C04 – Weakest Precondition calculus & Separation Logic

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Weakest Precondition calculus (cont.)

Separation Logic

Weakest Precondition calculus (cont.)

Weakest precondition calculus (WP)

WP is about evaluating a function:

- Given some code $\mathbb{C}$ and postcondition Q, find the unique P which is the weakest precondition such that Q holds after $\mathbb{C}$.
- $P = wp(\mathbb{C}, Q)$
- WP respects Hoare logic: $\{wp(\mathbb{C}, Q)\}\ \mathbb{C}\ \{Q\}$ is true in Hoare logic.

WP is about total correctness.

Total correctness = Termination + Partial correctness

Weakest precondition rules

Rule for Assignment $\boxed{wp(\texttt{x := } \mathbb{E}, Q) \;\equiv\; Q[x/\mathbb{E}]}$

(Q is an assertion involving a variable x and $Q[x/\mathbb{E}]$ indicates the same assertion with all occurrences of x replaced by the expression $\mathbb{E}$)

Rule for sequencing $\boxed{wp(\mathbb{C}_1\texttt{;}\;\; \mathbb{C}_2, Q) \;\equiv\; wp(\mathbb{C}_1, wp(\mathbb{C}_2, Q))}$

Rules for conditionals (equivalent)

$$\boxed{wp(\texttt{if } \mathbb{B} \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, Q) \;\equiv\; (\mathbb{B} \rightarrow wp(\mathbb{C}_1, Q)) \wedge (\neg\mathbb{B} \rightarrow wp(\mathbb{C}_2, Q))}$$

$$\boxed{wp(\texttt{if } \mathbb{B} \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, Q) \;\equiv\; (\mathbb{B} \wedge wp(\mathbb{C}_1, Q)) \vee (\neg\mathbb{B} \wedge wp(\mathbb{C}_2, Q))}$$

Loops

Suppose we have a while loop and some postcondition Q.

The precondition P that we seek is the weakest that:

- establishes Q
- guarantees termination

Loops

Suppose we have a while loop and some postcondition Q.

The precondition P that we seek is the weakest that:

- establishes Q
- guarantees termination

We can take hints for the corresponding rule for Hoare Logic. That is, think in terms of loop invariants.

Loops

Suppose we have a while loop and some postcondition Q.

The precondition P that we seek is the weakest that:

- establishes Q
- guarantees termination

We can take hints for the corresponding rule for Hoare Logic. That is, think in terms of loop invariants.

But termination is a bigger problem!

An undecidable problem

Determining if a program terminates or not on a given input is an **undecidable problem**!

An undecidable problem

Determining if a program terminates or not on a given input is an **undecidable problem**!

So there's no algorithm to compute $wp(\texttt{while}\ \mathbb{B}\ \texttt{do}\ \mathbb{C}, Q)$ in all cases.

But that doesn't mean there are no techniques to tackle this problem that at least work some of the time!

Guaranteeing termination

The precondition P we seek is the weakest that establishes Q and guarantees termination.

How can a loop terminate?

Guaranteeing termination

The precondition P we seek is the weakest that establishes Q and guarantees termination.

How can a loop terminate?

- If the loop body is never entered, then the postcondition Q must already be true and the loop condition $\mathbb{B}$ false.
  - We will call this precondition P_0.
  - $P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

Guaranteeing termination

The precondition P we seek is the weakest that establishes Q and guarantees termination.

How can a loop terminate?

- If the loop body is never entered, then the postcondition Q must already be true and the loop condition $\mathbb{B}$ false.
  - We will call this precondition P_0.
  - $P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$
- Suppose the loop body executes exactly once. In this case:
  - $\mathbb{B}$ must be true initially
  - after the first time through the loop body, P_0 must become true (so that the loop terminates next time through).
  - $P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\}\ \mathbb{C}\ \{P_0\}$

Guaranteeing termination

$P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\}\ \mathbb{C}\ \{P_0\}$

Guaranteeing termination

$P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\}\ \mathbb{C}\ \{P_0\}$

$P_2 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_1)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_1)\}\ \mathbb{C}\ \{P_1\}$

Guaranteeing termination

$P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\}\ \mathbb{C}\ \{P_0\}$

$P_2 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_1)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_1)\}\ \mathbb{C}\ \{P_1\}$

$P_3 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_2)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_2)\}\ \mathbb{C}\ \{P_2\}$

$\ldots$

Guaranteeing termination

$P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\}\ \mathbb{C}\ \{P_0\}$

$P_2 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_1)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_1)\}\ \mathbb{C}\ \{P_1\}$

$P_3 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_2)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_2)\}\ \mathbb{C}\ \{P_2\}$

...

P_k – the weakest precondition under which the loop terminates with postcondition Q after exactly k iterations.

We can capture the definition of P_k with an inductive definition.

An inductive definition

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

An inductive definition

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

If any of the P_k is true in the initial state, then we are guaranteed that the loop will terminate and establish the postcondition Q,

i.e. $\{P_0 \vee P_1 \vee \ldots\}$ `while` $\mathbb{B}$ `do` $\mathbb{C}$ $\{Q\}$ is true.

The weakest precondition for while loops (rule 4/4)

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k \ (k \geq 0 \wedge P_k)}$$

where P_k is defined inductively:

$$\begin{array}{rcl} P_0 & \equiv & \neg\mathbb{B} \wedge Q \\ P_{k+1} & \equiv & \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{array}$$

The weakest precondition for while loops (rule 4/4)

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k \; (k \geq 0 \wedge P_k)}$$

where P_k is defined inductively:

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

Interpretation:

- P_k is the weakest precondition that ensures that the body $\mathbb{C}$ executes exactly k times and terminates in a state in which the postcondition Q holds.
- If our loop is to terminate with postcondition Q, some P_k must hold before we enter the loop
  i.e. $\{P_0 \vee P_1 \vee \ldots\}$ $\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}$ $\{Q\}$ is true.

The weakest precondition for while loops

Applying the *wp* function to a while loop and postcondition will produce an assertion of the form

$$\exists k \ (k \geq 0 \land P_k)$$

P_k may be different for each k, so *wp* may produce an infinitely long assertion! Such an assertion is unsuitable for further manipulation.

The weakest precondition for while loops

Applying the *wp* function to a while loop and postcondition will produce an assertion of the form

$$\exists k \ (k \geq 0 \wedge P_k)$$

P_k may be different for each k, so *wp* may produce an infinitely long assertion! Such an assertion is unsuitable for further manipulation.

We can simplify matters by expressing P_k as a single, finite formula that is parameterised by k.

The weakest precondition for while loops

Applying the *wp* function to a while loop and postcondition will produce an assertion of the form

$$\exists k\ (k \geq 0 \wedge P_k)$$

P_k may be different for each k, so *wp* may produce an infinitely long assertion! Such an assertion is unsuitable for further manipulation.

We can simplify matters by expressing P_k as a single, finite formula that is parameterised by k.

Example

If $P_0 \equiv (n = 0)$, $P_1 \equiv (n = 1)$, $P_2 \equiv (n = 2)$, ..., then $P_k \equiv (n = k)$.

The weakest precondition for while loops

Applying the *wp* function to a while loop and postcondition will produce an assertion of the form

$$\exists k \; (k \geq 0 \land P_k)$$

P_k may be different for each k, so *wp* may produce an infinitely long assertion! Such an assertion is unsuitable for further manipulation.

We can simplify matters by expressing P_k as a single, finite formula that is parameterised by k.

Example

If $P_0 \equiv (n = 0)$, $P_1 \equiv (n = 1)$, $P_2 \equiv (n = 2)$, $\ldots$, then $P_k \equiv (n = k)$.

We must prove correctness of P_k by induction!

The weakest precondition for while loops

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k\ (k \geq 0 \wedge P_k)}$$

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

The weakest precondition for while loops

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k \ (k \geq 0 \wedge P_k)}$$

$$\begin{array}{rcl} P_0 & \equiv & \neg\mathbb{B} \wedge Q \\ P_{k+1} & \equiv & \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{array}$$

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We start by generating some of the P_k sequence:

The weakest precondition for while loops

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k \ (k \geq 0 \wedge P_k)}$$

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg\mathbb{B} \wedge Q$

The weakest precondition for while loops

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k\ (k \geq 0 \wedge P_k)}$$

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg\mathbb{B} \wedge Q$
- $P_1 \equiv (n > 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$

The weakest precondition for while loops

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k\ (k \geq 0 \wedge P_k)}$$

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg\mathbb{B} \wedge Q$
- $P_1 \equiv (n > 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$
- $P_2 \equiv (n > 0) \wedge wp(n := n - 1, n = 1) \equiv (n = 2)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_1)$

The weakest precondition for while loops

$$\boxed{wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q) \equiv \exists k \; (k \geq 0 \wedge P_k)}$$

$$\begin{aligned} P_0 &\equiv \neg\mathbb{B} \wedge Q \\ P_{k+1} &\equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k) \end{aligned}$$

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg\mathbb{B} \wedge Q$
- $P_1 \equiv (n > 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$
- $P_2 \equiv (n > 0) \wedge wp(n := n - 1, n = 1) \equiv (n = 2)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_1)$
- $\ldots$

so it looks pretty likely that $P_k \equiv (n = k)$

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

- We already checked the base case:

  $P_0 \quad \equiv \quad \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

- We already checked the base case:
  $P_0 \quad \equiv \quad \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$
- Now for our induction step:
  We assume $P_i \equiv (n = i)$ for some $i \geq 0$.
  Recall that $P_{i+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_i)$.

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

- We already checked the base case:
  $$P_0 \quad \equiv \quad \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$$
- Now for our induction step:
  We assume $P_i \equiv (n = i)$ for some $i \geq 0$.
  Recall that $P_{i+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_i)$.
  $$\begin{aligned} P_{i+1} \quad &\equiv \quad (n > 0) \wedge wp(n := n - 1, n = i) \\ &\equiv \quad (n > 0) \wedge (n - 1 = i) \\ &\equiv \quad (n > 0) \wedge (n = i + 1) \\ &\equiv \quad (n = i + 1) \end{aligned}$$

The weakest precondition for while loops

Example

Therefore we have

$$wp(\texttt{while n > 0 do n := n-1}, n = 0) \;\equiv\; \exists k\ (k \geq 0 \wedge n = k)$$

The weakest precondition for while loops

Example

Therefore we have

$$wp(\texttt{while n > 0 do n := n-1}, n = 0) \equiv \exists k \; (k \geq 0 \wedge n = k)$$

We can still simplify it further!

Useful trick: $\exists k \; ((k \geq 0) \wedge P_k) \equiv P_0 \vee P_1 \vee P_2 \vee \dots$

In this example we have $(n = 0) \vee (n = 1) \vee (n = 2) \vee \dots$

The weakest precondition for while loops

Example

Therefore we have

$$wp(\texttt{while n > 0 do n := n-1}, n = 0) \equiv \exists k \ (k \geq 0 \land n = k)$$

We can still simplify it further!

Useful trick: $\exists k \ ((k \geq 0) \land P_k) \equiv P_0 \lor P_1 \lor P_2 \lor \ldots$

In this example we have $(n = 0) \lor (n = 1) \lor (n = 2) \lor \ldots$

We can compress this infinite disjunction into a finite final result:

$$wp(\texttt{while n > 0 do n := n-1}, n = 0) \equiv (n \geq 0)$$

Total correctness

Example

We want to find

$$wp(\texttt{while n} \neq \texttt{0 do n := n-1}, n = 0)$$

Total correctness

Example

We want to find

$$wp(\texttt{while n} \neq \texttt{0 do n := n-1}, n = 0)$$

Step 1 - finding the P_k:

- $P_0 \equiv \neg(n \neq 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg\mathbb{B} \wedge Q$
- $P_1 \equiv (n \neq 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$
- $\ldots$
- $P_k \equiv (n = k)$ (induction omitted)

Total correctness

Example

Step 2 - finding the weakest precondition:

$$\begin{aligned} \exists k\ ((k \geq 0) \wedge P_k) &\equiv \exists k\ ((k \geq 0 \wedge (n = k))) \\ &\equiv (n \geq 0) \end{aligned}$$

Thus,

$$wp(\texttt{while n} \neq \texttt{0 do n := n-1}, n = 0) \equiv (n \geq 0)$$

Total correctness

Example

Step 2 - finding the weakest precondition:

$$\begin{aligned}\exists k\ ((k \geq 0) \wedge P_k) &\equiv \exists k\ ((k \geq 0 \wedge (n = k))) \\ &\equiv (n \geq 0)\end{aligned}$$

Thus,

$$wp(\texttt{while n} \neq \texttt{0 do n := n-1}, n = 0) \equiv (n \geq 0)$$

This is not really any different than the previous example.

But what is the trap in this while-loop?

Total correctness

Example

Step 2 - finding the weakest precondition:

$$\begin{aligned} \exists k\ ((k \geq 0) \wedge P_k) &\equiv \exists k\ ((k \geq 0 \wedge (n = k))) \\ &\equiv (n \geq 0) \end{aligned}$$

Thus,

$$wp(\texttt{while n} \neq \texttt{0 do n := n-1}, n = 0) \equiv (n \geq 0)$$

This is not really any different than the previous example.

But what is the trap in this while-loop?

We have automatically found that the while-loop will not terminate for initial values of n less than 0.

WP rules

- Rule for Assignment: $wp(\texttt{x := } \mathbb{E}, Q) \;\equiv\; Q[x/\mathbb{E}]$

  (Q is an assertion involving a variable x and $Q[x/\mathbb{E}]$ indicates the same assertion with all occurrences of x replaced by the expression $\mathbb{E}$)

- Rule for Sequencing: $wp(\mathbb{C}_1\,;\;\mathbb{C}_2, Q) \;\equiv\; wp(\mathbb{C}_1, wp(\mathbb{C}_2, Q))$

- Rule for Conditionals:

  $wp(\texttt{if } \mathbb{B} \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2, Q) \;\equiv\; (\mathbb{B} \to wp(\mathbb{C}_1, Q)) \wedge (\neg\mathbb{B} \to wp(\mathbb{C}_2, Q))$

- There is no algorithm to compute $wp(\texttt{while } \mathbb{B} \texttt{ do } \mathbb{C}, Q)$ in all cases!
  - But that doesn't mean there are no techniques to tackle this problem that at least work some of the time!
  - Inductive definition.

Separation Logic

Adding the heap

We extend our programming language with:

- Memory reads: `x := [`$\mathbb{E}$`]` *(dereferencing)*
- Memory writes: `[`$\mathbb{E}_1$`] :=` $\mathbb{E}_2$ *(update heap)*
- Memory allocation: `x := cons(`$\mathbb{E}_1$`,...`$\mathbb{E}_n$`)`
- Memory deallocation: `dispose` $\mathbb{E}$

Adding the heap

We extend our programming language with:

- Memory reads: `x := [`$\mathbb{E}$`]` *(dereferencing)*
- Memory writes: `[`$\mathbb{E}_1$`] :=` $\mathbb{E}_2$ *(update heap)*
- Memory allocation: `x := cons(`$\mathbb{E}_1$`,...`$\mathbb{E}_n$`)`
- Memory deallocation: `dispose` $\mathbb{E}$

The state is now represented by a pair of type $\mathit{Store} \times \mathit{Heap}$, denoted (σ, h), where

$$\sigma \in \mathit{Store}, \text{ where } \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$$

$$h \in \mathit{Heap}, \text{ where } \mathit{Heap} \triangleq \mathit{Loc} \to \mathit{Val}$$

where $\mathit{Loc} \subseteq \mathit{Val}$.

Adding the heap

Memory reads: `x := [`$\mathbb{E}$`]`

- evaluate expression $\mathbb{E}$ to get location l
- fault if location l is not in the current heap
- otherwise variable `x` is assigned the content of location l

Adding the heap

Memory reads: `x := [𝔼]`

- evaluate expression $\mathbb{E}$ to get location l
- fault if location l is not in the current heap
- otherwise variable `x` is assigned the content of location l

Example (`x := [y+1]`)

σ

y	0xAB

h

0xAB	1
0xAC	2

`x := [y+1]`

σ

y	0xAB
x	2

h

0xAB	1
0xAC	2

Adding the heap

Memory writes: $[\mathbb{E}_1] := \mathbb{E}_2$

- evaluate expression $\mathbb{E}_1$ to get location l
- fault if location l is not in the current heap
- otherwise make the content of location l the value of expression $\mathbb{E}_2$

Adding the heap

Memory writes: $[\mathbb{E}_1]$:= $\mathbb{E}_2$

- evaluate expression $\mathbb{E}_1$ to get location l
- fault if location l is not in the current heap
- otherwise make the content of location l the value of expression $\mathbb{E}_2$

Example (`[y+1] := 5`)

σ

y	0xAB

h

0xAB	1
0xAC	2

`[y+1] := 5`

σ

y	0xAB

h

0xAB	1
0xAC	5

Adding the heap

Memory allocation: `x := cons(`$\mathbb{E}_1$`,...`$\mathbb{E}_n$`)`

- extend the heap with n consecutive new locations $l, l+1, \ldots, l+n-1$
- put values of $\mathbb{E}_1$,...,$\mathbb{E}_n$ into locations $l, l+1, \ldots, l+n-1$ respectively
- extend the stack by assigning `x` the value l
- never fault

Adding the heap

Memory allocation: `x := cons(`$\mathbb{E}_1$`,...`$\mathbb{E}_n$`)`

- extend the heap with n consecutive new locations $l, l+1, \ldots, l+n-1$
- put values of $\mathbb{E}_1$,...,$\mathbb{E}_n$ into locations $l, l+1, \ldots, l+n-1$ respectively
- extend the stack by assigning `x` the value l
- never fault

Example (`p := cons(3,7)`)

σ h `p := cons(3,7)` σ h

h before:

⋮	⋮

σ after:

p	0xAB

h after:

⋮	⋮
0xAB	3
0xAC	7

Adding the heap

Memory deallocation: `dispose` $\mathbb{E}$

- evaluate expression $\mathbb{E}$ to get location l
- fault if location l is not in the current heap
- otherwise remove location l from the heap

Adding the heap

Memory deallocation: `dispose` $\mathbb{E}$

- evaluate expression $\mathbb{E}$ to get location l
- fault if location l is not in the current heap
- otherwise remove location l from the heap

Example (`dispose p+1`)

Before:

σ

p	0xAB

h

0xAB	5
0xAC	6

`dispose p+1`

After:

σ

p	0xAB

h

0xAB	5

Example

```
x := cons(3,3)
```

σ

x	0xAB

h

0xAB	3
0xAC	3

Example

```
x := cons(3,3); y := cons(4,4);
```

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	3
0xDD	4
0xDE	4

Example

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

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	4

Example

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

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

Example

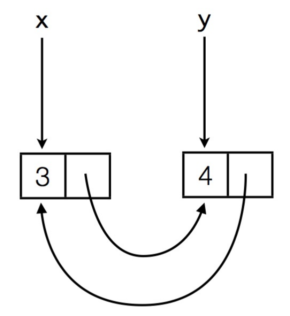
```
x := cons(3,3) ; y := cons(4,4); [x+1] := y; [y+1] := x;
```

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDE	4
0xDE	0xAB



Why separation logic?

Can you suggest a precondition such that this triple holds?

$\{???\}$

```
    [y] := 4; [z] := 5;
```

$\{(\exists y, z)(\mathtt{y} \mapsto y \wedge \mathtt{z} \mapsto z \wedge y \neq z)\}$

Why separation logic?

Can you suggest a precondition such that this triple holds?

$\{\mathtt{y} \neq \mathtt{z} \wedge \mathtt{y} \mapsto \_ \wedge \mathtt{z} \mapsto \_\}$

```
      [y] := 4; [z] := 5;
```

$\{(\exists y, z)(\mathtt{y} \mapsto y \wedge \mathtt{z} \mapsto z \wedge y \neq z)\}$

We need to assume that the locations pointed by $\mathtt{y}$ and $\mathtt{z}$ are different (aliasing).

Note that, for example, $\mathtt{y}$ is used to denote program variables, while y is used to denote logical variables.

Why separation logic?

And now?

```
[y] := 4; [z] := 5;
```

$\{(\exists y, z)(\mathtt{y} \mapsto y \wedge \mathtt{z} \mapsto z \wedge y \neq z \wedge \mathtt{x} \mapsto 3)\}$

We need to assume that the locations pointed by `y` and `z` are different (aliasing).

We also need to know when things stay the same.

Why separation logic?

And now?

$\{\mathtt{y} \neq \mathtt{z} \wedge \mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} \neq \mathtt{z} \wedge \mathtt{y} \mapsto \_ \wedge \mathtt{z} \mapsto \_ \wedge \mathtt{x} \mapsto 3\}$

```
      [y] := 4; [z] := 5;
```

$\{(\exists y, z)(\mathtt{y} \mapsto y \wedge \mathtt{z} \mapsto z \wedge y \neq z \wedge \mathtt{x} \mapsto 3)\}$

We need to assume that the locations pointed by $\mathtt{y}$ and $\mathtt{z}$ are different (aliasing).

We also need to know when things stay the same.

Framing

We want a general concept of things not being affected.

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\mathtt{x} \mapsto 3 \wedge P\}\ \mathbb{C}\ \{Q \wedge \mathtt{x} \mapsto 3\}}$$

What are the conditions on $\mathbb{C}$ and $\mathtt{x} \mapsto 3$?

These are very hard to define if reasoning about a heap and aliasing.

Framing

We want a general concept of things not being affected.

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{\mathtt{x} \mapsto 3 \land P\}\ \mathbb{C}\ \{Q \land \mathtt{x} \mapsto 3\}}$$

What are the conditions on $\mathbb{C}$ and $\mathtt{x} \mapsto 3$?

These are very hard to define if reasoning about a heap and aliasing.

This is where separation logic comes in:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{R * P\}\ \mathbb{C}\ \{Q * R\}}$$

The new connective $*$ ("sep" operator) is used to separate the heap.

Reasoning about pointers

For Hoare logic, we assumed no aliasing of variables!

In most real languages we can have multiple names for the same piece of memory.

How is aliasing a problem?

From Hoare logic to separation logic

- Robert W. Floyd 1967: gave some rules to reason about programs.
- Sometimes, our Hoare Logic is called Floyd-Hoare Logic in recognition.
- Many attempts made to extend Floyd-Hoare Logic to handle pointers.

- Only really solved around 2000 by Reynolds, O'Hearn and Yang using a connective $*$ called separating conjunction.
- To make the presentation less scary, we need to first extend Hoare Logic with an axiom due to Floyd.

Hoare Logic

Syntax	Semantics	Calculus
$\neg \wedge \vee \rightarrow \forall\exists$	FOL	N/A
$= + - \geq \leq \ldots$	Arithmetics	N/A
`:=` `;` `while` `if` `then` `else`	State maps variables to values (no pointers)	N/A
$\{P\}\ \mathbb{C}\ \{Q\}$	If initial state satisfies P and $\mathbb{C}$ terminates then final state satisfies Q	6 Inference Rules

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\}$ `x := ` $\mathbb{E}$ $\{Q\}$ (backward driven)

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\}$ `x := ` $\mathbb{E}$ $\{Q\}$ (backward driven)

Floyd axiom: $\{x = v\}$ `x := ` $\mathbb{E}$ $\{x = \mathbb{E}[x/v]\}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in $\mathbb{E}$
- $\mathbb{E}[x/v]$ means replace all occurrences of x in $\mathbb{E}$ by v

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\}$ `x := ` $\mathbb{E}$ $\{Q\}$ (backward driven)

Floyd axiom: $\{x = v\}$ `x := ` $\mathbb{E}$ $\{x = \mathbb{E}[x/v]\}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in $\mathbb{E}$
- $\mathbb{E}[x/v]$ means replace all occurrences of x in $\mathbb{E}$ by v

Example

Hoare instance: $\{x + 1 = 5\}$ `x := x+1` $\{x = 5\}$

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\}$ `x := ` $\mathbb{E}$ $\{Q\}$ (backward driven)

Floyd axiom: $\{x = v\}$ `x := ` $\mathbb{E}$ $\{x = \mathbb{E}[x/v]\}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in $\mathbb{E}$
- $\mathbb{E}[x/v]$ means replace all occurrences of x in $\mathbb{E}$ by v

Example

Hoare instance: $\{x + 1 = 5\}$ `x := x+1` $\{x = 5\}$

Floyd instance: $\{x = v\}$ `x := x+1` $\{x = v + 1\}$

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\}$ `x := ` $\mathbb{E}$ $\{Q\}$ (backward driven)

Floyd axiom: $\{x = v\}$ `x := ` $\mathbb{E}$ $\{x = \mathbb{E}[x/v]\}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in $\mathbb{E}$
- $\mathbb{E}[x/v]$ means replace all occurrences of x in $\mathbb{E}$ by v

Example

Hoare instance: $\{x + 1 = 5\}$ `x := x+1` $\{x = 5\}$

Floyd instance: $\{x = v\}$ `x := x+1` $\{x = v + 1\}$

- If we want the postcondition $x = 5$ then instantiate v to be 4
  $\{x = 4\}$ `x := x+1` $\{x = 5\}$

Note: does not solve the problem with pointers!

Quiz time!

https://www.questionpro.com/t/AT4NiZrf2O

References

- Lecture Notes on "Formal Methods for Software Engineering", Australian National University, Rajeev Goré.
- Mike Gordon, "Specification and Verification I", chapters 1 and 2.
- Michael Huth, Mark Ryan, "Logic in Computer Science: Modeling and Reasoning about Systems", 2nd edition, Cambridge University Press, 2004.
- Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, "Verification of Sequential and Concurrent Programs", 3rd edition, Springer.