形式验证、依赖类型与动态类型

aka 依赖类型传教文

前排提醒：你或许不需要一个游标卡尺。

本文面向一切有任何形式编程经验的读者，将会使用 Agda 和极少量其他语言展示代码例子。 读不懂的代码可以借助附近的文字描述理解它的含义，所以不用担心语言问题。

一切规矩照旧，这篇文章是一个 Agda 模块，我们导入一些基本库：

```
{-# OPTIONS --no-unicode #-}
{-# OPTIONS --without-K  #-}
{-# OPTIONS --safe       #-}
module DependentFunctionsVersusDynamicTyping where

open import lib.Basics
```

这次，我们需要使用 HoTT-Agda 中的自然数类型的相关定义：

```
open import lib.types.Nat
```

泛化阶：

```
variable i : ULevel
```

本人承诺不会在这种面向非 Agda 专业人士的文章里使用 Unicode 。

在这个没有 `null` 的语言里，我们有时还是需要使用 `null` 的：

```
data Nullable {i} (A : Type i) : Type i where
  nullptr : Nullable A
  new_    : A -> Nullable A
```

这样， `nullptr` 和 `new 1` 都是 `Nullable Nat` 类型的实例了（这些符号马上就介绍）：

```
_ = nullptr :> Nullable Nat
_ = (new 1) :> Nullable Nat
```

依赖类型的数组

定义一个数组类型，数组长度是类型的一部分：

```
infixr 5 _cat_
data Vec {i} (A : Type i) : Nat -> Type i where
  []    : Vec A 0
  _cat_ : forall {n} -> A -> Vec A n -> Vec A (S n)
```

这样，我们可以有空数组，以自然数类型为例：

```
_ = [] :> Vec Nat 0
```

其中， `:>` 左边是一个表达式（在这里是 `[]`），右边是它的类型。 `_ =` 表示把这个表达式赋值给一个无法被使用的变量，也就是丢掉这个表达式。 以这种形式写出来，我就可以在编译我的

博客的时候让 Agda 编译器检查我的这些代码片段是否在类型上是正确的。

把代码编译成网页是 Agda 编译器的一个功能，我目前正在着手改善它的这个功能。 也就是说，如果我给我的博客写一个 `main` 函数，我就可以运行我的博客了……

先不说这个，一个元素的数组，也就是一个元素和一个空数组连接的结果：

```
_ = (233 cat []) :> Vec Nat 1
```

两个元素：

```
_ = (666 cat 233 cat []) :> Vec Nat 2
```

观察发现，数组的类型 `Vec` 有两个参数，第一个参数是元素的类型 `Nat` ，也就是自然数，第二个是长度，这两个参数都是编译期已知的。 这个数组的定义常常被 Idris 语言的粉丝用于布道——因为它很安全，可以在编译的时候防止越界的情况发生，就像 C++ 程序员会推荐 `std::array` 一样。 ~~而由于 Agda 语言人气不足，别人布道的时间 Agda 程序员都写论文去了。~~

我们来尝试一下 Agda 的『小于』类型。小于关系是一个类型，如果我们有一个类型为 `a < b` 的变量，那么我们就『证明』了 `a < b` 。 也就是说，我们可以弄出一个类型为 `3 < 4` 的变量：

```
_ = ltS :> (3 < 4)
```

也可以有 `114514 < 114516` 类型的变量：

```
_ = ltSR ltS :> (114514 < 114516)
```

但是我们无论如何也弄不出 `1 < 1` 类型的对象——这是由 `<` 的定义决定的，这里就不展开说了（涉及的东西有点多）。

因此，我们可以写出这样一个函数：它『从长度为 `m` 数组中获取第 `n` 个元素』，并要求调用这个函数的人额外传入一个 类型为 `n < m` 的变量。

```
module GetAtIndex where
  _!!_<[_]> : {A : Type i} -> forall {l} -> Vec A l
          -> (n : Nat) -> n < l -> A
  (x cat _) !! O <[ _ ]> = x
  (_ cat a) !! S n <[ p ]> = a !! n <[ <-cancel-S p
```

这样，有了 `n < m` 的证明，我们无论如何都能安全地从数组里拿出一个元素了。 举一个简单的调用的例子。这是一个测试用的数组：

```
  sampleList : Vec Nat 4
  sampleList = 0 cat 1 cat 2 cat 3 cat []
```

我们试图取它的第 2 项，没有问题（下面的代码是让编译器试图验证『从刚才那个数组中取第 2 项得到的是 2』）：

```
  _ = idp :> ((sampleList !! 2 <[ ltSR ltS ]>) == 2
```

试试第一项？

```
  _ = idp :> ((sampleList !! 1 <[ ltSR (ltSR ltS) ]
```

运行时错误

```
module RuntimeErrors where
  open GetAtIndex
```

一个老生常谈（我在两个群里被问过了）的问题：如果下标和数组都是运行时获取的，编译器的验证不就凉了吗？

解决方法：在运行时进行判断——如果下标小于数组长度，那么在这样的条件下可以得到一个小于关系的证明， 就可以安全地进行函数调用。否则，你将不能调用这个函数，请自行处理非法输入。

给出一个简单的实现（库里面其实有一个 `Decidable` 的版本，但是它里面用了 Unicode ，我不想用，所以就自己写了一个）。 首先，我们需要判断两个自然数的大小，并在小于的时候返回小于的证明，其他时候返回 `nullptr`：

```
  lessThan : forall a b -> Nullable (a < b)
  lessThan O O = nullptr
  lessThan O (S b) = new O<S b
  lessThan (S a) O = nullptr
  lessThan (S a) (S b) with lessThan a b
  ... | nullptr = nullptr
  ... | new x   = new <-ap-S x
```

然后，我们就可以在不模式匹配、直接把参数拿来用的情况下，对 `_!!_<[_]>` 进行调用了。 这个函数通过接收任意的自然数和任意的数组来模拟运行时无法保证的输入，并返回可能不存在的输出：

```
runtimeInput : forall {m} -> Vec Nat m -> Nat ->
runtimeInput [] _ = nullptr
runtimeInput {m} v n with lessThan n m
... | nullptr = nullptr
... | new x   = new (v !! n <[ x ]>)
```

这个函数和 Java、JavaScript 函数就没什么可见的区别了，这也体现出了形式验证的一个小小的局限性。
但在运行时的情况下，我们还是能看出形式验证的好处的——类型签名里函数对参数需要满足的关系以非常清晰的形式呈现了（比如， `a > b` 这种表达式），表达的也很简洁， 函数的调用者无需阅读文档即可安全地进行 API 调用，运行时不需要任何错误处理——编译的时候就能保证所有的错误得到处理，运行时当然是把这些东西全部都擦除掉了。

稍微弱一点的类型系统

看到这里，我们会想起，像 Java 和 JavaScript 里类似的函数都会在参数不合法的时候抛出一个异常。 因此，如果我们的程序要足够健壮，我们会选择：

- 处理这个异常
- 判断下标是否小于长度，提前避免错误

具有证明命题能力的编程语言一般会强制程序员进行后者的操作，具有责任心的程序员会主动进行后者的操作， 绝大多数时间我们是忽略了这件事的——编译器、程序员、用户都没有去证明输入数据对程序来说一定合法。 当输入不合法时（尽管这很

少发生），我们的程序会崩溃掉（**运行时错误**），否则将正常运行。

运行时错误，从动态类型到静态类型的过程中已经大幅减少了。但这里可以看出，它还是存在的。

不知道 C++ 程序员有没有想到 SFINAE 呢？（笑）

但总而言之，把证明传来传去是一件很麻烦的事情，光是阅读命题本来就是额外工作，证明就更麻烦了。 像 C++ 的 SFINAE，在参数不合法的时候，如果不手动让编译器抛出一些可读的异常，编译器会产生大量的错误信息， 印在厕纸上都能用一年。

但是呢，会把程序写成这样的形式验证研究员都是垃圾（凭空制造复杂度），他们的思维都被限制了。

我们首先来看看大家觉得虽然邪恶但是写起来爽的『动态类型』。

更浅显的话题

程序员们常常对『动态类型和静态类型哪个更好』这一话题产生激烈的讨论。 这其实是一个完全没有意义的讨论，因为这首先是一个罗卜白菜的问题——两者都有能称得上是『优点』的地方； 其次不同的人对程序有不同的追求，有人想写出健壮可扩展的程序，有人只是想快速交付收钱； 再其次同一个人也有不同的需求，有时只是想批量处理一些文件，有时需要构建长期维护的大型项目。

对于简单的批处理需求，我们甚至会完全不考虑任何的可读性、可维护性、可扩展性、安全性、鲁棒性， 写出类似这样的代码（已经整理过了，再看不懂就是语言的问题）：

```
while (my $line = <$imguiHeader>) {
  chomp $line;
```

```
    $_ = $line;
    if (m!^// dear imgui, (.*)$!) {
      my $commandPrefix = 'cd ../imgui && git rev-parse';
      print "--- Generating for dear-imgui version $1\n", "---
      print "--- Branch ", `$commandPrefix --abbrev-ref HEAD`,
    }
    elsif (/^\s*enum\s(\w+)_/) {
      $currentEnum = $1;
      $currentEnumMangled = $currentEnum =~ s/^Im(Gui)?(.)([^\
      # =~ s/([a-z])([A-Z])/$1.@{[ lc $2 ]}/gr;
      print "\n", '--{', '{', "{ $currentEnumMangled\n", "---\
    }
    elsif (length $currentEnum) {
      if (/\}/) {
        print "$moduleName.$currentEnumMangled = $currentEnumM
        print "$currentEnumMangled = nil\n", '--}', '}' "} $cu
        $currentEnum = ''; $previousDefault = 0;
      }
      elsif (m!^\s*${currentEnum}_(\w+)((\s*=\s*([^,/]+))?),?\
        print "\n--- $6\n" if $5 and length $6;
        $mangledName = length $1 == 1 ? "_$1" : $1;
        $value = $2 ? calculate $4 : $previousDefault++;
        print "$currentEnumMangled.$mangledName = ", $value, "
      }
      elsif (m!^\s*//(.*)!) { print "---$1\n" }
    }
  }
```

这样的代码完全是一个精雕细琢的艺术品——它的每一个字符都是我（没错这是我写的……用来提取一段 C++ 代码中的一部分定义并翻译成 Lua） 小心翼翼写出来的，稍微改一点就会出错，输入数据有一点点变化也可能出错，静态分析工具对这个代码的正确性完全没有保障（Perl 的解释器在运行你的代码的时候也不知道你的程序的目的，只是一行一行地运行而已）。

动态类型往往因为其灵活性被一些程序员（不是我）喜欢。比如我们可以写出这样的 JavaScript 程序：

```
const Error = {}
const getAtIndex = (a, n) => n < a.length ? a[n] : Error;
```

简单地运行：

```
> getAtIndex([1, 2, 3], 2)
< 3
> GetAtIndex([], 1)
< {}
```

这个 `getAtIndex` 接收一个数组和一个整数，返回『有时是一个数组元素，有时是 `Error`』。 用 TypeScript 描述一下它的类型，就是：

```
<a>(a[], number): a|Error
```

而我们调用这个函数，它到底会返回什么类型，我们也必须看它的实现才能知道。 返回 `Error` 只是方便起见，我们大可使用异常来代替返回一个表达异常的对象。 但它做到了一点——在我

们不想处理错误的时候，我们可以不处理错误，让运行时炸。
在我们想处理错误的时候，我们可以处理错误。

但这给了我们一个启发。或许我们可以实现这样的函数：在很明显没有错误的时候，我们可以不处理错误。 在有可能错误的时候，我们需要处理错误。

或许有人会说，我们不还有异常吗？但首先异常这一概念首先本身就足够『运行时』， 我们对什么样的函数会抛出什么样的异常都一无所知。Java 的 Checked Exception 作为一个例外，是静态的异常， **但是这种语法结构和返回带有错误信息的类型**（可以理解为 Haskell 的 `Either`， Rust 的 `Result`） 是同构的（这篇文章本身说的很有道理，但请不要看它所引用的《给 Java 说句公道话》）。

> 首先，写 C# 代码时最让我头痛的事情之一，就是 C# 没有 CE。
> 每调用一个函数（不管是标准库函数，第三方库函数，还是队友写的函数，
> 甚至我自己写的函数），我都会疑惑这个函数是否会抛出异常。由于 C#
> 的函数类型上不需要标记它可能抛出的异常，
> 为了确保一个函数不会抛出异常，你就需要检查这个函数的源代码，
> 以及它调用的那些函数的源代码……
> 也就是说，你必须检查这个函数的整个“调用树”的代码，
> 才能确信这个函数不会抛出异常。这样的调用树可以是非常大的。说白了，
> 这就是在用人工对代码进行“全局静态分析”，遍历整个调用树。
> 这不但费时费力，看得你眼花缭乱，还容易漏掉出错。
> 显然让人做这种事情是不现实的，所以绝大部分时候，
> 程序员都不能确信这个函数调用不会出现异常。

在静态类型中，我们要么使用静态的带错误信息的返回类型（强制检查错误，即使很明显的正确的代码也必须处理错误）、要么

使用异常（静态变动态，你干啥不去用 Python）。

而且，刚才那段 JavaScript 根本不可能在静态类型语言里实现（当然，这里没有考虑支持和类型的语言。但是要考虑到 支持和类型的编程语言大部分都是渐进类型或者以动态类型编程语言为目标语言的编程语言）——我们写不出一个同时有两种可能的返回类型的语言。

也就是说，动态类型能做（即使做的不完美，比如那个 `getAtIndex` 并不会强制你处理异常，即使可能异常）的事静态类型做不到； 静态类型能做动态类型做不到的事就多了去了，在绝大多数情况下使用静态类型还是能带来远高于动态类型的编程体验的（ 包括 IDE 补全重构跳转、尽可能减少了类型错误等），这里就略过不提了。

难道鱼和熊掌真的不可兼得吗？

不，小孩才做选择。

我全都要

```
module IQuanDouWant where
  open RuntimeErrors
```

这时我们就需要用到依赖类型了。

我们除了传入证明作为参数之外，还可以使用依赖函数！

我们的 getAtIndex 函数，在下标越界的时候返回 Error，其他时候返回正常的值。 我们先定义一个表达错误的类型：

```
data ???? : Type0 where ??? : ????
```

我们的函数 getAtIndex，返回的类型，是根据两个参数决定的。 所以，我们需要先写一个函数，接收两个自然数，返回『我们的 getAtIndex 的返回类型』， 也就是说如果左边大于右边，就返回『自然数』这个类型，否则返回表达错误的类型。请注意，只有带有较好的依赖类型支持的类型系统才能做到这种事。如果一个语言声称自己支持依赖类型， 你可以问问他能不能写出这样的代码。

```
returnType : (_ _ : Nat) -> Type0
returnType O _ = ????
returnType (S _) O = Nat
returnType (S n) (S m) = returnType n m
```

然后，我们的 getAtIndex 需要返回的，就是这个类型。

```
getAtIndex : forall {m} -> (v : Vec Nat m) -> (n
getAtIndex [] _ = ???
getAtIndex (x cat _) O = x
getAtIndex (_ cat v) (S n) = getAtIndex v n
```

看看看！这个函数的实现真的好简单啊！
我们试试证明它的性质？我们先传入合法的参数，看看是不是就直接返回了数组的元素：

```
open GetAtIndex using (sampleList)
_ = idp :> (getAtIndex sampleList 2 == 2)
_ = idp :> (getAtIndex sampleList 1 == 1)
```

真的耶！那如果是超过长度上限的下标，是不是就会返回那个错误类型呢？

```
_ = idp :> (getAtIndex sampleList 233 == ???)
```

是的！

这个函数的返回类型真的在变耶！而且这是非常非常纯正的静态类型编程哦！

你看，静态类型做不到的事，依赖类型做到了。动态类型没法编译期进行类型检查，依赖类型可以。

实际应用

```
module FSharpLang where


  -- ...
```

就懒得写了，毕竟本来就是我的另一篇博客。这是一个依赖函数的经典应用——`printf`。它的类型是：`String -> ??`，其中`??`具体的值，取决于`String`参数中有多少个`%d`，`%c`等。

创建一个 issue 以申请评论

Create an issue to apply for commentary