Haskell大作业实验报告

软件61班 游凯超

学号 2016013234

基础部分：对AST的类型检查与求值

- 对应文件 EvalType.hs 与 EvalValue.hs
- 执行方法 根目录下 `stack ghci` 即可
- 实现目标
  - 所有的表达式类型都进行了实现
  - 支持了递归函数
  - 支持了高阶函数
- **加分项**：实现了对ECase的支持(没有对 `Non-exhaustive patterns in case` 进行处理，只是进行匹配，匹配上了就运行，没匹配上就报错)

```
-- 类型检查 PASS
-- 求值 = 1
case True of
  True -> 1

-- 类型检查 FAIL
case False of
  True -> 1
```

- 实现思路：
  - 类型检查思路
    - 采用 `Control.Monad.State`
    - 记录每个变量对应的类型
    - 对于重名的变量，采用最近的定义的类型
    - 具体来说，使用 `Map String [Type]` 来记录类型状态，如果一个名字已经有了定义，又有了新的定义，那么，新的类型插入到已有的列表的最前端，这样，后续的对这个名字的引用就能找到最近的类型了。

```
(\x        -- state: x -> [TInt]
  -> x +  -- typeof x is TInt
        (\x  -- state: x -> [TArrow TInt TInt, TInt]
          -> x 1) -- typeof x is TArrow TInt TInt
                (\y -> y + 1))
                            (1)
```

        - 高阶函数在这里实现得很自然，因为这里的函数都是一元函数，要定义高阶函数也只能把多个参数分开，逐个定义，所以应用的时候只要检查第一个参数的类型是否匹配，匹配了即可应用。
    - 求值的思路
        - 采用`Control.Monad.State`
        - 只有类型正确的表达式才会被求值
        - 记录每个变量(名字)对应的值
        - 对于重名的变量，采用最近的求出来的值
        - **对于函数类型的变量(名字)，需要记录它的函数体，及定义时的环境闭包(变量名与值的映射)**
        - **函数应用时，需要把当前的语境和函数体对应的闭包做并集，再去求值**
        - 所以函数的值的类型是**VFunc String Expr Context**
- 困惑点
    - 代码里面充满了模式匹配，但是又无法消除。比如对于`Value`的模式匹配，由于`Value`的几种模式里面包含了几种不同的类型，可能的做法是用newtype分别包装成新的类型，然后定义个`functor`，如`newtype IntValue Int = Value`，但是就算定义了`functor`，`fmap`里面也很复杂，而且有的地方还是`Int/Char/Bool`三种类型都可以，代码重复无可避免。弃疗了，就这样做模式匹配好了。
    - 尽力做了一点点的减少重复，比如类型检查时，把`-*/`的类型检查都直接调用`+`的类型检查函数。但是在求值时，与其抽象出一个函数，不如直接复制四份代码，只要改动一个运算符即可……

选做部分：AST编译到JavaScript代码

- 对应于文件`EvalJS.hs`里面的`evalJS :: Program -> String`函数
- 类型正确、能够求出值的表达式，才翻译成JS代码
- 核心是`eval :: Expr -> String`函数
- 选择JS，是因为JS和haskell有类似之处，比如都支持在函数内定义函数(因此都有闭包的概念)
- AST到JS的转换基本是按照语义逐个表达式翻译的，有一些坑点记录如下
    - 等号和不等号改为`=== !==`,严格判断
    - JavaScript的逻辑表达式返回值并不都是布尔值(类似python的逻辑表达式，`a || b`，如果`a`的真值为真，则返回`a`这个对象，而不是返回`false`)，所以，对于逻辑表达式的每一项，我使用了`!!a`的方法来把表达式强制转换为布尔值
    - JavaScript的整数除法的结果是小数，所以用`Math.floor`包装

- JavaScript的`if-else`也是有值的，**但是JS里面的If-Else语句在有些需要值的地方不能使用**，例如

```
// 报错 Uncaught SyntaxError: Unexpected token if
function f(x){return if(x>0){x}else{-x}}
```

  所以，我们必须使用`a?b:c`的三元表达式

- Let语句的翻译：
  - 转换成匿名函数的即时调用

```
-- haskell
let a = 1 in a * 2
```

```
// JavaScript
(function (a){return a * 2})(1)
```

  - 可能的内部函数变量名重名问题：由变量的作用域自行解决
- Lambda表达式翻译--使用匿名函数
- ELetRec表达式翻译：转化成匿名函数调用，参数为函数，参数名为函数名

```
-- haskell
let f x = if x > 0 then x * f (x - 1) else 1 in f 5
```

```
//JavaScript
(function (f){return f(5)}) (function f(x){return (x > 0 ? x * f(x-1)
: 1)})
```

  感谢JavaScript，即使是临时参数也可以起函数名，完美地解决了这一点问题。

- ECase模式匹配翻译
  - `case exp of …`先翻译成一个变量定义`$_tmp = exp`(随便用一个临时变量的名字即可)
  - 后续的`a -> aa`语句翻译成一系列的条件判断
  - 特殊的模式匹配语句`var -> xxx`翻译成`let in`，同时把条件判断的条件改为`$_tmp == $_tmp`，即永真。
  - 同样，条件判断通过三元表达式`a?b:c`完成