U. hs 编程入门

简单介绍

U.hs 是一门基于无类型 lambda 演算的纯函数式、懒惰求值的弱类型语言。它的灵感来源于 Unlambda.

在 U.hs 中，所有（除若干个特例外）的东西都是函数，每个函数接受一个参数（也是个函数），返回一个函数。

U.hs 使用 Lisp 风格的括号语法来表示函数的应用。同时，可以用 lambda 来构造函数。例如，我们可以定义一个单位函数 id:

(def id (\x x))

其中，\表示 lambda，即，我们定义 id 为接受参数 x 之后把 x 返回的函数。

函数通过括号来应用到参数上。例如，(id a)表示把 id 作用在 a 上. 括号内可以连续应用多个参数：

(a b c c)等价于(((a b) c) d)

多个变量可以串联在同一个 lambda 中：(\a b c)等价于(\a (\b c))

有几个已经为你定义好的函数，比如，

(def K (\x y x))

(def S (\f g x ((f x) (g x))))

实际上，任何的 lambda 表达式都可以通过 K 和 S 组合而成。比如，id 就是(S K K). 使用 S 和 K 代替 lambda 可以让你的程序变得人类不可读（如果你真的想这么做的话）。

为了让程序做点有用的事，U.hs 内置了整型和浮点型的数值（数值是唯数不多的不是函数的东西）。你可以用“前缀表达式”风格的方式来书写计算：

(+ (* 1 2) 3)　得到(1*2)+3 = 5

如何进行条件判断呢？U.hs 中预置了两个“布尔值”：

(def True K)

(def False (K (S K K)))

布尔值的特点是，如果 b 是一个布尔值，那么(b true_value false_value)就是：如果 b 是 True，那么 true_value；否则，false_value. 归功于懒惰求值，只有会被使用的那一分支会得到运行。于是，我们可以写出类似

(def safeinv (\x ((= x 0) 0 (/ 1 x))))

这样的函数出来。当 x=0 的时候，得到 0. 否则，得到 1/x.

下面的章节从不同的维度介绍 U.hs 的更多细节。

安装指南

随代码分发的有一份 readme.pdf. 里面详细说明了编译步骤。

最简单的运行 U.hs 程序的方法是用解释器运行它

```
$./urun helloworld.u
helloworld
```

另外，也可以把程序编译成可执行程序（需要你有一个可用的 llvm 工具链）。

```
$./ucomp helloworld.u
$./helloworld
helloworld
```

程序结构

每个 U.hs 程序都写在.u 文件中。一个文件中可以 import 若干别的文件，定义一些符号，同时有一个“主块”. 例如，一个实现按数值把字符串相加的.u 程序

```
(import* str)
(def strAdd (\a b (+ (atoi a) (atoi b))))    ;comments goes here. inline comments use #| |# notation
(print strAdd "10" "3")
```

这里，print 函数被定义为 id. 于是，最后一行也可以这样写：(strAdd “10” “3”). 主文件的主块(即，main.main)会被打印出来。你也可以这样书写主块：

(def main (strAdd “10” “3”))
如果想要输出更多的东西，需要后面章节介绍的 IO 机制。

import*表示把 str.u 中的所有符号导入进来。*表示“非限定的导入”。所谓“限定”，即必须用模块名.符号名的方法（例如，str.atoi）来引用符号。非限定的导入得到的符号可以直接使用。文件之间的导入关系必须是一个有向无环图。
(def name body)语句把 body 的值绑定到 name 上. body 中引用的都是之前已经定义好的值。例如，下面的语句是合法的：

```
(def x 1)
(def x (+ 1 x))
(def x (+ 1 x))
```

但是，这样的最终效果是得到了一个值为 3 的 x 而不是无穷循环。实际上，def 只不过一种 IIFE 风格的“语法糖”。以上的 def 块会被翻译为：

```
((\x ((\x ((\x …) (+ 1 x))) (+ 1 x))) 1)
```

那么如何定义递归函数呢？我们只需要一个把函数应用到自身上就行了：

```
((\f (f f)) (\x (+ 1 (x x))))
```

这个得到了一个(+ 1 (+ 1 (+ 1 ...)))的无穷的表达式（死循环）。
有一个更优雅的方式是使用 Y 组合子：

```
(def recur (\f ((\x (f (x x))) (\x (f (x x))))))
```

这样我们刚才写的无穷+1 函数就可以写成(recur (\x (+ 1 x)))或者(recur (+ 1)).

let 可以用来在一个模块内定义只在模块内可见的符号：

```
(let local_method body)
```

同时，let 也是一个表达式：

```
(let
    (a body_a)
    (b body_b)
    c
)
```

等价于((\a ((\b c) body_b)) body_a)

IO 操作

在纯函数式语言中进行 IO 是一个挑战。在 U.hs 中，所有的 IO 都是通过回调函数实现的。例如，可以使用 putChar 输出一个字符：

```
(putChar 'a' callback)
```

其中，callback 是一个函数。putChar 'a'的结果（0 表示成功，其他表示失败）会传给 callback 并继续运行。下面是把输入的一个字符送到输出并退出的程序：

```
(getChar (\a (putChar a (\_ (exit 0)))))
```

exit 使程序的运行结束，所以不需要回调函数。

一种理解“回调”的方法是，把(getChar c)想象成某种特殊的（无副作用的）值。这样，我们的整个程序就是一个由 getChar、putChar 等等串起来的“值”。之后，“运行环境”把这个值一层一层的拆开，将其中蕴含的副作用依次执行，直到得到(exit ?)为止。其实，Haskell 中我们也可以这样写程序：

```
(getLine >>=)  (\line ->  (putStrLn >>=)  (\_ -> return () ))
```

这样，也可以把它想象成是某种“回调”。

为了让这种级联的回调更好写，U.hs 中有一个 run 结构。上面的程序可以写作：

```
(run
    (a getChar)
    (_ (putChar a))
    (exit 0)
)
```

最后一个表达式之前的都是“绑定”语句。(name body) ...会变成(body (\name ...))
do 是和 run 类似的结构。(do ...)被定义为(\return (run ...))。例如，下面是如何读入两个字符：

```
(def readTwo (do
    (a getChar)
```

```
    (b getChar)
    (let c (pair a b))
    (return c)
))
```

上述程序中，let 用于在一个 do/run 块中绑定一个符号。pair 被定义为(\a b f (f a b))。

灵活使用 run/do 语法可以极大的方便程序的写作。例如，可以用下面的方法读两个字符然后返回第二个：

```
(do
    (a b readTwo)
    (return b)
)
```

(a b readTwo)会被翻译成(readTwo (\a b …))

甚至，可以使用空的绑定列表：

```
(do
    (readTwo)
    (K)
    return
)
```

这个就等价于(\return (readTwo (K return)))。你可以验证一下，K（即(\a b a)）会“吃掉”readTwo 回来的第一个参数。第二个参数被 return“吃掉”，即，被作为返回值。熟悉函数式编程的人可以看出，可以用函数复合运算符“.”进一步化简上式：

```
(. readTwo K)
```

当然，过分的使用这种“简结”的写法会显著降低程序的可读性。

除了 getChar、putChar，U.hs 中还有如下内置的 IO 函数：

open filename mode callback

filename 是文件名。mode 是 ReadMode、WriteMode、AppendMode、ReadWriteMode 之一. 这个操作到一个文件句柄并传给 callback.

getCharF handle callback

从句柄 handle 指向的文件读入一个字符，并把结果传给 callback. 如果遇到 EOF，那么得到-1. getChar 就是(getCharF stdin)

putCharF handle char callback

输出一个字符。返回 0 或者错误码。

peekCharF handle callback

非获取性的读入（读入但在缓存中保持这个字符）一个字符。-1 表示 EOF.

getArg callback

从命令行参数（可以把它想象成一个特殊的文件）读入一个字符。如果已经没有更多字符，返回-1.

close handle callback

关闭文件。返回 0 或者错误码（如果 handle 不合法的话）。

system cmdline callback

新建一个 shell，运行 cmdline. 退出值返回给 callback.

原则上，以上 IO 函数使得 U.hs 成为一个有可能自举的语言。当然，进行自举需要相当大的工作量。

基本数据类型

每个文件都隐式的导入了 prelude.u. prelude 中包含了各种常用的数据结构的定义。

例如，之前举过的布尔型的例子：

```
(def if id)
(def True (\a b a))
(def False (\a b b))
```

于是，可以像其他语言一样写 if then else 表达式：

```
(if condition
    true_block
    false_block
)
```

更加具有 U.hs 风格的写法是把 if 省去：

```
(condition true_block
```

```
    false_block
)
```

run 和布尔值是好朋友。你可以验证一下，下面的函数真的干了你认为它应该干的事情：

```
(run
    ( ((> a 10) “too large”))
    ( ((< a 5) “too small”))
    “good value”
)
```

布尔之外，最常用的就是 list 了。一个 list 定义为：

```
(def empty (\a b a))
(def cons (\head tail (\a b (b head tail))))
```

对于一个 list，你可以用（并且只能用）类似 pattern matching 方式来访问它：

```
(L null_value (\head tail non_null_value))
```

当列表 L 是空的时候，得到 null_value. 否则，针对 head 和 tail 对得到 non_null_value

注意，empty 和 True 是一模一样的。你可以大胆的混用它们。类型系统的缺乏（万物皆函数）使得 U.hs 事实上成为了一门弱类型的语言。如果你错误的混用了数据类型（例如把 False 当作了一个 list），除非涉及到内置的数值或函数（把数值作用到另一个数值上会产生一个异常），通常并不会导致立即报错。这也使得 U.hs 的程序调试起来很需要想象力。

cons 把一个元素和一段列表连接起来（放到头部）. 可以用 list“语法糖”简化列表的构造：(list a b c)等价于(cons a (cons b (cons c empty))

字符串被实现为 int 的列表。“abc”等价于(list 'a' 'b' 'c')或(list 97 98 99). 如果想在整型与浮点型之间互转，可以使用 toInt 和 toFloat. 整型、浮点型和字符串之间的转化由 str.u 中的 atoi/itoa/ftoa/atof 实现。

run 和 list 也是好朋友。下面的程序演示了如何得到字符串的头几个元素：

```
(run
    (line readLine)
    (let onerror (exit 1))
    (a (line onerror))
    (b (^2 onerror))
    (c (^2 onerror))
    (d _ (^2 onerror))
    (_ (putStrLn (list a b c d)))
    (exit 0)
)
```

其中，^2 的定义是(\x y f (f x y)). 它负责“吃掉”上一个语句未接受的一个参数，之后处理错误、把值喂给后面的函数。

当你确信列表的长度是多少的时候，可以使用“解构符”来进行列表的拆分：

```
(run
    (a b c (^list “abc”))
    …
)
```

其中，解构符被定义为(def ^list (flip (foldl id))). flip 的定义是(def flip (\f x y (f y x))); 有兴趣的读者可以研究一下这样写为什么是对的。

另外一些有用的数据类型包括：

```
(def pair (\a b f (f a b)))
```

于是，((pair a b) True) = a, ((pair a b) False) = b

另有 fst 和 snd 两个函数用于访问 pair

```
(def fst (feed True))
(def snd (feed False))
```

其中，feed 的定义是(\x f (f x))

更优雅的拆分 pair 的方法依然是使用 run 语句：

```
(run
    (a b (pair 1 2))  ;  a becomes 1, b becomes 2.
    …
)
```

同样的，还可以定义三元组(\a b c (f a b c))、四元组(\a b c d (f a b c d))等等。prelude.u 中提供了^2、^3、^4 等几个符号来帮助你定义元组。

还有一个用来处理“可能不存在值”的类型 Maybe:

```
(def nothing True)
(def just (\x (\a b (b x))))
```

对于一个类型的值 m，你可以用(m nothing_value (\x something_value))来访问它。

Maybe 在 run 语句中的作用类似于“异常处理”

```
(run
    (let m some_maybe_value)
    (x (m failure_case))
    (some_operation x)
)
```

U.hs 中使用 run 语句完成了回调级联、元组解析、guard 等一系列功能。那么，什么是 U.hs 中的 monad 呢？

例如，假设我们想实现 Haskell 中的 List monad.

```
do
    x <- [1,2,3]
    y <- [x,negate x]
    return y
```

由于 U.hs 中没有函数重载，所以我们需要定义

```
(def returnList (\x (list x)))
(def bindList (\x f (concat (map f x))))
```

然后程序就可以写成

```
(run
    (x (bindList (list 1 2 3)))
    (y (bindList (list x (neg x))))
    (returnList y)
)
```

函数重载的缺乏使得每个 bind 和 return 都必需指定类型。U.hs 中进行“重载”的方式是使用高阶函数。例如，一个广义的 sum：

```
(def sumT (\add zero l (foldr add zero l))))
```

针对 list monad 的例子，可能下面的语法变换之后的样子更接近 Haskell 一点：

```
(def runList (^2 bindList returnList))
(runList (\<- (do
    (x (<- (list 1 2 3)))
    (y (<- (list x (neg x)))
    (return y)
)))
```

标准库、示例程序

随源码附带了几个例子与模块。包括：

prelude.u：会被所有文件默认导入的库. 包含了基本定义（S、K、id、if 等）、列表操作（cons、map、foldr 及其他）、函数式操作（feed，.，recur 等）与另外一些常用的符号。

str.u : 字符串处理（atoi，itoa，splitStr 等）

io.u : 基本的输入、输出（readLine，putStrLn 等）

algorithm.u ： sort 等

有几个测试用的例子：

sorttest.u 把 stdin 读入的行按字典序排序之后输出

matmul.u 进行矩阵乘法。第一行读入三个数 n，k，m，表示矩阵的大小。后面跟一个 n 行 k 列的矩阵和一个 k 行 m 列的矩阵。输出它们的积（一个 n 行 m 列的矩阵）。

关于实现

源码中给出的是 U.hs 的解释与编译的一个 haskell 实现。U.hs 运行在一个“Lambda 表达式虚拟机”上。虚拟机提供的操作包括：

Abs：进行函数的 abstraction

Ref：引用一个 free variable

Apply：将一个函数作用到参数上

另外还有创建各种数值以及内置函数等。

ULambdaExpression 定义了 Lambda 表达式的类型 LExpr。

UParse 负责把.u 源码去掉所有的“语法糖”转换为 LExpr.

UModuleLoader 负责处理 import 和 def 的逻辑。

URuntime 和 UEnvironment 分别实现运行 LExpr 的虚拟机和负责提供与“真实世界”交互的“环境”。

UInterpret 实现 repl 与解释执行的功能。

UCompile 负责把 LExpr 翻译为 llvm-IR 表示的虚拟机代码，之后和 native 的虚拟机实现（simpleruntime.ll）一起通过汇编器变成可执行文件。

项目中，haskell 与 c 代码分别 1400 行左右，U.hs 代码 300 行左右。