*}\ q$ will be introduced later.

The True Story

- The presentation above was very simplified.
- In fact, all the predicates introduced above are predicate on *store and heap*, because we need the store to evaluate an expression.
- We will keep it simple for now. For a more precise account, see Reynolds [Rey02, Rey11].
- Keep in mind, for example, that $x \mapsto 3$, where x is a variable, actually means x is mapped to some a in the store, and a is mapped to 3 in the heap.
  - The predicate can be invalidated if either the value of x or the value stored in the heap changes.

Examples

- $x \mapsto 3, y$.
- $(x \mapsto 3, y) * (y \mapsto 3, x)$.
- $(x \mapsto 3, y) \wedge (y \mapsto 3, x)$.
- $(x \hookrightarrow 3, y) \wedge (y \hookrightarrow 3, x)$.

Separating Implication

- *Separating implication* is defined by:

$$(P \mathrel{-\!\!*} Q)\ h = \langle \forall h_0 : h_0 \perp h \wedge P\ h_0 : Q\ (h_0 \cdot h) \rangle\ .$$

- That is, $P \ \mathrel{-\!\!*}\ Q$ holds of h if, given any h_0 that is disjoint from h and satisfies P, we have $h_0 \cdot h$ satisfies Q.

Example

- Suppose P asserts various things, including $x \mapsto 3, 4$. Thus P holds of

$$\begin{array}{l} s : x = a \\ h : a \mapsto 3, a+1 \mapsto 4, \text{rest of heap} \end{array}$$

- $(x \mapsto 3, 4) \ \mathrel{-\!\!*}\ P$ holds of the following store and heap:

$$\begin{array}{l} s : x = a \\ h : \text{rest of heap} \end{array}$$

- $(x \mapsto 1, 2) * ((x \mapsto 3, 4) \mathrel{-\!\!*} P)$ holds of the following store and heap:

$$\begin{array}{l} s : x = a \\ h : a \mapsto 1, a+1 \mapsto 2, \text{rest of heap} \end{array}$$

Heap Mutation – Motivation

- From the example above we notice that

$$\begin{array}{l} \{(x \mapsto 1) * ((x \mapsto 3) \mathrel{-\!\!*} P)\} \\ {}^*x := 3 \\ \{P\} \end{array}$$

- To be slightly more general,

$$\begin{array}{l} \{(x \mapsto \_) * ((x \mapsto 3) \mathrel{-\!\!*} P)\} \\ {}^*x := 3 \\ \{P\} \end{array}$$

- We will see a more general rule later.

3 Commands and Rules

Rule of Constancy

- In logic systems, the following notation denotes "Q can be established by establishing P":

$$\frac{P}{Q}$$

- In Hoare logic, the following "rule of constancy" holds:

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

  where S does not mutate variables in R.

- It allows us to reason about programs in a more modular way.

- However, rule of constancy does not hold for programs allowing dynamic memory management. The following does not hold, for example.

$$\frac{\{x \mapsto \_\}\ {}^*x := 4\ \{x \mapsto 4\}}{\{x \mapsto \_ \wedge\ y \mapsto 3\}\ {}^*x := 4\ \{x \mapsto 4 \wedge\ y \mapsto 3\}}$$

- ${}^*x := 4$ does not mutate y. Yet the conclusion is invalidated when $x = y$.

Frame Rule

- With the introduction of separating conjunction, we do have (for those S that do not mutate variables in R):

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

- The rule above is called the "frame rule". With it we can again reason about programs modularly.

- Wanting to have such rule is the very reason why separation logic was developed.

Commands

- Now we discuss rules associated with each pointer manipulating command.

- Each command is associated with three types of rule: local, global (forward), and backward rules.

Mutation

- Local: $\{e \mapsto \_\}\ {}^*e := e'\ \{e \mapsto e'\}$.

- Global: $\{(e \mapsto \_) * R\}\ {}^*e := e'\ \{(e \mapsto e') * R\}$.

- Backwards: $\{(e \mapsto \_) * ((e \mapsto e') \mathrel{-\!\!*} P)\}\ {}^*e := e'\ \{P\}$.

- The global rule is often the result of applying the frame rule to the local rule.

Deallocation

- Local: $\{e \mapsto \_\}\ \mathit{free}\ e\ \{\mathbf{emp}\}$.

- Global: $\{(e \mapsto \_) * R\}\ \mathit{free}\ e\ \{R\}$.

- For this case, the global rule is also a backwards rule.

Allocation, Non-Overwriting

A simpler, *non-overwriting* case, where x does not occur free in e.

- Local: $\{\mathbf{emp}\}\ x := \mathbf{cons}\ e\ \{x \mapsto e\}$,

- Global: $\{R\}\ x := \mathbf{cons}\ e\ \{(x \mapsto e) * R\}$.

- The backwards rule and the general case is much more complex — we will discuss them later.

- We have not yet discussed the rule for looking up $(x := {}^*e)$ — which turns out to be surprisingly complex. Discussion postponed.

Example

The following code fragment tries to glue together adjacent cells, if possible.

$\{(x \mapsto \_) * (y \mapsto \_)\}$
if $y = x + 1 \ \rightarrow \mathit{skip}$
$\mid\ x = y + 1 \ \rightarrow x := y$
$\mid\ |x - y| > 1 \rightarrow \mathit{free}\ x; \mathit{free}\ y$
$\qquad\qquad\qquad\qquad x := \mathbf{cons}\ (1, 2)$
fi
$\{x \mapsto \_, \_\}$

Allocation, General Case

- Local:

$$\{x = X \wedge \mathbf{emp}\}\ x := \mathbf{cons}\ e\ \{x \mapsto e[x \backslash X]\}\ ,$$

  where X is distinct from x and does not occur free in e.

- Global:

  $\{R\}\, x := \mathbf{cons}\ e\, \{\langle \exists x_0 :: (x \mapsto e[x\backslash x_0]) * R[x\backslash x_0]\rangle\}$,

  where x_0 is distinct from x and does not occur free in e and R.

- Backwards:

  $\{\langle \forall x_1 :: (x_1 \mapsto e) \mathrel{-\!\!*} P[x\backslash x_1]\rangle\}\, x := \mathbf{cons}\ e\, \{P\}$,

  where x_1 is distinct from x and does not occur free in e and R.

Lookup, Non-Overwriting

Provided that x does not occur free in e,

- Local: $\{e \mapsto v\}\, x := {}^*e\, \{x = v \wedge e \mapsto x\}$.
- Global:

  $\{\langle \exists v :: (e \mapsto v) * R[x\backslash v]\rangle\}\, x := {}^*e\, \{(e \mapsto x) * R\}$,

  where $v \notin \mathit{free}\ e \cup (\mathit{free}\ R - \{x\})$.

Removing $\exists$

- Note that v in the global rule can be x, which gives us this special case:

  $$\{\langle \exists x :: (e \mapsto x) * R\rangle\}\, x := {}^*e\, \{(e \mapsto x) * R\} \ .$$

- That is, $x := {}^*e$ can be used to remove an existential quantification — a common usage.

Lookup, General

- Local:

  $\{x = x_0 \wedge e \mapsto v\}\, x := {}^*e\, \{x = v \wedge e[x\backslash x_0] \mapsto x\}$,

  where x, x_0, v distinct.

- Global:

  $$\{\langle \exists v :: (e \mapsto v) * R[x_0\backslash x]\rangle\}$$
  $$x := {}^*e$$
  $$\{\langle \exists x_0 :: (e[x\backslash x_0] \mapsto x) * R[v\backslash x]\rangle\}$$,

  where x, x_0, v distinct, x_0 and v not free in e, x not free in R.

- Backwards:

  $$\{\langle \exists v :: (e \mapsto v) * ((e \mapsto v) \mathrel{-\!\!*} P[x\backslash v])\rangle\}$$
  $$x := {}^*e$$
  $$\{P\}$$

- Backwards, in a shorter form:

  $$\{\langle \exists v :: (e \hookrightarrow v) \wedge P[x\backslash v]\rangle\}$$
  $$x := {}^*e$$
  $$\{P\}$$

Lookup — Example

- To comprehend the global rule we see an example. Let y, x and respectively points to two adjacent nodes in a list.
- Performing $x := {}^*(x+1)$ points x to its *next* node.

  $$\{\langle \exists v :: (y+1 \mapsto x) * (x+1 \mapsto v) * (v+1 \mapsto \mathbf{nil})\rangle\}$$
  $$x := {}^*(x+1)$$
  $$\{\langle \exists x_0 :: (y+1 \mapsto x_0) * (x_0+1 \mapsto x) * (x+1 \mapsto \mathbf{nil})\rangle\}$$

- In this example $R = (y+1 \mapsto x_0) * (v+1 \mapsto \mathbf{nil})$.
- Reynolds [Rey11] says that this global rule is the most commonly used among the three. I personlly prefer the backwards rule, with algebraic properties...

4 Algebraic Properties

Commutativity and Associativity

$$P * Q \equiv Q * P$$
$$(P * Q) * R \equiv P * (Q * R)$$
$$P * \mathbf{emp} \equiv P$$

Distributivity

$$(P \vee Q) * R \equiv (P * R) \vee (Q * R)$$
$$(P \wedge Q) * R \Rightarrow (P * R) \wedge (Q * R)$$

With x not free in Q,

$$\langle \exists x : R : P\rangle * Q \equiv \langle \exists x : R : P * Q\rangle$$
$$\langle \forall x : R : P\rangle * Q \Rightarrow \langle \forall x : R : P * Q\rangle$$

Monotonicity and Currying

- Monotonicity:

$$\frac{P \Rightarrow P' \qquad Q \Rightarrow Q'}{P * Q \Rightarrow P' * Q'}$$

- Currying and uncurrying:

$$((P * Q) \Rightarrow R) \equiv (P \Rightarrow (Q \mathrel{-\!\!*} R))$$

Rules regarding $\mapsto$ and $\hookrightarrow$

- $(e_0 \mapsto h_0) \wedge (e_1 \mapsto h_1) \equiv (e_0 \mapsto h_0) \wedge (e_0 = e_1) \wedge (h_0 = h_1)$
- $(e_0 \mapsto h_0) * (e_1 \mapsto h_1) \Rightarrow e_0 \neq e_1$
- $\mathbf{emp} \equiv \langle \forall x :: \neg\, (x \hookrightarrow \_) \rangle$
- $(e \hookrightarrow h) \wedge P \equiv\ (e \mapsto h) * ((e \mapsto h) \mathrel{-\!\!*} P)$

Purity

- P is *pure* if $P\ h$ implies $P\ h'$ for all h and h'.
- P is independent from heaps.
- $(P_0 \wedge P_1) \Rightarrow (P_0 * P_1)$ if P_0 or P_1 is pure.
- $(P_0 * P_1) \Rightarrow (P_0 \wedge P_1)$ if P_0 and P_1 are pure.
- $(P \wedge Q) * R \equiv (P * R) \wedge Q$ if Q is pure.
- $(P \mathrel{-\!\!*} Q) \Rightarrow (P \Rightarrow Q)$ if P_0 is pure.
- $(P \Rightarrow Q) \Rightarrow (P \mathrel{-\!\!*} Q)$ if P_0 and P_1 are pure.

Strictly Exactness

- P is *strictly exact* if it uniquely determines the heap. That is $P\ h$ and $P\ h'$ implies $h = h'$.
- $\mathbf{emp}$ is strictly exact; $e \mapsto h$ is strictly exact; $e \hookrightarrow h$ is not.
- $(Q * \mathit{True}) \wedge P \Rightarrow Q * (Q \mathrel{-\!\!*} P)$ if Q is strictly exact.
  - The other direction ($\Leftarrow$) holds for all Q. See below.
- There are other important classes of assertions: precise, intuitionistic, supported, etc., which we cannot cover here.
- See Reynolds [Rey11] for more information.

Derived Properties

- Some useful properties —
- $Q * (Q \mathrel{-\!\!*} P) \Rightarrow P$.
- $Q * (Q \mathrel{-\!\!*} P) \Rightarrow (Q * \mathit{True}) \wedge P$.
- $R \Rightarrow Q \mathrel{-\!\!*} (Q * R)$.

A Formal Definitions

We present a more precise definition (while not complete) of components of separation logic here.

To evaluate an expression we need to know the values of its free variables, which can be looked up in a store. Given a store $s : \mathit{Var} \to \mathit{Val}$, the value of an expression e with respect to s is often denoted by $[\![e]\!]_s$, which can be defined inductively on the structure of e:

$$
\begin{aligned}
&[\![x]\!]_s = s\ x \quad \text{, } x \text{ a variable} \\
&[\![e_0 + e_1]\!]_s = [\![e_0]\!]_s + [\![e_1]\!]_s \\
&[\![e_0 \times e_1]\!]_s = [\![e_0]\!]_s \times [\![e_1]\!]_s \\
&\ldots
\end{aligned}
$$

Other cases omitted.

Let $s : \mathit{Var} \to \mathit{Val}$ be a store, $h : \mathit{Addr} \to \mathit{Int}$ a heap. The notation

$$s, h \vDash P$$

denotes "the predicate P holds of s and h."

$$
\begin{aligned}
&s, h \vDash b \quad \equiv [\![b]\!]_s \quad \text{, for } b : \mathit{Bool} \\
&s, h \vDash \neg\, P \quad \equiv \neg\, (s, h \vDash P) \\
&s, h \vDash P \wedge Q \equiv (s, h \vDash P) \wedge (s, h \vDash Q) \\
&s, h \vDash P \vee Q \equiv (s, h \vDash P) \vee (s, h \vDash Q) \\
&\quad \text{similarly for } \Rightarrow, \equiv, \text{etc} \\
&s, h \vDash \mathbf{emp} \quad \equiv \mathit{dom}\ h = \emptyset \\
&s, h \vDash e \mapsto e' \quad \equiv \mathit{dom}\ h = \{[\![e]\!]_s\} \wedge h\ [\![e]\!]_s = [\![e']\!]_s \\
&s, h \vDash P * Q \quad \equiv \\
&\quad \langle \exists h_0, h_1 : h_0 \perp h_1 \wedge h = h_0 \cdot h_1 : \\
&\qquad (s, h_0 \vDash P) \wedge (s, h_1 \vDash Q) \rangle \\
&s, h \vDash P \mathrel{-\!\!*} Q \equiv \\
&\quad \langle \forall h' : h' \perp h : (s, h' \vDash P) \Rightarrow (s, (h' \cdot h) \vDash Q) \rangle \\
&s, h \vDash \langle \forall x :: P \rangle \equiv \langle \forall v :: (s : x \rightarrowtail v), h \vDash P \rangle \\
&s, h \vDash \langle \exists x :: P \rangle \equiv \langle \exists v :: (s : x \rightarrowtail v), h \vDash P \rangle
\end{aligned}
$$

We borrow the notation from the previous lectures:

$$
\begin{aligned}
(s : x \rightarrowtail v)\ y &= v \quad \text{, if } x = y; \\
&= s\ y \quad \text{, otherwise.}
\end{aligned}
$$

References

[O'H12] P. W. O'Hearn. A primer on separation Logic (and automatic program verification and analysis). In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security: Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series*, pages 286–318, 2012.

[O'H19] P. W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, February 2019.

[Rey02] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In G. D. Plotkin, editor, *Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society Press, 2002.

[Rey11] J. C. Reynolds. 15-818A3 Introduction to Separation Logic. Carnegie Mellon University. `https://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www15818As2011/cs818A3-11.html`, 2011.