CS401 Spring 2020 Final

Curt Wilson, BlazerID: curtisrw

23 April 2020

Honor Code: I, Curt Wilson, certify that I have done my own work on this exam. I have completed it alone, with the aid of the following sources:

Problem I

- https://uab.app.box.com/s/pphmmttbob4469ypghcsrzpv6soqgbaf
- https://thomas.gilray.org/classes/spring2020/cs401/slides/lambda-calc.pdf
- https://medium.com/@ayanonagon/the-y-combinator-no-not-that-one-7268d8d9c46

Problem II

- https://thomas.gilray.org/classes/spring2020/cs401/slides/lambda-calc.pdf
- https://www.seas.harvard.edu/courses/cs152/2010sp/lectures/lec10.pdf
- http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html - beta

Problem III

- https://thomas.gilray.org/classes/spring2020/cs401/slides/lambda-calc.pdf
- https://www.irif.fr/~mellies/mpri/mpri-ens/biblio/Selinger-Lambda-Calculus-Notes.pdf
- [I] https://chorasimilarity.wordpress.com/2012/12/21/conversion-of-lambda-calculus-terms-into-graphs/
- [II] https://www.irif.fr/~mellies/mpri/mpri-ens/biblio/Selinger-Lambda-Calculus-Notes.pdf (Page 52)

Problem I

Before deriving the Y-Combinator, I would like to clearly explain what a combinator is, what the Y-Combinator is, and how it works.
A **combinator** is simply a lambda expression that has no free variables, where a free variable is one that is not bound, or located inside of the body of a lambda expression. Moving on, the **Y-Combinator** is a generally known as a function in programming that enables recursion with one key exception: it cannot implement explicit recursion (it cannot refer to itself). In order to derive this combinator, we must know its definition:

```
(Y f) == (f (Y f))
```

This says, let us take **f** to be our fixpoint in a function, where **(Y f)** is similar to **f(x) == (x (f(x)),** a mathematical function.

In order to apply this to combinator definition, we will utilize lambda to make it into an application of lambda as follows:

```
Y = (λ (f) (f (Y f)))
```

At this point, the above definition refers to itself which will result in explicit recursion. There is one other underlying problem, which is that the order of evaluation is incorrect. To correct the explicit recursion, we implement the **U-Combinator,** as shown below:

```
(U U) == ((λ (u) (u u)) (λ (u) (u u)))
```

Now, apply this to the above **Y** function to imitate recursion non-explicitly through self-application. Implementing this step will eliminate the direct recursion issue with the initial **Y** function. Applying the U-Combinator on a **Y** helper function will help us attain our result. This will look like the following:

```
Y = (U (λ (mk-function) (λ (f) (f ((U mk-function) f)))))
```

At this point, we have a Y-Combinator that simulates recursion indirectly.
There is one last issue with our new implementation: in Call-By-Value (CBV), an infinite loop occurs because the Y-Combinator is rebuilt continuously through the following portion of the function:

((U mk-function) f)

To mitigate this issue, we can utilize what we learned about the lambda calculus η-expansion. To expand, adding a **(λ (x) ...) x)** function that takes in the portion from directly above and applies it on **x** should help. Expanding will result in the following:

Y = (U (λ (mk-function) (λ (f) (f (λ (x) (((U mk-function) f) x))))))

The above step is important because we need to have the ability to manage the order of evaluation. η-expansion also allows for the fix-point function to be created at each iteration throughout the duration of the function until achieving the base case, to which it would be returned. Now that the Y-Combinator is understood, it can be used to define a function. Below is a summation of **n** function that was written in Racket using the Y-Combinator:

```
(let* ([U (lambda (u) (u u))]
      [Y (U (lambda (y) (lambda (f) (f (lambda (x) (((U y) f) x))))))]
      [summate-n (Y (lambda (summate-n) (lambda (n) (if (= n 0) 0 (+ n (summate-n (- n
1)))))))])
  (pretty-print (summate-n  20))) ; 210
```

Explanation

In the first line of code, **let*** is chosen because it allows for nested let forms, which allows us to give meaning to each of the steps that have been discussed in deriving the Y-Combinator. The U-Combinator is the first case, which will mitigate the need for explicit recursion. Moving to the second line, the second case is made: the Y-Combinator that implements the U-Combinator with η-expansion. For the last case, the **summate-n** function is implemented, where the Y-Combinator takes in **summate-n**.

To go through one iteration of this combinator, the first lambda of the **summate-n** function is a **lambda (summate-n)** function that says that it has a **lambda (n)** function to be applied on the Y-Combinator. The second lambda, **lambda (n)**, creates the actual definition of the summation of **n** function. After checking **n** against the cases in this function, a fixpoint for that iteration is created.

Problem II

After working through examples, there are many terms that prove to textually grow forever through β-reduction. The term "reduction" in this case is a bit misleading, as β-reductions can grow, reduce, or reproduce the same lambda expression. The chosen expression is as shown below:

((λ (x) ((x x) x)) (λ (x) ((x x) x)))

Using Call-By-Value (CBN), we can try to reduce this expression, but will find that it grows at each reduction step:

Ɛ = ((λ (x) ((x x) x)) [])

r = (λ (x0) ((x0 x0) x0))

→β (((λ (x0) ((x0 x0) x0)) (λ (x0) ((x0 x0) x0))) (λ (x0) ((x0 x0) x0)))

$$\downarrow$$

Ɛ = (((λ (x0) ((x0 x0) x0)) []) (λ (x0) ((x0 x0) x0)))

r = (λ (x1) ((x1 x1) x1))

→β ((((λ (x1) ((x1 x1) x1)) (λ (x1) ((x1 x1) x1))) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

$$\downarrow$$

Ɛ = ((((λ (x1) ((x1 x1) x1)) []) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

r = (λ (x2) ((x2 x2) x2))

→β (((((λ (x2) ((x2 x2) x2)) (λ (x2) ((x2 x2) x2))) (λ (x2) ((x2 x2) x2))) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

$$\downarrow$$

Ɛ = (((((λ (x2) ((x2 x2) x2)) []) (λ (x2) ((x2 x2) x2))) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

r = (λ (x3) ((x3 x3) x3))

→β ((((((λ (x3) ((x3 x3) x3)) (λ (x3) ((x3 x3) x3))) (λ (x3) ((x3 x3) x3))) (λ (x2) ((x2 x2) x2))) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

$$\downarrow$$

Ɛ = ((((((λ (x3) ((x3 x3) x3)) []) (λ (x3) ((x3 x3) x3))) (λ (x2) ((x2 x2) x2))) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

r = (λ (x4) ((x4 x4) x4))

→β (((((((λ (x4) ((x4 x4) x4)) (λ (x4) ((x4 x4) x4))) (λ (x4) ((x4 x4) x4))) (λ (x3) ((x3 x3) x3))) (λ (x2) ((x2 x2) x2))) (λ (x1) ((x1 x1) x1))) (λ (x0) ((x0 x0) x0)))

r ::= (v e)

Ɛ ::= (Ɛ v) | [] "hole"

$$\frac{r \to_\beta e}{\mathcal{E}[e] \to \mathcal{E}[e']}$$

((λ (x) E0) E1) →β E0 [x ← E1]

Explanation

I saw this example on the last slide of our lambda calculus slides as a term similar to Omega and wanted to work with it to better understand how it nonterminates. In doing so, I discovered that it grows exponentially through β-reduction. Through CBN (lazy), you are reducing by the lambda term at the leftmost-outermost position, which in this case grows the reduction overall by one more term.

Problem III

Understanding the differences between plain untyped and simply typed lambda calculus: In plain untyped lambda calculus, the domain or codomain of functions is not specified, nor is the type. This gives much leeway for function definition and application. It is easy to run into cases where applying functions to specific arguments proves to be problematic. In simply typed lambda calculus, things are quite the opposite; there are similarities to set theory, where the type of each expression is *always* completely specified. The application of a function to an argument is *not* allowed unless the type matched the function's domain. Let us take a look at a simply typed lambda calculus:

$$A, B ::= \iota \mid A \to B \mid A \times B \mid 1$$

Now, let us look at a plain untyped lambda calculus:

$$M, N ::= (\lambda\ (x)\ M) \mid (M\ N) \mid x$$

Not all terms are typable in untyped lambda calculus, and this can be shown through the nonterminating lambda term that is known as Omega (Ω). This term is defined below:

$$((\lambda\ (u)\ (u\ u))\ (\lambda\ (u)\ (u\ u)))$$

β-reducing the Omega term results in the following:

$$\begin{tikzcd}
((\lambda\ (u)\ (u\ u))\ (\lambda\ (u)\ (u\ u))) \arrow[d] \\
\to_\beta\ ((\lambda\ (u)\ (u\ u))\ (\lambda\ (u)\ (u\ u))) \arrow[d] \\
((\lambda\ (u)\ (u\ u))\ (\lambda\ (u)\ (u\ u))) \arrow[d] \\
{}
\end{tikzcd}$$

$\rightarrow_\beta$ ((λ (u) (u u)) (λ (u) (u u)))

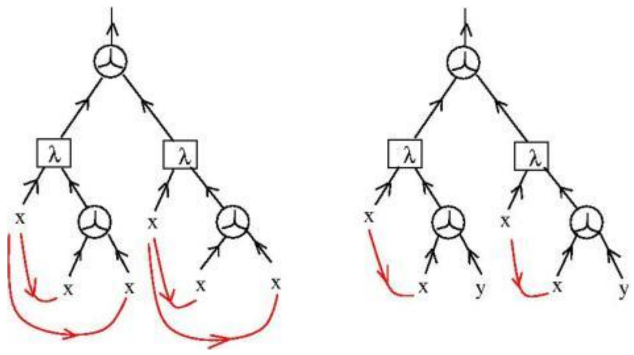
.

.

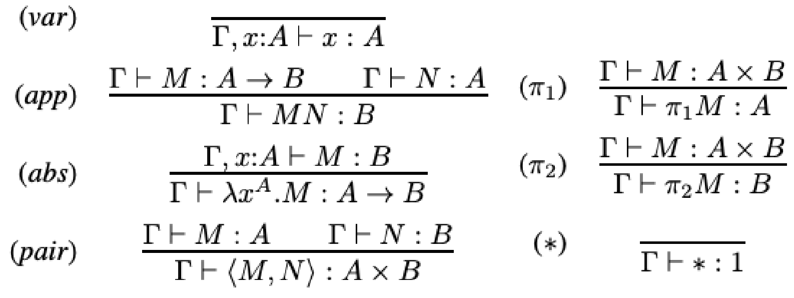
.

Further Explanation

In the redex (λ (u) (u u)), **(u u)** cannot be well-typed in the simply typed lambda calculus, as the first **u** would need to be assigned a specific type, and the other **u** would need to be assigned a different type. This is illegal because **u** cannot have two separate types. This situation will be true for expressions that have the same parameters that are applied in function or argument position. It can be done in the untyped (loose), but not in the typed (strict). The term may not be simply typed, but it is confluent based on the Church-Rosser theorem. Below is a diagram that gives a visual for how Omega works. Following that diagram there is a table of different typing rules for simply typed lambda calculus that should aid in understanding what simply typed means.



[I] Graphical Syntax Tree of Omega (Ω)



$$(var) \quad \frac{}{\Gamma, x{:}A \vdash x : A}$$

$$(app) \quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad (\pi_1) \quad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A}$$

$$(abs) \quad \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x^A . M : A \to B} \qquad (\pi_2) \quad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B}$$

$$(pair) \quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad (*) \quad \frac{}{\Gamma \vdash * : 1}$$

Table 4: Typing rules for the simply-typed lambda calculus

[II] Typing Rules