霍尔逻辑

断言：就是一个描述程序状态的数学性质

布尔表达式一定能算出来，数学命题不一定能算出来（断言）断言表示的东西为静态的，但是实际的程序语言的表达式可以更改程序状态（不是断言）

布尔表达式可能有side-effect，可以在内部调用函数

1 霍尔三元组

对于任意程序状态 s_1 与 s_2，如果 s_1 满足性质 P 并且 $(s_1, s_2) \in [\![c]\!]$，那么 s_2 满足性质 Q。这一性质写作 $\{P\}c\{Q\}$，称为霍尔三元组，P 称为前条件，Q 称为后条件。

```
{ x == 0 }
y = 0;
while (y < 6) do {
  x = x + y;
  y = y + 1
}
{ x == 1 + 2 + 3 + 4 + 5 }
```

P = false, 则均为三元组

True
while(I) do skip
False是三元组

2 顺序执行规则、空语句规则与条件分支语句规则

下面是顺序执行规则：

如果 $\{P\}c_1\{Q\}$ 并且 $\{Q\}c_2\{R\}$，那么 $\{P\}c_1; c_2\{R\}$。

下面是一个顺序执行规则的例子。我们想要证明，下面程序确实交换了变量`x`与`y`的值。

```
{ x == m && y == n }
t = x;
x = y;
y = t
{ x == n && y == m }
```

首先，下面的霍尔三元组显然为真。

```
{ x == m && y == n }
t = x
{ t == m && y == n }
```

M，n都是逻辑变量；

```
{ t == m && y == n }
x = y
{ t == m && x == n }
```

```
{ t == m && x == n }
y = t
{ x == n && y == m }
```

由其中的第二第三条可以得到：

{True}
C
{False} 这段程序在任何情况下都运行不终止。

```
{ t == m && y == n }
x = y;
y = t
{ x == n && y == m }
```

最后再用顺序执行规则和第一条霍尔三元组可知：

```
{ x == m && y == n }
t = x;
x = y;
y = t
{ x == n && y == m }
```

下面是空语句规则：

$\{P\}$ skip $\{P\}$

下面是条件分支语句规则：

如果 $\{P\&\&e\}c_1\{Q\}$ 并且 $\{P\&\&!e\}c_2\{Q\}$，那么

$$\{P\} \text{ if } (e) \text{ then } \{ c_1 \} \text{ else } \{ c_2 \} \{Q\}$$

{True}
If (x <= 0)
Then y = 2
Else y = x + 1
{x <= y}

3 While 语句规则与循环不变量

While 语句规则

如果 $\{ P \&\& e \}$ c $\{ P \}$，那么

$$\{ P \} \text{ while } (e) \text{ do } \{ c \} \{ P \&\& !e \}$$

说明 {P} c1 {Q}, {P} c2 {Q} 的证明能力是不够的

跳出循环的时候e一定不成立

其中，我们一般会称 P 为循环不变量。

例如，要证明下面霍尔三元组

```
{ x == 0 }
while (x < 10) do
{
  x = x + 1
}
{ x == 10 }
```

就只需证明：

```
  P
{ x <= 10 }
while (x < 10) do
{
  x = x + 1
}                 e
{ x <= 10 && ! (x < 10) }
```

这又可以被规约为：

```
{ x <= 10 && x < 10 }
  x = x + 1
{ x <= 10 }
```

习题 1. 如何证明：

```
{ x >= 0 }
while (0 < x) do {
  x = x - 1
}
{ x == 0 }
```

{x >= 0 && x > 0}
X = x – 1
{x >= 0}

下面通过一个例子分析如何找循环不变量。

假如 m 与 n 是给定正整数，如何证明：

```
{ x == m && y == 0 }
while (! (x < n)) do {
  x = x - n;
  y = y + 1
}
{ n * y + x == m && 0 <= x < n }
```

!e

考虑`n`的值为 3、`m`的值为 10 的具体情况。

```
{ x == 10 && y == 0 }
x = x - 3;
y = y + 1
{ x == 7 && y == 1 }
```

```
{ x == 7 && y == 1 }
x = x - 3;
y = y + 1
{ x == 4 && y == 2 }
```

```
{ x == 4 && y == 2 }
x = x - 3;
y = y + 1
{ x == 1 && y == 3 }
```

循环不变量 P 应当使得下面断言能推出 P：

```
x == 10 && y == 0 ||
x == 7 && y == 1 ||
x == 4 && y == 2 ||
x == 1 && y == 3
```

任意一种都可以推出循环不变量

否则以下两个条件总有一个不成立：

- 循环的前条件能推出 P；
- { P &&循环条件 } 循环体 { P }。

另一方面，循环不变量 P 也不能太弱，否则

$$P \;\&\&\; !\text{循环条件}$$

不足以推出循环整体的后条件。

结合这两点考虑，我们在刚才的例子中可以取循环不变量 P 为：

```
x == 10 && y == 0 ||
x == 7 && y == 1 ||
x == 4 && y == 2 ||
x == 1 && y == 3
```

不难检查，它满足下面三条性质：

- 循环前条件`x == 10 && y == 0`能推出 P
- 循环体能保持循环不变量

```
{ P && ! (x < 3) }
x = x - 3; y = y + 1
{ P }
```

- `P && ! ! (x < 3)`能推出循环后条件`3 * y + x == 10 && 0 <= x < 3`。

当然，循环不变量也可以是弱一些的断言，例如：`3 * y + x == 10 && 0 <= x`
构造循环不变量的一般思路如下：

- 循环的前条件要能推出循环不变量
- 循环不变量不能太强，至少要保证程序运行中每次循环体执行结束后的程序状态应当满足循环不变量，否则循环体不满足霍尔三元组：

$$\{ P \ \&\& \ e \}c\{ P \};$$

- 循环不变量不能太弱，否则 $P \ \&\& \ !e$ 不足以推出循环的后条件；
- 通常情况下，可以考虑选择一个循环不变量，使得满足这个循环不变量的程序状态恰好是所有循环体执行结束之后的程序状态。

习题 2. 假如 m 与 n 是给定正整数，如何证明：

```
{ x == m }
while (! (x < n)) do {
  x = x - n
}
{ exists y'. n * y' + x = m && 0 <= x < n }
```

所有执行完一次循环体到达的状态，并起来，概括为一个性质，就可以叫做循环不变量

习题 3. 假如 m 与 n 是给定正整数，如何证明：

```
{ x == m && y == n }
while (! (! (x < 0) && ! (0 < x))) do {
  y = y - 1;
  x = x - 1
}
{ y == n - m }
```

x!= 0 的时候进行循环

y − x = n − m

关键在于从所有可能的 其实状态出发，执行循环体一次或多次能够到达的集合，把他们放在一起，用程序断言去刻画性质，得到一个循环不变量

习题 4. 假如 m 是给定正整数，如何证明：

```
{ x == m && i == res == 0 }
while (i < x) do {
  res = res + x;
  i = i + 1
}
{ res == m * m }
```

P ：= x == m && Res = I * m && m >= i

P && (x <= i) => (res == x * x)

4 变量赋值语句规则（正向）与最强后条件

如何严格定义最佳后条件？

把所有的有用信息都表示出来

- $\{P\}c\{Q\}$ 成立；
- 如果 $\{P\}c\{Q'\}$ 成立，那么 Q 能推出 Q'；

亦称为最强后条件：strongest postcondition。

x >= 0 && x = y

习题 5. 对于前条件`0 <= y`与程序`x = y`，它们的最强后条件是什么？

X >= 0 && n * (y + 1) + x == m && x >= –n

习题 6. 对于前条件`n * y + x == m && 0 <= x && n <= x`与程序`x = x - n`，它们的最强后条件是什么？

习题 7. 对于前条件`n * y + x + n == m && 0 <= x`与程序`y = y + 1`，它们的最强后条件是什么？X >= 0 && n

习题 8. 对于前条件`x == n && y == m`与程序`x = x + y`，它们的最强后条件是什么？ y == m && x = n + m

变量赋值规则（正向）：

$$\{P\}\ x = e\ \{\exists x'.\ e[x \mapsto x'] = x\ \&\&\ P[x \mapsto x']\}$$

P: N * y + x == m && 0 <= x < n
cmd:Y = 0
Q: ??

这里，我们用 $P[x \mapsto x']$ 表示将断言 P 中出现的每一个 x 都替换成为 x'。例如：

Y = 0 && 0 <= x < n &&
exists y', y' * N + x == m

- $(x \geqslant 0)[x \mapsto x']$ 表示 $x' \geqslant 0$，
- $(\exists k.\ x = a_k)[x \mapsto x']$ 表示 $\exists k.\ x' = a_k$，
- $(x + y)[x \mapsto x']$ 表示 $x' + y$。

  请注意，此处 $(x + y)$ 是一个 SimpleWhile 语言的表达式（变量加变量），将其中的 x 替换为 x' 之后得到的 $(x' + y)$ 还是一个程序表达式（常量加变量），它的求值结果是 $x' + y$，这里的加号是数学上的加号。

- $y[x \mapsto x']$ 表示 y。
- $(x + 1)[x \mapsto x']$ 表示 $x' + 1$。

例如

```
{ x == m && y == n }
x = x + y
{ exists x'. x' + y == x && x' == m && y == n }
```

X’ 为旧值

```
{ x == m && y == n }
temp = x
{ exists temp'. x == temp && x == m && y == n }
```

Floyd 推理规则

习题 9. 对于前条件`x == temp == m && y == n`与程序`x = y`，它们的最强后条件是什么？

存在x', x’ == temp == m && y = n && x = y

5 变量赋值语句规则（反向）与最弱前条件

后条件 Q 与程序 c 的最弱前条件 P 是指满足下面条件的断言 P。

- $\{P\}c\{Q\}$ 成立；
- 如果 $\{P'\}c\{Q\}$ 成立，那么 P' 能推出 P；

例如，后条件`temp == m && y == n`与程序`temp = x`的一个最弱前条件是：`x == m && y == n`。

下面我们写出变量赋值规则（反向）：

那么 $\{\ P[x \mapsto e]\ \}\ x = e\ \{\ P\ \}$

6 可靠性证明

7 导出规则

除了上述霍尔逻辑规则之外，其实也可以在保持逻辑可靠性的基础上增加一些其他的规则。例如，我们可以增加单侧的 Consequence 规则。

```
Lemma hoare_conseq_pre_sound:
  forall (P P' Q: assertion) (c: com),
    valid {{ P' }} c {{ Q }} ->
    P |-- P' ->
    valid {{ P }} c {{ Q }}.
(* 证明详见Coq源代码。 *)

Lemma hoare_conseq_post_sound:
  forall (P Q Q': assertion) (c: com),
    valid {{ P }} c {{ Q' }} ->
    Q' |-- Q ->
    valid {{ P }} c {{ Q }}.
(* 证明详见Coq源代码。 *)
```

然而，我们并不需要将其添加到霍尔逻辑的原始规则（primitive rules）集合中去。因为，这一规则可以由双侧的 Consequence 规则导出。

```
Lemma hoare_conseq_pre:
  forall (P P' Q: assertion) (c: com),
    provable {{ P' }} c {{ Q }} ->
    P |-- P' ->
    provable {{ P }} c {{ Q }}.
```

下面的证明即是导出证明。

```
Proof.
  intros.
  apply (hoare_conseq P P' Q Q).
  + tauto.
  + tauto.
  + unfold derives.
    intros; tauto.
Qed.
```

上面证明中用到了 `Q |-- Q` 这一性质。之后的证明中还会用到许多关于断言推导的命题逻辑性质。证明中可以使用 `assn_tauto` 指令用于证明。具体而言，`assn_tauto H1 H2 ... Hn` 表示在将 `H1` 等前提考虑在内的情况下使用命题逻辑证明。

```
Lemma hoare_conseq_post:
  forall (P Q Q': assertion) (c: com),
    provable {{ P }} c {{ Q' }} ->
    Q' |-- Q ->
    provable {{ P }} c {{ Q }}.
Proof.
  intros.
  apply (hoare_conseq P P Q Q').
  + tauto.
  + assn_tauto.
  + assn_tauto H0.
Qed.
```

类似的，可以用变量赋值规则（正向）与顺序执行规则导出下面规则。在 Coq 证明中，`eapply` 表示使用 `apply` 但是相关参数暂时空缺。

```
Lemma forward_symbolic_exe:
  forall P x e c Q,
    provable {{ exists x',
                  exprZ_subst [[ e ]] x [[ x' ]] == [[ x ]] &&
                  assn_subst P x [[ x' ]] }} c {{ Q }} ->
    provable {{ P }} x = e; c {{ Q }}.
Proof.
  intros.
  eapply hoare_seq.
  + apply hoare_asgn_fwd.
  + apply H.
Qed.
```