

# Introduction to Separation Logic

Heavily borrows from slides by Cristiano Calcagno, Imperial College  
London

October 16, 2019

# Table of Contents

- 1 Introducing Separation Logic
- 2 Relation with Pointer Logic
- 3 Inductive predicates

# Table of Contents

1 Introducing Separation Logic

2 Relation with Pointer Logic

3 Inductive predicates

# Syntax of Separation Logic

- Given a decidable base-theory  $T$ , the syntax of separation logic  $SL(T)_{Loc, Data}$  is presented
- $Loc$  and  $Data$  represent the type of the address and the values  
[Sriv: 1. Is  $T$  different from  $Loc$ ,  $Data$ ? I thought they would be part of  $T$   
2.  $Data$  can be  $Loc$  too, right otherwise you couldn't express indirection]
- $Loc$  and  $Data$  can be any sorts, but  $Loc$  should be countably infinite, for the purpose of the decision procedure
- For example,  $Loc$  and  $Data$  can be  $Int$

$$\begin{aligned} P, Q ::= & \text{false} \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \\ & \mid P * Q \mid P \multimap Q \\ & \mid E = E' \mid E \hookrightarrow E' \mid \text{empty} \end{aligned}$$

We use  $E$  and  $E'$  to denote expressions in the base theory, where heap indirection is not used. This is needed to syntactically rule out formulas

like  $F : (x \hookrightarrow v_1) = (y \hookrightarrow v_2)$  [Sriv: What are the operators allowed in  $E$ ? Obviously it can't have

$\text{dref}(*):$  can it have arithmetic over  $Loc$ ? I guess yes: Can it have  $\&$ ? I guess not?]

# Semantics of Separation Logic

The model consists of an interpretation ( $I$ ) and a heap ( $h$ )

$$I : \text{Var} \rightarrow \text{Loc}$$

$$h : \text{Loc} \rightarrow \text{Data}$$

$I, h \models \text{false}$	never satisfied
$I, h \models P \wedge Q$	$I, h \models P$ and $I, h \models Q$
$I, h \models P \vee Q$	$I, h \models P$ or $I, h \models Q$
$I, h \models P \rightarrow Q$	$I, h \models P$ implies $I, h \models Q$
$I, h \models E = E'$	$\llbracket E \rrbracket_I = \llbracket E' \rrbracket_I$

We use  $\llbracket E \rrbracket_I$ , to denote the value of  $E$  under the interpretation  $I$ . Also, the domain of  $h$  should be a finite subset of  $\text{Loc}$  and it can be a partial function.

Empty heap

$$\begin{aligned} I, h &\models \text{empty} \\ \text{iff } h &= \phi \end{aligned}$$

Separating conjunction

$$\begin{aligned} I, h &\models P * Q \\ \text{iff } \exists h_1, h_2. &(h_1 \# h_2) \wedge (h = h_1 \circ h_2) \wedge I, h_1 \models P \wedge I, h_2 \models Q \end{aligned}$$

Where  $h_1 \# h_2$  denotes that the heap domains are disjoint and  $h_1 \circ h_2$  means their union.

## Separating Implication

$$\begin{aligned} I, h &\models P \multimap Q \\ \text{iff } \forall h'. (h \# h') \wedge (I, h' &\models P) \rightarrow I, h \circ h' \models Q \end{aligned}$$

Interpretation : If we extend the current heap with a disjoint heap satisfying  $P$ , then the new heap satisfies  $Q$ . In some ways, we can imagine that our current heap is only missing the records of  $P$ , to make it satisfy  $Q$ .

Points to

$$\begin{aligned} I, h &\models E \hookrightarrow E' \\ \text{iff } h(\llbracket E \rrbracket_I) &= \llbracket E' \rrbracket_I \end{aligned}$$

# Examples

Points to,

$$F : x \hookrightarrow 10$$

$$I : \{(x, 0)\}$$

$$h : \{(0, 10)\}$$

$$I, h \models F$$

Separating conjunction,

$$F : x \hookrightarrow 10 * y \hookrightarrow 20$$

$$I : \{(x, 0), (y, 1)\}$$

$$h : \{(0, 10), (1, 20)\}$$

$$I, h \models F$$



Another example,

$$F : x \hookrightarrow y * y \hookrightarrow x$$

$$I : \{(x, 0), (y, 1)\}$$

$$h : \{(0, 1), (1, 0)\}$$

$$I, h \models F$$

## Separating Implication

$$I, h \models P \multimap Q$$
$$\text{iff } \forall h'. (h \# h') \wedge (I, h' \models P) \rightarrow I, h \circ h' \models Q$$

Example,

$$F : (x \hookrightarrow 10) \multimap (x \hookrightarrow 10 * y \hookrightarrow 20)$$

$$I : \{(x, 0), (y, 1)\}$$

$$h' : \{(0, 10)\}$$

$$h : \{(1, 20)\}$$

$$h \circ h' : \{(0, 10), (1, 20)\}$$

$$I, h \models F$$

[Srivas: Show some negative examples of  $\multimap$  formula that are not satisfiable by a  $I$  and  $H$ ; also does  $\phi \multimap \psi \models \phi \multimap \psi$ ?

# Applications

Separation logic has been useful in Hoare logic for program verification. We will have a quick look at Hoare logic.

[Srivvas: This is *strongest post condition* and must be introduced as such]

$\{P\}S\{Q\}$

$P$  : Logical assertion on states - precondition

$S$  : Code section that modifies state

$Q$  : Logical assertion on states - postcondition

Example,

$\{x = 1\}x := 2\{x = 2\}$

The programs  $S$  are defined against a language specification with imperative commands such as skip, loops, variable and pointer assignments.

# Applications

Now given a post-condition  $Q$  and program  $S$ , we want to calculate the set of states, that the program can be in before, to end-up in  $Q$  after execution.

Example,

$$\{x > 0\}x := x + 1\{x \geq 0\}$$

[Srivias: This is correct as a ordinary triple; but strongest post condition would be  $x > 0$ ]

This is also valid,

$$\{x > -1\}x := x + 1\{x \geq 0\}$$

In fact,  $(x > -1) \rightarrow (x > 0)$ , thus,  $(x > -1)$  is weaker. Finding the weakest unique precondition, can then let us reason about all the states, that the program can be in before, to guarantee  $Q$  to hold. [Srivias: This can be confusing; first introduce SP, then formally define WP in a slide after this and talk about it before you show the next to next slide]

# Applications

The first application of separation logic in Hoare-style verification is to do local reasoning, which is defined as,

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

Example,

```
{root ↦ (left, right) * tree(left) * tree(right)}  
deletetree(left)  
{root ↦ (left, right) * emp * tree(right)}  
deletetree(right)  
{root ↦ (left, right) * emp * emp}  
free(root)  
{emp * emp * emp}  
{emp}
```

# Applications

Another application is to have an operation similar to modus-ponens for heap predicates.

$$P * (P \multimap Q) \models Q$$

Example,

$$(x \hookrightarrow v_1) * (x \hookrightarrow v_1 \multimap y \hookrightarrow v_2) \models (y \hookrightarrow v_2)$$

Also, this is used in weakest pre-condition computation, where if we have a triple,

$$\{P\}x := 3\{Q\}$$

where the weakest precondition would be,

$$wp(x := 3, Q) \equiv (x \hookrightarrow -) * ((x \hookrightarrow 3) \multimap Q)$$

The trivial case being  $P = x \hookrightarrow 3$ .

# Table of Contents

1 Introducing Separation Logic

2 Relation with Pointer Logic

3 Inductive predicates

# Translating Separation Logic into Pointer Logic

Points to,

$$\begin{aligned} I, h &\models x \hookrightarrow v \\ &\iff \\ L, M &\models *x = v \end{aligned}$$

Separating conjunction,

$$\begin{aligned} I, h &\models x \hookrightarrow v_1 * y \hookrightarrow v_2 \\ &\iff \\ L, M &\models *x = v_1 \wedge *y = v_2 \wedge x \neq y \end{aligned}$$



# Table of Contents

1 Introducing Separation Logic

2 Relation with Pointer Logic

3 Inductive predicates

# Need for inductive predicates

- Most interesting data structures in programs are defined as inductive systems
- For example : linked lists, trees, graphs
- Being able to reason about these in SL is useful
- But inductive predicates introduce quantifiers

This way of specifying a list is quite cumbersome,

$$p \hookrightarrow v_1, p_1 \wedge$$

$$p_1 \hookrightarrow v_2, p_2 \wedge$$

$$p_2 \hookrightarrow v_3, p_3 \wedge$$

...

# Lists in pointer logic

First try at defining lists without explicit quantifiers. We define the following predicates for the  $i^{th}$  element of the list,

$$\text{list-elem}(p, 0) \equiv p$$

$$\text{list-elem}(p, i) \equiv * \text{list-elem}(p, i - 1)$$

In this formulation, a null-terminated list would be

$$\text{list-elem}(p, l) = \text{NULL}$$

and a cyclic list would be,

$$\text{list-elem}(p, l) = p$$

But the last predicate is also satisfied by an element with length 1.

To actually get a disjoint list with unique pointer elements, we need to add an extra constraint,

$$\text{overlap}(p, q) \equiv p = q \vee p + 1 = q \vee p = q + 1$$

$$\text{list-disjoint}(p, 0) \equiv \text{TRUE}$$

$$\text{list-disjoint}(p, l) \equiv \text{list-disjoint}(p, l - 1) \wedge$$

$$\forall 0 \leq i < l - 1. \neg \text{overlap}(\text{list-elem}(p, i), \text{list-elem}(p, l - 1))$$

The set of clauses grows quadratically upon the quantifier instantiation.

# Lists - Try 2

We now try to define a list with inductive predicates,

$$\text{list } 0 \ x \equiv x = \text{nil}$$

$$\text{list } n \ x \equiv \exists y. (x \hookrightarrow v, y) \wedge (\text{list } (n - 1) \ y)$$

This is a satisfying assignment for *list* 3 *x*,

$$l = \{ (x, 0), (y, 0), (z, 1), (w, \text{nil}) \}$$

$$h = \{ (0, (v, 1)), (1, (v, w)) \}$$

The pointers *x* and *y* got aliased to point to *z*.

$$\text{list } 3 \ x \equiv$$

$$(x = 0) \hookrightarrow v, 1 \wedge$$

$$(y = 0) \hookrightarrow v, 1 \wedge$$

$$(z = 1) \hookrightarrow v, \text{nil} \wedge$$

# Lists in Separation Logic

In separation logic, the separating conjunction takes care of ensuring the pointers don't alias.

$$\text{list } 0 \ x \equiv x = \text{nil}$$

$$\text{list } n \ x \equiv \exists y. (x \hookrightarrow v, y) * (\text{list } (n - 1) \ y)$$