Logic in Computer Science

Part.0 : Program ↔ Logic

Programming Language is the real world application of Logic.

程序设计语言理论(以下简称PL)，简而言之，就是有关如何设计一门程序语言的学问。

起源于Turing & Church，从λ Calculus到Turing Machine，从函数式语言到指令式语言，从Predicate Logic到Linear Logic，我们会发现自诞生以来，PL和Logic之间就存在着密不可分的联系。

一门程序语言，可以看作是Software Abstraction of Logic. 它是对底层逻辑电路的一种软件抽象。

也正因为如此，当我们在学习数字电路或是计算理论这类与Software Foundations密切相关的学科时，Logic几乎无处不在。

尽管如今大多数高级语言在使用上已经与底层或理论上的Logic产生了一定的隔断，但在考虑程序本身的正确性时，却往往需要Logic的参与。

本文的一大主题便是如何利用Logic来验证程序的正确性。

当谈及Logic in Computer Science这一Topic时，程序验证和模型检查领域往往是逃不开的话题。

除此之外，当我们在使用Haskell或是Rust这样的语言编写程序时，熟悉类型系统往往是实现高效编程所必需的。

一门程序语言的类型系统以及衍生出的各种类型多态、泛型的本质也与Logic直接相关。因此，我们也将介绍类型论这一有趣的Logic分支。

在此基础上，我们将尝试形式化验证一个常用算法——插入排序的正确性。

最后，为了处理涉及到指针和并发的程序验证，我们将引入Separation Logic.

本文所有示例均配备Coq形式化证明脚本，正确性能够得到验证。

Part.1 : Program Verification

在软件工程领域，用于确保程序正确性的手段有很多，大体可分为两类：动态方法和静态方法。

动态方法需要将软件源码编译为可执行文件，再运行软件，通过特定的输入考察软件的运行结果。

动态方法中使用得比较广泛的是软件测试，有手动其测试（如单元测试）、自动化测试（如模糊测试）等等。

动态方法效率高，错误定位准确，但应用场合受限，它必须要求具有程序能运行起来的环境。而且，其最大的问题在于，动态方法不能覆盖程序的所有输入和全部的执行路径，故只能证明被测试所覆盖的部分程序没有问题，而不能证明整个程序不存在错误。

静态方法则是基于程序源代码进行分析，不需要编译运行，所以应用场景更加广泛。常用的静态方法有静态分析、符号执行、定理证明、（自动化）程序验证等等。不同的静态方法各有优劣，其应用门槛也比较高，一般用于对程序正确性要求较高的场景。

常规的静态分析方法比较轻量级，适用于大规模代码，但比较大的问题是较高比例的误报。所以静态分析报出的结果，需要逐一人工排查，从而导致耗费大量的人力资源。静态分析引擎比较有名的是Coverity，目前已经实现较大规模商业化应用。

符号执行可以看作是更加准确的测试方法，它通过符号值来静态「执行」程序，积累路径条件，直到到达目标位置，再对路径条件进行约束求解，判断目标位置的可达性。由于需要使用约束求解，而且对循环不友好，所以符号执行方法比较难以大规模应用。符号执行比较经典的工具是KLEE.

定理证明方法是使用高阶逻辑对程序及其需要满足的性质进行建模，然后使用机器辅助证明的方法，一步一步证明程序能够满足要求的性质。定理证明方法主要的缺陷是需要大量的专业人力参与，编写证明代码，而且对软件的快速更新迭代不友好。辅助定理证明的典型工具是Coq.

在本文中，程序验证是指能够自动化证明程序的正确性的方法。这里的正确性包括两个方面：程序在运行的过程中不会出错并且程序的功能可以被满足。

想要验证一个程序的正确性，就要从理解这个程序的语义开始。

对于一个可执行的C语言程序（或其他高级语言程序）而言，比较直观的语义理解应该是，内存中有一部分存储着程序的二进制指令，然后有一个指向当前执行指令的特殊指针变量（即程序计数器），还有一部分存储着每个变量的值（栈和堆）。

随着程序指令的依次执行，内存的状态发生变化，比如赋值语句导致变量的值改变，控制流语句产生指令跳转，导致计数器发生变化。

将这个过程形式化地定义出来，就是直观的操作语义。它定义了一个状态机，将程序每一步执行所产生的影响，反应到状态机的状态变化上。

这样的程序语义和程序的实际运行情况非常切合，但却不利于逻辑和数学表达。因此，在进行程序分析和验证时，我们会求诸于更加贴近于数学表达的公理语义。

而这引出了程序验证的理论基础Hoare Logic.

霍尔逻辑是一种典型的建立在谓词逻辑基础上的公理语义。在其基础上，我们可以在程序代码和谓词逻辑公式之间，建立起「等语义关系」的转化，从而确保我们的程序验证方法是有效的。

霍尔逻辑的核心概念是**霍尔三元组**，一般记为以下形式：

$$\{P\}c\{Q\}$$

其中，P和Q是一阶逻辑公式，分别表示前置条件和后置条件，c表示一段程序源代码。

霍尔三元组的含义是：假定前置条件P是有效的，那么在执行完程序c后，后置条件Q将会是有效的。

例如，我们可以非形式化地写出以下霍尔三元组：

$$\{a,b\in\mathbb{Z}\}c=(a+b)^2\{c=a^2+2ab+b^2\}$$

这个霍尔三元组说明，只要输入是两个整数a,b且有$c=(a+b)^2$，那么$c=a^2+2ab+b^2$.

在霍尔逻辑的帮助下，我们能够将程序以及属性（也就是程序的功能）转换为一阶逻辑公式进行求解。

程序验证的优点是能够自动化地进行程序的正确性证明。但其缺点也很显著：其一是对需要使用复杂逻辑描述的功能属性不友好，一般用于证明一些低阶属性，比如程序中是否存在「除0错误」、「悬垂指针」、「数组越界」和「缓冲区溢出」等问题；其二是程序验证一般也依赖于约束求解，所以同样难以直接大规模地应用。但随着验证算法和约束求解引擎的不断进步，程序验证的可用性将变得越来越好。

一般来说，程序验证会遵循以下四个步骤：

1. 给出程序源码;
2. 标注我们需要验证的**属性**;
3. 将我们需要验证的问题转换为逻辑公式;
4. 证明逻辑公式的正确性.

也就是说，程序的正确性最后由它所表征的逻辑公式的真假来决定。

例如，考虑一个简单的C程序：

```c
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d %d", &a, &b);
    a = a + b;
    b = a - b;
    a = a - b;
    printf("%d %d", a, b);
    return 0;
}
```

这段代码希望实现的**功能属性**是：交换`a`和`b`两个整型的值。

遵照上面的流程，我们可以尝试验证这个程序：

在以上代码中，我们需要证明，对于任意的`int a`和`int b`，经过执行代码后，两者的值能够互换。

我们遇到的第一个问题就是如何用数学公式来表达C语言中的赋值语句`a = a + b`.

这也是许多初次接触编程的人遇到的第一个问题，即数学上的「等号」和程序语言中的「赋值」在语义上的区别。

各位应该都能理解这两者的区别，不过现在我们需要在这两者之间建立联系。

为此，我们将程序转换为**静态单赋值形式**(简称SSA)，即每个变量都仅会被定义和赋值一次。

转换的方法非常简单：每次对某个变量进行写操作时，我们都引入一个新的别名，来代替这个变量。而在后续的程序中使用这个变量时，我们会用这个别名。

例如，我们会把`a = a + b`转化为`a1 = a + b`，而在`b = a - b`中实际引用的是`a1`，所以会被转为`b1 = a1 - b`.

以下是对整个程序的SSA转化：

```
// SSA
int a0 = a;
int b0 = b;
a1 = a + b;  // a → a1
b1 = a1 - b;  // b → b1
a2 = a1 - b1;  // a1 → a2
assert(a2 == b0 && b1 == a0);
```

SSA程序的优点在于，程序中的「等号」和「赋值」能够不加区分地等同起来，也就是说数据是不可变的，熟悉函数式编程的读者应该较为清楚。

程序中的每一条语句，都对应一条逻辑公式，比如赋值语句`a1 = a + b`对应$a_1 = a + b$. 从而，我们可以将SSA程序直接转化为逻辑公式：

$$\forall a, b \in int.\ a_0 = a \wedge b_0 = b \wedge$$
$$a_1 = a + b \wedge b_1 = a_1 - b \wedge a_2 = a_1 - b_1$$
$$\Rightarrow$$
$$a_2 = b_0 \wedge b_1 = a_0$$

如上公式的蕴含符号($P \Rightarrow Q$)的前一部分(P)是对输入以及程序本身的描述,本例中对输入要求为int；后一部分(Q)是对程序的功能属性的描述。

该公式描述了：对于所有的输入变量，只要其值满足对输入的要求和程序的执行逻辑约束(蕴含运算的前提为真)，那么在程序运行后，要求的属性也应当被满足(蕴含运算的结论为真)。

因此，只要以上逻辑公式是永真的，那所要验证的属性也就是成立的。

将得到的逻辑公式代入化简不难得到：$P \Rightarrow Q \leftrightarrow P \Rightarrow P$. 这个式子毫无疑问是永真的，因此可以确信这段程序能够满足要求。

从以上的过程中，可以发现，程序源码到SSA是可以自动化进行的；从SSA到逻辑公式也是可以自动化进行的；而证明逻辑公式的正确性，也可以借助于约束求解器来自动化实现。

所以，在程序验证中，理论上(似乎)只要提供程序源码和描述属性的断言，就可以通过自动化的方法，证明属性成立。

单纯从这样一个简单的程序，不足以说明逻辑在程序验证中的威力，不妨给原程序加上一个分支结构：

```c
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    if (a ≥ b) {
        printf("%d\n", a);
    } else {
        a = a + b;
        b = a - b;
        a = a - b;
        printf("%d\n", a);
    }
```

```
    return 0;
}
```

改写为更接近自然语言的伪代码：

```
if(a1 < a2) {
    a1 = a1 + a2;
    a2 = a1 - a2;
    a1 = a1 - a2;
}
assert(a1 ⩾ a2);
```

以上程序实际上有两条路径，即`if`语句的`a1 < a2`和`a1 >= a2`两个分支。所以，我们需要对两个分支分别处理。其中，`a1 < a2`分支的代码对应的SSA为：

```
// SSA 1
assume(a1 < a2);  // assumption
a1_1 = a1 + a2;  // a1 → a1_1
a2_1 = a1_1 - a2;  // a2 → a2_1
a1_2 = a1_1 - a2_1;  // a1_1 → a1_2
assert(a1_2 ⩾ a2_1);
```

我们将进入这个分支所要求满足的条件`a1 < a2`用语句`assume(a1 < a2)`来表示。我们将`assume(a1 < a2)`的语义暂时解释为为假定条件`a1 < a2`成立，再执行后面的代码。将以上SSA代码转化为逻辑公式可得：

$$\forall a_1, a_2.\ \ a_1 < a_2 \wedge$$
$$a_{11} = a_1 + a_2 \ \wedge\ a_{21} = a_{11} - a_2 \ \wedge\ a_{12} = a_{11} - a_{21}$$
$$\Rightarrow$$
$$a_{12} \geq a_{21}$$

代入化简得到：

$$\forall a_1, a_2.\ \ a_{21} < a_{12}$$
$$\Rightarrow$$
$$a_{12} \geq a_{21}$$

这个式子显然也是永真的。所以在`a1 < a2`分支下，我们证明了属性成立。而在原程序`if`语句的另一个分支`a1 >= a2`中，属性也显然是成立的。

```
// SSA 2
assume(a1 ⩾ a2);
assert(a1 ⩾ a2);
```

所以，我们证明了整个程序片段的功能属性的正确性。

我们已经看到了程序验证中对顺序程序，以及带分支程序的处理。在接下来的例子中，我们将演示程序验证对带循环程序的处理。

给出一个简单的例子：

```
#include <stdio.h>
#define N 10

int main() {
    int i;
    scanf("%d", &i);
    if (i < N) {
        while(i < N) {
            i = i + 1;
        }
        printf("%d\n", i == N ? 1 : 0);
    }
    return 0;
}
```

现在需要验证的属性是：符合条件的i经过循环程序的处理后最后是否一定和N相等？

验证带循环的程序最大的问题在于，我们无法直接将循环转写为逻辑公式，因为循环语句无法直接转为SSA的形式。

然而，循环恰恰是一般程序的重要组成部分（这也是程序语言的图灵完备性的关键所在）。

所以，我们首先要解决循环。注意到在循环入口处，i是小于N的。

同时，在循环执行的过程中，i的值始终不会超过N，即在循环头位置，我们始终有$\mathtt{i} \in (-\infty, \mathrm{N}]$.

我们将这个式子称为**循环不变式**，即在任意多次循环执行后，位于该位置的这个式子始终是成立的。

有了这个信息，我们就可以使用循环头位置的循环不变式，来替代循环实际执行的效果，进而表示循环的执行对程序变量的值的影响。

循环头位置的循环不变式意味着，不论循环执行多少次，对程序变量的值所造成的影响，都会在循环不变式所约束的范围内。需要注意的是，循环头位置的循环不变式还有另一层含义，即从循环之前的代码执行到循环头时，循环不变式也要成立。

这实际上可以看作是循环执行0次（即循环还未执行）的情况。

所以，该循环不变式也包含了循环头之前的代码`(if (i < N))`的执行，对程序变量的值产生的影响。

那么，可以得到等价的SSA程序：

```
1 assume(i ⩽ N); // Loop Invariant
2 assume(i ⩾ N); // Jump out of the loop
3 assert(i == N);
```

所以，我们使用循环不变式来替代循环头之前的代码以及循环本身，从而无需考虑进入循环体的情况，仅考虑跳出循环的情况`(assume(i >= N))`.

如上所示，我们移除循环以及其之前的代码，代之以循环不变式和跳出循环的条件。

接着，我们将其转化为逻辑公式，可得：

$$\forall i.\ \ i \leq N \ \wedge\ i \geq N$$
$$\Rightarrow$$
$$i = N$$

很显然，这个逻辑公式也是永真的，故我们要验证的属性是成立的。即，倘若循环能退出，那么变量`i`的值最终将被增加到`N`.

我们也可以把之前的一些程序验证的例子写为霍尔三元组的形式：

- 交换两个数

$$\{x = x_0 \ \wedge \ y = y_0\}$$
$$x = x + y; y = x - y; x = x - y;$$
$$\{x = y_0 \ \wedge \ y = x_0\}$$

- 排列两个数

$$\{true\}$$
$$\mathrm{if}(\mathrm{x}_1 < \mathrm{x}_2)\{\mathrm{x}_1 = \mathrm{x}_1 + \mathrm{x}_2; \mathrm{x}_2 = \mathrm{x}_1 - \mathrm{x}_2; \mathrm{x}_1 = \mathrm{x}_1 - \mathrm{x}_2\}$$
$$\mathrm{else}\{\mathrm{skip};\}$$
$$\{x_1 \geq x_2\}$$

注意到在这些例子中，我们并未显式地要求x和y这些变量是整数，因为默认的理论语境已经做了这些限制。

以上的霍尔三元组，都能表达我们对代码实现的功能正确性的要求。由此可见，霍尔三元组非常适合用来表达程序验证，它包含了程序验证所需的程序源码以及属性这两个核心元素。

> 注意到排列两个数的霍尔三元组中，我们额外引入了一个skip语句，它表示空语句，意味着无事发生。许多常用的编程语言中，都有这一特性。

通过以上的几个例子，相信大家对程序验证已经有了比较初步和直观的认识。

首先，程序验证的目的是证明程序的正确性，这既包括程序不会有运行时错误，也包括程序功能的正确性。

其次，程序验证的基本要素是程序的源码，要验证的属性，这是实现自动化验证的前提。同时，在必要时，我们需要有循环不变式。

最后，程序验证的基本思路是，在给定源码和属性的基础上，通过验证算法，将我们需要验证的问题转化为逻辑公式，再证明逻辑公式的正确性。

验证算法是程序验证所研究的核心内容，对程序验证的效率，有着至关重要的影响。以上几个例子展示了最基础的验证算法，它直接简单粗暴地将程序转化为逻辑公式进行求解。

Part.2 : Type Theory

前面我们已经了解到程序验证的基本方法，使用程序验证的手段来确保程序正确性固然是可靠的。

但就像之前提到的，程序验证在目前来看依然不够成熟，大多数应用程序都不会采用这样一种复杂且耗时的手段来排除潜在的错误。

那么，当我们的应用场景并不要求严格的正确性，可我们又希望能够尽可能地确保程序的稳定性，方便后续的运营维护时，选用一门**静态强类型语言**通常是不错的解决方案。

当我们需要稳健且性能优越的程序时，往往会选用静态强类型语言；而当我们需要敏捷开发和快速迭代时，选用一门动态弱类型语言会是更好的选择。

这两个事实告诉我们，一门程序语言本身**类型系统的设计**与这门语言的易用性、可靠性都有不可分割的联系。

一门语言的类型系统越严格，它的可靠性往往越高，但在学习门槛上一般而言会更高。

有趣的是，类型最早是Russell为了解决Russell悖论而提出的，它是为Logic而生的，却在PL领域大放异彩。

在1901年，Russell对于当时的集合论提出了一个问题：所有集合的集合还是不是一个集合?

他构造出下面的集合：

$$R = \{x | x \notin x\}$$

这个问题也就转化为了：$R \in R$(即R也是一个集合)还是$R \notin R$(R不是一个集合)呢?

如果R仍是一个集合，那么根据定义$\mathrm{R} \in \mathrm{R}$，则$\exists x \in R \rightarrow x \notin R$. 进而$R$作为全体集合的集合产生了一个悖论。

从直观上看，或许有些读者能够感觉出问题所在——全部集合的集合太大了，没有办法仍然是一个集合了。

Russell也和我们持相同的观点，除此之外，他还将这样的R定义为一个Type，叫作Type_1. 而在Type_1之上又还存在着$\mathrm{Type}_2, \mathrm{Type}_3 \ldots$

这些类型满足：$\mathrm{n} \in \mathbb{N} \rightarrow \mathrm{Type}_{\mathrm{n}} \in \mathrm{Type}_{\mathrm{n}+1}$.

如此一来，我们也就解决了Russell悖论以及由它衍生的问题(所有的R的集合是不是一个Type_1等等)。

在这里，我们不妨多谈一谈Type Theory后续的发展：

如今的类型理论的应用范围已经超过了Russell最初的设想，它不再只是公理集合论中的一个特殊的约定，而将拓展整个Logic宇宙的疆域。

从1900年开始，Hilbert，Frege，Russell，Brouwer等人就开始试图对定理证明进行形式化。

在1930年Church提出了λ Calculus，这也是Functional Programming的理论基础。

λ Calculus与Type Theory的联系极其紧密，在Church的论文中，他提出了一个有趣的类型系统——Typed λ Calculus，这一计算系统与Turing Machine等价。

在1960年，Curry&Howard发现λ Calculus里面的类型规则和逻辑里面的规则是对应的，这个结果就是大名鼎鼎的Curry-Howard Isomorphism.

于是在1970年De Bruijn, Coquand开始试图用程序来形式化定理证明。在同一时期，Martin Lof&Coquand提出了Dependent Type的概念，正式拉开了类型理论在Logic中大规模应用的序幕。

近年来，最值得一提的类型理论或许是Homotopy Type Theory(简称HoTT). 在2013年Voevodsky进一步把Type等同于一个空间从而得到了Homotopy Type.

而这一切的核心在于"Propersition as Type"，即Type Theory中的一个类型可以对应到Logic中的一个命题。

本章的主题内容也将围绕着这一话题展开。

让我们从λ Calculus开始正式进入Type Theory的世界吧!

在学习函数式编程时，接触到λ Calculus这个概念几乎是不可避免的。

λ Calculus由三个要素组成：变量，函数与应用。其中，函数是一个表达式，写成：`lambda x . body`,表示"一个参数为`x`的函数，它的返回值为`body`的计算结果"。这时，我们说Lambda表达式绑定了参数`x`.

变量就是一个名字，这个名字用于匹配函数表达式中的某个参数名；应用可以理解为以一个实参调用函数的过程，通常写成把函数值放在它的参数前面的形式,如`(lambda x . plus x x) y`.其中的`y`就是我们的实参。

λ Calculus建立在函数概念的基础上，在纯粹的λ Calculus中，一切都是函数，甚至连值的概念都没有。

尽管λ Calculus如此简单纯粹，但能够从无到有地建立出我们熟悉的数学结构。

最基本的函数是$\lambda x.\, x$，也即恒等函数$f(x) = x$.其中，第一个x是函数的参数，第二个是函数体。

λ Calculus的特殊之处在于它能够用于构造高阶函数—既能够以函数为参数传入并返回一个函数结果的函数。高阶函数是函数式编程极其重要的特性，在JavaScript, Python等主流语言中也十分常见。常用的高阶函数有`map`,`filter`,`reduce`等等。

$\lambda x.\, x$中x被称为约束变量，因为它既在函数体中又是形式参数。而$\lambda x.\, y$中y被称为自由变量，因为它没有被预先声明。

λ Calculus只有两条真正的公理：α和β公理.

- α-Conversion

α-Conversion是一个重命名操作(以下称为α公理)，简单来说，就是变量的名称并不重要。给定λ Calculus中的任意表达式，我们可以修改函数参数的名称，只要我们同时修改函数体内所有对它的自由变量，使它们依然自由。

例如，现在有一个这样的表达式$\lambda x.\, x^2$,我们用α公理将x替换为y(写作$\alpha[\frac{x}{y}]$)会有$\lambda y.\, y^2$.

这样丝毫不会改变表达式的含义，但能够修改函数参数名称这一点又至关重要。

- β-Reduction

λ Calculus中的求值是通过β-Reduction来完成的，在λ Calculus中这可以被视为是一条公理(以下简称为β公理)，其本质是词法作用域的替换。具体形式如下：

$$(\lambda x.\, M)N = M[x := N]$$

那么，显然会有：$(\lambda x.\, x)a = a$和$(\lambda x.\, y)a = y.$

根据β公理，我们可以构造高阶函数$(\lambda x.\, (\lambda y.\, x))a = (\lambda y.\, x)[x := a] = \lambda y.\, a.$它的返回是一个函数，而这个式子本身也是一个函数。

在纯函数式语言Haskell中，对λ Calculus有天然的支持，例如

```haskell
--(λx.x²+1)3=10
(\x → x ^ 2 + 1) 3
10
```

虽然λ Calculus传统上只支持单参数函数，但可以通过一种叫Currying的技巧来创造多参数函数：

对于二元函数$f(x, y)$，我们可以定义$F_x = \lambda y.\, f(x, y), F = \lambda x.\, F_x$,那么就有：

$$(Fx)y = F_x y = f(x, y)$$

在Haskell中,`curry`可以这样表示：

```haskell
curry :: ((a, b) → c) → a → (b → c)
curry f a b = f(a, b)
--f(x,y)=x+y
f :: Num a ⇒ a → a → a
f = (\ x y → x + y)
```

又例如$(\lambda x.\, \lambda y.\, \lambda z.\, xyz)$等同于$f(x, y, z) = ((xy)z).$

细心的读者可能已经注意到了在λ Calculus中并没有定义数字和算术运算，在λ Calculus中真正存在的只有函数。因此，我们需要发明某种使用函数创建数字的方法，这样我们才能安心地在λ Calculus中使用数字。

幸运的是，Alonzo Church提出了一种函数化的数字版本—Church Numerals.

在学习丘奇计数之前，我们需要引入一些语法糖来命名函数.

我们可以用`let`来引入全局函数，从而不需要在每一个λ Calculus中声明这个函数。

它的Haskell实现如下：

```
let square = \x → x ^ 2
```

这个表达式声明了一个名为`square`的函数，其定义是`\x . x ^ 2`.如果我们要求`square 4`,则它的等价表达式为：

```
(\square → square 4) (\x → x ^ 2)
```

接下来，让我们正式进入丘奇计数的学习。

所有丘奇数都是带有两个参数的函数：

- 0是$\lambda sz.\ z.$
- 1是$\lambda sz.\ sz.$
- 2是$\lambda sz.\ s(sz)$
- 对于任何自然数`n`，它的丘奇数是将第一个参数应用到第二个参数上`n`次的函数.

我们可以把`z`看作是对于`0`值的命名，将`s`看作是后继函数的简称。因此，`0`是一个仅返回`0`值的函数；`1`是将后继函数应用于`0`上一次的函数；`n`是将后继函数应用到`0`上n次的函数。

现在，让我们来定义加法`x + y`，我们需要的是一个有四个参数的函数：两个需要相加的数字，以及推导数字时用到的`s`和`z`.

利用Haskell的实现如下：

```
let add = \ s z x y → x s (y s z)
```

将其柯里化，会有：

```
let add = \ x y → (\ s z → (x s (y s z)))
```

现在，让我们来看看`add`的含义：

- 它接受两个参数，这是我们需要做加法的两个值;

- 它需要正则化这两个参数，以使它们都对`z`和`s`进行绑定，将参数都写成`s`和`z`的组合形式。

为了将`x`和`y`相加，先用参数创建正则化的丘奇数`y`，然后将`x`应用于丘奇数`y`，得到由`s`和`z`定义的新丘奇数`y`.也就是说，要计算`x + y`，先计算`y`是`z`的几号后继，然后计算`x`是`y`的几号后继。

以`2 + 3`为例，我们来看计算过程：

```
add (\ s z → s (s z)) (\ s z → s (s (s z))) news newz
```

对`2`和`3`运用α公理，`2`中用`s2`和`z2`代替`s`和`z`，`3`中用`s3`和`z3`代替`s`和`z`,于是不需要参数`newz`和`news`：

```
add (\s2 z2 → s2 (s2 z2)) (\ s3 z3 → s3 (s3 (s3 z)))
```

用`add`的定义做替换：

```
(\ x y → (\ s z → (x s y s z))) (\ s2 z2 → s2 (s2 z2)) (\ s3 z3 → s3 (s3 (s3 z3)))
```

对上面的式子运用β公理：

```
\ s z → (\ s2 z2 → s2 (s2 z2)) s (\ s3 z3 → s3 (s3 (s3 z3)) s z)
```

对`3`再次运用β公理,将`3`正则化：把`3`定义中的`s3`和`z3`替换成`add`参数列表中的`s`和`z`：

```
\ s z → (\ s2 z2 → s2 (s2 z2)) s (s (s (s z)))
```

最后，对`2`运用β公理，会得到：

```
\ s z → s (s (s (s (s z))))
```

得到的刚好就是丘奇数`5`！

在λ Calculus中引入数字之后，我们还需要能够表达分支，也就是说，希望能够提供`if / then / else`语句的表达式，就像我们在大多数编程语言里面做的那样。

在学习了丘奇计数将数字表示为函数之后，利用λ Calculus的特性，我们不妨将`true`和`false`值也表示为对其参数执行一个`if-then-else`操作的函数：

```
let True = \ x y → x
let False = \ x y → y
```

于是，我们可以写出一个`if`函数，它的第一个参数是一个条件表示式，第二个参数式如果条件为真时进行运算的表达式，第三个参数是如果条件为假时进行运算的表达式：

```
let If_Then_Else = \ cond true_expr false_expr → cond
true_expr false_expr
```

此外，我们还需要定义常用的逻辑运算：

```
let And = \ x y → x y False
let Or = \ x y → x True y
let Not = \ x → x False True
```

让我们来验证一下这些定义的正确性，以`And`为例,先来看`And True False`：

```
And True False = And (\ x y → x) (\ x y → y)
```

对`True`和`False`采用α公理：

```
And (\ xt yt → xt) (\ xf yf → yf)
```

展开`And`表达式：

```
(\ x y → x y False) (\ xt yt → xt) (\ xf yf → yf)
```

最后对式子采用β公理，会有：

```
(\ xt yt → xt) (\ xf yf → yf) False = \ xf yf → yf
= False
```

于是我们有`And True Fasle = False`.接下来，验证`And False True`:

```
And (\ x y → y) (\ x y → x) = And (\ xf yf → yf) (\
xt yt → xt)
                            = (\ x y → x y False)
(\ xf yf → yf) (\ xt yt → xt)
                            = (\ xf yf → yf) (\ xt
yt → xt) False
                            = False
```

最后，我们来看`And True True`:

```
And True True = And (\ x y → x) (\ x y → x)
              = And (\ xa ya → xa) (\ xb yb → xb)
              = (\ x y → x y False) (\ xa ya → xa)
(\ xb yb → xb)
              = (\ xa ya → xa) (\ xb yb → xb) False
              = (\ xb yb → xb)
              = True
```

因此`And True True = True`.

PS: 以下的内容对于初学者来说，是有一定挑战性的：一方面，数学推导过程比较多；另一方面，也要求读者对于之前学习的λ Calculus内容比较熟悉.如果读者感觉到阅读有困难，可以先跳过本节.

现在，我们距离将λ Calculus变成一个完备的算术系统只差最后一步——用λ Calculus表示递归。

但是，由于λ Calculus中，函数没有名字(这样的函数被称为*匿名函数*)，我们需要采取一些特殊手段，也就是所谓的Y Combinator，又叫λ不动点算符.

我们先来看一个λ Calculus之外的递归函数——阶乘函数:

```
factorial 0 = 1
factorial (n + 1) = (n + 1) * factorial n
```

利用Haskell中的`product`函数会更加简洁：

```haskell
factorial n = product [1..n]
```

如果我们要使用λ Calculus来编写这个阶乘函数，需要几个工具——一个测试是否为0的函数，一个乘法函数以及一个减1的函数。

我们将第一个函数命名为`IsZero`，它有三个参数：一个数字，两个值。如果数字为0，返回第一个值，如果不为0，则返回第二个值。

乘法函数本身就是一个递归结构，在实现递归前，我们姑且假设一个乘法函数`Mult x y`.

最后，减1函数可以看作是数字的前驱，我们用`Pred x`表示。

那么，阶乘函数就可以表示为：

```haskell
\ n → IsZero n 1 (Mult n (something (Pred n)))
```

这个`something`部分就是我们需要构造的递归结构.

关键就是我们要如何实现`something`的递归功能呢?

这就要借助到λ Calculus中的重要技巧——组合子：一个组合子可以看成是一种特殊的高阶函数，它们只引用函数应用。

这里，我们用到的是Y Combinator，它的出现使得递归成为可能，其实现如下：

```haskell
let Y = \ y → (\ x → y (x x)) (\ x → y (x x))
```

它的语法树形似一个Y，因此得名Y Combinator.

Y Combinator的特别之处在于它应用自身来创造本身，换句话说，`(Y Y) = Y (Y Y)`.

我们可以证明这个等式：

```
(Y Y) = (\ y → (\ x → y (x x)) (\ x → y (x x))) Y
      = (\ x → Y (x x)) (\ x → Y (x x))
      = (\ x → Y (x x)) (\ z → Y (z z))
      = (\ a → (\ b → a (b b))
        (\ b → a (b b)))
        ((\ z → Y (z z))
        (\ z → Y (z z)))
      = (\ b → ((\ z → Y (z z))
        (\ z → Y (z z))) (b b))
        (\ b → ((\ z → Y (z z))
        (\ z → Y (z z))) (b b))
      = Y (Y Y)
```

因此，我们会有`(Y Y) = Y (Y Y) = Y (Y (Y Y))=...`,这也就是所谓的**Y组合子利用自身创造本身**。

那么，要如何使用Y组合子呢?

我们可以尝试重写阶乘函数，并给它重命名为`fact`函数:

```
let fact = \ n → IsZero n 1 (Mult n (fact (Pred n)))
```

现在的问题是，表达式右边的`fact`并不是`fact`函数中定义的标识符，并不能直接出现在`fact`函数表达式中，如何让`fact`引用`fact`自身呢?

在这里，可以编写一个`metafact`函数:

```
let metafact = \ fact → (\ n → IsZero n 1 (Mult n
(fact (Pred n))))
```

这时，我们可以发现:

```
fact n = (metafact metafact) n
```

这就为引入Y组合子做好了准备，我们可以构造`metafact (Y metafact)`这样一个结构，它正是我们想要的递归结构。将这个式子展开后，会有:

```
(\ fact → (\ n → IsZero n 1 (Mult n (fact (Pred n))))) (Y (\ fact → (\ n → IsZero n 1 (Mult n (fact (Pred n))))))
```

`(Y metafact)`实际上就是我们的参数`fact`的值，如果`n = 0`，那么它只是返回`1`，如果`n`不为`0`，那么它将调用`fact (Pred n)`,这个环节将延续下去，得到输出`metafact (Y metafact) (Pred n)`.

因此，`metafact (Y metafact) = metafact (Y metafact) (Pred n)`就是我们希望得到的递归结构。

按照这样的思路，我们也能够定义出`Mult`函数，这里不再赘述...

使用Y实现阶乘函数(为了方便，只有主体部分使用Y组合子，乘法和`if`仍然采用原生Haskell语法):

```
y f = f (y f)

fac = y (\ f n → if (n == 0) then 1 else n * f (n-1))
```

解决了这个问题，我们再来看Y组合子与不动点之间的联系:

一个函数$f(x)$的不动点是指满足条件$f(\lambda) = \lambda$的参数λ.而Y组合子的作用就是计算函数的不动点，它对所有函数`f`都满足`f(Y(f))=Y(f)`.推导如下:

```
Y (f) = (\ y → (\ x → y (x x)) (\ x → y (x x))) f
      = (\ x → f (x x)) (\ x → f (x x))
      = (\ x → f (x x)) (\ a → f (a a))
      = f ((\ a → f (a a)) (\ a → f (a a)))
      = f ((\ x → f (x x)) (\ x → f (x x)))
      = f (Y (f))
```

这也解决了我们前面提到的一个问题——`fact`函数不能引用自身.从现在的视角来看，我们通过Y组合子计算出`fact`的不动点，在不引用自身的情况下，构造出需要的递归结构.

正因如此，Y组合子也叫λ不动点算子.

最后，顺带提一下惰性求值，这是Haskell相比于其他函数式语言的特殊之处。

对于一个Y组合子，存在两种计算方式：

给定以下表达式：

```
(\ x y → x * y) 3 ((\ z → z ^ 2) 4)
```

我们可以按不同的顺序来计算，可以先对`\ x y -> x * y`运用β公理：

```
3 * ((\ z → z ^ 2) 4)
```

或者，我们先对`(\ z -> z ^ 2) 4`运用β公理：

```
(\ x y → x * y) 3 (4 ^ 2)
```

在这种情况下，两种方式得到的结果相同，**但在很多时候，它们的结果不同！**

第一种计算顺序就是我们所说的惰性求值：我们只在需要的时候计算函数的参数.这有点像高中解析几何"设而不求"的思想。Haskell语言默认的求值形式就是惰性求值。

第二种叫做严格求值，即总是在把参数传递给函数之前进行计算。这也是大多数程序语言的求值形式。

两种求值方式互有利弊，惰性求值更具灵活性，能够更自由地构造我们想要的结构，但如果利用不好，对性能会产生很大的拖累；严格求值节约内存空间，在性能上也更具突出，但相对而言没有那么高的自由度。

我们学习的Y组合子是惰性求值形式的，如果我们在Haskell中采用严格求值形式的Y组合子,是无法导出正确的递归结构的。

现在，让我们正式进入到组合子编程！

在前面的内容中，我们已经利用λ Calculus构造了一个与熟知的算术系统相似的运算体系；与此同时，也学习了组合子的相关概念。

那么就有一个问题值得思考——能否用几个简单的组合子表示所有的无类型的λ Calculus计算呢?

答案是肯定的！这也就是著名的SKI组合子的由来，SKI组合子与无类型的λ Calculus系统等价，也是一个简约的图灵完备的计算系统。

SKI组合子一般的定义如下：

```
S = \ x y z → x z (y z)
K = \ x y → x
I = \ x → x
```

实际上，我们连`I`组合子都不需要，只需要有`S`和`K`两个组合子即可，我们会有：

```
S K K x = K x (K x) = x
```

细心的读者可能会发现，上面的表达式和我们的`I`组合子的形式不完全相同，并不能规约为`\ x -> x`的形式。

到目前为止，我们说在λ Calculus中，`x = y`，当且仅当`x`与`y`相同，或在α公理替换后相同，这实际上是一种内涵等价，这种等价关系是比较严格的；我们还可以引入另一种相对不是那么严格的等价形式——外延等价：

当外延等价`x = y`时，当且仅当`x ≡ y (mod α)`或是`forall \ a -> X a = Y a`，因此，在外延等价的意义上，会有`x = \ x -> x`.

我们可以将任何λ Calculus转化为外延等价的组合子形式，例如，用`S`和`K`表示的Y组合子如下：

```
Y = S S K (S (K (S S (S (S S K)))) K)
```

这里，特别注意`S`的应用机制——它并不是接受两个参数`x`和`y`，并应用`x`到`y`，而是新增了一个值`z`，先将`x`应用到`z`上，再将`y`应用到`z`上，最后将前者的结果应用到后者的结果上。

最后，定义一个从λ Calculus到组合子形式的变换函子`C`满足：

1. `C{x} = x`
2. `C{E1 E2} = C{E1} {E2}`
3. `C{\ x -> E} = K C{E}`,如果`x`在`E`中非自由.
4. `C{\ x -> x} = I`
5. `C{\ x -> E1 E2} = (S C{\ x -> E1} C {\ x -> E2})`
6. `C{\ x -> (\ y -> E)} = C{\ x -> C{\ y -> E}}`,如果`x`在`E`中是自由变量.

利用函子`C`，我们能够将任意的λ Calculus翻译为SKI组合子的形式，例如：

```
C{\ x y → y x} = C{\ x → (\ y → y x)}
               = C{\ x → C{\ y → y x}}
               = C{\ x → S C{\ y → y} C {\ y → x}}
               = C{\ x → S I C {\ y → x}}
               = C{\ x → S I (K C{x})}
               = C{\ x → S I (K x)}
               = S C{\ x → S I} C {\ x → (K x)}
               = S (K (S I)) C {\ x → (K x)}
               = S (K (S I)) (S C{\ x → K} C {\ x -> x})
               = S (K (S I)) (S C{\ x → K} I)
               = S (K (S I)) (S (K K) I)
```

由此，我们正式建立了SKI组合子与λ Calculus间的同构！

之前我们学习的λ Calculus都是简单的无类型λ演算，在计算λ Calculus时，并没有将类型纳入考虑范围。

没有类型系统，对于一种编程语言来说，是致命的缺陷；而λ Calculus又被称为最小的通用程序设计语言，因此，引入类型的概念也十分必要，并发展出类型化λ Calculus.

从类型化的角度来看，无类型λ Calculus可以看作是类型化λ Calculus的一个简单到病态的特例——只有一个类型的λ Calculus.

在这里，我们将介绍简单类型λ Calculus，它是最简单的类型化λ Calculus,从它的基础上还可以衍生出更多类型化的λ Calculus，感兴趣的读者可以阅读*λ Calculi With Types*.

简单类型λ Calculus的主要变化在于增加了**基本类型**的概念——可以使用一些由原子值构成的论域将这些值分为不同的简单类型.这个概念看起来十分抽象，那么不妨跳过概念，让我们直接从例子入手：

我们可以构造一个类型$\mathbb{N}$，它是包含所有自然数的集合；也可以构造一个类型$\mathbb{B}$，它包含Bool值true/false,还可以构造一个对于字符串的类型$\mathbb{S}$.

这三个类型就可以看作是我们预设的基本类型，接下来就可以引入函数类型，这里的函数类型与我们之前在第二章中讲到类似，函数类型的元素就是将一种类型的值映射到第二种类型的值的函数.

对于一个接受类型`A`的输入参数，并返回类型`B`的值的函数，我们定义它的类型为`A -> B`.其中`->`可以看作是函数类型构造器，它是右关联的，满足`A -> B -> C = A -> (B -> C)`.

为了将类型应用于λ Calculus，我们还需要做几件事：首先，我们需要更新语法，使我们能够在λ表达式中加入类型信息；其次，我们需要添加一些规则，声明哪些类型化操作是合理的。

语法部分很容易解决，我们添加一个`:`符号(在Haskell中一般是`::`)；冒号左边是表达式或变量的绑定，右侧是类型规范。它表明，其做出拥有右侧指定的类型。举几个例子：

- `\ x :: N -> x + 3`表示参数`x`的类型为`N`.这里没有指明函数结果的类型；但已知函数`+`的类型是`N -> N`，于是可以推断函数结果的类型也是`N`.
- `(\ x -> x + 3) :: N -> N`,和上面一样，但我们把类型声明提了出来，它给出了这个λ表达式作为一个整体的类型，我们可以推出`x :: N`.
- `\ x :: N, y :: B -> if y then x * x else x`.这是两个参数的函数，第一个参数类型是`N`，第二个的类型是`B`.我们可以推断返回结果的类型为`N`.于是，这个函数的类型为`N -> B -> N`.对这个函数采用柯里化可能会更容易看出这一点：`\ x :: N -> (\ y :: B -> if y then x * x else x)`.其中，内层λ表达式的类型是`B -> N`;外层类型为`N -> (B -> N) = N -> B -> N`.

为了讨论程序本身是否是类型安全的，我们需要引入一套类型推理规则。

当使用这些规则推理一个λ表达式的类型时，称这个过程为类型判断；类型推理和判断总能让我们推断出λ表达式的类型；如果表达式的任一部分与类型推断结果不一致，则表达式不合理。

现在，让我们来介绍类型推断法则，这里我采用数学语言(而不是Haskell语言)：

1. $\frac{\mathbb{x} : A \in \Gamma}{\Gamma \vdash \mathbb{x} : A}$，如果我们只知道变量的类型声明，那么这个变量是它所声明的类型.
2. $\frac{\Gamma \vdash M : (A \to B) \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$，知道函数类型为`A -> B`，把它应用到类型为`A`的值上，返回值类型为`B`.
3. $\frac{\Gamma \vdash \mathbb{x} : A \quad M : B}{\Gamma \vdash (\lambda \mathbb{x}.M) : A \to B}$，如果函数参数的类型为`A`，函数返回值的类型为`B`，则函数类型为`A -> B`.

其中，Γ被叫做类型上下文，它是类型中的值与类型绑定关系的集合。

只要我们的λ表达式中每一项的类型判断均符合这三条规则，我们就说这个表达式是类型安全的。

至此，我们得到了一个简单的类型化λ Calculus，也借此了解到一些类型推导的原理。

感兴趣的读者还可以尝试建立SKI组合子与简单类型化λ Calculus之间的对应关系。

这里，我们直接给出SKI组合子的类型化描述：

```haskell
I :: A → A
I = (\ x → x)

K :: A → B → A
K = (\ x :: A → ((\ y :: B → x) :: B → A))

S :: (A → B → C) → (A → B) → (A → C)
S = (\ x :: (A → B → C)
    → (\ y :: (A → B)
    → (\ z :: A → (x z :: B → C) (y z :: B))))
```

之所以说这种类型化很简单，是因为简单类型λ Calculus对于类型的处理方式很少：建立新类型的唯一途径就是通过`->`这个类型构造器.其他的λ Calculus类型化方法一般还会定义出参数类型，它能将类型表示为不同类型间的函数(实际上也就是Haskell中使用到的`type`声明)。

有了上面学到的前置内容，我们终于到达了λ Calculus的终极命题——“程序即证明”.我们可以将一段程序看作是一个逻辑命题，并从数学上严格证明或证伪其正确性.

这里为熟悉逻辑学的读者补充一下：每一种类型化的λ Calculus都对应一种直觉逻辑；我们可以说，类型化λ Calculus语义上等同于直觉逻辑。

当然，从主题上来说，我们并不需要了解直觉逻辑，也能够看懂"程序即证明"这个结论。

先考虑简单类型λ Calculus中的类型，任何可以从下面语法中生成的形式都是λ Calculus类型：

```
type :: primitive | function | (type)
primitive :: A | B | C | D | ...
function :: type → type
```

这段代码表明：简单类型λ Calculus中的类型要么是预先定义的基本类型；要么是函数类型。

这个语法中存在的问题就是：你可以创建一个类型的表达式，而且它们是合理的类型定义，但你无法写出一个拥有该类型的单独的完整的封闭表达式(封闭表达式是指没有自由变量的表达式)。如果一个表达式有类型，我们说表达式“居留”该类型，从而该类型是一个居留类型。反之，如果没有表达式能够居留类型，则这个类型是不可居留的。

那么，什么时候我们构造的类型才是居留类型呢?

Curry-Howard Isomorphism为我们提供了答案：每个类型化λ Calculus都有对应的直觉逻辑；类型表达式是可居留的，当且仅当该类型是对应逻辑上的定理。

先看类型`A -> A`，我们将`->`视为一种逻辑蕴涵，则`A implies A`显然是直觉逻辑中的定理。那么，类型`A -> A`就是可居留的。

再来看类型`A -> B`，这不是一个定理，除非能在某个`context`中下证明它。作为一个函数类型，这表示一类函数，但在不包含任何条件的情况下，你无法做到以`A`类型中的值为参数，返回到一个不同的类型`B`中的值。必须有某个`context`提供`B`类型中的值——为了访问这个`context`，就必

须存在某种允许函数访问的方式：一个自由变量。这一点在逻辑学上也是相同的，我们需要某个`context`能够将`A -> B`作为一个定理。

如果有一个封闭的λ Calculus，其类型对应直觉逻辑中的定理，那么，该类型的表达式就是定理的一个证明。每一次运用β公理则等同于逻辑中的一个推理步骤，对应这个λ Calculus的逻辑被称为它的模型。从哲学意义上来说，λ Calculus与直觉逻辑只是同一件事物的不同反映，从数学上来说，可以认为两者同构。

要证明这个同构，就需要借助组合子演算.

由于直觉逻辑本身是完备的，我们需要做的就是将λ Calculus中的计算翻译为直觉逻辑中的证明.通过组合子我们将很容易导出直觉逻辑：

直觉逻辑中的所有证明都依赖于两条基本公理

- `A implies B implies A`
- `(A implies B implies C) implies ((A implies B) implies (A implies C))`

我们用`->`重写它们，改为类型语言：`A -> B -> A`以及`(A -> B -> C) -> ((A -> B) -> (A -> C))`.

结合之前学习的SKI组合子定义，这分别对应了我们的`K`和`S`组合子的类型!

进而，λ Calculus中的类型对应于直觉逻辑中的原子命题，函数是推理规则，每个函数都能转化为SKI组合子的表达式；每个组合子表达式都对应直觉逻辑的某个基本推理规则的实例。

于是，函数就成了对应逻辑中定理的一个构造性证明。Curry-Howard同构到此结束.....

Part.3 : Insertion Sort

本章，我们将尝试使用Dafny语言来自动化验证插入排序算法，以此来展示程序验证是如何工作的。

Dafny语言结合了来自函数性和命令性范式的特点，并且支持部分面向对象的特性，可以编译成其他程序语言，如C#或Java，并通过前置条件、后置条件和循环不变量(也就是Hoare Logic)实现形式验证。

Dafny支持包括泛型类、动态分配、归纳数据类型以及用于副作用推断的"隐式动态框架"等为自动程序验证量身定制的特性。

Dafny的设计目的是让使用者能够体验到简单的自动化形式验证，在教学中得到了广泛的应用。

下面是插入排序算法的Dafny语言实现，我们将验证这个排序算法的实现在功能上是正确的。

```
method InsertionSort(a:array<int>)
  requires a ≠ null;
  //ensures forall k:: forall l:: 0 ⩽ k ⩽ l < a.Length ⟹ a[k] ⩽ a[l];
  modifies a;
{
  var i:int := 1;
  while(i < a.Length)
  {
    var t:int := a[i];
    var j:int := i - 1;
    while (j ⩾ 0)
    {
      if (a[j] ⩽ t)
      {
        break;
      }
      a[j + 1] := a[j];
      j := j - 1;
    }
    a[j + 1] := t;
    i := i + 1;
  }
}
```

我们先将以下代码复制到编辑器中。

保存后，Dafny没有报出任何错误。

因为此时我们没有说明需要验证的功能属性，所以Dafny没有得到需要验证的任务。但值得一提的是，对于一些简单的属性，比如数组越界，Dafny会自动对数组访问的位置进行验证。

例如我们将`var j:int := i - 1`中的`i - 1`改为`i`，Dafny会立刻在这个位置报出`index out of range`，也就是说这个位置可能会发生数组越界。

接下来，我们逐步添加需要证明的程序功能属性。对于程序验证而言，是最重要的内容之一。 属性需要使用形式化的描述，比如一阶逻辑公式，对所要证明的内容的一种表达。接下来，我们先讨论一下所要描述的属性。

对单个函数的验证一般称之为模块化验证，由于没有函数被调用的具体环境，所以我们需要对函数的输入参数添加一些要求。我们将这些要求称为函数的**前置条件**。

在Dafny中，前置条件用`requires`关键字表示。可以按照如下的形式加入描述前置条件的语句：

```
...
  requires a ≠ null
  modifies a
  {
     ...
  }
```

保存后，Dafny仍然没有报出任何错误。这是因为前置条件仅仅描述了对函数输入的前提假设，并未提供需要验证的属性。

那么，接下来加入描述后置条件：

我们一般使用后置条件来描述需要验证的属性，有时也会在程序中使用断言来描述需要验证的属性。对`InsertionSort(a)`而言，所有满足条件`a != null`的调用，都应该返回我们所预期的结果。

那么接下来的问题是，如何形式化地表示我们预期的结果，即如何正确地写出所需要的后置条件？

这里，我们需要额外定义模块`sorted`来表示一个数组是否已经排序。

那么，很显然，后置条件是`sorted(a)`.

也即，我们可以按照如下的形式加入描述后置条件的语句：

```
...
  requires a ≠ null
  ensures sorted(a)
  modifies a
  {
     ...
  }
```

保存后，可以发现Dafny报出了错误，显示说此位置的多个后置条件可能不会满足。

这说明了后置条件可能不会满足，Dafny验证器目前无法证明这些后置条件是成立的。

其核心原因是程序中包含循环，而我们并没有提供对应的循环不变式。缺乏必要的不变式，Dafny验证器无法自动完成对带循环程序的证明。

因此，循环不变式是需要人为给出并在编译器的提示下一步步优化的，这是因为当给出的循环不变式不够强大，受限于Dafny的推理能力，依旧无法验证后置条件。

具体的细节我们在此就不做展示，感兴趣的读者可以自行探索。

当我们手动地给出对应的循环不变式，也即：

```
predicate sorted (a: array<int>)
    requires a ≠ null
    reads a
{
    sorted'(a, a.Length)
}

predicate sorted' (a: array<int>, i: int)
    requires a ≠ null
    requires 0 ≤ i ≤ a.Length
    reads a
{
    forall k :: 0 < k < i ⟹ a[k-1] ≤ a[k]
}

```

```
method FindMin (a: array<int>, i: int) returns (m: int)
    requires a ≠ null
    requires 0 ⩽ i < a.Length
    ensures i ⩽ m < a.Length
    ensures forall k :: i ⩽ k < a.Length ⟹ a[k] ⩾ a[m]
{
    var j := i;
    m := i;
    while(j < a.Length)
        invariant i ⩽ j ⩽ a.Length
        invariant i ⩽ m < a.Length
        invariant forall k :: i ⩽ k < j ⟹ a[k] ⩾ a[m]
        decreases a.Length - j
    {
        if(a[j] < a[m]) { m := j; }
        j := j + 1;
    }
}

method InsertionSort (a: array<int>)
    requires a ≠ null
    modifies a
    ensures sorted(a)
{
    var c := 0;
    while(c < a.Length)
        invariant 0 ⩽ c ⩽ a.Length
        invariant forall k, l :: 0 ⩽ k < c ⩽ l < a.Length ⟹ a[k] ⩽ a[l]
        invariant sorted'(a, c)
    {
        var m := FindMin(a, c);
        a[m], a[c] := a[c], a[m];
        //assert forall k :: c ⩽ k < a.Length ⟹ a[k] ⩾ a[c];
        c := c + 1;
    }
}
```

则Dafny显示后置条件`sorted(a)`是成立的。

从以上验证的例子中我们可以看出，验证低阶属性，如数组越界，是比较有希望实现完全自动化地完成的。这实际上也是程序验证相关研究的主攻方向。

然而，对于复杂程序的功能正确性验证，却是一件比较麻烦的事情：

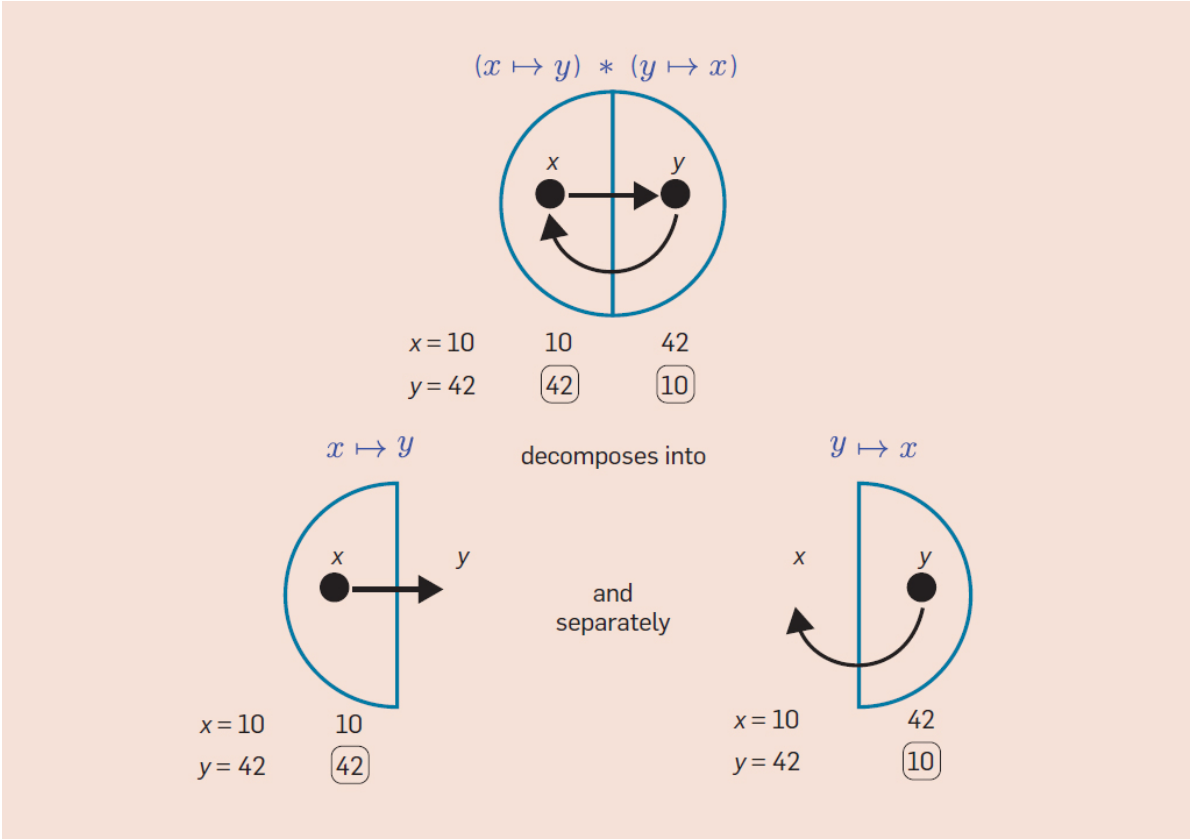
- 其一，针对功能属性正确性的前置条件和后置条件的描述需要对谓词逻辑有比较深刻的理解。
- 其二，针对功能属性验证的不变式，也非常复杂，基本上都需要人工依据程序的语义来提供。

不过，在前置条件以及后置条件的基础上，进行人工的修正，然后交互式地一步步得到不变式，或许会是一个不错的解决方法。

Part.4 : Separation Logic

之前我们讲到了程序验证的基础——霍尔逻辑，并初步了解霍尔逻辑是如何将一段程序转化为待验证的逻辑公式的。

尽管霍尔逻辑在处理顺序、分支、循环结构时都能取得很好的效果，但在面对C/Cpp这类系统级程序语言时，依旧存在着一个盲区——涉及到内存管理的操作，它们能直接操作变量指向的内存空间。这意味着，如果有两个变量指向同一片内存空间，那么这段程序带来的执行结果将同时影响两个变量。



正如图中所展示的那样，$*x$、$*y$是同一片内存空间上的指针，当我们试图对$*x = 10$的值进行操作$*x = 42$时，同一片内存地址上的$*y = 42$，此时，我们令$*y = 10$，则对应地会使得$*x = 10$.

因此，$*x + *y = 20$而不是52.

我们将上面的这个结果用霍尔逻辑描述：

$$
\begin{gathered}
\{true\} \\
*x = 42; \\
*y = 10; \\
z = *x + *y; \\
\{z = 52\}
\end{gathered}
$$

我们尝试推导这个错误的结论：

$$
\frac{\{*x + *y = 52\}}{\begin{gathered} z = *x + *y; \\ \{z = 52\} \end{gathered}}
$$

这一步不存在问题。继续推导：

$$\frac{\{*x + 10 = 52\}}{*y = 10;}$$
$$\{*x + *y = 52\}$$
$$z = *x + *y;$$
$$\{z = 52\}$$

这一步同样也不存在问题。继续向前计算：

$$\frac{\{42 + 10 = 52\}\ \times}{*x = 42;}$$
$$\{*x + 10 = 52\}$$
$$*y = 10;$$
$$\{*x + *y = 52\}$$
$$z = *x + *y;$$
$$\{z = 52\}$$

这一步貌似成立，我们好像已经证明了霍尔三元组。但是这实际上是有问题的，因为两个指针可以指向同一个地址，这在C语言中是允许的。倘若指针`x`和`y`指向了同一个地址，那么`*x`和`*y`的值都会是10. 这也就是说，我们的推理是不正确的。

为了在涉及指针操作时，仍然保证霍尔逻辑推理的正确性。我们需要加入额外的谓词来描述以下实事：当我们在对`x`指向的内存区进行操作时，`y`指向的内存区不会发生改变，反过来也一样。

我们将这个问题称为Frame问题，把额外加入内存约束的谓词称为Frame约束。

在使用霍尔逻辑对系统程序进行证明时，需要花费大量的精力来解决Frame问题。有时，花费在证明Frame相关的约束上的时间，甚至比证明目标本身更多，这造成了极大的资源浪费。我们亟需一套通用的方法来解决Frame问题。

分离逻辑就是在这样的背景下，总结出的一套解决方法，它是对一般霍尔逻辑的一种扩展。

分离逻辑的核心内容是引入了分离与运算，记为$A * B$.

在介绍分离与的语义前，我们先介绍分离逻辑的语义模型。分离逻辑中假定有一个值函数s，将变量x映射到它的值$s(x) = x$.

例如$s(x) = 100 \to x = 100$.

在一般霍尔逻辑中，当提及一个变量x时，我们默认对做取值处理，所以我们一般会忽略值函数$s(x)$，直接用x表示。

但在分离逻辑中情况不同，因为分离逻辑需要单独考虑内存，所以对分离逻辑中的变量，不仅仅有取值处理，我们还可以将它们视为内存中的地址。例如，在C语言中，变量可以是一个整型值，也可以作为一个指针。

所以，在分离逻辑中，除了值函数，还有一个堆函数h，将地址映射到堆中的值。

例如$h(x) = 100$意味着在内存地址x处的存储的值为100，可以理解为C语言代码`*x == 100`.

如果两个堆函数h_1和h_2的定义域没有公共的地址，我们说它们不相交。

我们也可以把堆函数理解成实际的内存堆，不相交意味着两个堆没有公共内存地址。

例如，我们可以将一个堆$h : 1, 2, 3, 4, 5$划分为两个不相交的子堆$h_1 : 1, 2, 3 \mid h_2 : 4, 5$.

在分离逻辑中，我们还会引入一个谓词emp，表示堆是空的。

此外，还会有一个指向运算符$\mapsto$，$x \mapsto y$表示x的值对应的地址，在堆上对应的值等于y的值，即$h(s(x)) = s(y)$，可以简写为$h(x) = y$.

指向运算符$\mapsto$可以直观地理解成C语言的指针取值运算，可以大概理解为`*x == y`.

分离与$*$可以看作是一种新的与运算，它能够以如下方式形式化定义：

$$f \cdot g = \begin{cases} f \biguplus g & \text{if } \mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

其中

$$(f \biguplus g)(x) = \begin{cases} f(x) & x \in \mathrm{dom}(f) \\ g(x) & x \in \mathrm{dom}(g) \\ \text{undefined} & \text{otherwise} \end{cases}$$

满足这一条件的值集$\mathbb{H} : \mathbb{N} \to X$在代数上实际构成一个 $\mathrm{Partial\ Commutative\ Monoid}$.

也就是说，分离与$*$可以看作是部分交换幺半群上的$\wedge$运算。

同普通的逻辑与$\wedge$相比，分离与不仅要求运算符两边的公式都成立，还要求它们在原堆划分出的两个不相交子堆中分别成立。这一点是非常重要的，正是因为要求子堆不相交，我们在进行分离逻辑推理时，不再需要显示地添加Frame约束。

对于逻辑与，显然有公式$P \wedge P \Rightarrow P$成立。但是对于分离与，$P * P \Rightarrow P$不一定成立。例如我们令P为$x \mapsto v$，那么公式$x \mapsto v * x \mapsto v \Rightarrow x \mapsto v$是不成立的。

因为我们无法将一个堆分割为不相交的两部分，使得两部分都有x对应的地址。

所以$x \mapsto v * x \mapsto v$等值于$\mathrm{false}$. 进而$P * P \Rightarrow P$不是永真式。

有了新的与运算，那么对应也会有新的其他运算，典型的就是分离蕴含，记为$-\!\!\circ$.

它的形式化定义如下：

$$f \leq g \Longleftrightarrow \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \wedge \forall x \in \mathrm{dom}(f), f(x) \simeq g(x)$$

分离蕴含同样有着双重含义：f所在的子堆h_1同时也是g所在子堆h_2的子堆$h_1 \cap h_2 = h_1$，且在h_1中两者等价。

类比逻辑蕴含和逻辑与，我们可以从$P \wedge (P \Rightarrow Q)$中得到$Q$；在分离逻辑中，我们也可以从$P * (P -\!\!\circ Q)$得到$Q$.

分离逻辑对于具有验证复杂数据结构的动态内存程序似乎非常实用。在定理证明领域，分离逻辑使用得非常广泛。

它在一定程度上解决了Frame问题，并且和递归数据结构以及递归函数都非常契合。但是，在自动化的程序验证领域，分离逻辑使用得非常少。

这其中比较主要的原因是分离逻辑缺乏全自动的逻辑公式可满足性求解器（比如一阶逻辑理论的SMT求解器）。其次，自动化的程序验证主要针对低阶的安全属性，如数组越界，除错误等等。而分离逻辑与高阶的功能属性比较契合，所以自动化的程序验证的分析对象不太需要使用分离逻

辑来描述。虽然如此，分离逻辑仍然魅力十足。

它仅对霍尔逻辑进行了简单的扩展，就非常优雅地解决了Frame问题，并且实现了理论上的完备性。

分离逻辑使得对复杂数据结构动态内存程序的验证变得更加简单。同时，分离逻辑在针对共享内存的并行程序的验证有着非常好的适性。我们可以使用如下的并行语句规则，来描述并行进程的组合产生的影响：

$$\text{Concurrency}\frac{\{P_1\}\ \text{process}_1\ \{Q_1\}\ \ \{P_2\}\ \text{process}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \text{process}_1 \parallel \text{process}_2\{Q_1 * Q_2\}}$$

那么，我们对分离逻辑的介绍到此告一段落，如果有对更强的高阶分离逻辑感兴趣的读者可自行阅读相关论文...

Part.5 : The End

通过前四章浮光掠影地介绍计算机科学中的逻辑学，想必大家对这一TCS Track B的分支领域有了基本的了解。

无论各位读者通过阅读本文对LICS是满怀热情也好；毫无感觉也好；被劝退也罢，只要能够有所收获，不认为阅读本文是浪费时间，便也值得...

LICS是一门深邃而有趣的学科，本文也仅仅谈及皮毛，如果想要了解振奋人心的最新成果，接受系统的逻辑学训练是必不可少的。

在本文的最后，给大家推荐一些相关的论文资料，以便感兴趣的读者自行探索学习：

1. Higher Order Separation Logic
2. An Introduction to Linear Logic
3. Iris – An Implementation of Separation Logic in Coq

至此，Logic in Computer Science正式完结！！！