课后阅读：符号执行

基于霍尔逻辑，我们就可以使用程序断言标注来表示一段程序的正确性证明。例如下面例子演示了我们可以使用断言表述程序的正确性性质和证明框架。

首先，人们通常会用 `require` 与 `ensure` 条件描述一段程序执行前的假设以及执行后的保证。这其实就对应着霍尔三元组中的前后条件。其次，循环不变量往往也是很难通过自动推导得到的，所以这也需要事先通过 `inv` 标注提供。

```
//@ require true
//@ ensure x == 10
x = 0;
//@ inv x <= 10
while (x < 10) do
{
  x = x + 1
}
```

事实上，上面这样标注的程序就已经表述了一个霍尔逻辑证明。例如，利用赋值语句规则可以生成 `x = 0` 这一语句的最强后条件，如下面 `generated` 断言所示。

```
//@ require true
//@ ensure x == 10
x = 0;
//@ [generated] exists x', 0 == x && true
//@ inv x <= 10
while (x < 10) do
{
  x = x + 1
}
```

从整体来看，整个程序是用分号连接的顺序执行程序，分号前是 `x = 0` 分好后是一个 `while` 循环语句。要证明这个程序符合 `require` 与 `ensure` 断言描述的霍尔三元组就需要顺序执行规则，以上面自动生成的 `generated` 断言为中间条件。在需要证明的两个霍尔三元组中（如下所示），前一个是赋值语句规则的直接结论，后一个需要我们继续证明。

```
{ true }
  x = 0;
{ exists x', 0 == x && true }
```

```
{ exists x', 0 == x && true }
  while (x < 10) do
  {
    x = x + 1
  }
{ x == 10 }
```

这里我们可以看到，存在量词只出现在后续证明的前提中，所以我们也就可以省去此处存在量词，并简写

为：

```
//@ require true
//@ ensure x == 10
x = 0;
//@ [generated] 0 == x && true
//@ inv x <= 10
while (x < 10) do
{
  x = x + 1
}
```

程序中的 `inv` 标注要求我们用 `x <= 10` 为循环不变量进行霍尔逻辑证明，这里我们首先需要检查 `0 == x && true` 这一断言能否推出循环不变量。即检查以下断言推导是否成立：

```
0 == x && true |-- x <= 10
```

在整个证明的过程中，会生成许多这样的断言推导性质需要检查，这些性质都成为“验证条件”，英文名为 Verification Condition，简称 VC。

完成上述检查后，就需要验证循环体的安全性（或功能正确性），循环体的前条件（下面阴影部分的第一个 `generated` 断言）和后条件（下面阴影部分的 `target` 断言）都是由循环不变量确定的。要证明由这两个断言表述的霍尔三元组，我们可以对循环体赋值语句再做一次最强后条件推导，这里我们直接省去后续不再需要的存在量词（见下面阴影部分的第二个 `generated` 断言）。

```
//@ require true
//@ ensure x == 10
x = 0;
//@ [generated] 0 == x && true
//@ inv x <= 10
while (x < 10) do
{
  //@ [generated] x <= 10 && x < 10
  x = x + 1
  //@ [generated] x' + 1 == x && x' <= 10 && x' < 10
  //@ [target] x <= 10
}
```

这里就又产生了一个验证条件需要检查。

```
x' + 1 == x && x' <= 10 && x' < 10 |-- x <= 10
```

在完成循环体相关的验证后，我们也就完成了循环的验证。循环的后条件可以依据循环不变量直接写出。

```
//@ require true
//@ ensure x == 10
x = 0;
//@ [generated] 0 == x && true
//@ inv x <= 10
while (x < 10) do
{
  //@ [generated] x <= 10 && x < 10
  x = x + 1
  //@ [generated] x' + 1 == x && x' <= 10 && x' < 10
  //@ [target] x <= 10
}
//@ [generated] x <= 10 && !(x < 10)
```

到这里为止，验证还没有结束。我们还需要检查，这一循环的后条件是否可以推出整个程序的 ensure 后条件。

```
x <= 10 && !(x < 10) |-- x == 10
```

上述验证过程就是符号执行（Symbolic Execution），其中我们需要检查下面三个验证条件。

```
0 == x && true |-- x <= 10
```

```
x' + 1 == x && x' <= 10 && x' < 10 |-- x <= 10
```

```
x <= 10 && !(x < 10) |-- x == 10
```