Lambda Calculus Notes

Mac Radigan

Y-Combinator

Description of the Y Combinator based on Mike Mvanier's blog post [1]. see http://mvanier.livejournal.com/2897.html

Cannonical Expression

Curry's Y Combinator [2] is defined as:

$$\mathbf{Y} = \lambda f.\,(\lambda x.f\,(xx))\,(\lambda x.f\,(xx)) \tag{1}$$

When applied to a function g, the expansion follows [2]

$$\begin{aligned}\mathbf{Y}g &= (\lambda f.\,(\lambda x.f\,(xx))\,(\lambda x.f\,(xx)))\,g \\ &= (\lambda x.g\,(xx))\,(\lambda x.g\,(xx)) \\ &= g\,((\lambda x.g\,(xx))\,(\lambda x.g\,(xx))) \\ &= g\,(\mathbf{Y}g)\end{aligned} \tag{2}$$

Connonical Form in Scheme

Direct implementation of the above expression for the Y Combinator will not terminate during applicative order [1].

Strict Scheme (Chicken)

Chicken scheme is a strict scheme, and evaluates in applicative order.

```
(define Y
  (lambda (f)
    (f (Y f))))

(define almost-factorial
  (lambda (f)
```

```
        (lambda (n)
          (if (= n 0)
              1
              (* n (f (- n 1)))))))

    (define factorial (Y almost-factorial))

    (display (factorial 6)) ; infinite loop
```

Using Lazy Evaluation (Racket #lang lazy)

This will work in a lazy language, as shown using the lazy extension in Racket.

```
#lang lazy

;;; Eliminating (most) explicit recursion (lazy version)

(define Y
  (lambda (f)
    (f (Y f))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(println (factorial 6)) ; 720
```

Normal Order Y Combinator

The Normal Order Y Combinator will not terminate during applicative order [1].

Strict Scheme (Chicken)

```
;;; The lazy (normal-order) Y combinator

(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
```

```
(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(prn (factorial 6)) ; infinite loop
```

Using Strict Evaluation (Racket)

```
#lang lazy

;;; The lazy (normal-order) Y combinator

(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(println (factorial 6)) ; 720
```

Using Lazy Evaluation (Racket #lang lazy)

However, it will work under lazy evaluation.

```
#lang lazy

;;; The lazy (normal-order) Y combinator

(define Y
  (lambda (f)
```

```
      ((lambda (x) (f (x x)))
       (lambda (x) (f (x x))))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(println (factorial 6)) ; 720
```

Applicative-Order) Y Combinator

The Strict (Applicative-Order) Y Combinator can be used with both applicative order and lazy evaluation [1].

Strict Scheme (Chicken)

```
;;; The strict (applicative-order) Y combinator

(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (lambda (y) ((x x) y)))))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define (part-factorial self)
   (let ((f (lambda (y) ((self self) y))))
     (lambda (n)
       (if (= n 0)
         1
         (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(display (factorial 6)) ; 720
```

Using Strict Evaluation (Racket)

```racket
#lang racket

;;; The strict (applicative-order) Y combinator

(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (lambda (y) ((x x) y)))))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))

(define (part-factorial self)
   (let ((f (lambda (y) ((self self) y))))
     (lambda (n)
       (if (= n 0)
         1
         (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(println (factorial 6)) ; 720
```

Using Lazy Evaluation (Racket #lang lazy)

```racket
#lang lazy

;;; The strict (applicative-order) Y combinator

(define Y
  (lambda (f)
    ((lambda (x) (x x))
     (lambda (x) (f (lambda (y) ((x x) y)))))))

(define almost-factorial
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
```

```
          (* n (f (- n 1)))))))

(define (part-factorial self)
   (let ((f (lambda (y) ((self self) y))))
     (lambda (n)
       (if (= n 0)
         1
         (* n (f (- n 1)))))))

(define factorial (Y almost-factorial))

(println (factorial 6)) ; 720
```

References

[1] M. Mvanier, "The y combinator (slight return) or how to succeed at recursion without really recursing," http://mvanier.livejournal.com/2897.html, 2010.

[2] Wikipedia, "Fixed-point combinator — Wikipedia, the free encyclopedia," 2011, [Online; accessed 11-July-2016]. [Online]. Available: http://en.wikipedia.org/Fixed-point_combinator