Introduction to Separation Logic

Heavily borrows from slides by Cristiano Calcagno, Imperial College London

October 18, 2019

Table of Contents

Table of Contents

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 [Srivas: 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*

$$
\begin{aligned}
P, Q ::= \ & \mathit{false} \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \\
& \mid E = E' \mid E \hookrightarrow E' \mid \mathit{empty}
\end{aligned}
$$

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

$$I : \mathrm{Var} \to \mathrm{Loc}$$
$$h : \mathrm{Loc} \to \mathrm{Data}$$

$$
\begin{aligned}
I, h &\models \mathit{false} && \text{never satisfied} \\
I, h &\models P \wedge Q && I, h \models P \text{ and } I, h \models Q \\
I, h &\models P \vee Q && I, h \models P \text{ or } I, h \models Q \\
I, h &\models P \to Q && I, h \models P \text{ implies } I, h \models Q \\
I, h &\models E = E' && [\![E]\!]_I = [\![E']\!]_I
\end{aligned}
$$

We use $[\![E]\!]_I$, to denote the value of E under the interpretation I. Also, the domain of h should be a finite subset of *Loc* and it can be a partial function.

Semantics of Separation Logic:

Empty heap: $I, h \models empty \text{iff } h = \phi$

Points to: $I, h \models E \hookrightarrow E' \text{iff } h([\![E]\!]_I) = [\![E']\!]_I$

Separating conjunction: defines two separate areas of heap

$$I, h \models P * Q$$
$$\text{iff } \exists h_1, h_2.(h_1 \# h_2) \wedge (h = h1 \circ h2) \wedge I, h_1 \models P \wedge I, h_2 \models Q,$$

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

Examples

Points to: $F : x \hookrightarrow 10; \quad I : \{(x,0)\}; \quad h : \{(0,10)\}; \quad I, h \models F$

Separating conjunction,

$$F1 : x \hookrightarrow 10 * y \hookrightarrow 20$$

$$I : \{(x,0),(y,1)\}; \quad h : \{(0,10),(1,20)\}; \quad I, h \models F1$$

$$I : \{(x,0),(y,1)\}; \quad h' : \{(0,10),(1,10)\}; \quad I, h' \models F1\ ?;$$

$$\quad yes,\ if\ 10\ is\ (non\ Loc)\ Data$$

$$F2 : (x \hookrightarrow 10 \wedge 10 \hookrightarrow 1) * (y \hookrightarrow 10)$$

$$I : \{(x,0),(y,1)\}; h : \{(0,10),(1,10),(10,15)\}; I, h \models F2?$$

$$\quad No;\ no\ sharing\ allowed\ at\ any\ level$$

Examples

Another example,

$$F1 : x \hookrightarrow y * y \hookrightarrow x$$

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

$$I, h \models F1$$

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

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

$$I', h' \not\models F1$$

Separating Implication: $-\!\!*$

Separating Implication

$$
\begin{gathered}
I, h \models P \mathrel{-\!\!*} Q \\
\text{iff } \forall h'.(h \# h') \wedge (I, h' \models P \rightarrow I, h \circ h' \models Q)
\end{gathered}
$$

Intiution: Defines condition under which P is *separable* from Q

P is *separable* from Q if for every heap h' that satisfies P, we can make another heap h satisfy Q by *disjointedly extending* h by h'

Examples

Separating Implication:

$$I, h \models P \mathrel{-\!\!*} Q \iff \forall h'.(h \# h') \wedge (I, h' \models P) \rightarrow I, h \circ h' \models Q$$

Examples:

$F1 : (x \hookrightarrow 10) \mathrel{-\!\!*} (x \hookrightarrow 10 * y \hookrightarrow 20)$

$I, h \models F1$ with $I : \{(x, 0), (y, 1)\}$; $h : \{(1, 20)\}$ because:

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

$F2 : (x \hookrightarrow 10) \mathrel{-\!\!*} (x \hookrightarrow 20 * y \hookrightarrow 20)$

$I, h \not\models F2$ with $I : \{(x, 0), (y, 1)\}$; $h : \{(1, 20)\}$ because:

$h \circ h' \not\models (x \hookrightarrow 20 * y \hookrightarrow 20)$ with $h' : \{(0, 10)\}$; $h \circ h' : \{(0, 10), (1, 20)\}$

$h \models (emp \mathrel{-\!\!*} \phi)$, if $h \models \phi$; $h_2 \models (\phi_1 \mathrel{-\!\!*} \phi_2)$, if $h_1 \circ h_2 \models (\phi_1 * \phi_2)$

Table of Contents

Translating Separation Logic into Pointer Logic

Points to,

$$I, h \models x \hookrightarrow v$$
$$\iff$$
$$L, M \models \ ^*x = v$$

Separating conjunction,

$$I, h \models x \hookrightarrow v_1 * y \hookrightarrow v_2$$
$$\iff$$
$$L, M \models \ ^*x = v_1 \ \wedge \ ^*y = v_2 \wedge x \neq y$$

Table of Contents

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,

$$
\begin{aligned}
&p \hookrightarrow v_1, p_1 \wedge \\
&p_1 \hookrightarrow v_2, p_2 \wedge \\
&p_2 \hookrightarrow v_3, p_3 \wedge \\
&\ldots
\end{aligned}
$$

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,

$$\begin{aligned}\text{list-elem}(p, 0) &\equiv p \\ \text{list-elem}(p, i) &\equiv *\text{list-elem}(p, i - 1)\end{aligned}$$

In this formulation, a null-terminated list would be

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

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.

Lists in pointer logic

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

$$\begin{aligned}
&\mathrm{overlap}(p, q) \equiv p = q \vee p + 1 = q \vee p = q + 1 \\
&\mathrm{list\text{-}disjoint}(p, 0) \equiv \mathit{TRUE} \\
&\mathrm{list\text{-}disjoint}(p, l) \equiv \mathrm{list\text{-}disjoint}(p, l - 1) \wedge \\
&\forall 0 \leq i < l - 1. \neg\, \mathrm{overlap}(\mathrm{list\text{-}elem}(p, i), \mathrm{list\text{-}elem}(p, l - 1))
\end{aligned}$$

The set of clauses grows quadratically upon the quantifier instantiation.

Lists - Try 2

We now try to define a list with inductive predicates,

$$\text{list } 0\ x \equiv x = nil$$
$$\text{list } n\ x \equiv \exists y.(x \hookrightarrow v, y) \wedge (list\ (n-1)\ y)$$

This is a satisfying assignment for $list\ 3\ x$,

$$I = \{\ (x, 0), (y, 0), (z, 1), (w, nil)\ \}$$
$$h = \{\ (0, (v, 1)), (1, (v, w))\ \}$$

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

$$list\ 3\ x \equiv$$
$$(x = 0) \hookrightarrow v, 1 \wedge$$
$$(y = 0) \hookrightarrow v, 1 \wedge$$
$$(z = 1) \hookrightarrow v, nil \wedge$$

Lists in Separation Logic

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

$$\begin{aligned} \text{list } 0\ x &\equiv x = \mathit{nil} \\ \text{list } n\ x &\equiv \exists y.(x \hookrightarrow v, y) * (\mathit{list}\ (n-1)\ y) \end{aligned}$$

Applications: Hoare Logic

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

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

P : Logical assertion on states - precondition

S : Code section that modifies state

Q : Logical assertion on states - postcondition

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

Hoare Triple : holds if “whenever you start in a state that satisfies the precondition and execute the program statement S and terminate then you will be in a state that satisfies the postcondition”

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

Strongest Postcondition and Weakest Precondition

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,

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

$(x > 0)$ is the *strongest* postcondition of $(x \geq 0)$: $SP(x := x + 1, x \geq 0)$

Similarly, we can reason backwards using *weakest* precondition defining all the states that the program *should* be in before, to guarantee Q to hold.
$WP(x := x + 1, x \geq 0) = x \geq -1$

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,

$$\{\mathrm{root} \hookrightarrow (\mathrm{left}, \mathrm{right}) * \mathrm{tree}(\mathrm{left}) * \mathrm{tree}(\mathrm{right})\}$$
$$\mathrm{deletetree}(\mathrm{left})$$
$$\{\mathrm{root} \hookrightarrow (\mathrm{left}, \mathrm{right}) * \mathrm{emp} * \mathrm{tree}(\mathrm{right})\}$$
$$\mathrm{deletetree}(\mathrm{right})$$
$$\{\mathrm{root} \hookrightarrow (\mathrm{left}, \mathrm{right}) * \mathrm{emp} * \mathrm{emp}\}$$
$$\mathrm{free}(\mathrm{root})$$
$$\{\mathrm{emp} * \mathrm{emp} * \mathrm{emp}\}$$
$$\{\mathrm{emp}\}$$

Applications

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

$$P * (P \mathrel{-\!\!*} Q) \models Q$$

Example,

$$(x \hookrightarrow v_1) * (x \hookrightarrow v_1 \mathrel{-\!\!*} y \hookrightarrow v_2) \models (y \hookrightarrow v_2)$$

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,

$$wp(x := 3, Q) \equiv (x \hookrightarrow -) * ((x \hookrightarrow 3) \mathrel{-\!\!*} Q)$$

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