2016060104002_2017060103023

Haodong Liao & Fan Gao

July 17, 2019

Preface

This is a LaTeXreport written by Haodong Liao and Fan Gao, students of UESTC who participated in the summer school of UCPH. More codes of this summer school are at Repository.

Introduction

Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution[1]. It is one of the three fundamental conputational models and is the mother of all functional programming languages.

In this assignment, we needed to implement an interpreter for terms of untyped λ calculus as presented in the given slides. We used methods like pattern-matching, tuple and so on to meet the requirements and we had to say that figured out the logic of recursion was very challenging for us, thankfully, things went well.

Analysis and Design

Term: According to [1], the objects of study in λ-calculus are $\lambda - terms$, and the set of *pre-terms* has three kinds of elements: *variable, abstraction and application*, so it was not hard for us to implement the embedded domain specific language *Term*, for the grammar *pre-term* with the given slide *Lambda calculus AST*.

Substitution: In this sub-task, we needed to implement the function *substitute* which performed the substitution.

Since we could ignore the free variable side condition, there were three branches needed to be handled.

Var x: if x was equaled to the given substitute argument, then we substitued it with the target argument.

Abs (x,y): according to the structure of abstraction, x was a variable, so we needed to compare y with the given substitute argument and call this function recursively.

App (x,y): same as abstraction, but it required to substitute both x and y recursively with the given argument (if they were equivalent).

Reduction: To implement the function **reduction**, we substituted in the appropriate order recursively until a β-normal form was reached.

Var x: if an β-reduction had already been performed on this term of *VarType*, then returned itself directly.

Abs(x, y): y was the *abstraction body* and needed to continue performing reduction on it until reaching a *variable*.

App(x, y): if x was an *App*, reduction would be performed on both parts of it. If x was an *abstraction*, a substitution would be performed and then a reduction on the result.

Evaluation: We performed β-reduction on the *abstraction* in an *application* recursively by calling **reduce** in this sub-task. The order of evaluation also deserved special notice to avoid incomplete reduction.

Additionally, a problem was that some λ-term would end in an infinite β-reduction loop. To solve this, we defined two variable *red* and *cnt*, which represented whether a β-reduction was performed on this λ-term and count the times. When the reduction iterated times reached 100, this loop would be ended immediately and returned its original term. This worked well for special situations in our program.

Program description

Our implementation of **Term** was as follows:

```
type Term =
    | Var of VarType
    | App of Term * Term
    | Abs of VarType * Term
```

As it was shown, *Var, App and Abs* represented *variable, abstraction and application* separately.

Our implementation of **substitute** function was as follows:

```
let rec substitute (term: Term) (var: VarType) (sub: Term) =
    match term with
    | Var x when x = var -> sub
```

[1] https://en.wikipedia.org/wiki/Lambda_calculus

```
        | Abs(x, y) -> Abs(x, substitute y var sub)
        | App(x, y) -> App(substitute x var sub, substitute y var sub)
        | _ -> term
```

The key to this function was to compare and substitute the type *Term* to target argument.

Our implementation of **reduce** function was as follows:

```
let mutable cnt = 0
let rec reduce ((term, red, cnt): Term * bool * int) =
    match term with
    | Var x when red = true -> Var x
    | Abs(x, y) ->
        if cnt >= 100 then Abs(x, y) else Abs(x, reduce (y, red, cnt + 1))
    | App(left, right) ->
        if cnt >= 100 then App(left, right) else
        match reduce (left, true, cnt + 1) with
        | Var _ -> App(left, reduce (right, red, cnt + 1))
        | App(x,y) -> App(reduce (x,red,cnt+1),reduce (y,red,cnt+1))
        | Abs(x, y) -> reduce ((substitute y x right), red, cnt + 1)
    | _ -> term
```

The key to this function was to substitute the abstraction in an application recursively. When the iteration ends in an infinite reduction loop, it would return the original λ-term.

Our implementation of **eval** function was as follows:

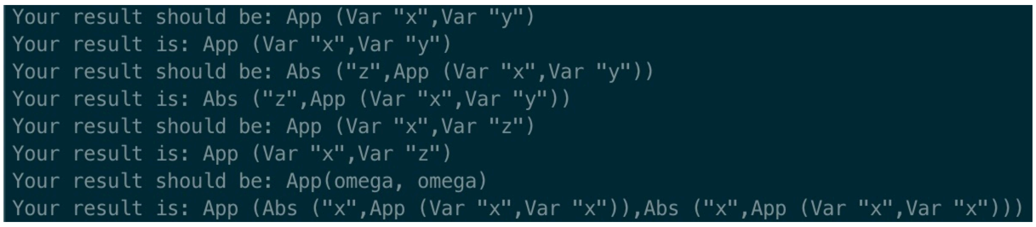
```
let rec eval (term: Term) =
    match term with
    | Var x -> Var x
    | Abs(x, y) -> Abs(x, y)
    | App(t1, t2) ->
        match eval t1 with
        | Var _ -> t1
        | App(t3, t4) -> App(t3, t4)
        | Abs(v, t) -> reduce (term, false, 0)
```

The key to this function was to call the **reduce** to evaluate the expression. It would call it with argument *term, false, 0*, which marked the beginning of the reduction.

Evaluation

The testing environment was macOS Mojave 10.14.5 system with iTerm and Microsoft (R) F# Compiler version 4.1.

We complied and tested the program, part of results were showed in Figure 1 (infinite situation was not shown there):



```
Your result should be: App (Var "x",Var "y")
Your result is: App (Var "x",Var "y")
Your result should be: Abs ("z",App (Var "x",Var "y"))
Your result is: Abs ("z",App (Var "x",Var "y"))
Your result should be: App (Var "x",Var "z")
Your result is: App (Var "x",Var "z")
Your result should be: App(omega, omega)
Your result is: App (Abs ("x",App (Var "x",Var "x")),Abs ("x",App (Var "x",Var "x")))
```

Figure 1: Testing of program

Discussion & Conclusion

As shown in Figure 1, for the first three results, if we did it manually: the first input was xy, the output was itself, i.e., $App(Var"x", Var"y")$. The second input was $\lambda z.xy$, the output was itself too, i.e., $Abs("z", App(Var"x", Var"y"))$. For the third test, it substituted the y to z, i.e., $xy[y := z]- > xz$. We took Ω as the last test input, since there was a infinite substitute loop, we returned itself as result. Our program worked well and all of our results were correct.

In this assignment, we implemented an interpreter for terms of untyped λ calculus. We spent a lot of time to figure out the logic of recursion and the structure of our "tree". It was challenging and rewarding.

References

[1] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.