Programming Languages: Imperative Program Construction 11. Separation Logic I

Shin-Cheng Mu

Autumn Term, 2021

National Taiwan University and Academia Sinica

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

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

$$\textbf{proc}\ \mathit{swap}\ (\mathit{ref}\ x, y : \mathit{Int}) = \\ x := x - y; y := x + y; x := y - x$$

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

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 **nil** be a unique address.
- $p := \textbf{cons}\ (1, 2)$ — 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 **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.
- **free** e — free the cell having address
- In the last three cases the address e must have been allocated.

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).

- 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:

$$\begin{aligned} &34 \mapsto 1, 92 \\ &60 \mapsto 3, \textsf{nil} \\ &92 \mapsto 2, 60 \quad . \end{aligned}$$

- (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.

$$
\begin{array}{l}
\{\ i \text{ represents } XS\ \} \\
j := \mathbf{nil} \\
\mathbf{do}\ i \neq \mathbf{nil} \rightarrow k := {}^*(i+1) \\
\qquad\qquad\qquad\quad {}^*(i+1) := j \\
\qquad\qquad\qquad\quad j, i := i, k \\
\mathbf{od} \\
\{\ j \text{ represents } \mathit{reverse}\ XS\ \}
\end{array}
$$

- 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:

$$
\begin{aligned}
&i \text{ represents } xs \wedge \\
&\quad j \text{ represents } ys \wedge \ldots \\
&\qquad i \text{ and } j \text{ share only } \textbf{nil}.
\end{aligned}
$$

- 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:

$$
\begin{aligned}
&i \text{ represents } xs \wedge \\
&\quad j \text{ represents } ys \wedge \ldots \\
&\qquad i \text{ and } j \text{ share only } \textbf{nil} \wedge \\
&\qquad\quad \text{k and } (i \text{ union } j) \text{ share only } \textbf{nil}.
\end{aligned}
$$

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

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.

- 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: $Store = Var \rightarrow Val$, where $Val = Int \cup Bool \cup \ldots$.
- A *heap* is a (partial) function from addresses to integers: $Heap = Int \rightarrow Int$ — 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

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

Given a heap h,

- **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.
- $e \mapsto e_0, e_1, .. e_n \equiv (e \mapsto e_0) * (e + 1 \mapsto e_1) * .. (e + n \mapsto e_n)$.
- $e \mapsto \_ \equiv \langle \exists v :: e \mapsto v \rangle$.
- $e \hookrightarrow e' \equiv (e \mapsto e') * True$.
- *separting 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.
- 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* is defined by:

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

- 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

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

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

$$s : x = a$$
$$h : \text{rest of heap}$$

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

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

- From the example above we notice that

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

- To be slightly more general,

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

- We will see a more general rule later.

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:

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

- 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.

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

- (What if x and y evaluate to the same address?)

Frame Rule

- With the introduction of separating conjunction, we do have:

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

- 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.

- Now we discuss rules associated with each pointer manipulation command.
- Each command is associated with three types of rule: local, global (forward), and backward rules.

- 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 frame rule to the local rule.

- 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.

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

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

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

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

- Local:

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

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

- Global:

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

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

- Backwards:

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

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

Provided that x does not occur free in e,

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

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

where v does not occur free in e and R.

Lookup, General

- Local:

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

  where x, x_0, v distinct.

- Global:

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

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

- Backwards:

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

- Backwards, in a shorter form:

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