Separation Logic

Presenter: Sigurd Schneider

UdS, Graduate School of Computer Science
Seminar on Concurrent Program Logics

May 16, 2011

The presented material is a compilation from
O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*
Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Last Week: Rely/Guarantee

- Enables Hoare-style reasoning for parallel programs
- Allows fine-grained sharing
- Improved modularity over Owicki-Gries approach using rely and guarantee conditions to formulate sharing protocol

This Week: Separation Logic

- Enables Hoare-style reasoning about programs with dynamic resource allocation/deallocation
- Main challenges: *frame problem* and *aliasing*
- Key idea: reasoning should be kept confined to cells that are actually touched by the program; unmentioned cells remain automatically unchanged
- Supports fine-grained heap partition separation, and allows generalization of specifications to heaps with other unmentioned cells
- Due to Reynolds (1999/2000), Ishtiaq and O'Hearn (2001). Early ideas already appear in Burstall (1972).

Reversing an Imperative List

```
j:=nil
while i != nil do (
  k := [i+1];
  [i+1] := j;
  j := i;
  i := k
)
```

- i holds the adress of the current element
- [i+1] is the location of i's next pointer
- k is a temporary for current next pointer
- j holds the address of the previous element

From page 1 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Specifying the List Reversal Procedure

$$\mathsf{j} := \mathbf{nil}\,;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$$
$$(\mathsf{k} := [\mathsf{i}+1]\,;\ [\mathsf{i}+1] := \mathsf{j}\,;\ \mathsf{j} := \mathsf{i}\,;\ \mathsf{i} := \mathsf{k}).$$

Invariant: i,j represent sequences α, β s.t. rev of initial value α_0 is concat of rev α onto β:

$$\exists\alpha,\beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta,$$

where

$$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\mathrm{def}}{=} \mathsf{i} = \mathbf{nil}$$

$$\mathsf{list}(\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \stackrel{\mathrm{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\ \alpha\ \mathsf{j}$$

But this is not enough because i,j could share structure.

So disjointness must be asserted:

$$(\exists\alpha,\beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta)$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reach}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}),$$

Even worse, suppose there is some other list *x* representing a seq. γ that shall not be affected:

$$(\exists\alpha,\beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta) \wedge \mathsf{list}\ \gamma\ \mathsf{x}$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reach}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil})$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{x},\mathsf{k}) \wedge (\mathbf{reach}(\mathsf{i},\mathsf{k}) \vee \mathbf{reach}(\mathsf{j},\mathsf{k}))$$
$$\Rightarrow \mathsf{k} = \mathbf{nil})$$

From page 1/2 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Overview

Syntax of a Simple Programming Language

$\langle comm \rangle ::=$	commands
$\mid \ \ldots$	usual cmds + exps
$\mid \ \langle var \rangle := \mathbf{cons}(\langle exp \rangle, \ldots, \langle exp \rangle)$	allocation
$\mid \ \langle var \rangle := [\langle exp \rangle]$	lookup
$\mid \ [\langle exp \rangle] := \langle exp \rangle$	mutation
$\mid \ \mathbf{dispose}\ \langle exp \rangle$	deallocation

- Low level programming language
- There are *three* different forms of assignment
- Features a command for deallocation

From page 2 Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Denotational Semantics for Expressions

$$\begin{aligned}\text{Values} &\overset{\text{def}}{=} \mathbb{Z} \\ \text{Atoms} \uplus \text{Addresses} &\subseteq \mathbb{Z} \\ \text{Heaps} &\overset{\text{def}}{=} \text{Addresses} \overset{\textit{fin}}{\rightharpoonup} \text{Values}\end{aligned}$$

$$\begin{aligned}\textbf{nil} &\in \text{Atoms} \\ \text{Variables} &\overset{\text{def}}{=} \{x, y, \ldots\} \\ \text{Stores} &\overset{\text{def}}{=} \text{Variables} \overset{\textit{fin}}{\rightharpoonup} \text{Values} \\ \text{States} &\overset{\text{def}}{=} \text{Stores} \times \text{Heaps}\end{aligned}$$

- Records of arbitrary size provided by assumption
- Denotation of expressions $[\![e]\!]$, and boolean expressions $[\![b]\!]$ as usual mapping to store-taking functions
- Expressions are side-effect free (that's why there are several different assignment statements)

From page 3 O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Small Step Semantics for Commands

$$\textsc{Allocation}$$
$$\langle v := \mathbf{cons}(e_1, \ldots, e_n), (s, h)\rangle \rightsquigarrow ([s \mid v : l], [h \underset{i \in [1,n]}{\mid} l + i - 1 : [\![e_i]\!]s])$$

$$\textsc{Lookup}$$
$$\frac{}{\langle v := [e], (s, h)\rangle \rightsquigarrow ([s \mid v : h([\![e]\!]s)], h)}\ [\![e]\!]s \in dom\, h$$

$$\textsc{Mutation}$$
$$\frac{}{\langle [e] := e', (s, h)\rangle \rightsquigarrow (s, [h \mid [\![e]\!]s \mapsto [\![e']\!]s])}\ [\![e]\!]s \in dom\, h$$

$$\textsc{Deallocation}$$
$$\frac{}{\langle \mathbf{dispose}\ e, (s, h)\rangle \rightsquigarrow (s, h\rceil(dom(h) \setminus \{[\![e]\!]s\}))}\ [\![e]\!]s \in dom\, h$$

- Non-determinism in Allocation.
- If $[\![e]\!]s \notin dom\, h$ for any command, then it reduces to **abort**.

From page 3 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Modelling Memory Faults

- **abort** is an error state indicating a memory fault
- For heaps h_0, h_1 we write $h_0 \perp h_1$ if $dom\, h_0 \cap dom\, h_1 = \emptyset$, and $h_0 \cdot h_1$ for their union.
- Let $h_0 \subseteq h$:
  - If $\langle c, \langle s, h \rangle \rangle \rightsquigarrow^* \textbf{abort}$ then $\langle c, \langle s, h_0 \rangle \rangle \rightsquigarrow^* \textbf{abort}$. (The contraposition is also interesting!)
  - If $\langle c, \langle s, h \rangle \rangle \rightsquigarrow^* \langle s', h' \rangle$ then either $\langle c, \langle s, h_0 \rangle \rangle \rightsquigarrow^* \textbf{abort}$ or $\langle c, \langle s, h_0 \rangle \rangle \rightsquigarrow^* \langle s', h'_0 \rangle$ where $h'_0 \perp h_1$ and $h' = h'_0 \perp h_1$ (he seemed to have assumed that $h = h_0 \cdot h_1$ and $h_0 \perp h_1$).
  - If $\langle c, \langle s, h \rangle \rangle \uparrow$ then either $\langle c, \langle s, h_0 \rangle \rangle \rightsquigarrow^* \textbf{abort}$ or $\langle c, \langle s, h_0 \rangle \rangle \uparrow$.

From page 4 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

The Separation Logic Assertion Language

$\langle \mathit{assert} \rangle ::=$	assertions
$\mid \ \ldots$	connectives and quantifiers
$\mid$ **emp**	empty heap
$\mid \langle \mathit{exp} \rangle \mapsto \langle \mathit{exp} \rangle$	singleton heap
$\mid \langle \mathit{assert} \rangle * \langle \mathit{assert} \rangle$	separating conjunction
$\mid \langle \mathit{assert} \rangle \mathrel{-\!\!*} \langle \mathit{assert} \rangle$	separating implication

- Idea for $*$ and $\mathrel{-\!\!*}$ from Logic of Bunched Implications*
- Features a command for deallocation (As far as I know this is not present in the intuitionistic version)

$\star$ O'Hearn and Pym, *The Logic of Bunched Implications*

-2mm From page 4 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

A Satisfaction Judgement for Separation Assertions

A satisfaction judgement $s, h \models p$ is defined

$$
\begin{aligned}
s, h \models \mathbf{emp} &\iff \mathit{dom}\, h = \emptyset \\
s, h \models e \mapsto e' &\iff \mathit{dom}\, h = \{[\![e]\!]\, s\} \text{ and } h([\![e]\!] s) = [\![e']\!]\, s \\
s, h \models p_0 * p_1 &\iff \exists h_0, h_1 . h_0 \perp h_1 \wedge h_0 \cdot h_1 = h \wedge [\![p_0]\!]\, s\, h_0 \wedge [\![p_1]\!]\, s\, h_1 \\
s, h \models p_0 \mathrel{-\!\!*} p_1 &\iff \forall h_0 . (h_0 \perp h \wedge s, h_0 \models p_0) \Rightarrow s, h_0 \cdot h \models p_1 \\
s, h \models \forall x . p &\iff \forall v \in \text{Values}. [s \mid x \mapsto v], h \models p
\end{aligned}
$$

- Propositional constants and connectives as usual
- Note that $\mapsto$ admits no other cells in the heap
- Note non-determinism in definition of $*$
- Quantification is available only over Values

From page 4 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Useful Abbreviations

$$e \doteq e' \overset{\text{def}}{=} (e = e') \land \textbf{emp}$$

$$e \mapsto - \overset{\text{def}}{=} \exists x. e \mapsto x \qquad x \text{ not free in } e$$

$$e \hookrightarrow e' \overset{\text{def}}{=} e \mapsto e' * \textbf{true}$$

$$e \mapsto e_1, \ldots, e_n \overset{\text{def}}{=} e \mapsto e_1 * \ldots * e + n - 1 \mapsto e_n$$

$$e \hookrightarrow e_1, \ldots, e_n \overset{\text{def}}{=} e \hookrightarrow e_1 * \ldots * e + n - 1 \hookrightarrow e_n$$

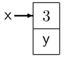
$$\Leftrightarrow e \hookrightarrow e_1, \ldots, e_n * \textbf{true}$$

- Note that $\hookrightarrow$ permits other things to be in a disjoint fragment of the heap (but $\mapsto$ does not).

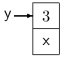
From Page 4 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*
and Page 4 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

An Overview of Sharing Patterns

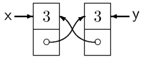
1. $\mathsf{x} \mapsto 3, \mathsf{y}$ asserts that x points to an adjacent pair of cells containing 3 and y (i.e., the store maps x and y into some values α and β, α is a address, and the heap maps α into 3 and $\alpha + 1$ into β):



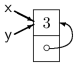
2. $\mathsf{y} \mapsto 3, \mathsf{x}$ asserts that y points to an adjacent pair of cells containing 3 and x:



3. $\mathsf{x} \mapsto 3, \mathsf{y} * \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for separate parts of the heap:



4. $\mathsf{x} \mapsto 3, \mathsf{y} \wedge \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same:



5. $\mathsf{x} \hookrightarrow 3, \mathsf{y} \wedge \mathsf{y} \hookrightarrow 3, \mathsf{x}$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

From page 4/5 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

More Examples

There are also simple examples that reveal the occasionally surprising behavior of separating conjunction. Suppose $s\,\mathsf{x}$ and $s\,\mathsf{y}$ are distinct addresses, so that

$$h_1 = \{\langle s\,\mathsf{x}, 1\rangle\} \qquad \text{and} \qquad h_2 = \{\langle s\,\mathsf{y}, 2\rangle\}$$

are heaps with disjoint singleton domains. Then

If p is:	then $[\![p]\!]_{\text{asrt}} s\,h$ is:
$\mathsf{x} \mapsto 1$	$h = h_1$
$\mathsf{y} \mapsto 2$	$h = h_2$
$\mathsf{x} \mapsto 1 \ast \mathsf{y} \mapsto 2$	$h = h_1 \cdot h_2$
$\mathsf{x} \mapsto 1 \ast \mathsf{x} \mapsto 1$	**false**
$\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2$	$h = h_1$ or $h = h_2$
$\mathsf{x} \mapsto 1 \ast (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	$h = h_1 \cdot h_2$
$(\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2) \ast (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	$h = h_1 \cdot h_2$
$\mathsf{x} \mapsto 1 \ast \mathsf{y} \mapsto 2 \ast (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	**false**
$\mathsf{x} \mapsto 1 \ast \textbf{true}$	$h_1 \subseteq h$
$\mathsf{x} \mapsto 1 \ast \neg\, \mathsf{x} \mapsto 1$	$h_1 \subseteq h.$

To illustrate separating implication, suppose that an assertion p holds for a store that maps the variable x into an address α and a heap h that maps α into 16. Then

$$(\mathsf{x} \mapsto 16) -\!\!\ast\; p$$

holds for the same store and the heap $h\rceil(\operatorname{dom} h - \{\alpha\})$ obtained by removing α from the domain of h, and

$$\mathsf{x} \mapsto 9 \ast ((\mathsf{x} \mapsto 16) -\!\!\ast\; p)$$

holds for the same store and the heap $[\,h \mid \alpha\colon 9\,]$ that differs from h by mapping α into 9. (Thus, anticipating the concept of partial-correctness specification introduced in the next section, $\{\mathsf{x} \mapsto 9 \ast ((\mathsf{x} \mapsto 16) -\!\!\ast\; p)\}\ [\mathsf{x}] := 16\ \{p\}$.)

From page 5 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Behaviour of Separating Conjunction

$p_1 * p_2 \iff p_2 * p_1$	commutative
$(p_1 * p_2) * p_3 \iff p_1 * (p_2 * p_3)$	associative
$p * \mathbf{emp} \iff p$	neutral element
$(p_1 \vee p_2) * q \iff (p_1 * q) \vee (p_2 * q)$	distributes over $\vee$
$(p_1 \wedge p_2) * q \implies (p_1 * q) \wedge (p_2 * q)$	
$(\exists x.p) * q \iff \exists x.(p * q)$	x not free in q
$(\forall x.p) * q \implies \forall x.(p * q)$	x not free in q
$(p \doteq q) * r \iff (p = q) \wedge r$	

- Why are the two missing directions unsound?

From Page 5 Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Non-Behaviour of Separating Conjunction

To see the **unsoundness** of

$$p \implies p * p \qquad \text{contraction}$$
$$p \impliedby q * p \qquad \text{weakening}$$

consider a heap $\{(sx,1),(sy,2)\}$ where $sx \neq sy$, and let $p = x \mapsto 1$ and $q = y \mapsto 2$. To see the **unsoundness** of

$$(p_1 \wedge p_2) * q \impliedby (p_1 * q) \wedge (p_2 * q)$$
$$(\forall x.p) * q \impliedby \forall x.(p * q) \qquad x \text{ not free in } q$$

consider a heap $\{(sx,1),(sy,2),(sz,3)\}$, sx, sy, sz p.w. distinct, and let $q = \exists a.a \hookrightarrow 1 \vee \exists a.a \hookrightarrow 3$, $p_1 = \exists a.a \hookrightarrow 2 \wedge \exists a.a \hookrightarrow 3$, and $p_2 = \exists a.a \hookrightarrow 2 \wedge \exists a.a \hookrightarrow 1$. (And similarly for $\forall$).

Interaction of Implication, Separating Conjunction, and Separating Implication

Separating Conjunction distributes over Implication:

$$\frac{p_1 \Rightarrow p_2 \quad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2}$$

Relationship between separating implication and implication:

$$p_1 * p_2 \Rightarrow p_3 \iff p_1 \Rightarrow (p_2 -\!\!* p_3)$$

The following implication is used throughout the paper:

$$p * (p -\!\!* q) \Rightarrow q$$

From Page 5 Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Pure, Intuitionistic, and Exact Assumptions

An assertion is

pure if it is independent from the heap (for every store)
$\wedge$ and $*$ coincide

intutitionistic if it is invariant under heap-extension
$(p * \mathbf{true}) \Rightarrow p$ and $p \Rightarrow (\mathbf{true} -\!\!* p)$

strictly exact if any two heaps satisfying the assertion (w.r.t the same store) are identical, i.e. for strictly exact q:
$(q * \mathbf{true}) \wedge p \Rightarrow q * (q -\!\!* p)$

domain exact if any two heaps satisfying the assertion (w.r.t the same store) have the same domain
semi-distributive laws become distributive laws

The classes are defined *syntactically* based on the contained connectives.

From page 5/6 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Glueing together Commands and Assertions

$\langle \mathit{spec} \rangle ::=$	specifications
$\mid \{\langle \mathit{assert} \rangle\}\langle \mathit{comm} \rangle\{\langle \mathit{assert} \rangle\}$	partial
$\mid [\langle \mathit{assert} \rangle]\langle \mathit{comm} \rangle[\langle \mathit{assert} \rangle]$	total

Let V be the free variables in p, c, q

- $\{p\}c\{q\}$ holds, iff $\forall (s, h) \in \text{States}. s, h \models p$ implies
  - it is not the case that $\langle c, (s, h)\rangle \rightsquigarrow^* \textbf{abort}$, and
  - forall $(s', h') \in \text{States}$ such that $\langle c, (s, h)\rangle \rightsquigarrow^* (s', h')$ we have $s', h' \models q$
- $[p]c[q]$ additionally demands termination
- **abort** ensures that c must mention every cell it uses

From page 6 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Small Axioms

These axioms suffice to get all others using the structural rules

$$\{e \mapsto -\}[e] := e'\{e \mapsto e'\}$$

$$\{e \mapsto -\}\textbf{dispose } e\{\textbf{emp}\}$$

$$\{x \doteq m\}x := \textbf{cons}(e_1, \ldots, e_k)\{x \mapsto e_1[m/x], \ldots, E_k[m/x]\}$$

$$\{x \doteq n\}x := e\{x \doteq e[n/x]\}$$

$$\{e \mapsto n \wedge x = m\}x := [e]\{x = n \wedge e[m/x] \mapsto n\}$$

- x, m, n are assumed to be distinct variables

From page 5 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Consequence and Auxilliary Variable Elimination

$$\textsc{AuxVarE}\quad \frac{\{p\}\mathrm{c}\{q\}}{\{\exists x.p\}\mathrm{c}\{\exists x.q\}}\ x \notin \mathrm{Free}(c)$$

$$\textsc{Consequence}\quad \frac{\mathrm{p'}\Rightarrow\mathrm{p} \quad \{p\}\mathrm{c}\{q\} \quad \mathrm{q}\Rightarrow\mathrm{q'}}{\{p'\}\mathrm{c}\{q'\}}$$

From page 6 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Variable Substitution

$$\textsc{VarSubst}\ \frac{\{p\}c\{q\}}{(\{p\}c\{q\})[e_1/x_1, \ldots, e_k/x_k]}\ \star$$

$\star$ $\{x_1, \ldots, x_k\} \supseteq \text{Free}(p, c, q)$ and if $x_i \in \text{Modifies}(c)$ then e_i is a variable not free in any other e_j.

The restrictions on this rule are needed to avoid aliasing. For example, in

$$\{x = y\}\ x := x + y\ \{x = 2 \times y\},$$

one can substitute $x \to z$, $y \to 2 \times w - 1$ to infer

$$\{z = 2 \times w - 1\}\ z := z + 2 \times w - 1\ \{z = 2 \times (2 \times w - 1)\}.$$

But one cannot substitute $x \to z$, $y \to 2 \times z - 1$ to infer the invalid

$$\{z = 2 \times z - 1\}\ z := z + 2 \times z - 1\ \{z = 2 \times (2 \times z - 1)\}.$$

The substitution rule will become important when we consider procedures.

From page 78 of Reynolds, *Introduction to Separation Logic, Lecture Notes*

The Frame Rule

$$\textsc{FrameRule}\quad \frac{\{p\}\mathrm{c}\{q\}}{\{p * r\}\mathrm{c}\{q * r\}}\ \mathrm{Modifies}(c) \cap \mathrm{Free}(r) = \emptyset$$

- p specifies an area of storage + logical properties sufficient for c to run and (if it terminates) establish q
- makes up for the promise of the first slide
- *tightness*: every cell used by c must be allocated or asserted to be active in p (enforced by semantics)
- *locality*: p only asserts cells actually used by c (programmers burden)

From page 6 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Simple Example

$$\textsc{FrameRule}\quad \frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}\ \mathsf{Modifies}(c) \cap \mathsf{Free}(r) = \emptyset$$

As a warming-up example, using the Frame Rule we can prove that assigning to the first component of a binary cons cell does not affect the second component.

$$\dfrac{\dfrac{\{x \mapsto a\}\,[x] := b\,\{x \mapsto b\}}{\{(x \mapsto a) * (x + 1 \mapsto c)\}\,[x] := b\,\{(x \mapsto b) * (x + 1 \mapsto c)\}}\ \text{Frame}}{\{x \mapsto a, c\}\,[x] := b\,\{x \mapsto b, c\}}\ \text{Syntactic Sugar}$$

The overlap of free variables between $x + 1 \mapsto c$ and $[x] := b$ is allowed here because $\text{Modifies}([x] := b) = \{\}$.

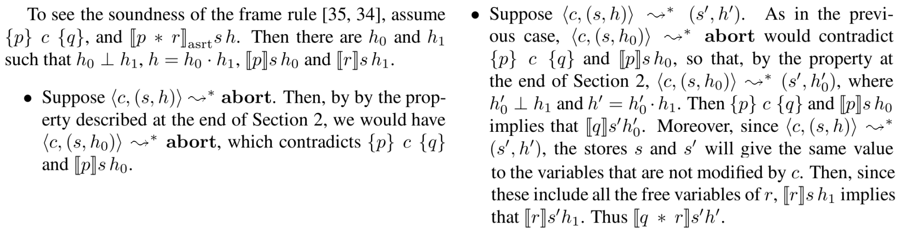
From page 7 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

The General Frame Problem

- "To most AI researchers, the frame problem is the challenge of representing the effects of action in logic without having to represent explicitly a large number of intuitively obvious non-effects."
- "epistemological issue[...:]Is it possible, in principle, to limit the scope of the reasoning required to derive the consequences of an action?"
- "And, more generally, how do we account for our apparent ability to make decisions on the basis only of what is relevant to an ongoing situation without having explicitly to consider all that is not relevant?"

From Stanford Enceclopedia of Philosopy, `http://plato.stanford.edu/entries/frame-problem/`

Soundness argument for the frame rule



To see the soundness of the frame rule [35, 34], assume $\{p\}\ c\ \{q\}$, and $[\![p\ *\ r]\!]_{\mathrm{asrt}} s\, h$. Then there are h_0 and h_1 such that $h_0 \perp h_1$, $h = h_0 \cdot h_1$, $[\![p]\!] s\, h_0$ and $[\![r]\!] s\, h_1$.

- Suppose $\langle c, (s, h)\rangle \rightsquigarrow^* \mathbf{abort}$. Then, by by the property described at the end of Section 2, we would have $\langle c, (s, h_0)\rangle \rightsquigarrow^* \mathbf{abort}$, which contradicts $\{p\}\ c\ \{q\}$ and $[\![p]\!] s\, h_0$.
- Suppose $\langle c, (s, h)\rangle \rightsquigarrow^* (s', h')$. As in the previous case, $\langle c, (s, h_0)\rangle \rightsquigarrow^* \mathbf{abort}$ would contradict $\{p\}\ c\ \{q\}$ and $[\![p]\!] s\, h_0$, so that, by the property at the end of Section 2, $\langle c, (s, h_0)\rangle \rightsquigarrow^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$. Then $\{p\}\ c\ \{q\}$ and $[\![p]\!] s\, h_0$ implies that $[\![q]\!] s' h'_0$. Moreover, since $\langle c, (s, h)\rangle \rightsquigarrow^* (s', h')$, the stores s and s' will give the same value to the variables that are not modified by c. Then, since these include all the free variables of r, $[\![r]\!] s\, h_1$ implies that $[\![r]\!] s' h_1$. Thus $[\![q\ *\ r]\!] s' h'$.

- Note crucial use of **abort**

From page 7 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Completeness of the frame rule

Yang[⋆] showed the frame rule is complete in the following sense:

- If all we know about a command c is that $\{p\}c\{q\}$ is valid,
- and if the validity $\{p'\}c\{q'\}$ is a semantic consequence of this knowledge,
- then $\{p'\}c\{q'\}$ is derivable using the four structural rules.

From page 8 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

⋆ Yang, *Local Reasoning for Stateful Programs*, Yang and O'Hearn, *A semantic basis for local reasoning*

Derived Laws

- Frame rule yields immediatelly
  $\{(E \mapsto -) * R\}\, [E] := F\, \{(E \mapsto F) * R\}$
- Same for cons:

$$\cfrac{\cfrac{\{\mathtt{emp}\}\, x := \mathtt{cons}(E_1, \ldots, E_k)\, \{x \mapsto E_1, \ldots, E_k\}}{\{\mathtt{emp} * R\}\, x := \mathtt{cons}(E_1, \ldots, E_k)\, \{(x \mapsto E_1, \ldots, E_k) * R\}}\ \text{Frame}}{\{R\}\, x := \mathtt{cons}(E_1, \ldots, E_k)\, \{(x \mapsto E_1, \ldots, E_k) * R\}}\ \text{Consequence}$$

From Page 8 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Weakest Preconditions

- Using adjunction in consequence step:

$$\cfrac{\cfrac{\{E \mapsto -\}\,[E] := F\,\{E \mapsto F\}}{\{(E \mapsto -) * ((E \mapsto F) -\!\!* Q)\}\,[E] := F\,\{(E \mapsto F) * ((E \mapsto F) -\!\!* Q)\}}\ \text{Frame}}{\{(E \mapsto -) * ((E \mapsto F) -\!\!* Q)\}\,[E] := F\,\{Q\}}\ \text{Consequence}$$

- Note that the modifies clause of **dispose** *E* is empty:

$$\cfrac{\cfrac{\{E \mapsto -\}\,\mathtt{dispose}(E)\,\{\mathtt{emp}\}}{\{(E \mapsto -) * R\}\,\mathtt{dispose}(E)\,\{\mathtt{emp} * R\}}\ \text{Frame}}{\{(E \mapsto -) * R\}\,\mathtt{dispose}(E)\,\{R\}}\ \text{Consequence}$$

- If $x \notin \mathsf{Free}(E, P, x)$:

  $\{P[E/x]\}x := E\{P\}$

  $\{\exists n. (\mathtt{true} * E \mapsto n) \wedge P[n/x]\}x := [E]\{P\}$

- If $x \notin \mathsf{Free}(E, R)$ and $y \notin \mathsf{Free}(E)$:

  $\{(E \mapsto y) * R\}\, x := [E]\, \{(E \mapsto x) * R[x/y]\}$

From Page 9 of O'Hearn, Reynolds, and Yang, *Local Reasoning about Programs that Alter Data Structures*

Revisiting the Initial Example

Consider the list reversal program:

$$\mathsf{j} := \mathbf{nil}\,;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$$
$$(\mathsf{k} := [\mathsf{i}+1]\,;\ [\mathsf{i}+1] := \mathsf{j}\,;\ \mathsf{j} := \mathsf{i}\,;\ \mathsf{i} := \mathsf{k}).$$

Using separation logic we can define

$$\mathsf{list}\ \epsilon\ (\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{list}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{k}) \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{list}\ \alpha\ (\mathsf{j},\mathsf{k})$$

and prove

$$\mathsf{list}\ \mathsf{a}\ (\mathsf{i},\mathsf{j}) \Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j}$$

$$\mathsf{list}\ \alpha{\cdot}\beta\ (\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{list}\ \beta\ (\mathsf{j},\mathsf{k})$$

$$\mathsf{list}\ \alpha{\cdot}\mathsf{b}\ (\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{b},\mathsf{k}.$$

So whenever it holds that list $(\alpha_1 \cdot \ldots \cdot \alpha_n)(i_0, i_n)$ we have

$$\exists \mathsf{i}_1, \ldots \mathsf{i}_{n-1}.$$
$$(\mathsf{i}_0 \mapsto \alpha_1, \mathsf{i}_1)\ *\ (\mathsf{i}_1 \mapsto \alpha_2, \mathsf{i}_2)\ * \cdots *\ (\mathsf{i}_{n-1} \mapsto \alpha_n, \mathsf{i}_n)$$

Thus $i_0, \ldots, i_{n-1}$ are p.w. distinct, but i_n is not constrained, so it may hold list $(\alpha_1 \cdot \ldots \cdot \alpha_n)(i, i)$ for any $n \geq 0$. I.e. the following does not define emptiness:

$$\mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j}) \Rightarrow (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil}))$$

$$\mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j}) \Rightarrow (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon)$$

But these implications hold:

$$\mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \hookrightarrow - \Rightarrow (\mathsf{i} = \mathsf{j} \Leftrightarrow \alpha = \epsilon)$$

$$\mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{list}\ \beta\ (\mathsf{j},\mathbf{nil}) \Rightarrow (\mathsf{i} = \mathsf{j} \Leftrightarrow \alpha = \epsilon)$$

From page 9/10 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Revisiting the Initial Example

$\{\mathbf{list}\ \mathsf{a}\cdot\alpha\,(\mathsf{i},\mathsf{k})\}$

$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j} * \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\{\mathsf{i} \mapsto \mathsf{a} * \exists \mathsf{j}.\ \mathsf{i}+1 \mapsto \mathsf{j} * \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\mathsf{j} := [\mathsf{i}+1]\ ;$

$\{\mathsf{i} \mapsto \mathsf{a} * \mathsf{i}+1 \mapsto \mathsf{j} * \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\mathbf{dispose}\ \mathsf{i}\ ;$

$\{\mathsf{i}+1 \mapsto \mathsf{j} * \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\mathbf{dispose}\ \mathsf{i}+1\ ;$

$\{\mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\mathsf{i} := \mathsf{j}$

$\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{k})\}$

Example of a command that deletes the head of the list.

$$\mathsf{list}\ \epsilon\,(\mathsf{i},\mathsf{j}) \overset{\mathrm{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{list}\ \mathsf{a}\cdot\alpha\,(\mathsf{i},\mathsf{k}) \overset{\mathrm{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j} * \mathsf{list}\ \alpha\,(\mathsf{j},\mathsf{k})$$

$$e \mapsto e_1, e_2 \overset{\mathrm{def}}{=} e \mapsto e_1 * e+1 \mapsto e_2$$

Lookup (backwards reasoning)

$$\frac{}{\{\exists v'.\ (e \mapsto v') * ((e \mapsto v') -\!\!* \ p')\}\ v := [e]\ \{p\}}$$

where v' is not free in e, nor free in p unless it is v.

$$\{e \mapsto n \wedge x = m\} x := [e]$$
$$\{x = n \wedge e[m/x] \mapsto n\}$$

From page 10 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Revisiting the Initial Example

$$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta \wedge \mathsf{i} \neq \mathbf{nil}\}$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathbf{list}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$\{\exists \mathsf{a}, \alpha, \beta, \mathsf{k}.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$\mathsf{k} := [\mathsf{i} + 1]\ ;$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$[\mathsf{i} + 1] := \mathsf{j}\ ;$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = (\mathsf{a}{\cdot}\alpha)^\dagger{\cdot}\beta\}$$
$$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \mathsf{a}{\cdot}\beta\ (\mathsf{i}, \mathbf{nil}))$$
$$\quad \wedge\ \alpha_0^\dagger = \alpha^\dagger{\cdot}\mathsf{a}{\cdot}\beta\}$$
$$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{i}, \mathbf{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}$$
$$\mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}$$
$$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\ (\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\ (\mathsf{j}, \mathbf{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger{\cdot}\beta\}.$$

Example: Body of the while loop in the list reversal procedure

$$\mathsf{j} := \mathbf{nil}\ ;\ \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$$
$$\qquad (\mathsf{k} := [\mathsf{i} + 1]\ ;\ [\mathsf{i} + 1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).$$

$$\mathsf{list}\ \epsilon\ (\mathsf{i}, \mathsf{j}) \overset{\mathrm{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{list}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{k}) \overset{\mathrm{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ (\mathsf{j}, \mathsf{k})$$

The first lookup step is justified by AuxVarE and Lookup.

From page 10 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

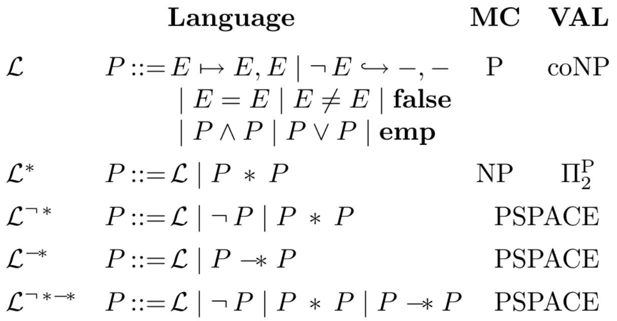
Computability

- Existence of weakest preconditions for each of the axioms ensures that verification conditions can be derived which ensure the original specification (why are weakest preconditions needed)
- Undecidable, even without arithmetic and without **emp**, $\mapsto$, $*$, $-\!\!*$ (but not $e \hookrightarrow$); the idea is to remove separation
- On the other hand, if quantification is omitted, validity is decidable, but NP or worse (without address arithmetic)
  - **MC**: model checking, i.e. $s, h \models p$ for specific s, h
  - **VAL**: model checking, i.e. $s, h \models p$ for all s, h

From page 15 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Complexity

The results displayed below are Calcagno and Yang; The language does neither allow quantification nor address arithmetic (or predicates)



	Language	MC	VAL
$\mathcal{L}$	$P ::= E \mapsto E, E \mid \neg\, E \hookrightarrow -, -$ $\mid E = E \mid E \neq E \mid \mathbf{false}$ $\mid P \wedge P \mid P \vee P \mid \mathbf{emp}$	P	coNP
$\mathcal{L}^{*}$	$P ::= \mathcal{L} \mid P * P$	NP	Π_2^{P}
$\mathcal{L}^{\neg *}$	$P ::= \mathcal{L} \mid \neg\, P \mid P * P$	PSPACE	PSPACE
$\mathcal{L}^{-\!*}$	$P ::= \mathcal{L} \mid P -\!\!* P$	PSPACE	PSPACE
$\mathcal{L}^{\neg * -\!*}$	$P ::= \mathcal{L} \mid \neg\, P \mid P * P \mid P -\!\!* P$	PSPACE	PSPACE

From page 15 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Future Directions

- New forms of inference: Separation Logic is not only theoretically incomplete, but also practically (proofs must rely on semantic arguments, i.e. a rule is missing)
- Address arithmetic: *aliasing* is an undecidable sub-problem
- Concurrency: priciple of ownership; another problem are systems that cannot be specified using input/output relation (such as reactive systems)
- Storage allocation and garbage collection
- Relationship to type systems
- Embedded code pointers

From page 16-18 of Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*

Questions?

Thank you for listening!