

# Compilation and Program Analysis (#11) :

## Hoare triples and shape analysis

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/capM1.html>

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

dec 2017



- 1 Floyd-Hoare Logic
- 2 Separation Logic
- 3 Recursive Data Structures

# Hoare triples

- $\{A\} p \{B\}$  a Hoare triple

partial correctness :

if the initial state satisfies assertion  $A$ , and *if the execution of program  $p$  terminates*, then the final state satisfies assertion  $B$

- inference rules
- expressive properties

*functional correctness* rather than absence of runtime errors

# Hoare logic | main ingredients

**programmers**

$X := Y+3$

**(Hoare) logicians**

$X \geq Y+3$

ingredients in Hoare logic :

a language for programs  $p$  IMP

a language for assertions  $A$

inference rules

important aspects :

- invariants in loops
- logical deduction rule
- backward reasoning (in the rule for assignment)

## Hoare Logic - rules

$$\begin{array}{c}
 \frac{}{\{A[a/X]\} X := a \{A\}} \qquad \frac{}{\{A\} \text{skip} \{A\}} \qquad \frac{\{A_1\} p_1 \{A_2\} \quad \{A_2\} p_2 \{A_3\}}{\{A_1\} p_1; p_2 \{A_3\}} \\
 \\
 \frac{\{A \wedge a \geq 0\} p_1 \{B\} \quad \{A \wedge \neg(a \geq 0)\} p_2 \{B\}}{\{A\} \text{if } a \geq 0 \text{ then } p_1 \text{ else } p_2 \{B\}} \\
 \\
 \frac{\{A_I \wedge a \geq 0\} p \{A_I\}}{\{A_I\} \text{while } a \geq 0 \text{ do } p \{A_I \wedge \neg(a \geq 0)\}} \\
 \\
 \frac{A_1 \Rightarrow A_2 \quad \{A_2\} p \{B_2\} \quad B_2 \Rightarrow B_1}{\{A_1\} p \{B_1\}}
 \end{array}$$

# Hoare logic : metatheoretical properties 1/2

- **operational semantics and validity**

- big step operational semantics for IMP :  $(\sigma, p) \rightarrow \sigma'$ 
  - $\sigma$  is an *environment*
  - $\sigma : \mathcal{V} \rightarrow \mathbb{Z}$  a map from variables to integers  
given some program  $p$ ,  $\sigma$  is a partial mapping from a *finite set of variables* to  $\mathbb{Z}$
- the triple  $\{A\} p \{B\}$  is **valid** :  
for all  $\sigma$ , if  $\sigma$  satisfies  $A$  and  $(\sigma, p) \rightarrow \sigma'$ , then  $\sigma'$  satisfies  $B$

## Hoare logic : metatheoretical properties 2/2

- **correctness** If the triple  $\{A\} p \{B\}$  can be derived using the inference rules of Hoare logic, then it is valid.
  - NB : we could also rely on *denotational semantics*  
associate to each program  $p$  some function  $F_p$  from environments to environments
- **(relative) completeness** any valid triple can be constructed in Hoare logic, *provided* we can decide validity of the assertions (*i.e.*, *decide whether  $A$  always holds*)
- logic rules capture the properties we want to express

## Correct rules and completeness

- the 6 rules of Hoare logic are not the only correct rules
- for instance, the **rule of constancy** is correct too

$$\frac{\{A\}p\{B\}}{\{A \wedge C\}p\{B \wedge C\}} \text{ no variable in } C \text{ is modified by } p$$

- completeness : no new Hoare triple can be established if we add the rule of constancy
  - the 6 rules “tell everything”
  - using the rule of constancy makes proofs easier/more natural/more readable

somehow, completeness is not only a theoretical question



# The axiom for assignment

the axiom for assignment goes backwards

$$\frac{}{\{A[a/X]\} X := a \{A\}}$$

(consider  $X := X + 3$  to convince yourself)

## One example

$u := 0;$

**While**  $x > 1$  **Do**

**if** *pair*( $x$ ) **then**  $x := \frac{x}{2}; y := y * 2$  **else**  $x := x - 1; u := u + y$

**fi**

**od;**

$y := y + u;$

**Show :**

$$\{x = x_0 \wedge y = y_0 \wedge x_0 > 0\} S \{y = x_0 * y_0\}$$

- 1 Floyd-Hoare Logic
- 2 Separation Logic
- 3 Recursive Data Structures

# Programs manipulating pointers 1/2

- Hoare logic deals essentially with **control**

`if  $a \geq 0$  then  $p_1$  else  $p_2$      $p_1; p_2$     while  $a \geq 0$  do  $p$`

- move to a *richer language* :

add (some kind of) pointers and handling of **memory**

- allocation
- modification    (move pointers around)
- liberation/deallocation

## Programs manipulating pointers 2/2

- different kinds of properties
  - typical runtime errors we want to detect :  
memory leaks, invalid disposal, invalid accesses  
  
typically, other approaches either *assume* memory safety,  
or forbid dynamic memory allocation
  - describe what programs manipulating pointers do
- adopt the same methodological framework

Separation Logic is an enrichment of Floyd-Hoare logic

# Extending Mu

- structure of memory at runtime
  - in (traditional) Hoare-Floyd logic, programs manipulate **variables**
    - the **environment** just records the (integer) value of each variable
      - that is all we know about the memory*
  - dynamically allocated memory : add a **heap** component.

# Extending the Mu language

Extending the programming language with new constructions :  
nil, cons, [x]. (slides from M. Parkinson)

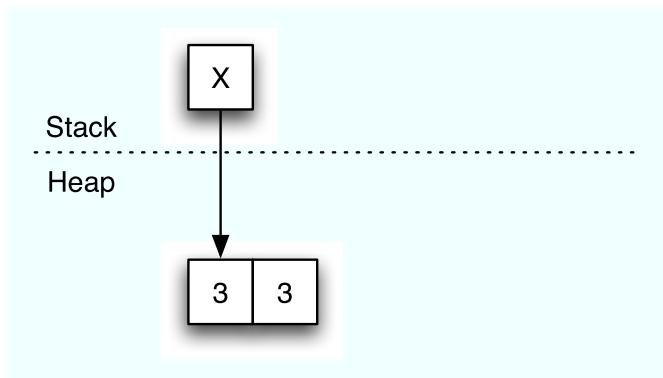
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



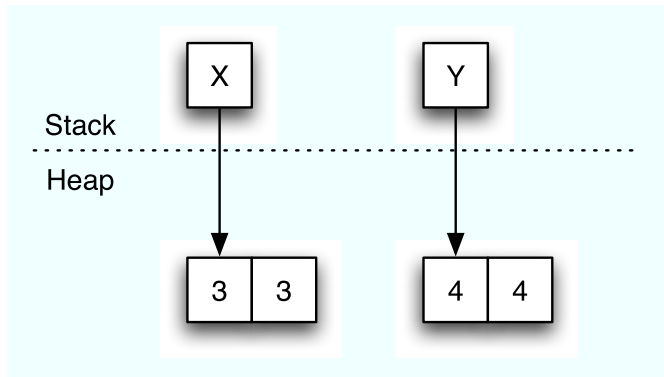
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



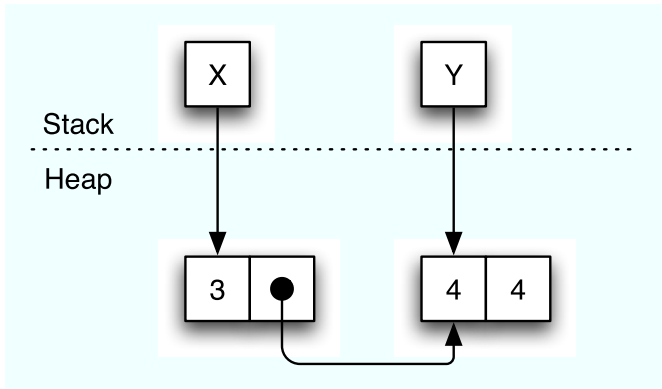
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



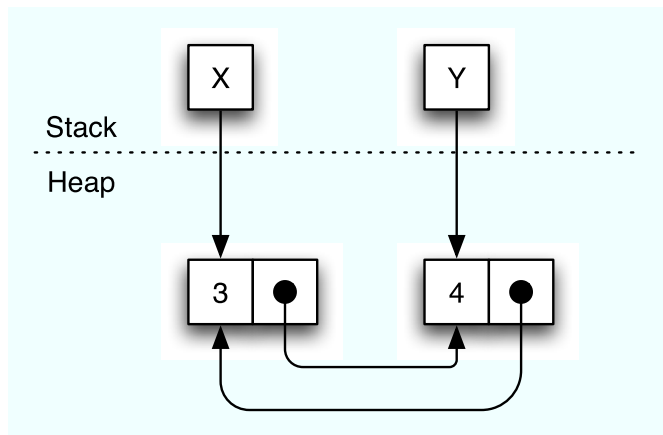
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



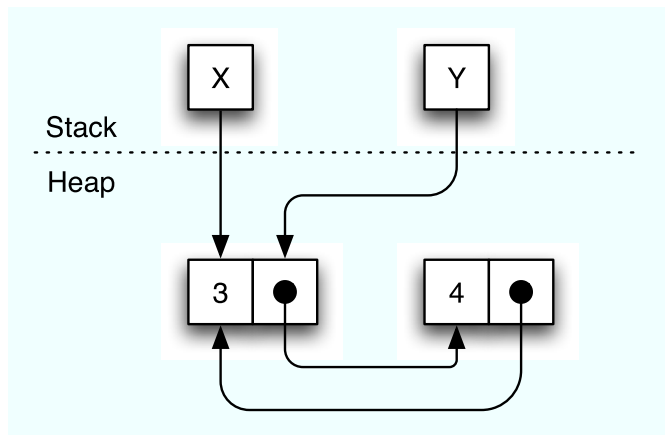
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



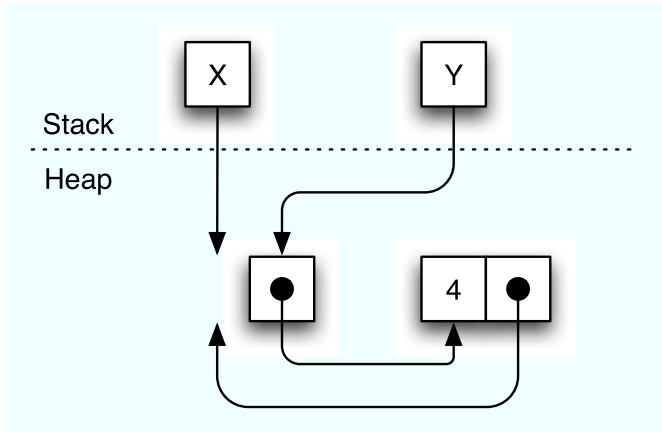
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



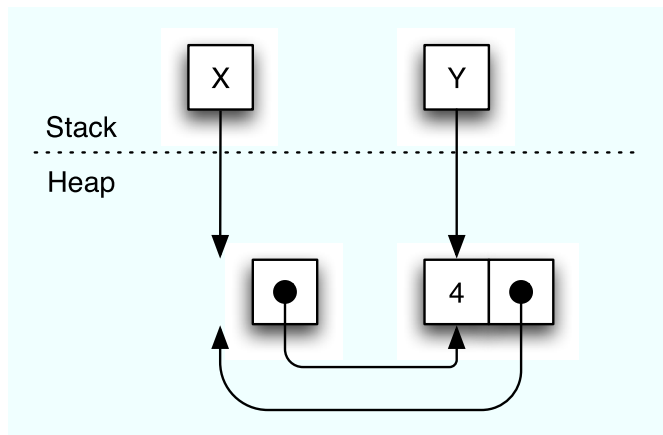
## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



## + Example program

```
x = cons(3,3);  
y = cons(4,4);  
[x+1] = y;  
[y+1] = x;  
y = x+1;  
dispose x;  
y = [y];
```



# Extending Mu - example

- ▶ what does this program do ?

```
J := nil ;  
while I != nil do  
  K := [I + 1] ;  
  [I + 1] := J ;  
  J := I ;  
  I := K
```



## Extending the semantics 1/2

- a memory state is  $(\sigma, h)$  where
  - $\sigma$  is a store : Variables  $\rightarrow$  Values (addresses OR constants).
  - $h$  is a heap : Adresses  $\rightarrow$  Values. We denote by  $dom(h)$  the set of adresses on which  $h$  is defined.
- Hoare logic assertions state properties about the environment

$$X \geq Y * Z + Q \wedge T > 0$$

- add formulas to reason about the heap ( $*$ ,  $\mapsto$ ).
- NB :  $X \mapsto 52$  usually makes more sense than  $32 \mapsto 52$

(both are assertions)

# Operational semantics of the new operators

Let us define  $(\sigma, h) \Downarrow (\sigma', h')$  the semantics :

- **Lookup** :  $(x := [a]) (\sigma, h) \Downarrow (\sigma[k/x], h)$  if  $\mathcal{A}(a)\sigma = i$ ,  $i \in \text{dom}(h)$  (else error), and  $h(i) = k$ .
- **Mutation** :  $([a_1] := a_2) (\sigma, h) \Downarrow (\sigma, h[k/i])$  if  $\mathcal{A}(a_1)\sigma = i$ ,  $i \in \text{dom}(h)$  (else error), and  $\mathcal{A}(a_2)\sigma = k$
- **Allocation** :  $(X = \text{cons}(a_1, \dots, a_n))$  Allocation cannot fail :  $\mathcal{A}(a_j)\sigma = k_j$  for all  $j$ ,  $i$  is a new fresh address, then  $h' = h[k_1/i, k_2/i + 1 \dots]$  and  $\sigma' = \sigma[x \mapsto i]$ .
- **Deallocation** :  $(\text{free}(a)) (\sigma, h) \Downarrow (\sigma, h \setminus i)$   $\mathcal{A}(a_2)\sigma = i$  and  $i \in \text{dom}(h)$ , (else error).

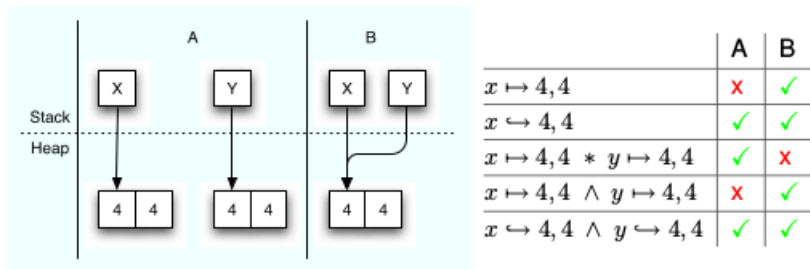
$[k/x]$  denotes “x mapped to k”

## A new logic on states

- $(\sigma, h) \models a \geq 0$  iff  $\llbracket a \rrbracket_\sigma = k \geq 0$ .
- $(\sigma, h) \models \neg A$  iff not  $(\sigma, h) \models A$ .
- $(\sigma, h) \models A \wedge B$  iff  $(\sigma, h) \models A$  and  $(\sigma, h) \models B$ .
- $(\sigma, h) \models \exists x, A(x)$  iff there exists  $x \in \mathbb{N}$  such that  
 $(\sigma, h) \models A(x)$
- $(\sigma, h) \models emp$  iff  $dom(h) = \emptyset$ .
- $(\sigma, h) \models a_1 \mapsto a_2$  iff  $dom(h) = \{i\}$ ,  $h(i) = k$  where  $\llbracket a_1 \rrbracket_\sigma = i$   
and  $\llbracket a_2 \rrbracket_\sigma = k$ .
- $(\sigma, h) \models A_1 * A_2$  iff  $h = h_1 \uplus h_2$  and  $(\sigma, h_i) \models A_i$ .

**Here  $\mapsto$  is a new (logic) symbol**

# Example of heaps



where  $E \mapsto E_0, \dots, E_n \stackrel{\text{def}}{=} E \mapsto E_0 * E + 1 \mapsto E_1 * \dots * E + n \mapsto E_n$

and  $E \hookrightarrow E' \stackrel{\text{def}}{=} E \mapsto E' * \text{true}$

# Hoare triples in Separation logic | interpretation

$\{A\} p \{B\}$  holds iff

$\forall \sigma, h.$ , if  $(\sigma, h) \models A$ ,       $((\sigma, h)$  satisfies  $A$ )  
then

- $(\sigma, h), p \not\Downarrow$  error, and
- if  $(\sigma, h), p \Downarrow (\sigma', h')$ , then  $(\sigma', h') \models B$

like in traditional Hoare logic, but :

- the state has a heap component
- absence of forbidden access to the memory

## Small axioms (Hoare Triples)

- **Lookup** :  $\{a \mapsto i \wedge X = j\} X := [a] \{X = i \wedge a[j/X] \mapsto i\}$ .

If  $X$  is not in  $vars(a)$ , this rule becomes

$$\{a \mapsto i\} X := [a] \{X = i \wedge a \mapsto i\}.$$

- **Mutation** :  $\{\exists i, a_1 \mapsto i\} [a_1] := a_2 \{a_1 \mapsto a_2\}$ .
- **Allocation** :  $\{X = i \wedge emp\} X := cons(a_1, \dots, a_n) \{X \mapsto a_1[i/X] * X + 1 \mapsto a_2[i/X] * \dots * X + n - 1 \mapsto a_n[i/X]\}$
- **Desallocation** :  $\{a \mapsto -\} free(a) \{emp\}$

► axioms for heap-accessing operations are **tight**, i.e. they only refer to the part of the heap they need to access.

## About tightness

Being tight tells us the following :

- suppose we can prove  $\{10 \mapsto 32\} p \{10 \mapsto 52\}$  whatever  $p$  is
- then we know that

if we run  $p$  in a state where cell 11 is allocated, then  $p$  will not change the value of 11

# The frame rule

- the rules of Hoare logic remain sound

- the rule of consistency  $\frac{\{A\} p \{B\}}{\{A \wedge C\} p \{B \wedge C\}}$  no variable in  $C$   
is modified by  $p$   
becomes unsound

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

what if  $x = y$ ?



# The frame rule

- the rules of Hoare logic remain sound
- the rule of consistency  $\frac{\{A\} p \{B\}}{\{A \wedge C\} p \{B \wedge C\}}$  no variable in  $C$   
is modified by  $p$   
becomes unsound

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

what if  $x = y$ ?

- the Frame Rule**

$$\frac{\{A\} p \{B\}}{\{A * C\} p \{B * C\}} \quad \begin{array}{l} \text{no variable in } C \\ \text{is modified by } p \end{array}$$

- separation logic is inherently **modular**

as opposed to *whole program verification*

# Separation logic : sum up

- inference rules
    - those of Hoare logic
    - those for the new programming constructs
  - important things :
    - invariants in while loops, backward rule for assignment, consequence rule
    - (tight) small axioms, footprint, frame rule
  - metatheoretical properties
    - correctness
    - completeness
- control  
memory

- 1 Floyd-Hoare Logic
- 2 Separation Logic
- 3 Recursive Data Structures

## Reasoning about lists 1/2



- a linked list in memory is something like

$$(X_1 \mapsto k_1, X_2) * (X_2 \mapsto k_2, X_3) * \dots * (X_n \mapsto k_n, \underline{nil})$$

$(X \mapsto a, b)$  stands for  $X \mapsto a * (X + 1) \mapsto b$

- describe the structure using assertions :

add the possibility to write **(recursive) equations**

$$list(i) = (i = \underline{nil} \wedge emp) \vee (\exists j, k. (i \mapsto k, j) * list(j))$$

## Linked lists

The preceding formula just specifies that we have a list in memory

We can rely on “mathematical lists” ( $[] , k :: ks$ ) to provide a more informative definition

$$\begin{aligned} list([], i) &= emp \wedge i = \underline{nil} \\ list(k :: ks, i) &= \exists j. (i \mapsto k, j) * list(ks, j) \end{aligned}$$

# Recursive data structures

- we can specify similarly various kinds of data structures
- we can give a meaning to such recursive definitions using Tarski's theorem

# Recursive data structures

- we can specify similarly various kinds of data structures
- we can give a meaning to such recursive definitions using Tarski's theorem
- an **exercise**

$$list(i) = (i = \underline{nil} \wedge emp) \vee (\exists j, k. (i \mapsto k, j) * list(j))$$

- write the code for a while loop that deallocates a linked list,
- and prove  $\{list(X)\} p \{emp\}$ , where  $p$  is your program

# More : Reasoning about concurrent programs

concurrent separation logic

- shared memory, several threads
- permissions, locks, critical sections
- ownership