

# 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.

## REMARK: PROCEDURE CALLS

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

```
proc swap (ref x, y : Int) =  
  x := x - y; y := x + y; x := y - x
```

- `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.

# 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.
- $p := \text{cons}(1, 2)$  — 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  $f$ .
- $\text{free } e$  — free the cell having address
- In the last three cases the address  $e$  must have been allocated.

## EXAMPLE

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

## 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:

$34 \mapsto 1, 92$

$60 \mapsto 3, \text{nil}$

$92 \mapsto 2, 60$  .

- (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  $\text{nil}$  and  $j$  represents the *reverse* of  $XS$ .

```
{  $i$  represents  $XS$  }  
 $j := \text{nil}$   
do  $i \neq \text{nil} \rightarrow k := *(i + 1)$   
           $*(i + 1) := j$   
           $j, i := i, k$   
od  
{  $j$  represents reverse  $XS$  }
```

- That is, the program reverts a linked list without using additional space.
- Can we prove that it is correct?

## IN-PLACE LIST REVERSAL

- 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  $ys \wedge \dots$   
 $i$  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  $ys \wedge \dots$   
 $i$  and  $j$  share only **nil**  $\wedge$   
 $k$  and  $(i \text{ union } j)$  share only **nil**.

- 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  $Val = Int \cup Bool \cup \dots$
- 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

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

## SOME PRIMITIVES

Given a heap  $h$ ,

- **emp**  $h$  holds if  $\text{dom } h = \emptyset$ .
  - **emp** says that nothing is allocated in the heap.
- $e \mapsto e'$  holds of  $h$  if  $\text{dom } h = \{e\}$  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*.
- $e \mapsto e_0, e_1, \dots e_n \equiv (e \mapsto e_0) * (e + 1 \mapsto e_1) * \dots (e + n \mapsto e_n)$ .
- $e \mapsto \_ \equiv \langle \exists v :: e \mapsto v \rangle$ .
- $e \hookrightarrow e' \equiv (e \mapsto e') * \text{True}$ .
- *separating implication*  $p \multimap q$  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:

$$(P \multimap Q) h = \langle \forall h_0 : h_0 \perp h \wedge P h_0 : Q (h_0 \cdot h) \rangle .$$

- That is,  $P \multimap Q$  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

$s : x = a$

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

- $(x \mapsto 3, 4) \multimap P$  holds of the following store and heap:

$s : x = a$

$h : \text{rest of heap}$

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

$s : x = a$

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

## HEAP MUTATION – MOTIVATION

- From the example above we notice that

$$\begin{array}{l} \{(x \mapsto 1) * ((x \mapsto 3) \multimap P)\} \\ *x := 3 \\ \{P\} \end{array}$$

- To be slightly more general,

$$\begin{array}{l} \{(x \mapsto -) * ((x \mapsto 3) \multimap P)\} \\ *x := 3 \\ \{P\} \end{array}$$

- 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:

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

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.

- 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.

- Local:  $\{e \mapsto \_ \} * e := e' \{e \mapsto e'\}$ .
- Global:  $\{(e \mapsto \_) * R \} * e := e' \{(e \mapsto e') * R\}$ .
- Backwards:  $\{(e \mapsto \_) * ((e \mapsto e') \multimap P) \} * e := e' \{P\}$ .
- The global rule is often the result of applying the frame rule to the local rule.

- Local:  $\{e \mapsto \_ \} \text{free } e \{ \text{emp} \}$ .
- Global:  $\{ (e \mapsto \_) * R \} \text{free } e \{ R \}$ .
- 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:  $\{\text{emp}\} x := \text{cons } e \{x \mapsto e\},$
- Global:  $\{R\} x := \text{cons } e \{(x \mapsto e) * R\}.$
- 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 ↦ -) * (y ↦ -)}  
if y = x + 1 → skip  
| x = y + 1 → x := y  
| |x - y| > 1 → free x; free y  
               x := cons (1, 2)  
fi  
{x ↦ -, -}
```

## 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:

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

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

- Backwards:

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

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



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

- 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:

$$\{\langle \exists v :: (e \mapsto v) * ((e \mapsto v) \multimap P[x \backslash v]) \rangle\}$$

$$x := *e$$

$$\{P\}$$

- Backwards, in a shorter form:

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

## 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.

$$\{\langle \exists v :: (y + 1 \mapsto x) * (x + 1 \mapsto v) * (v + 1 \mapsto \text{nil}) \rangle\}$$

$$x := *(x + 1)$$

$$\{\langle \exists x_0 :: (y + 1 \mapsto x_0) * (x_0 + 1 \mapsto x) * (x + 1 \mapsto \text{nil}) \rangle\}$$

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

# 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:

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

- Currying and uncurrying:

$$((P * Q) \Rightarrow R) \equiv (P \Rightarrow (Q \multimap R))$$



## RULES REGARDING $\mapsto$ AND $\hookrightarrow$

- $(e_0 \mapsto h_0) \wedge (e_1 \mapsto h_1) \equiv$   
 $(e_0 \mapsto h_0) \wedge (e_0 = e_1) \wedge (h_0 = h_1)$
- $(e_0 \mapsto h_0) * (e_1 \mapsto h_1) \Rightarrow e_0 \neq e_1$
- **emp**  $\equiv \langle \forall X :: \neg (X \hookrightarrow \_) \rangle$
- $(e \hookrightarrow h) \wedge P \equiv$   
 $(e \mapsto h) * ((e \mapsto h) \multimap P)$

- $P$  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.
- $(P \multimap Q) \Rightarrow (P \Rightarrow Q)$  if  $P_0$  is pure.
- $(P \Rightarrow Q) \Rightarrow (P \multimap Q)$  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  $h = h'$ .
- **emp** is strictly exact;  $e \mapsto h$  is strictly exact;  $e \hookrightarrow h$  is not.
- $(Q * \text{True}) \wedge P \Rightarrow Q * (Q \multimap P)$  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 for more information.

## DERIVED PROPERTIES

- Some useful properties —
- $Q * (Q \multimap P) \Rightarrow P$ .
- $Q * (Q \multimap P) \Rightarrow (Q * \text{True}) \wedge P$ .
- $R \Rightarrow Q \multimap (Q * R)$ .