定理证明与形式化方法

总述

最近接触到了一些软件验证和辅助证明方面的知识，感觉非常神奇~

(阅读本文需要一定的函数式编程基础，如知晓什么是**ADT**(Algebraic Data Type))

软件验证应该算是软件工程中一个比较冷门的分支，毕竟不是所有程序都有必要在一种接乎数学的角度上严格证明其正确性的。但在类似于核工程，地铁系统等等对程序正确性要求较高，且一旦出错损失巨大的领域来说，软件形式化验证就显得至关重要了。

当然，在某种程度上，这也和如今软件形式化发展的状况有关。不得不说，从技术层面来看，想要让所有的程序都经过严格验证在现在看来是**几乎不可能的**；哪怕是对于一段只有1000行的代码，对其进行形式化验证也至少需要**9000**行左右的代码，极大增加了工程量。

为了降低形式化验证的复杂度，使软件验证的普及度增加，许多人也在自动化验证的方向上不断努力。尽管自动化验证仍然是一门新生的学科，但我相信，随着自动化验证的不断推进，未来形式化验证会越来越流行。除此之外，形式化验证也推动了与之相关的另一个领域—程序分析(Static Analysis)的发展。程序分析更加注重于分析程序本身的性能,可靠性和安全性，在工业中的应用也更为常见。

回过头来谈谈计算机辅助证明，这可谓是数学和计算机科学的交叉领域。众所周知，数学中应用最为广泛的公理化基础是ZFC公理集合论(以下简称ZFC)。集合论的语言符合人类自身的思维习惯，也极大推动了严格数学的发展。但ZFC不能够与计算机形式化语言适配，使得即便在如今，数学证明的审阅也仍然依靠人工。例如Perelman对Poincare猜想的证明的验证，就耗费了数位几何学家几年的精力(而且还不能保证**绝对**正确)。

也正因为如此，机器化证明，尽管令人反感，却仍然在蓬勃发展。需要说明的是，机器化证明并不代表计算机能够代替人类证明数学定理，而仅仅是起到辅助验证的作用。

自从**同伦类型论**(Homotopy Type Theory，简称HoTT)指出了构建数学新的公理化基础的可能性之后，使用依赖类型的编程语言(如Coq, Agda)从类型论和同伦数学的角度上描述数学公理在这一领域的流行度逐渐上升（详细的可以参考HoTT-Coq和HoTT-Agda）。我们接下来的演示代码也将会涉及到Agda这一有趣的语言。

这里的**同伦**(Homotopy)是代数拓扑中的概念，存在许多深刻的几何和代数背景等待挖掘。而与代数拓扑密切相关的另一个领域—**范畴论**在HoTT中也有广泛应用。

这篇文章的参考文献主要有Benjamin C. Pierce的《Software Foundations》和Philip Wadler的《Programming Language Foundations in Agda》.

从GADT开始

我最早接触到定理证明可能是在解决Codewars上的kyu1题目(A+B=B+A? Prove it! & A * B=B * A? Prove it!)时，要求使用Haskell证明加法交换律和乘法交换律。

前面我们提到过，几种常用的辅助证明器都是**依赖类型**(Dependent Type)的语言，暂时不深究什么是依赖类型，只需要知道依赖类型能够给定理证明提供巨大便利。但与此同时大多数依赖类型的语言都存在一定缺陷(Idris2可能是一个例外)，无论是Coq 还是 Agda都是Turing Incomplete Language，很难用于实际生产环境。

而Haskell本身并不是依赖类型的，这就导致Haskell在证明定理时会遇到一定的困难。那有什么方法模拟依赖类型呢?

对于Haskell来说，**GADT**(General Algebraic Data Type)就是这样一种扩展.

在Haskell中，我们这样定义一个列表：

```
List a = Nil | Cons a (List a)
```

进一步地，我们可以写出Nil和Cons的函数签名：

```
Nil :: List a
Cons :: a -> List a -> List a
```

我们可以发现，这两个函数都是返回List a这个类型，其中a是一个虚幻类型，没有一个实际的类型叫"a". 而在GADT中情况则稍有不同，我们通过一定的方式为原本的虚幻类型"a"添加一定的类型信息，从而对类型做出需要的限制，达到依赖类型语言的效果。

使用*GADT*重新定义列表：

```
data List a where
  IntNil :: List Int
  Nil :: List a
  Cons :: a -> List a -> List a

a :: List Int
a = IntNil

c :: List Char
c = Nil
```

如何利用GADT进行定理证明呢？由于篇幅限制，不能展开描述，感兴趣可以参考Haskell中的"定理证明".

Haskell-like Language — Agda

Agda基础知识

看完了GADT，我们来介绍一种真正的定理证明器—**Agda**.

Agda是一种类Haskell的依赖类型语言，可以作为构建构造性证明的证明辅助工具。

与另一个常用的辅助证明语言Coq不同，Agda的设计更多地参考了Haskell(Coq更多参考的是OCaml),它本身并不提供单独的证明策略语法(Tactics,在Coq中至关重要)，所有的证明均以函数式编程的方式书写。也正因为如此，Agda具有许多常规函数式程序语言的要素，如模式匹配(Pattern Matching),模块(Modules),数据类型(Data Types)等等。

定理证明对于计算机科学和数学学科最大的意义可能在于它揭示了逻辑与计算之间最深刻的联系：形式化的结构可以看作是逻辑中的命题，也可以视作计算中的类型。而使用Agda进行定理证明不容易陷入证明策略语法细节中，更能体现这种"命题即类型(Propositions as Types)"的核心思想。

对于我们来说，最为熟悉的形式化结构，莫过于自然数了~自然数是无限集合，想要依靠传统的枚举，是不可能在计算机当中真正表示自然数的，但如果我们将其视之为特定的类型，则只需要简单的几条推导规则，便能定义自然数。实际上，自然数在Agda中被叫做**归纳数据类型**(Inductive Datatype).

我们将自然数的类型记为$\mathbb{N}$,记0，1，2，3等自然数是类型$\mathbb{N}$的**值**(Value)，表示为$0:\mathbb{N}$,$1:\mathbb{N}$,$2:\mathbb{N}$,$3:\mathbb{N}$等等.

```
data N : Set where
  zero : N
  suc : N -> N
```

其中$\mathbb{N}$是我们定义的数据类型的名字，而$zero$和suc(*Successor*的简写)称为该数据类型的**构造子**(*Constructor*).

我们仔细考察自然数的定义过程，可以发现和我们之前在Haskell中学习的GADT十分类似.

```
--定义起始自然数zero
zero : N
--归纳推导得到另一个自然数suc zero
suc zero : N
--重复上面的步骤
suc (suc zero) : N
--以此类推
suc (suc (suc zero)) : N
suc (suc (suc (suc zero))) : N
......
```

进一步地，我们还能定义自然数上的运算，如常见的加法和乘法~

```
_+_ : N -> N -> N
zero + n = n
(suc m) + n = suc (m + n)
```

```
_*_ : N -> N -> N
zero * n = zero
(suc m) * n = n + (m * n)
```

这个构造过程说明：加法和乘法的定义都是**递归**(Recursion)的.

Currying

在Agda主题之外，我还想介绍一个函数式编程中的重要概念—**柯里化**(Currying).

柯里化从某种程度上来说，可以视为一种技巧(或者说**语法糖**)，但其意义又不只是技巧那么简单。

我们可以回想一下高数中学习的二元函数$f(x,y)$，我们可以说$f(x,y)$是一个接受两个参数的函数，唯有输入两个参数时才能返回具体的值；那么很显然，当我们的$f(x,y)$只接受了一个参数时，返回的就应该是一个接受另一个参数的一元函数$f(x_0,y)$.

而这样一个将接受两个参数的函数表示为**"接受第一个参数，返回接受第二个参数的函数"**的技巧就被叫做柯里化。

我们可以拿自然数加法举例，`_+_ 2 3`可以表示为`(_+_ 2) 3`，而`(_+_ 2)`就表示一个**"将参数加2"**的函数，它接受参数3得到5.

柯里化的思想看似简单，却为函数式编程(**Functional-Programming**)的诞生提供了理论基础。通常的面向对象编程(Object-Oriented-Programming)接受参数后是不能返回一个**函数**的。而函数式编程则借助柯里化能够返回一个函数，进而还能将函数视为参数输入，构造**高阶函数**(Higher-Order Function).

定理证明

我们以证明加法结合律为例：

```
(m + n) + p ≡ m + (n + p)
```

实际上这等价于证明两条规则：

```
--------------------------------
(zero + n) + p ≡ zero + (n + p)

(m + n) + p ≡ m + (n + p)
--------------------------------
(suc m + n) + p ≡ suc m + (n + p)
```

这几段代码就是在阐述我们常用的数学归纳法的证明思路：如果对`zero`来说加法结合律成立，那么我们假设对某个$m \in \mathbb{N}$加法结合律成立(这一部分是我们的**归纳假设**)；只要我们证明在归纳假设的前提下，$m+1 \in \mathbb{N}$也符合加法结合律，则加法结合律对全体自然数都成立。

我们可以写出如下代码进行证明：

```
+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc zero n p =
  begin
    (zero + n) + p
  ≡⟨⟩
    n + p
  ≡⟨⟩
    zero + (n + p)
  ∎
+-assoc (suc m) n p =
  begin
    (suc m + n) + p
  ≡⟨⟩
    suc (m + n) + p
  ≡⟨⟩
    suc ((m + n) + p)
  ≡⟨ cong suc (+-assoc m n p) ⟩
    suc (m + (n + p))
  ≡⟨⟩
    suc m + (n + p)
  ∎
```

更多有关Agda定理证明的内容可以去阅读[《PLFA》](#) ~

形式化验证初探

我使用**Idris**编写了一个简单的程序：

```
findIndex : DecEq a => Vect n a -> a -> Maybe (Fin n)
findIndex x [] = Nothing
findIndex x (y :: xs) =
   case decEq x y of
     Yes _ => pure FZ
     No _ => [| FS (findIndex x xs) |]
```

这段程序的功能是在数组中检索元素并返回下标或不存在~

那么要如何对这段程序的正确性进行验证呢?

首先将这段程序的正确性转化为形式化结构—**逻辑命题**:

```
findIndex : DecEq a => (x : a) -> (xs : Vect n a) ->
            case findIndex x xs of
              Just i => Elem x xs
              Nothing => Not (Elem x xs)
```

我们采用最常见的Coq进行形式化验证:

```
notHeadNotTail : Not (x = y) -> Not (Elem x xs) -> Not (Elem x (y :: xs))
notHeadNotTail notHead notTail Here = notHead Refl
notHeadNotTail notHead notTail (There tl) = notTail tl

findIndexOk : DecEq a => (x : a) -> (xs : Vect n a) ->
              case findIndex x xs of
                Just i => Elem x xs
                Nothing => Not (Elem x xs)
findIndexOk x [] = absurd
findIndexOk x (y :: xs) with (decEq x y)
  | Yes Refl = Here
  | No notHead with (findIndexOk x xs)
    | ih with (findIndex x xs)
        | Nothing = notHeadNotTail notHead ih
        | Just z  = There ih
```

经过形式化验证的程序的正确性是毋庸置疑的~

当然了，对于Idris这样的依赖类型语言来说，形式化验证并不是非常必要的;(对于依赖类型来说)如果这段程序能够通过编译，那么它几乎百分百是正确的。

而对于我们日常使用的语言来说，形式化验证会更加复杂且有意义。水平所限，无法进行这方面的展示。

好了，本文的内容就此告一段落了.....

期待下一次再会!