CS 565

Programming Languages (graduate)
Spring 2025

Week 14

Separation Logic

Readings

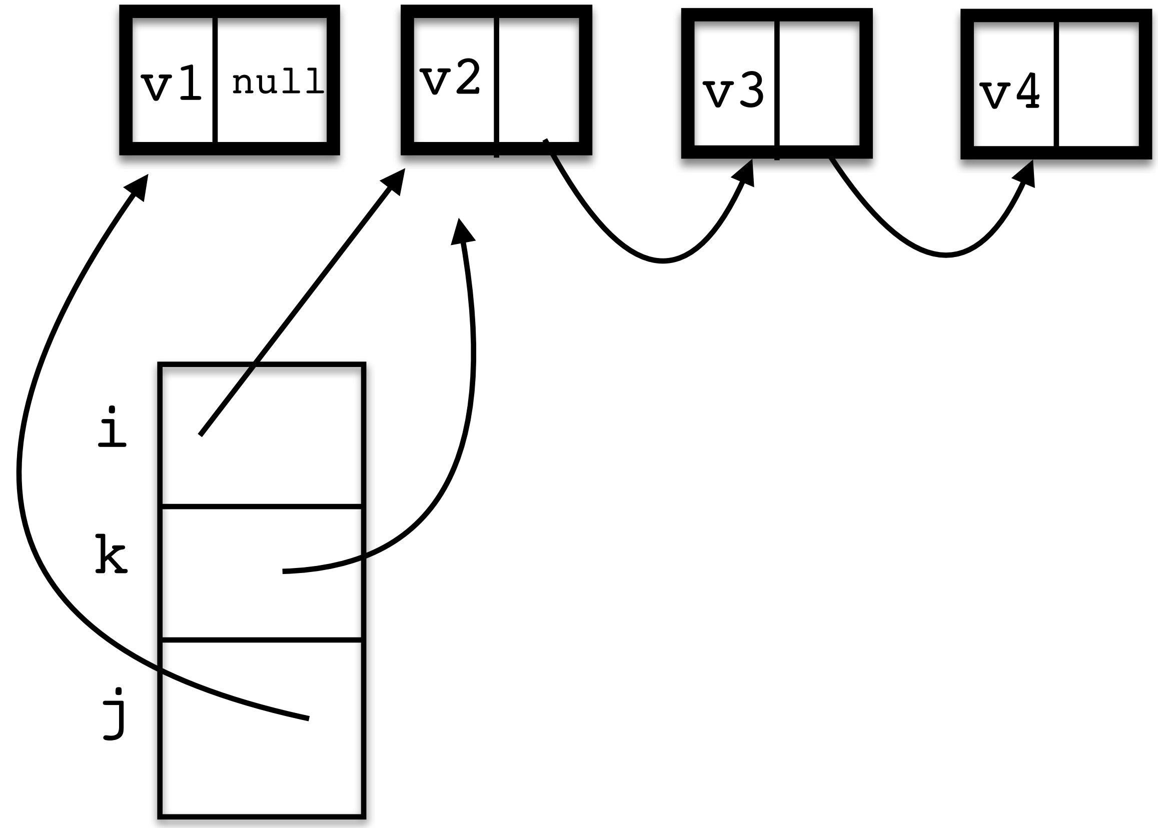
- Software Foundations, Vol. 6
- Separation Logic, O'Hearn CACM, 2/19
- Local Reasoning about Programs that Alter Data Structures, O'Hearn, Reynolds, Yang, CSL, 10/01
- Separation Logic: A Logic for Shared Mutable Data Structures, LICS, 2002

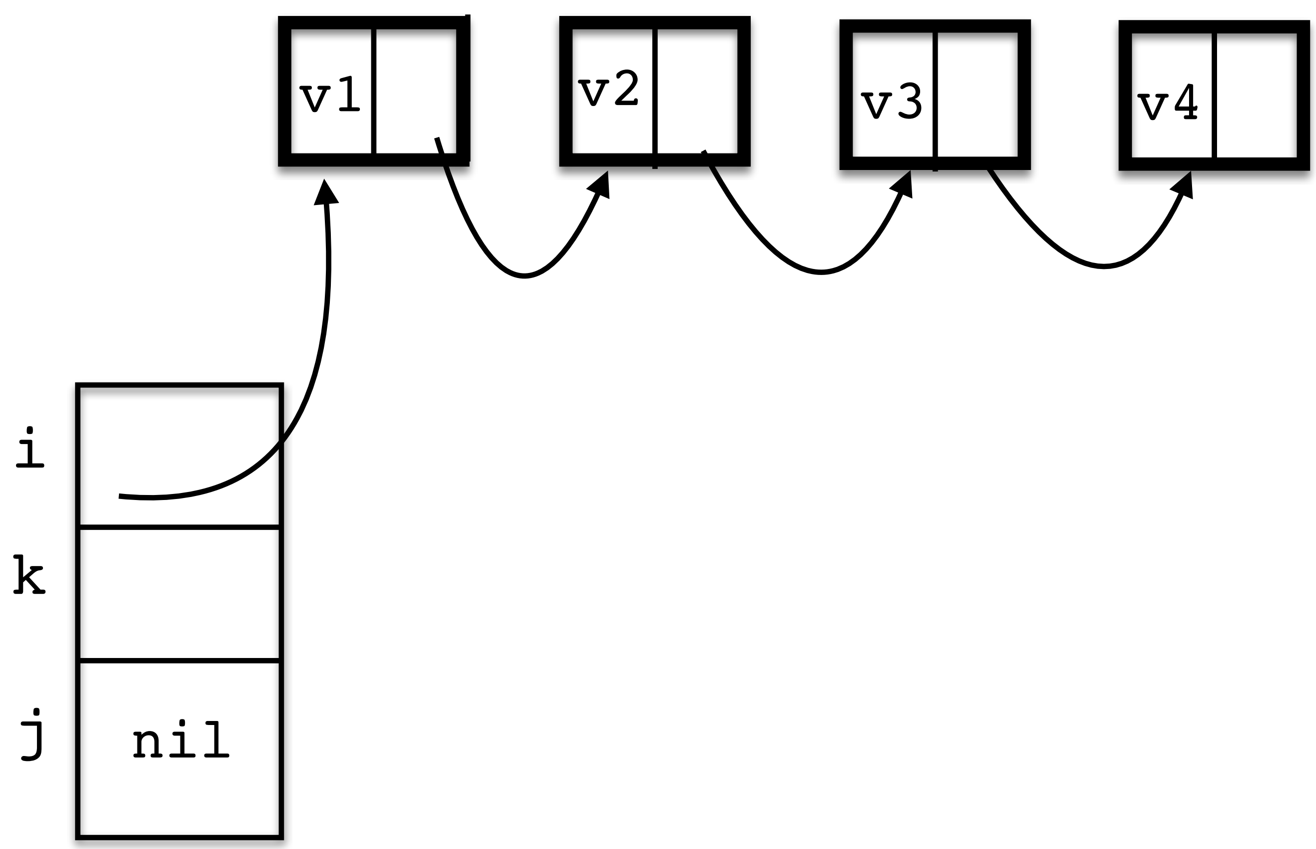
Motivation

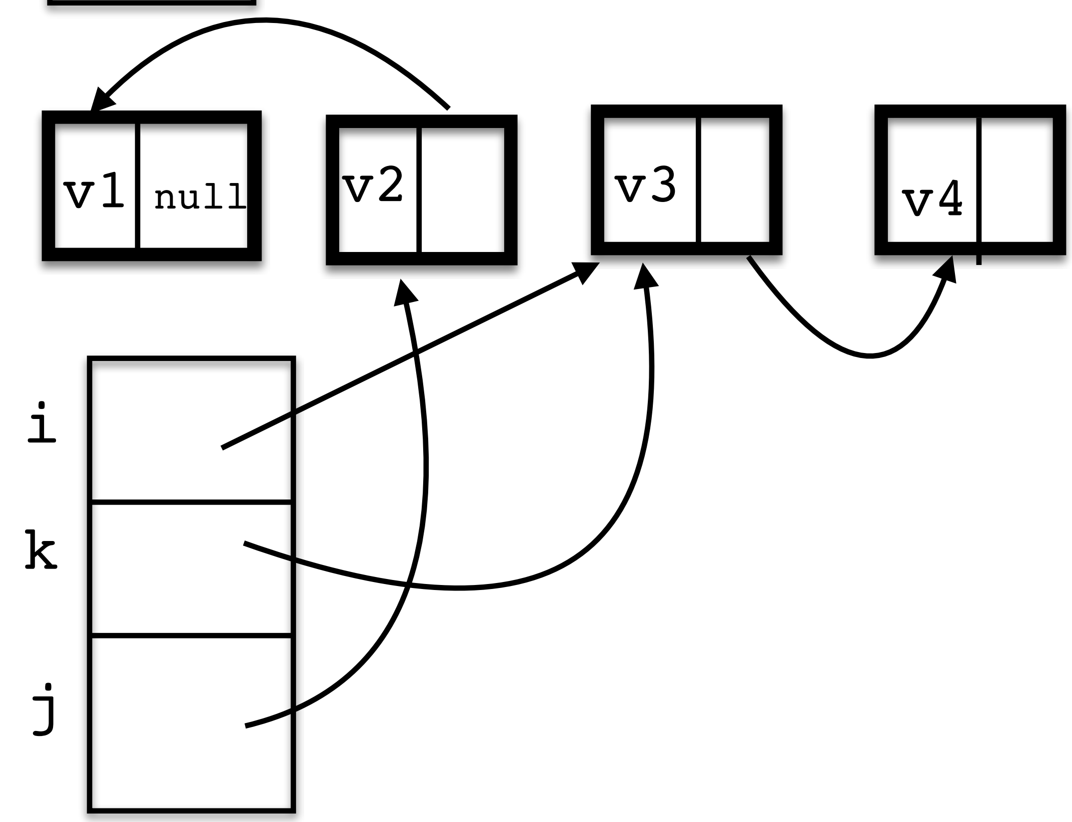
- Hoare Logic is defined in terms of assertions on states:
  - states are maps from variables to their values
  - most programming languages also support the notion of a heap:
    - variables map to addresses
    - the contents at a given address can be shared and aliased
  - Embedding notions of sharing 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

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

$$\{\mathtt{e}_1 \mapsto n\} \quad \mathtt{x} := [\mathtt{e}_1] \quad \{\mathtt{x} = n\}$$

Store

$$\{\mathsf{True}\} \quad [\mathtt{e}_1] := \mathtt{v} \quad \{\mathtt{e}_1 \mapsto v\}$$

Consider the following rule:

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

Is this a reasonable assertion?

In general, the validity of a triple must take aliasing properties into account, either in the precondition (establish that w <> z) or in the post-condition (establish that [[y]] maybe 3 or 4)

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:

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

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:

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

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

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

Points-to

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

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

That is, the assertion holds in a singleton heap that only contains the resource at location `x`

{ emp } `x = new(3)` { `x` ↦ 3}

{ `x` ↦ v } `free x` { emp }

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

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

$lseg(a, b)$ 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 ↦ 3 => x ↦ 3 * x ↦ 3`
- `x ↦ 3 => x ↦ 3 * y ↦ 42`
- `x ↦ 3 =>  0 ≤ [x]`
- `(x ↦ - ) * (x ↦ - )`
- `(P ↦ - ) * (Q ↦ - ) => P ≠ Q`
- `(P ↦ 3) * (Q ↦ 3) => P ≠ Q`

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:

$$\{\ \mathtt{emp}\ \}$$

Postcondition:

$$\{\ \mathtt{x} \mapsto 42\ \}$$

Proof rule for malloc:

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

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:

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

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

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

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

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

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

Precondition:

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

Postcondition:

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

Loop invariant:

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

Example


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

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

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

list:

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

$\mathit{list}(\mathtt{i}, s')$: unreversed list remaining
$\mathit{list}(\mathtt{j}, v :: s_2)$: reversed list extend by head node (justified by assignment - j := i
$\mathit{rev}(v :: s_2) = \mathit{rev}(s_2) + [v]$
- $s_0 = \mathit{rev}(s_2) + (v :: s') = \mathit{rev}(v :: s_2) + s'$

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

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

Example

A function invokes a method (free_list) to deallocate a list

Spec for free_list: { list(x) } free_list(x) {emp}
where

$$\text{list}(x) \equiv (x = \text{null} \wedge \text{emp}) \vee (\exists y \,.\, x \mapsto y * \text{list}(y))$$

Suppose the heap state prior to the call is determined to be:

$$\text{Pre} \equiv (x\,.\,\text{next} \mapsto y * y\,.\,\text{next} \mapsto z)$$

Is the call to free_list safe?

Can we extend Pre with a heap A such that $\text{Pre} * A \vdash \text{list}(x)$

- Infer an *anti-frame* that z is a list in the heap:

$$\equiv (x\,.\,\text{next} \mapsto y * y\,.\,\text{next} \mapsto z) * \text{list}(z)$$

Suppose the client expects a post-condition: $\{ a \mapsto v \}$

- But, free_list's post-condition is { emp }. Infer a *frame* that represents the portion of the heap consistent with the heap returned by free_list and the post-condition:

$$\{\text{emp} * a \mapsto v\}$$