Lecture 11: Separation Logic

1 动机：经典 Hoare 逻辑处理指针的局限

从 `malloc/free` 看动态内存。`malloc(size)` 申请一块给定大小的内存并返回指针；成功分配后，程序才能通过该指针访问这块内存。`free(ptr)` 释放对应的已分配内存，使其可以被重新使用。

在不涉及堆操作的经典 **Hoare** 逻辑中，我们验证形如 $\{P\}\ C\ \{Q\}$ 的程序规约，其中 C 表示代码语句，程序数据仅保存在固定的一组变量（x, y, z）中。程序状态记录这些变量当前的值。然而，实际程序还常操作链表、树等动态数据结构，它们的节点存放在程序运行时申请的内存中，这部分内存称为堆（**Heap**）。

从固定变量转向堆指针后，经典 Hoare 逻辑面临一个挑战：刻画无界动态数据结构，也就是描述长度没有预先确定上限的链表等结构。假设前置条件要求 x 指向一条合法（无环、以 `null` 结尾）的单链表。若链表只有两个节点，可以写出如下长度为 2 的公式：

$$x \neq \texttt{null} \ \wedge\ x.\texttt{next} \neq \texttt{null} \ \wedge\ x.\texttt{next.next} == \texttt{null} \ \wedge\ \texttt{x} \neq \texttt{x.next}$$

但如果链表长度 n 任意，就无法通过固定次数地读取 `next` 字段来描述整条链表。即使已经说明 x 和 y 各自指向一条合法链表，也不能保证两条链表占据互不重叠的内存——它们可能共享节点。因此，还需要一种方式来直接表达两处结构的节点互不重叠。

引例（`delete`）： 考虑如下递归释放整条单链表的过程：

```
procedure delete(x: Node) {
    if (x != null) {
        delete(x.next);
        free(x);
    }
}
```

我们先用自然语言梳理对 `delete(x)` 的正确性推理：

- 前置条件： `x` 要么指向一条长度不限（≥ 1）且以 `null` 结尾的单链表，要么为空链表（`x == null`）。
- 若 `x == null`： 跳过 `if` 语句体；链表为空，没有需要释放的链表节点。
- 若 `x != null`： 链表由头节点 `x` 以及从 `x.next` 开始的子链表（尾链表）组成。假设递归调用能正确释放更短的链表，那么 `delete(x.next)` 会释放整个尾链表，同时不会改动头节点 `x`（因此 `x` 成为该链表唯一尚未释放的节点）。最后，`free(x)` 释放 `x`，此时该链表的所有节点均已释放完毕。

2 分离逻辑（Separation Logic）

2.1 核心思想

为了把 `delete(x)` 的自然语言推理转化为严格的形式化规约，分离逻辑（**Separation Logic**）引入了两个核心思想：

1. 说明拥有哪些节点： 断言不仅描述变量的值，还描述当前可以读写或释放哪些节点，这就是节点的所有权。节点释放后，就不再拥有它，也不能继续访问它。
2. 说明哪些内存互不重叠： 链表的头节点与尾链表占据不同的节点。分离合取 $F * G$ 用来表达这种关系：F 和 G 分别描述两块互不重叠的内存。因此，递归释放尾链表时，可以保留头节点，最后再单独释放它。

2.2 基本命题

描述堆时，先掌握以下四种写法。这里的“堆”指当前断言所拥有的那部分内存。

- emp：堆为空，不拥有任何节点。
- $\mathsf{acc}(x)$：堆中恰好只有一个已分配节点，由 x 指向。
- $F * G$：堆可以分成互不重叠的两部分，一部分满足 F，另一部分满足 G。
- $x = y$：堆为空，并且 x 与 y 的值相等。后面的程序和断言也用 `==` 表示相等。

因此，emp 只说明当前断言不拥有节点，不表示整个程序都没有已分配内存。要同时描述一个节点及其字段值，就写成 $\mathsf{acc}(x) * x.\texttt{next} = y$：拥有 x 指向的节点，而且它的 `next` 字段存放 y。

3 链表的归纳谓词（Inductive Predicates）

我们可以给描述链表的断言取一个名字，再在定义中用同一个名字描述更短的链表，这样的定义称为递归（归纳）谓词。借助 $*$ 将头节点和剩余链表分开，就能描述任意有限长度的无环链表，从而解决第 1 节中刻画无界动态数据结构的挑战。

3.1 无环链表段（`lseg`）

从头指针 x 到终止指针 y 的无环链表段记作 $\mathsf{lseg}(x, y)$，归纳定义如下：

$$\mathsf{lseg}(x, y) \;\equiv\; (x == y) \;\vee\; \big(x \neq y * \mathsf{acc}(x) * \mathsf{lseg}(x.\texttt{next}, y)\big)$$

- 基础情形（$x == y$）： 从 x 到 y 的链表段为空，不拥有任何节点，用 emp 表示。对于以 `null` 结尾的完整单链表 $\mathsf{lseg}(x, \texttt{null})$，基础情形即为 $x == \texttt{null}$。
- 递归情形（$x \neq y$）： 指针 x 拥有头节点 $\mathsf{acc}(x)$，且在与之不相交的另一块堆内存（$*$）中，后继指针 $x.\texttt{next}$ 指向通往 y 的剩余链表段 $\mathsf{lseg}(x.\texttt{next}, y)$。（注意：终点地址 y 本身并不包含在 $\mathsf{lseg}(x, y)$ 所拥有的内存中。）
- 为什么 $*$ 能保证链表无环？ 因为头节点 $\mathsf{acc}(x)$ 与剩余链表段 $\mathsf{lseg}(x.\texttt{next}, y)$ 是用 $*$（而非 $\wedge$）连接的，头节点 x 必然与后续链表段中的所有节点互不重叠；配合归纳定义，就排除了链表内部成环的可能。

3.2 带数据域的归纳谓词：有界链表与有序链表

当每个节点还存有数据字段 `x.data` 时，可以在归纳谓词中同时约束链表结构与节点数据。下面的 min、max 是允许的数据范围的下限和上限：

1. 有界链表段（$\mathsf{bnd_lseg}$）： 链表段中每个节点的 `data` 值都位于区间 $[\min, \max]$ 内：

$$\begin{aligned}\mathsf{bnd_lseg}(x, y, \min, \max) \;\equiv\; &(x == y) \vee \\ &\big(x \neq y * \mathsf{acc}(x) * (\min \le x.\texttt{data} \le \max) * \mathsf{bnd_lseg}(x.\texttt{next}, y, \min, \max)\big)\end{aligned}$$

2. 有序链表段（$\mathsf{srt_lseg}$）： 非递减顺序是指后一个值不小于前一个值，允许相邻值相等。只需在递归调用时，将当前节点的值 `x.data` 作为后续节点允许的最小值传入：

$$\begin{aligned}\mathsf{srt_lseg}(x, y, \min, \max) \;\equiv\; &(x == y) \vee \\ &\big(x \neq y * \mathsf{acc}(x) * (\min \le x.\texttt{data} \le \max) * \mathsf{srt_lseg}(x.\texttt{next}, y, \texttt{x.data}, \max)\big)\end{aligned}$$

4 Hoare 三元组的有效性与内存安全

在分离逻辑中，证明一个 Hoare 三元组 $\{P\}\ C\ \{Q\}$ 不仅证明了功能正确性，还同时确立了内存安全（**Memory Safety**）（即“通过规约验证的程序不会发生内存错误”）：

- 内存安全： 代码语句 C 只能在当前拥有相应权限的已分配区域中读写或释放内存；这些权限来自前置条件，或来自执行过程中的分配及被调用函数的规约。缺少所需权限时，验证不能通过。典型错误包括：访问有效范围之外的位置（越界访问）、继续访问已经释放的内存（释放后使用），以及再次释放同一块已释放的内存（重复释放）。链表例子中，全部节点通过 `free` 释放后得到 emp，表示该调用负责的链表已全部释放，而非整个进程的堆都为空。
- 后置条件保证： 若 C 正常终止，则终止时的堆与变量状态满足后置条件 Q。

Hoare 三元组 $\{P\}\ C\ \{Q\}$	是否有效	原因
$\{\mathsf{acc}(x)\}$ `x.next := y;` $\{\mathsf{acc}(x) * x.\texttt{next} == y\}$	有效	前置条件拥有 $\mathsf{acc}(x)$，写入 `x.next` 安全。
$\{\mathsf{acc}(y)\}$ `x.next := y;` $\{\mathsf{acc}(y) * x.\texttt{next} == y\}$	无效	前置条件仅拥有 y，当 $x \neq y$ 时无法保证拥有 x。
$\{\mathsf{emp}\}$ `x := new Node;` $\{\mathsf{acc}(x)\}$	有效	分配前不拥有节点；分配后拥有新节点 $\mathsf{acc}(x)$。
$\{\mathsf{emp}\}$ `free(x);` $\{\mathsf{emp}\}$	无效	当前不拥有任何节点，无法提供释放 x 所需的权限。
$\{\mathsf{acc}(x)\}$ `free(x);` $\{\mathsf{emp}\}$	有效	释放原来拥有的节点 x 后，不再拥有任何节点，因此得到 emp。
$\{\mathsf{acc}(x) * \mathsf{acc}(y) * y.\texttt{next} == z\}$ `x.next := y;` $\{\mathsf{acc}(x) * x.\texttt{next} == y * \mathsf{acc}(y) * y.\texttt{next} == z\}$	有效	由 $*$ 知 x 与 y 不相交（$x \neq y$），修改 `x.next` 不影响 y。

5 使用 VeriFast 验证链表程序

VeriFast 是支持 C 与 Rust 的形式化验证工具。它检查内存安全和用户指定的后置条件，但不是完全自动的证明器：用户需要给出规约、谓词及必要的证明说明。这些内容写在特殊注释 `/*@ ... @*/` 与 `//@ ...` 中，供验证器使用，不是程序运行时执行的语句：

概念	VeriFast 语法	含义
$F * G$	`F &*& G`	分离合取：F 与 G 描述的内存互不重叠。
字段所有权	`x->next |-> y`	字段所有权：赋予读写 `x->next` 的权限，且当前存储值为 `y`。
未知字段值	`x->next |-> ?next`	用名字 `next` 记录该字段当前的值，供后续规约使用。
$\mathrm{acc}(x)$	`x->next |-> ?next &*& malloc_block_node(x)`	完整节点所有权：字段读写权限，以及该节点由 `malloc` 分配的记录；释放时两项都需要。
$\{P\}\ C\ \{Q\}$	`requires P; ensures Q;`	前置与后置条件规约（`result` 表示返回值）。

5.1 验证链表的创建与销毁

下面的 C 示例定义链表谓词、递归删除过程和创建头节点的函数。`nodes(0)` 表示空链表；非空时，`?next` 用名字 `next` 记录头节点中存储的后继地址。

```
#include <stdlib.h>

struct node { struct node *next; };

/*@
predicate nodes(struct node *x) =
    x == 0 ? emp : x->next |-> ?next &*& malloc_block_node(x) &*& nodes(next);
@*/

void delete(struct node *x)
    //@ requires nodes(x);
    //@ ensures emp;
{
    if (x != 0) {
        struct node *next = x->next;
        delete(next);  // Free the tail; keep the head node allocated.
        free(x);       // Free the head node.
    }
}

struct node *node_create(struct node *next)
    //@ requires nodes(next);
    //@ ensures nodes(result);
{
    struct node *x = malloc(sizeof(struct node));
    x->next = next;
    return x;
}
```

这里的证明针对满足 `nodes(x)` 的任意输入链表，而不依赖某个固定长度。递归调用只释放尾链表；头节点与尾链表的内存不重叠，因此头节点仍然存在，最后由 `free(x)` 释放。

5.2 用 VeriFast 检查释放后使用错误

如果在 `delete(x)` 中误将 `free(x)` 与 `struct node *next = x->next` 顺序颠倒，会发生什么？

```
void delete_buggy(struct node *x)
    //@ requires nodes(x);
    //@ ensures emp;
{
    if (x != 0) {
        free(x);                        // Consumes x->next |-> ?next and malloc_block_node(x)!
        struct node *next = x->next; // ERROR: Reading released memory (Use-After-Free)!
        delete_buggy(next);
    }
}
```

由于 `free(x)` 立即消耗了字段权限 `x->next |-> ?next`，VeriFast 会在验证阶段拒绝随后的读取操作 `x->next`：

```
error: No matching heap chunks: [_]node_next(x, _)
```

这条错误表示：验证器找不到读取 `x->next` 所需的权限。

6 更复杂的例子：快速排序

链表释放中，我们把头节点和尾链表分开；快速排序先选取一个基准值（pivot），再把数组拆为左子数组、**pivot** 单元和右子数组。递归调用只修改交给它的区域，其余不重叠的区域保持不变。这里除了内存安全，还需要证明两件事：结果有序，并且每个值的出现次数不变。

6.1 主过程：只递归处理 pivot 两侧

下面的实现使用数组最后一个元素作为 pivot。`partition` 将其他元素分到它的两侧，返回它最终所在的下标 p，随后递归排序两侧的子数组：

```
void quicksort(int *a, int n)
{
    if (n <= 1) return;
    int p = partition(a, n);
    quicksort(a, p);
    quicksort(a + p + 1, n - p - 1);
}
```

第一处递归处理原数组的区间 $[0, p)$，也就是下标从 0 到 $p-1$ 的元素，长度为 p。这种包含左端、不包含右端的写法称为半开区间。第二处递归的起始地址是 $a+p+1$，长度为 $n-p-1$，对应原数组的 $[p+1, n)$。下标 p 处的 **pivot** 不属于任何一个递归子数组，因此后续递归不会再移动它。

例如，对数组 $[5,2,9,1,5,6]$ 的递归分解可逐层写成：

$$
\begin{aligned}
[5,2,9,1,5,6] &\longrightarrow [5,2,1,5] \mid \boxed{6} \mid [9],\\
[5,2,1,5] &\longrightarrow [2,1] \mid \boxed{5} \mid [5],\\
[2,1] &\longrightarrow [] \mid \boxed{1} \mid [2].
\end{aligned}
$$

把各层已经固定的 pivot 和排序后的两侧接起来，得到 $[1,2,5,5,6,9]$。中间出现两个 5，说明这里要求的是非递减有序，并且必须保留重复元素。

6.2 `partition`：扫描并将 pivot 放到分界处

```
int partition(int *a, int n)
{
    int pivot = a[n - 1];
    int i = 0;
    for (int j = 0; j < n - 1; j++) {
        int aj = a[j];
        if (aj < pivot) {
            if (i < j) swap(a, i, j);
            i++;
        }
    }
    if (i < n - 1) swap(a, i, n - 1);
    return i;
}
```

`partition` 要求 $n > 0$，因此读取 $a[n-1]$ 有意义。主过程只在 $n > 1$ 时调用它。j 是当前待检查位置，i 是已处理区域中**“小于 pivot”**与**“不小于 pivot”**的边界；两者都表示下标，而 `pivot` 表示元素值。

扫描只访问 pivot 之前的元素。结束时，$[0, i)$ 中的元素均小于 pivot，$[i, n-1)$ 中的元素均不小于 pivot。最后把 pivot 从 $n-1$ 放到 i，返回 i。若 $i = n-1$，所有其他元素都较小，pivot 已在正确位置，无需交换。

在每次扫描 $a[j]$ 之前，数组可分为以下四部分。这里所有区间均为左闭右开的半开区间，空区间也是允许的。

$$
\underbrace{a[0\ldots i)}_{\text{low: } <pivot} \mid \underbrace{a[i\ldots j)}_{\text{high: } \geq pivot} \mid \underbrace{a[j\ldots n-1)}_{\text{todo: 尚未处理}} \mid \underbrace{a[n-1]}_{\text{pivot}}
$$

初始化。$i = j = 0$ 时，low 和 high 都为空，因而关于它们的大小关系自动成立；所有非 pivot 元素都在 todo 区域。

保持性。设当前读到的元素是 $a_j = a[j]$：

- 若 $a_j \geq pivot$，不交换、不增加 i；当 j 增加 1 时，该元素并入 high 区域。
- 若 $a_j < pivot$ 且 $i < j$，则 $a[i]$ 原来属于 high。交换后，小元素进入 $a[i]$，原来的大元素移到 $a[j]$；再增加 i 和 j，两个已处理区域仍满足各自的界限。

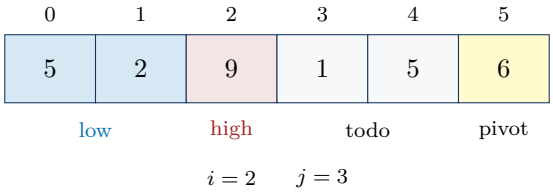
- 若 $a_j < pivot$ 且 $i = j$，high 为空，小元素已经在 low 的下一格；直接增加 i 和 j 即可，不需要把同一格与自己交换。

退出。$j = n-1$ 时，todo 为空；若 $i < n-1$，交换 $a[i]$ 和 pivot 单元，把 pivot 放到 i，并把原来不小于 pivot 的 $a[i]$ 放到最右端。于是

$$\forall k\ (0 \le k < i \to a[k] < a[i]), \qquad \forall k\ (i < k < n \to a[k] \ge a[i]), \qquad 0 \le i < n.$$

这只是分区完成：左右两侧内部还不一定有序，必须继续递归排序。

一次具体扫描。在 $[5,2,9,1,5,6]$ 上取 pivot 为 6，处理完下标 $0,1,2$ 后，$i=2, j=3$。此时四个区域是：



步骤	数组	解释
处理 $j=3$ 前	$[5,2,9,1,5,6]$	小元素为 5, 2；9 已归入 high。
`swap(a,2,3)` 后	$[5,2,1,9,5,6]$	$1<6$，交换后 i 增为 3。
`swap(a,3,4)` 后	$[5,2,1,5,9,6]$	$5<6$，交换后 i 增为 4，扫描结束。
交换 pivot：`swap(a,4,5)`	$[5,2,1,5,6,9]$	返回下标 4；左边四个元素小于 6，右边为 9。

7 快速排序的规约

7.1 在规约中记录数组的范围和值

`quicksort` 的规约同时描述可以访问的数组范围与排序结果：

```
void quicksort(int *a, int n)
    //@ requires a[0..n] |-> ?vs;
    //@ ensures a[0..n] |-> ?ws &*& sorted(ws) && perm(vs, ws);
```

`a[0..n] |-> vs` 表示拥有从 $a[0]$ 到 $a[n-1]$ 的 n 个连续、已经存有整数的内存单元，其当前值按顺序记在列表 `vs` 中。这里不包含 $a[n]$。`?vs` 为输入时的值列表取名为 `vs`，`?ws` 为返回时的值列表取名为 `ws`。这两个列表用于比较排序前后的值，不会在程序运行时另外分配链表；数组发生修改后，`vs` 仍记录原来的值。

这个规约同时要求：返回同一地址范围的完整所有权，输出序列 `ws` 有序，并且 `ws` 与输入序列 `vs` 的每个值出现次数相同。`&*&` 表示内存所有权的分离；`sorted(ws) && perm(vs,ws)` 表示关于值列表的两个条件必须同时成立，它们不额外占有数组单元。

7.2 `sorted`：递归定义非递减次序

```
fixpoint bool le(int x, int y) { return x <= y; }
fixpoint bool ge(int x, int y) { return x >= y; }

fixpoint bool sorted(list<int> vs) {
    switch (vs) {
        case nil: return true;
        case cons(h, t):
            return forall(t, (le)(h)) && sorted(t);
    }
}
```

`fixpoint` 定义在规约中计算结果、但不修改程序内存的函数。`nil` 是空列表；`cons(h,t)` 是把首元素 h 放在列表 t 前面得到的非空列表。`(le)(h)` 把第一个参数固定为 h，所以 `forall(t,(le)(h))` 表示对 t 中的每个元素 v 都有 $h \le v$。空列表和单元素列表都满足 `sorted`；由于使用 $\le$，相等的重复值可以相邻出现。

7.3 `perm`：每个整数的出现次数都保持不变

```
fixpoint bool eq(int x, int y) { return x == y; }

fixpoint bool same_count(list<int> xs, list<int> ys, int v) {
    return count(xs, (eq)(v)) == count(ys, (eq)(v));
}

fixpoint bool perm(list<int> xs, list<int> ys) {
    return forall_int((same_count)(xs, ys));
}
```

`count(xs,(eq)(v))` 统计 v 在 `xs` 中出现的次数；`(same_count)(xs,ys)` 固定两个列表，接着对给定整数 v 检查它在两边的出现次数是否相同。`forall_int(f)` 表示：对每个整数 v，`f(v)` 都成立。因此，`perm(xs,ys)` 要求每个整数在 `xs` 和 `ys` 中的出现次数都相同。 证明 `perm` 才能排除丢失、增加或改变元素的情况。

8 子函数规约：`swap` 与 `partition`

8.1 `swap`

```
void swap(int *a, int i, int j)
    //@ requires a[i] |-> ?vi &*& a[j] |-> ?vj;
    //@ ensures a[i] |-> vj &*& a[j] |-> vi;
{
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

前置条件提供两个内存单元的读写权限，用 `vi`、`vj` 记录它们原来的值；后置条件返还同样两个单元的权限，但交换它们的值。

8.2 `partition`

```
int partition(int *a, int n)
    //@ requires a[0..n] |-> ?vs &*& 0 < n;
    /*@
    ensures 0 <= result &*& result < n &*&
        a[0..result] |-> ?low &*& a[result] |-> ?p &*&
        a[result + 1..n] |-> ?high &*&
        forall(low, (ge)(p)) && forall(high, (le)(p)) &&
        perm(vs, append(low, cons(p, high)));
    @*/
```

这里 `result` 是 pivot 下标，`?p` 用名字 `p` 记录 pivot 的值；而 `quicksort` 程序里的局部变量 `p` 存的是返回的下标。为避免混淆，下面记下标为 $k = \texttt{result}$，pivot 值为 v。

该后置条件包含四层信息：

1. 范围：$0 \le k < n$，因此左右子数组长度 k 和 $n-k-1$ 非负。
2. 内存范围：完整数组拆成 $[0,k)$、单元 k、$[k+1,n)$ 三块，彼此不重叠，合起来恰好覆盖原范围。
3. 大小关系：因为 `(ge)(p)` 将第一个参数固定为 v（对元素 x 检查 $v \ge x$），所以 `forall(low, (ge)(p))` 表示左边每个值 $\le v$；同理 `forall(high, (le)(p))` 表示右边每个值 $\ge v$。
4. 内容保持：`cons(p,high)` 把 pivot 值放在右侧列表前面，`append` 再把左侧列表接到它们前面。`perm` 要求这样合并后的列表与输入列表相比，每个值的出现次数都不变。

8.3 使用 VeriFast 进行模块化验证

VeriFast 按函数分别验证：检查某个函数时，调用其他函数的效果由其前置、后置条件描述；每个被调用函数的实现也必须分别满足自己的规约。