CS 565

Programming Languages (graduate)
Spring 2026

Week 12

Separation Logic

Background

DOI:10.1145/3211968

Separation logic is a key development in formal reasoning about programs, opening up new lines of attack on longstanding problems.

BY PETER O'HEARN

Separation Logic

A FUNDAMENTAL TECHNIQUE in reasoning about programs is the use of logical assertions to describe properties of program states. Turing used assertions to argue about the correctness of a particular program in 1949,[40] and they were incorporated into general formal systems for program proving starting with the work of Floyd[21] and Hoare[22] in the 1960s. Hoare logic, which separation logic builds upon, is a formal system for proving specifications of the form

$$\{precondition\}code\ \{postcondition\}$$

where the precondition and postcondition are vassertions describing properties of the input and output states. For example,

$$\{x == N\}code\ \{x == N \wedge y == N!\}$$

can serve as a specification of an imperative program that computes the factorial of the value held in variable x and places it in y.

Hoare logic and related systems worked very well for programs manipulating simple primitive data types such as for integers or strings, but proofs became more complex when dealing with structured data containing embedded pointers. One of the founding papers of separation logic summarized the problem as follows.[32]

"The main difficulty is not one of finding an in-principle adequate axiomatization of pointer operations; rather there is a mismatch between simple intuitions about the way that pointer operations work and the complexity of their axiomatic treatments....when there is aliasing, arising from several pointers to a given cell, an alteration to a cell may affect the values of many syntactically unrelated expressions."

Bornat provided a good description of the struggles in reasoning about mutable data structures up to 2000.[6]

In joint work with John Reynolds and others we developed separation logic (SL) to address the fundamental problem of reasoning about programs that mutate data structures. From a special logic for heaps, it gradually evolved into a general theory for modular reasoning about concurrent as well as sequential programs. Efforts by many researchers established that the logic provides a basis for efficient proof search in automatic and semi-automatic proof tools, for example, giving rise to the Infer static analyzer, a tool that is in deployment at Facebook where it catches thousands of bugs per month before code reaches production in products used daily by over one billion people.

Separation logic is an extension of Hoare logic, which employs novel logical operators, most importantly the separating conjunction * (pronounced "and

» key insights

- **Separation logic supports in-place updating of facts as we reason, in a way that mirrors in-place update of memory during execution, and this leads to logical proofs about imperative programs that match computational intuition.**
- **Separation logic supports scalable reasoning by using an inference rule (the frame rule) that allows a proof to be localized to the resources that a program component accesses (its footprint).**
- **Concurrent separation logic shows that modular reasoning about threads that share storage and other resources is possible.**

Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap, and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

1. Introduction

The use of shared mutable data structures, i.e., of structures where an updatable field can be referenced from more than one point, is widespread in areas as diverse as systems programming and artificial intelligence. Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size. (A partial bibliography is given in Reference [28].)

The problem faced by these approaches is that the correctness of a program that mutates data structures usually depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

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

(Here the notation $[e]$ denotes the contents of the storage at address e.)

The invariant of this program must state that i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of 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 the predicate **list** α i is defined by induction on the length of α:

$$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathsf{i} = \mathbf{nil} \qquad \mathsf{list}(\mathsf{a}\cdot\alpha)\ \mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\ \alpha\ \mathsf{j}$$

(and $\hookrightarrow$ can be read as "points to").

Unfortunately, however, this is not enough, since the program will malfunction if there is any sharing between the lists i and j. To prohibit this we must extend the invariant to assert that only **nil** is reachable from both i and j:

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta) \wedge (\forall \mathsf{k}.\ \mathbf{reach}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reach}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}), \quad (1)$$

where

$$\mathbf{reach}(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \exists n \geq 0.\ \mathbf{reach}_n(\mathsf{i},\mathsf{j})$$
$$\mathbf{reach}_0(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \mathsf{i} = \mathsf{j}$$
$$\mathbf{reach}_{n+1}(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \exists \mathsf{a}, \mathsf{k}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{k} \wedge \mathbf{reach}_n(\mathsf{k},\mathsf{j}).$$

Even worse, suppose there is some other list x, representing a sequence γ, that is not supposed to be affected by

*Portions of the author's own research described in this survey were supported by National Science Foundation Grant CCR-9804014, and by the Basic Research in Computer Science (http://www.brics.dk/) Centre of the Danish National Research Foundation.

Local Reasoning about Programs that Alter Data Structures

Peter O'Hearn[1], John Reynolds[2], and Hongseok Yang[3]

[1] Queen Mary, University of London
[2] Carnegie Mellon University
[3] University of Birmingham and University of Illinois at Urbana-Champaign

Abstract. We describe an extension of Hoare's logic for reasoning about programs that alter data structures. We consider a low-level storage model based on a heap with associated lookup, update, allocation and deallocation operations, and unrestricted address arithmetic. The assertion language is based on a possible worlds model of the logic of bunched implications, and includes spatial conjunction and implication connectives alongside those of classical logic. Heap operations are axiomatized using what we call the "small axioms", each of which mentions only those cells accessed by a particular command. Through these and a number of examples we show that the formalism supports local reasoning: A specification and proof can concentrate on only those cells in memory that a program accesses.

This paper builds on earlier work by Burstall, Reynolds, Ishtiaq and O'Hearn on reasoning about data structures.

Motivation

- Hoare Logic is defined in terms of assertions on states:
  - states are maps from variables to their values
  - most programming languages also support the notion of a heap:
    - variables map to addresses
    - the contents at a given address can be shared and aliased
  - Embedding notions of sharing and mutation into the logic is problematic
- Separation Logic enables local reasoning about memory
  - It is a *substructural* logic that controls how memory (heaps) are constructed and used
  - In classical logical systems (e.g., Hoare logic) can:
    - add (weaken) or contract (duplicate) assumptions. Here, think of assumptions as claims we can make about resources (aka states or memory)
    - Substructural logics restrict how assumptions can be introduced:
      - can't invent extra memory to satisfy predicates
      - can't duplicate memory

Substructural Logic

Classical logic permits three kinds of 'structural' rules:

- *Weakening*: if there exists a proof of A -> G, then there exists a proof from A /\ B -> G.
  - * Extra assumptions are free!
- *Contraction*: if there exists a proof of A /\ A -> G, then there exists a proof from A -> G
- *Exchange*: the order of assumptions does not matter

When we model a system with resources, these rules are not applicable.
Weakening and contraction do not apply to heap resources:

- Weakening would allow new heap cells to materialize
- Contraction would allow heap cells to freely disappear

Overview

Foundations

- Motivation: why Hoare Logic fails for heap programs (aliasing)
- Heap model: addresses, load/store, axiomatic rules
- Core operators: emp, points-to (↦), separating conjunction (*)
- Frame rule: local reasoning and why classical conjunction is unsound

Reasoning about Data Structures

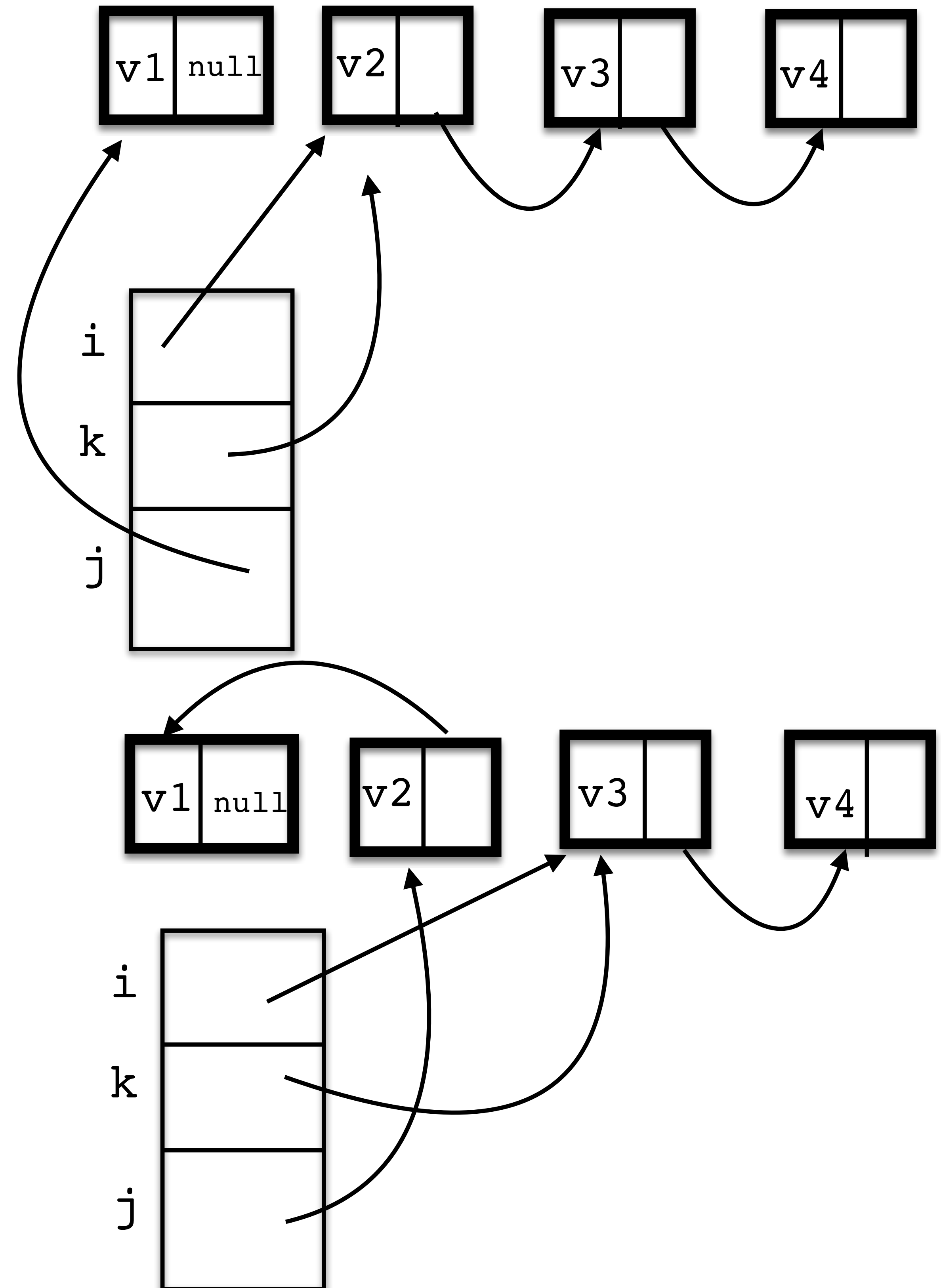
- Magic wand (-*) and separating implication
- Recursive predicates: list, lseg – definition and unfolding
- Consequence rule: applying predicate unfolding in proofs
- Worked examples: swap, malloc/free, list reversal

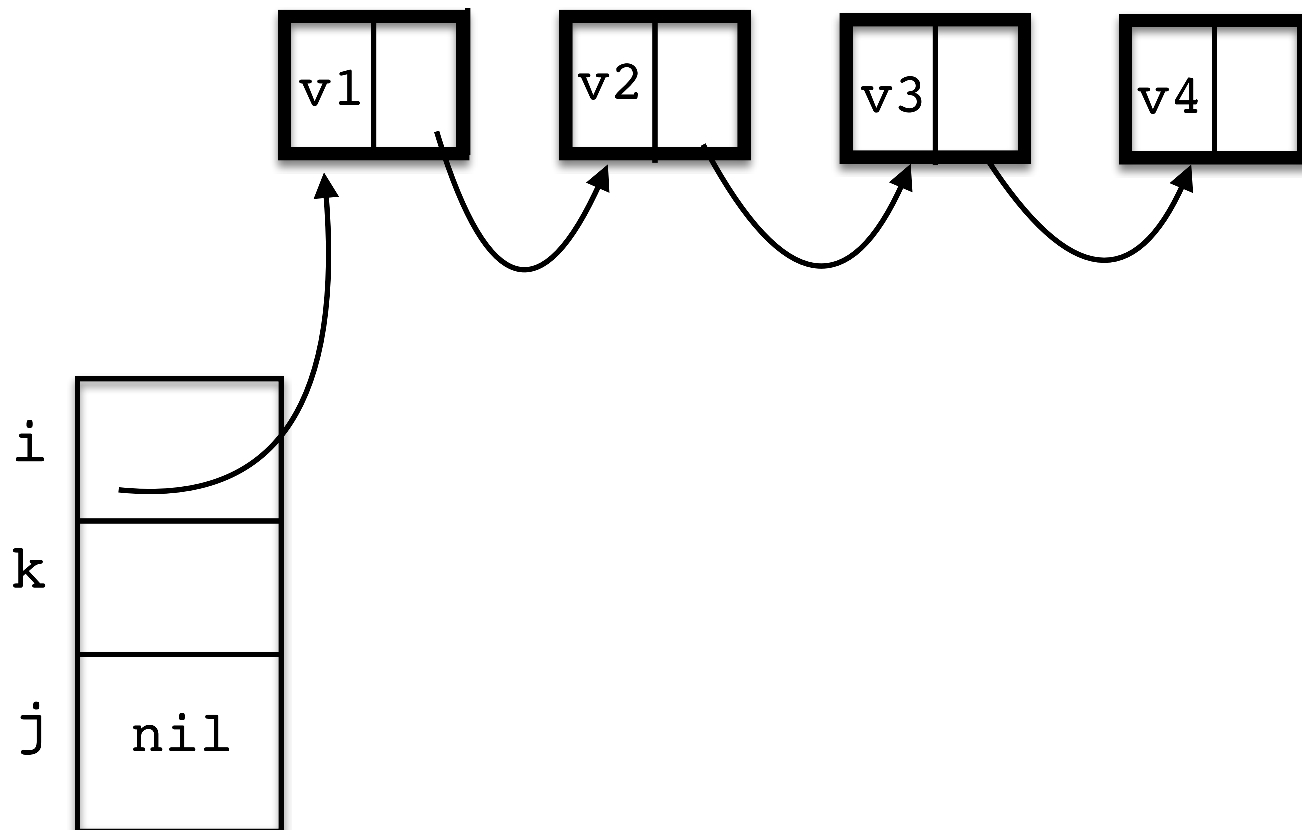
Tools and Scale

- Bi-abduction: inferring pre/postconditions automatically
- Facebook Infer: SL at industrial scale

Example


```
j := null;
while i <> null do {
   k = *(i + 1);
   *(i + 1) = j;
   j = i;
   i = k;
}
```





Example


```
j := nil;
while i <> nil do {
   k = *(i + 1);
   *(i + 1) = j;
   j = i;
   i = k;
}
```

Invariants:

- i and j are pointers into a list structure
- the reversal of the original list can be obtained by concatenating the reversal of the list reachable from i to the list reachable from j

The correctness of the invariant crucially relies on the assumption that there is no aliasing among list elements:

- augment invariant to assert that only nil is reachable from i and j

- What about other structures that may point into this list?

The Heap

- A heap is a partial function that maps addresses to values
  - assume addresses and values are both ints
  - thus, addresses are also values
- Two new instructions:
  - x := [e]   // load the contents of the address referenced by e into x
  - [e1] := e2 // store the value of e2 into the address referenced by e1

Semantics:

- a state consists of a heap *h* and local memory γ

$$\frac{\gamma \vdash \mathsf{e} \Downarrow addr \quad \gamma' = \gamma[\mathtt{x} \mapsto h(addr)]}{\langle h, \gamma \rangle \vdash \mathtt{x} := [\mathsf{e}] \Rightarrow \langle h, \gamma' \rangle}$$

$$\frac{\gamma \vdash \mathtt{e1} \Downarrow addr \quad \gamma \vdash \ e2 \Downarrow v \quad h' = h[addr \mapsto v]}{\langle h, \gamma \rangle \vdash [\mathtt{e1}] := \mathtt{e2} \Rightarrow \langle h', \gamma \rangle}$$

Axiomatic Rules

Load

$$\{e_1 \mapsto n\} \quad x := [e_1] \quad \{x = n\}$$

Store

$$\{\text{True}\} \quad [e_1] := v \quad \{e_1 \mapsto v\}$$

Consider the following rule:

$$\{\ [x] \mapsto z \wedge [y] \mapsto w \wedge [z] \mapsto 3\}\ [z] := 4\ \{\ [x] \mapsto z \wedge [y] \neq [z]]\ \}$$

Is this a reasonable assertion? How about:

$$\{\ [x] \mapsto z \wedge [y] \mapsto w \wedge [z] \mapsto 3\}\ [z] := 4\ \{\ [x] \mapsto z \wedge ([y] \mapsto w \vee [y] \mapsto 4) \wedge [z] \mapsto 4\}$$

Or: $\{\ [x] \mapsto z * [y] \mapsto w * [z] \mapsto 3\}\ [z] := 4\ \{\ [x] \mapsto z * [y] \mapsto w * [z] \mapsto 4\}$

In general, the validity of a triple must take aliasing properties into account, either in the precondition (establish that $w \neq z$) or in the post-condition

Separation Logic

Rather than trying to explicitly reason about heap structure and aliasing within Hoare Logic, introduce new logical operators to reason about how heaps (aka resources) are used

- emp: empty heap
- x $\mapsto$ v: heap has a cell at x with value v
- P * Q: separating conjunction (disjoint parts of the heap)
- P $-*$ Q: separating implication (hypothetical heap extension)

Assertions now describe heap and variable conditions

- Conjunction (*), implication (-*)
- Emp and points-to relation

Example:

$$h \models P * Q$$

means: the heap can be divided into disjoint parts, one which satisfies P ($h \models P$)and the other which satisfies Q ($h \models Q$)

More Formally...

$$h \models P * Q$$

means

$$\exists h_1, h_2 . h_1 \oplus h_2 = h \wedge h_1 \models P \wedge h_2 \models Q$$

where

$$h_1 \oplus h_2$$

means the union of the resources "owned" by h_1 and the resources owned by h_2

$$h_1 \oplus h_2 = h_2 \oplus h_1$$

$$h_1 \oplus (h_2 \oplus h_3) = (h_1 \oplus h_2) \oplus h_1$$

It is expected that heaps be disjoint (resources owned by one are not also owned by the other)

Memory

- The heap consists of a collection of memory cells, each indexed by an address
- Each memory cell provides a resource
- The assertion:

$$\{ \ x \mapsto 2 \ * \ y \mapsto 3 \ \}$$

means:

the heap can be split into two disjoint regions, one that satisfies the assertion that the memory cell with address x contains 2 and the other that satisfies the assertion that the memory cell with address y contains 3

In other words, x and y are not aliases for the same cell

So, the following proof rule is sound:

$$\{\mathrm{x} \mapsto v_1 * \mathrm{y} \mapsto v_2\}[\mathrm{x}] := \mathrm{v}_3\{\mathrm{x} \mapsto v_3 * \mathrm{y} \mapsto v_2\}$$

Points-to

What does $\mathtt{x} \mapsto v$ mean?

$$(h, \gamma) \vDash \mathtt{x} \mapsto v \equiv \; h(x) = v \land dom(h) = \{\mathtt{x}\}$$

That is, the assertion holds in a singleton heap that only contains the resource at location $\mathtt{x}$

$\{ \text{emp} \}$ `x = new(3)` $\{ \mathtt{x} \mapsto 3 \}$

$\{ \mathtt{x} \mapsto \mathrm{v} \}$ `free x` $\{ \text{emp} \}$

Frame Rule

Note that in the rule:

$$\{\mathrm{x} \mapsto v_1 * \mathrm{y} \mapsto v_2\}[\mathrm{x}] := \mathrm{v}_3\{\mathrm{x} \mapsto v_3 * \mathrm{y} \mapsto v_2\}$$

the assertion on y is unused and provides no meaningful information relevant to the proof

$$\frac{\{\psi\} \quad \mathsf{c} \quad \{\phi\}}{\{\psi * \mathsf{F}\} \quad \mathsf{c} \quad \{\phi * \mathsf{F}\}}$$

Importantly, the following is not valid:

$$\frac{\{\psi\} \quad \mathsf{c} \quad \{\phi\}}{\{\psi \wedge \mathsf{F}\} \quad \mathsf{c} \quad \{\phi \wedge \mathsf{F}\}}$$

Frame Rule

The following is a valid inference:

$$\frac{\{\mathtt{x} \mapsto \mathtt{w}\}\ [\mathtt{w}] := 4\ \{\mathtt{x} \mapsto \mathtt{w} * \mathtt{w} \mapsto 4\}}{\{\mathtt{x} \mapsto \mathtt{w} * \mathtt{y} \mapsto \mathtt{z} * \mathtt{w} \mapsto 3 * \mathtt{z} \mapsto v\}\ [\mathtt{w}] := 4\ \{\mathtt{x} \mapsto \mathtt{w} * \mathtt{y} \mapsto \mathtt{z} * \mathtt{w} \mapsto 4 * \mathtt{z} \mapsto v\}}$$

While the following is not, because $\mathtt{z}$ and $\mathtt{w}$ may denote the same address:

$$\frac{\{\mathtt{x} \mapsto \mathtt{w}\}\ [\mathtt{w}] := 4\ \{\mathtt{x} \mapsto \mathtt{w} * \mathtt{w} \mapsto 4\}}{\{\mathtt{x} \mapsto \mathtt{w} * \mathtt{y} \mapsto \mathtt{z} * \mathtt{w} \mapsto 3 \land \mathtt{z} \mapsto v\}\ [\mathtt{w}] := 4\ \{\mathtt{x} \mapsto \mathtt{w} * \mathtt{y} \mapsto \mathtt{z} * \mathtt{w} \mapsto 4 \land \mathtt{z} \mapsto v\}}$$

Local Reasoning

Why the Frame Rule Matters: Local vs Global Reasoning

Classical Hoare Logic (no frame rule): global burden

- To verify `[x] := v`, must assert every OTHER heap cell is unchanged
- Precondition must name ALL heap cells y1,...,yn and assert x ≠ yi
- Proof scales with the SIZE of the heap, not the size of the command

Separation Logic (frame rule): local reasoning

- Prove `{ x ↦ v } [x] := w { x ↦ w }` in isolation
- Frame rule lifts this to any heap F disjoint from x:

```
{ x ↦ v * F } [x] := w { x ↦ w * F }
```

- Disjointness from * guarantees F is unmodified - no aliasing check needed
- Proof complexity is proportional to the FOOTPRINT of the command

Why classical /\ doesn't work as frame

- `{ P } c { Q }` does NOT imply `{ P /\ F } c { Q /\ F }` in general
- F might share addresses with P - * requires disjointness, /\ does not

Consequence Rule

Consequence Rule in Separation Logic

- Same as Hoare Logic: strengthen pre, weaken post

$$\frac{P' \models P \qquad \{ P \}\ c\ \{ Q \} \qquad Q \models Q'}{\{ P' \}\ c\ \{ Q' \}}$$

Strengthening the precondition:

$$x \mapsto v * y \mapsto w \vDash x \mapsto v$$

Note that entailment is a logical property, and does not say anything about the shape of the heap (we're not inventing resources). The consequence rule operates at the level of proof/specification, not heap semantics (unlike separating conjunction).

Weakening the postcondition:

$$x \mapsto 3 \vDash \exists v.\, x \mapsto v$$

Consequence Rule

Consequence Rule in Separation Logic

- Same as Hoare Logic: strengthen pre, weaken post

```
P'  |= P      { P } c { Q }      Q |= Q'
-----------------------------------------
           { P' } c { Q' }A
```

Additional challenge: entailment between SL assertions

- Must prove `P' |= P` where `P`, `P'` involve *, emp, recursive predicates
- Primary technique: UNFOLD recursive predicates at case boundaries

Example: unfolding list(i, s1) when i ≠ null

```
list(i, s1) /\ i ≠ null   |=
        ∃v, n, s'.  i ↦ v * (i+1) ↦ n * list(n, s')  /\  s1 = v :: s'
```

- This unfolding step is what justifies reading [i] and [i+1] inside the loop body
- After the loop body, FOLD back into `list(j, v::s2)`

More Formally ...

Formal Semantics of Separating Conjunction

The satisfaction relation (h, γ) |= P * Q is defined as:

$$\exists h1,h2.\ h = h1 \uplus h2 \wedge (h1,\gamma) \vDash P \wedge (h2,\gamma) \vDash Q$$

where h1 $\uplus$ h2 means disjoint union: dom(h1) ∩ dom(h2) = ∅

γ is the local store that is shared unchanged between both parts – only the heap is split."

Key consequences

- `P * Q  =>  P` is NOT valid in general: the right part of the heap is discarded
- `P  =>  P * P` is NOT valid: would require duplicating heap cells
- `(x ↦ v) * (x ↦ w)` is UNSATISFIABLE: x cannot appear in both disjoint parts

emp semantics

- (h, γ) |= `emp` iff dom(h) = ∅ (the heap is empty)
- `P * emp  ⇔  P` (emp is the unit of separating conjunction)

Magic Wand (Separating Implication)

$$P \quad -\!\!* \quad Q$$

reads:

Extending a heap *h* with another (disjoint) heap that satisfies *P*, results in a new heap that satisfies *Q*

$$\frac{\forall h'. h' \perp h, h' \models P \rightarrow \; h \oplus h' \models Q}{\langle h, \gamma \rangle \models P -\!\!* Q}$$

$$\frac{\langle h, \gamma \rangle \models P * (P -\!\!* Q)}{\langle h, \gamma \rangle \models Q}$$

Magic Wand Example

$$\mathtt{x} \mapsto 1 \vdash \mathtt{y} \mapsto 2 \quad -\!\!* \quad (\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 2)$$

Starting from a heap that stores 1 at address x, if we add another heap that stores 2 at address y, then we can conclude that the combined heap maps x to 1 and y to 2

$$lseg(\mathtt{x}, \mathtt{y}) \vdash lseg(\mathtt{y}, \mathtt{z}) \quad -\!\!* \quad lseg(\mathtt{x}, \mathtt{z})$$

$\mathtt{lseg}(a, b)$ represents a list indexed at a upto but not including b

The formula states:

- Assuming a list whose root is at address x that does not include the node indexed at y
- If there is another (disjoint) list indexed at y that does not include the node indexed at z
- The heap containing both list segments contains a list segment from x to z (exclusive)

Concept Check

Which assertions are valid?

- `P => P * P`
- `P * Q => P`
- $x \mapsto 3 \Rightarrow x \mapsto 3 * x \mapsto 3$
- $x \mapsto 3 \Rightarrow x \mapsto 3 * y \mapsto 42$
- $x \mapsto 3 \Rightarrow 0 \leq [x]$
- $(x \mapsto -) * (x \mapsto -)$
- $(P \mathbin{-*} -) * (Q \mathbin{-*} -) \Rightarrow P \neq Q$
- $(x \mapsto 3) * (y \mapsto 3) \Rightarrow x \neq y$

Concept Check – Answers

Recall: Which assertions are valid?

- **P => P * P** ❌ FALSE: P * P requires splitting the heap into two parts both satisfying P. Can't duplicate a heap cell.
- **P * Q => P** ❌ FALSE: * is not weakening. The Q part of the heap is not discardable in SL.
- **x ↦ 3 => x ↦ 3 * x ↦ 3** ❌ FALSE: same cell cannot be in two disjoint heaps simultaneously.
- **x ↦ 3 => x ↦ 3 * y ↦ 42** ❌ FALSE: requires a cell at y which is not in scope.
- **x ↦ 3 => 0 ≤ [x]** ✅ TRUE: x ↦ 3 means [x]=3 and 0≤3 holds.
- **(x ↦ _) * (x ↦ _)** ❌ UNSATISFIABLE: x appears in both disjoint parts – impossible.
- **(P -* _) * (Q -* _) => P ≠ Q** ❌ FALSE in general: magic wand makes no claim about current heap ownership.
- **(P ↦ 3) * (Q ↦ 3) => P ≠ Q** ✅ TRUE: both cells are in disjoint heap parts, so they must be different addresses.

Summary

Separation Logic is useful to verify properties of programs that make use of references (i.e., memory addresses). It can help identify errors involving:

- using memory before allocation or using it after freeing
- inadvertent use of aliased memory
- freeing memory that is not allocated
- allocation without freeing

Generalizes to any system that manipulates resources

- networks
- concurrency
- distributed programming

The frame rule enables compositional (local) reasoning

- to verify a property involving the heap, we can safely ignore all parts of the heap unrelated to the parts reachable from the command being analyzed

Example

A program that swaps the value of two memory cells

```
t := [x];
b := [y]
[x] := b;
[y] := t
```

Precondition:

$$(\mathrm{x} \mapsto v_1 * \mathrm{y} \mapsto v_2)$$

Postcondition:

$$(\mathrm{x} \mapsto v_2 * \mathrm{y} \mapsto v_1)$$

Example

$$(\mathtt{x} \mapsto v_1 * \mathtt{y} \mapsto v_2)$$

```
t := [x]      // local assignment, t = v1
```

$$(\mathtt{x} \mapsto v_1 * \mathtt{y} \mapsto v_2)$$

```
b := [y]      // local assignment, b = v2
```

$$(\mathtt{x} \mapsto v_1 * \mathtt{y} \mapsto v_2)$$

```
[x] := b      // store
```

$$(\mathtt{x} \mapsto v_2 * \mathtt{y} \mapsto v_2)$$

```
[y] := t      // store
```

$$(\mathtt{x} \mapsto v_2 * \mathtt{y} \mapsto v_1)$$

Example


```
x := malloc();
[x] := 42
```

Precondition:

$$\{ \texttt{emp} \}$$

Postcondition:

$$\{ \texttt{x} \mapsto \texttt{42} \}$$

Proof rule for malloc:

$$\{\texttt{emp}\} \quad \texttt{x} := \texttt{malloc}() \quad \{x \mapsto - \}$$

Example


```
x := malloc();
[x] := 42;
free(x)
```

Both the pre- and post-condition should be { emp }

```
{ emp }
   x := malloc();
{ x ↦ - }
   [x] := 42;
{ x ↦ 42 }
   free(x);
{ emp }
```

Proof rule for free:

$$\{\mathtt{x} \mapsto v\} \quad \mathtt{free(x)} \quad \{\mathtt{emp}\}$$

Example

Deep copy contents of one list to another:

```
p := x;
q := y
while (p != null) {
    temp1 := [p];
    temp2 := [q];
    [p] := temp2;
    [q] := temp1;
    p := [p + 1];
    q := [q + 1]
}
```

Shallow copy much simpler:

```
t := x;
x := y;
y := t;
```

List predicate:

$$\mathit{list}(\mathrm{x}, \mathrm{s}) \equiv$$
$$\mathrm{x} = \mathrm{null} \wedge \mathrm{s} = [\,] \wedge \mathrm{emp}$$
$$\vee\ \exists v, \mathrm{n}, \mathrm{s}'.\, \mathrm{x} \mapsto v * \mathrm{x} + 1 \mapsto \mathrm{n} * \mathit{list}(\mathrm{n}, \mathrm{s}') \wedge \mathrm{s} = v :: \mathrm{s}'$$

Example

Precondition:

$$\{list(\mathtt{x}, \mathtt{s}_1) * list(\mathtt{y}, \mathtt{s}_2)\} \text{ where } |\mathtt{s}_1| = |\mathtt{s}_2|$$

Postcondition:

$$list(\mathtt{x}, \mathtt{s}_2) * list(\mathtt{y}, \mathtt{s}_1)$$

```
p := x;
q := y
while (p != null) {
   temp1 := [p];
   temp2 := [q];
   [p] := temp2;
   [q] := temp1;
   p := [p + 1];
   q := [q + 1]
}
```

$$\mathtt{list_seg}(r, r, []) \equiv \mathtt{emp}$$

$$\mathtt{list_seg}(r, s, v :: vs) \equiv \exists n \,.\, r \mapsto v, n * \mathtt{list_seg}(n, s, vs) \text{ if } r \neq s$$

Loop invariant (spatial):

$$\exists s_{1_a}, s_{1_b}, s_{2_a}, s_{2_b} \,.\, \mathtt{list_seg}(\mathtt{x}, \mathtt{p}, s_{1_a}) * \mathtt{list}(\mathtt{p}, s_{1_b}) * \mathtt{list_seg}(\mathtt{y}, \mathtt{q}, s_{2_a}) * \mathtt{list}(\mathtt{q}, s_{2_b})$$

Loop invariant (content):

$$|s_{1_a}| = |s_{2_a}| \wedge \forall i < |s_{1_a}[i] \,.\, s_{1_a}[i] = \mathtt{old}(s_{2_a}[i]) \wedge s_{2_a}[i] = \mathtt{old}(s_{1_a}[i])$$

Example


```
j := null;
while i ≠ null do {
  k := [i + 1];
  [i + 1] := j;
  j := i;
  i := k;
}
```

$$list(\mathtt{x}, \mathtt{s}) \equiv$$

$$\mathtt{x} = \mathtt{null} \land \mathtt{s} = [] \land \mathtt{emp}$$

$$\lor \exists v, \mathtt{n}, \mathtt{s}' . \mathtt{x} \mapsto v * \mathtt{x} + 1 \mapsto \mathtt{n} * list(\mathtt{n}, \mathtt{s}') \land \mathtt{s} = v :: \mathtt{s}'$$

Precondition:

$$\{list(\mathtt{i}, s_0)\}$$

Postcondition:

$$\{list(\mathtt{j}, \mathtt{rev}(s_0))\}$$

Loop invariant:

$$\exists s_1, s_2 . list(\mathtt{i}, s_1) * list(\mathtt{j}, s_2) \land s_0 = rev(s_2) + s_1$$

Example


```
j := null;
while i ≠ null do {
  k := [i + 1];
  [i + 1] := j;
  j := i;
  i := k;
}
```

$list(\mathtt{i}, s_0)$ from precondition

j = null => $list(\mathtt{j}, [])$

$s_1 = s_0, s_2 = []$

Invariant holds

$$\exists s_1, s_2 \,.\, list(\mathtt{i}, s_1) * list(\mathtt{j}, s_2) \wedge s_0 = rev(s_2) + s_1$$

Example


```
j := null;
while i ≠ null do {
  k := [i + 1];
  [i + 1] := j;
  j := i;
  i := k;
}
```

invariant:

$$\exists s_1, s_2 \,.\, list(\mathtt{i}, s_1) * list(\mathtt{j}, s_2) \wedge s_0 = rev(s_2) + s_1$$

list:

$$\exists v, \mathtt{n}, \mathtt{s}' \,.\, \mathtt{x} \mapsto v * \mathtt{x} + 1 \mapsto \mathtt{n} * list(\mathtt{n}, \mathtt{s}') \wedge \mathtt{s} = v :: \mathtt{s}'$$

$list(\mathtt{i}, s')$: unreversed list remaining

$list(\mathtt{j}, v :: s_2)$: reversed list extended by head node (justified by assignment - j := i)

$rev(v :: s_2) = rev(s_2) + [v]$

- $s_0 = rev(s_2) + (v :: s') = rev(v :: s_2) + s'$

$list(i, s') * list(j, v :: s_2) \wedge s_0 = rev(v :: s_2) + s'$

Example


```
j := null;
while i ≠ null do {
  k := [i + 1];
  [i + 1] := j;
  j := i;
  i := k;
}
```

invariant:

$$\exists s_1, s_2 \,.\, list(\mathtt{i}, s_1) * list(\mathtt{j}, s_2) \wedge s_0 = rev(s_2) + s_1$$

When i = null,

- $s_1 = [], list(\mathtt{i}, []) = \mathtt{emp}$

$$list(\mathtt{j}, s_2) \wedge s_0 = rev(s_2) \Rightarrow list(j, rev(s_0))$$

Bi-Abduction

A logical proof typically asserts the validity of statements of the form:
$A \vdash B$ which states that B holds assuming A is true

To infer properties of problems in Separation Logic, an alternative proof inference technique called Bi-abduction is used:

$$A * antiFrame \vdash B * frame$$

Here, the anti-frame refers to *missing part of the heap* that may be accessed, while the frame is the part of the heap that is implied by the original heap that is valid after B is satisfied

A bi-abduction inference procedure enables modular inter-procedural reasoning

Anti-frame

Current heap (Pre): $x \mapsto 1$

Wish to call a function f whose specification requires:

$(x \mapsto _ * y \mapsto *)$

Add an anti-frame to the caller's heap that includes y

Facebook Infer

Facebook Infer: Separation Logic at Industrial Scale

Infer is a static analysis tool built on SL and bi-abduction

- Deployed at Meta (Facebook), Amazon, Mozilla, Spotify
- Analyzes Java, C, C++, Objective-C, Kotlin
- Runs on every code diff before it lands in Meta's codebase

What it detects (no annotations required)

- Null dereferences, use-after-free, memory leaks
- Resource leaks (file handles, locks, database cursors)
- Concurrency bugs via RacerD (SL generalized to thread resources)

The bi-abduction connection

- No function specs needed: bi-abduction infers them from call sites
- Frame = what the caller's heap still owns after the call
- Anti-frame = resources the callee needs from the caller

Automated Verification

Infer reasons via symbolic execution over SL assertions

The program state is a symbolic heap — a SL assertion updated at each step using SL proof rules

Example: increment a heap cell

```
Precondition: { x ↦ 3 }
t := [x]          symbolic heap: x ↦ 3 ∧ t = 3          (Load rule)
[x] := t + 1      symbolic heap: x ↦ 4                  (Store rule)
Postcondition: { x ↦ 4 }  ✔
```

At call sites: bi-abduction fills in missing heap context

- Anti-frame: what the callee needs that is missing from the current symbolic heap
- Frame: the portion of the current heap untouched by the call (preserved by the frame rule)
- Each function analyzed once; inferred specs propagate across call boundaries without re-analysis

Errors surface when the symbolic heap becomes inconsistent

- Null dereference: accessing address x when x ↦ _ is not in the symbolic heap
- Memory leak: symbolic heap is non-empty (contains x ↦ _) at a free point

Concurrent Separation Logic

Problem: sequential SL assumes exclusive ownership per thread

- Concurrency requires threads to share heap cells — exclusive ownership is too restrictive
- Solution: attach a fractional permission π ∈ (0, 1] to each points-to assertion

```
x ↦^π v          where π ∈ (0, 1]
```

Permission semantics

- $\pi = 1$ — **exclusive write permission:** no other thread holds any permission on x; thread may read or write
- $0 < \pi < 1$ — **shared read permission:** read only; multiple threads may hold read fractions (sum ≤ 1)

Splitting and recombining permissions

- `x ↦¹ v   ⊢   x ↦^(1/2) v  *  x ↦^(1/2) v   // split`
- `x ↦^(1/2) v  *  x ↦^(1/2) v   ⊢   x ↦¹ v   // recombine`

CSL: Concurrent Frame Rule

Concurrent frame rule (O'Hearn, 2007)

- Two threads c_1 and c_2 may run in parallel if their heap footprints are compatible under the permission model:

$$\frac{\{\ P_1\ \}\ c_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ c_2\ \{\ Q_2\ \} \qquad P_1 * P_2 \text{ defined } (\pi_1 + \pi_2 \leq 1 \text{ for any shared cell } x)}{\{\ P_1 * P_2\ \}\ \ c_1 \parallel c_2\ \ \{\ Q_1 * Q_2\ \}}$$

What the permission side condition enforces

- Disjoint ($\pi_1 = \pi_2 = 1$): threads own separate cells — sequential frame rule applies
- Shared read-only ($\pi_1 + \pi_2 < 1$): safe concurrent reads — writes forbidden
- Conflicting writes ($\pi_1 + \pi_2 > 1$): $P_1 * P_2$ is *unsatisfiable* — the rule cannot be applied, flagging a data race at proof time