

分离逻辑

1 基本概念

考虑 WhileDeref 语言中有关内存地址的读写，需要新的程序逻辑推理规则。

- 断言 $P * Q$ 表示：可以将程序状态中的内存拆分为互不相交的两部分，其一满足 P 另一个满足 Q ；
- 即，程序状态 s 满足性质 $P * Q$ 当且仅当存在 s_1 与 s_2 使得：
 - s_1 满足 P ,
 - s_2 满足 Q ,
 - $s.\text{vars} = s_1.\text{vars} = s_2.\text{vars}$ 并且 $s.\text{mem} = s_1.\text{mem} \uplus s_2.\text{mem}$
简写为 $s_1 \oplus s_2 \Downarrow s$,
- $P * Q$ 中的星号称为分离合取。

例如：

- `* x == m && * y == n && x != y` 可以写作 `store(x, m) * store(y, n)` ,
- `** x == 0 && * x != x` 可以写作 `exists u. store(x, u) * store(u, 0)` 。

这里，我们使用 `store(a, b)` 表示地址 a 上存储了 b 这个值，并且仅仅拥有此内存权限。

下面是一些霍尔三元组的例子：

```
{ store(0x40, 0) }  
* 0x40 = 0x80  
{ store(0x40, 0x80) }
```

```
{ store(0x40, 0) * store(0x80, 0) }  
* 0x40 = 0x80  
{ store(0x40, 0x80) * store(0x80, 0) }
```

假设 $0 \leq m \leq 100$,

```
{ store(x, m) * store(y, n) }  
* x = * x + 1  
{ store(x, m + 1) * store(y, n) }
```

```
{ store(x, m) * store(y, n) }  
* x = * y  
{ store(x, n) * store(y, n) }
```

直观上，断言 $P * Q * R$ 表示：可以将程序状态中的内存拆分为互不相交的三部分，分别满足 P 、 Q 与 R 。

```
* x == n && * y == m && * z == 0 &&
x != y && y != z && z != x
```

可以写作: `store(x, n) * store(y, m) * store(z, 0)`。

```
* x == 10 && * (x + 8) == u &&
* u == 100 && * (u + 8) == 0 &&
x != u && x + 8 != u && x - 8 != u
```

可以写作: `store(x, 10) * store(x + 8, u) * store(u, 100) * store(u + 8, 0)`。

以下霍尔三元组成立:

```
{ store(x, 10) * store(x + 8, u) * store(u, n) }
* x = * * (x + 8)
{ store(x, n) * store(x + 8, u) * store(u, n) }
```

2 霍尔逻辑规则

内存赋值规则 (正向):

- 如果 P 能推出
 - e_1 能够安全求值并且求值结果为 a
 - e_2 能够安全求值并且求值结果为 b
 - $\exists u. \text{store}(a, u) * Q$,
- 那么 $\{P\} * e_1 = e_2 \{\text{store}(a, b) * Q\}$
- 其中 a 与 b 都是与内存无关的数学式子。

变量赋值规则 (正向):

- 如果 P 能推出 e 能够安全求值并且求值结果为 a ,
- 那么 $\{P\} x = e \{\exists x'. a[x \mapsto x'] = x \ \&\& \ P[x \mapsto x']\}$
- 其中 a 是与内存无关的数学式子。

框架规则:

- 如果 F 中不出现被 c 赋值的变量, 并且 $\{P\} c \{Q\}$,
- 那么 $\{P * F\} c \{Q * F\}$

存在量词规则:

- 如果对于任意 a 都有, 并且 $\{P(a)\} c \{Q\}$,
- 那么 $\{\exists a, P(a)\} c \{Q\}$

3 例子 - 交换地址上的值

```
{ store(x, m) * store(y, n) }
  t = * x;
  * x = * y;
  * y = t
{ store(x, n) * store(y, m) }
```

4 例子 - 单链表取反

下面程序描述了单链表取反的过程。

```
while (x != 0) do {
  t = x;
  x = * (x + 8);
  * (t + 8) = y;
  y = t
}
```

利用分离逻辑，我们可以定义一些新的谓词，从而简洁描述数据结构。以单链表为例：

- 可以用谓词 $\text{sll}(p)$ 表示以 p 地址为头指针存储了一个单链表
- $\text{sll}(p)$ 定义为：

```
p == 0 && emp ||
exists u q, store(p, u) * store(p + 8, q) * sll(q)
```

利用 $\text{sll}(p)$ 谓词，可以如下描述单链表取反程序的内存安全性性质：

```
{ y == 0 && sll(x) }
while (x != 0) do {
  t = x;
  x = * (x + 8);
  * (t + 8) = y;
  y = t
}
{ x == 0 && sll(y) }
```

可以选用 $\text{sll}(x) * \text{sll}(y)$ 作为循环不变量。

首先，前条件可以推出循环不变量。

```
y == 0 && sll(x)
|-- y == 0 && emp * sll(x)
|-- (y == 0 && emp) * sll(x)
|-- sll(y) * sll(x)
|-- sll(x) * sll(y).
```

其次，循环体能保持循环不变量。

```

{ x != 0 && sll(x) * sll(y) }
{ exists u z, store(x, u) * store(x + 8, z) * sll(z) * sll(y) }
  // Given u z,
{ store(x, u) * store(x + 8, z) * sll(z) * sll(y) }
  t = x;
{ t == x && store(x, u) * store(x + 8, z) * sll(z) * sll(y) }
  x = * (x + 8);
{ exists x', x == z && t == x' && store(x', u) * store(x' + 8, z) * sll(z) * sll(y) }
{ x == z && store(t, u) * store(t + 8, z) * sll(z) * sll(y) }
  * (t + 8) = y;
{ x == z && store(t, u) * store(t + 8, y) * sll(z) * sll(y) }
  y = t
{ exists y', y == t && x == z && store(t, u) * store(t + 8, y') * sll(z) * sll(y') }
{ exists y', store(y, u) * store(y + 8, y') * sll(x) * sll(y') }
{ sll(x) * sll(y) }

```

最后，退出循环时后条件成立。

```

!(x != 0) && sll(x) * sll(y)
|-- (x == 0) && sll(x) * sll(y)
|-- (x == 0) && sll(y).

```

5 例子 - 单链表的连接 1

下面程序描述了单链表的连接。

```

if (x == 0)           第一个
then {               分支
    res = y
}
else {
    res = x;           第二个
    nx = * (x + 8);    分支, x
                       不为空
    while (nx != 0) do {
        x = nx;
        nx = * (x + 8)
    };
    * (x + 8) = y
}

```

```

/*@ require sll(x) * sll(y)  初始条件
/*@ ensure sll(res)
if (x == 0)
then {
    res = y
}
else {
    res = x;
    nx = * (x + 8);
    while (nx != 0) do {
        x = nx;
        nx = * (x + 8)
    };
    * (x + 8) = y
}

```

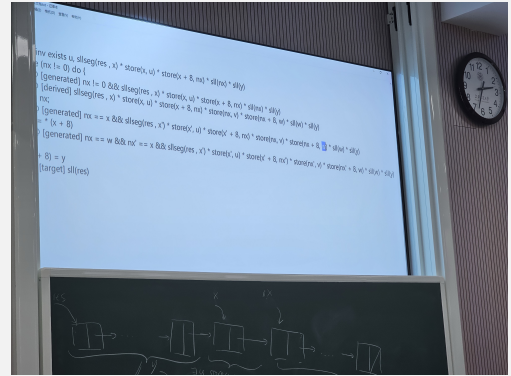
- 可以用谓词 `sllseg(p, q)` 表示以 `p` 地址为头指针开始到 `q` 为止是单链表中的一段

- `sllseg(p, q)` 定义为:

这一段为空, 下一个就是p或q

```
p == q && emp ||
exists u r, store(p, u) * store(p + 8, r) * sllseg(r, q)
```

```
//@ require sll(x) * sll(y)
//@ ensure sll(res)
if (x == 0)
then { res = y }
else {
  res = x;
  nx = * (x + 8);
  //@ [generated] v == nx && x == res && store(x, u) *
    store(x + 8, v) * sll(v) * sll(y)
  //@ inv exists u, sllseg(res, x) * store(x, u) *
    store(x + 8, nx) * sll(nx) * sll(y)
  while (nx != 0) do {
    x = nx;
    nx = * (x + 8)
  };
  * (x + 8) = y
}
```



6 例子 - 单链表的连接 2

习题 1.

下面是用二阶指针实现的单链表连接。

Require store(head, Uninitialized) * sll(x) * sll(y)

```
* phead = x;
pt = phead;
while ( * pt != 0 ) {
  pt = * pt + 8;
};
* pt = y;
res = * phead
```

请证明, 其满足:

```
{ exists u, store(phead, u) * sll(x) * sll(y) }
* phead = x;
pt = phead;
while ( * pt != 0 ) {
  pt = * pt + 8;
};
* pt = y;
res = * phead
{ exists u, store(phead, u) * sll(res) }
```

generated中的exist可以去掉, inv和ensure里的exist不能扔掉, 因为会当后条件

证明中应当指明循环语句的循环不变量, 每条赋值语句的最强后条件, 以及每次使用 Consequence rule 改写前后的断言。另外, 证明中可以考虑使用下面谓词:

P q 为二阶指针

- `SllSeg(p, q)` 定义为:

```
p == q && emp ||
exists u r, store(p, r) * store(r, u) * SllSeg(r + 8, q)
```

R + 8 是一个二阶指针

7 例子 - 双向链表的操作