Login

Assignment 12: Introduction to Monads

Our biggest mistake: using the scary term “monad” rather than “warm fuzzy thing”.

Assignment

There are an abundance of lecture notes you could use for this assignment.

- Cameron's lecture notes are probably those which hew closest to the homework.
- Dan and Adam's View of Monads and its earlier iteration …
- … Dan's Schemer's View may also be of some use.
- Cameron, of Cameron's notes, also suggests this explanation from Phil Wadler [http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf]
- Dave Herman wrote up this Schemer's Monads Introduction [http://www.ccs.neu.edu/home/dherman/research/tutorials/monads-for-schemers.txt]

Requirements

In order to receive credit:

- You should use the monads.rkt file. You should add it to your C311 directory. This contains all the monad definitions you should need.
- You should also pull down and use the monads-student-tests.rkt test file.
- Use monadic style.
- Do not use `set!` or any another Racket procedures that perform side effects.

Maybe Monad

Recall the definition of the `maybe` monad presented in lecture.

1. The Racket function `findf` takes a list and a predicate against which to test the list's elements. In our implementation, we will return either the leftmost list element to pass the predicate or `(Nothing)` if none of the list's elements pass.

```
> (findf-maybe symbol? '(1 2 c))
(Just 'c)
> (findf-maybe boolean? '(#f 1 2 c))
(Just #f)
> (findf-maybe number? '(a b c))
(Nothing)
```

Writer Monad

The `writer` monad provides a mechanism to write data separately from the actual return value. If we use a list to represent these writes, we can use this monad to implement some rather useful functions.

2. The function `partition` takes a list and a predicate, returning a dotted pair with the values that do not pass the predicate in the writer part of the monad and the values that do in the value part. Implement this using the `writer` monad.

```
> (run-writer (partition-writer even? '(1 2 3 4 5 6 7 8 9 10)))
((1 3 5 7 9) . (2 4 6 8 10))

> (run-writer (partition-writer odd? '(1 2 3 4 5 6 7 8 9 10)))
((2 4 6 8 10) . (1 3 5 7 9))
```

3. Exponentiation by squaring is a method for quickly raising numbers to integer powers. Here is the definition of `power`, a function that raises a base `x` to a power `n` using this algorithm:

```
(define power
  (lambda (x n)
    (cond
      [(zero? n) 1]
      [(zero? (sub1 n)) x]
      [(odd? n) (* x (power x (sub1 n)))]
      [(even? n) (let ((nhalf (/ n 2)))
                   (let ((y (power x nhalf)))
                     (* y y)))])))
```

Using the writer monad, implement the `powerXpartials` procedure, which also takes a base and an exponent. It should return the answer as a natural value, along with each partial result computed along the way.

```
> (run-writer (powerXpartials 2 6))
((2 4 8) . 64)

> (run-writer (powerXpartials 3 5))
((3 9 81) . 243)

> (run-writer (powerXpartials 5 7))
((5 25 125 15625) . 78125)
```

State Monad

Recall from lecture that the `state` monad uses a state and works with `inj-state` and `bind-state`.

4. Given a symbol `x` and a binary tree of symbols `tr` (i.e. a tree with symbols at the leaves), via a preorder walk replace every occurrence of `x` with the number of `x`s that have been seen so far.

```
> ((run-state (replace-with-count 'o '(a o (t o (e o t ((n . m) . o) . f) . t) . r))) 0)
(4 . (a 0 (t 1 (e 2 t ((n . m) . 3) . f) . t) . r))

> ((run-state (replace-with-count 'o '(((h (i s . o) . a) o s o e . n) . m))) 0)
(3 . ((h (i s . 0) . a) 1 s 2 e . n) . m))

> ((run-state (replace-with-count 'o '(o (h (o s . o) . o) . o))) 1)
(6 . (1 (h (2 s . 3) . 4) . 5))
```

Mixed Monads Problems

One of the neat things about monadic code is that it can reveal the underlying structure in the code that *uses* them. This enables you to parameterize your code over the monad. You can then drop in a different monad and monadic operation, and get different behavior as a result. We'll do that here.

You'll use the following `traverse` in the next three problems.

```
> (define traverse
    (lambda (inj bind f)
      (letrec
        ((trav
           (lambda (tree)
             (cond
               [(pair? tree)
                (go-on ([a (trav (car tree))]
                        [d (trav (cdr tree))])
                  (inj (cons a d)))]
               [else (f tree)]))))
        trav)))
```

5. The reciprocal of a number `n` is computed by `(/ 1 n)`. Note that `0` has no reciprocal. Implement `reciprocal` using the `maybe` monad, returning any value computed and `(Nothing)` when `0` is provided.

```
> (reciprocal 0)
(Nothing)

> (reciprocal 2)
(Just 1/2)
```

Using this, we can return a tree of reciprocals, and instead signal failure if the tree contains a `0`.

```
> (define traverse-reciprocal
    (traverse Just bind-maybe reciprocal))
```

```
> (traverse-reciprocal '((1 . 2) . (3 . (4 . 5))))
(Just ((1 . 1/2) . (1/3 . (1/4 . 1/5))))

> (traverse-reciprocal '((1 . 2) . (0 . (4 . 5))))
(Nothing)
```

6. Halve. Implement the function `halve` that, given a number, either will return in the monad half the number, or, if the number is not divisible by two, will instead leave the original number in place, and also log that number (using the writer monad).

```
> (run-writer (halve 6))
(() . 3)

> (run-writer (halve 5))
((5) . 5)
```

Using this, we can return a tree in which the even numbers have been halved, the odds remain in place, and in which we've logged the odd numbers (which are not cleanly divisible by 2).

```
> (define traverse-halve
    (traverse inj-writer bind-writer halve))

> (run-writer (traverse-halve '((1 . 2) . (3 . (4 . 5)))))
((1 3 5) . ((1 . 1) . (3 . (2 . 5))))
```

7. State/sum. Implement a function `state/sum` which will, when given a number, return the current state as the value, and add that number to the current state. Below, we give state/sum the number to sum the state with, then we call the result of `run-state` with the initial state. So in the first case, the state is 0 and the sum is 5.

```
> ((run-state (state/sum 5)) 0)
(5 . 0)

> ((run-state (state/sum 2)) 0)
(2 . 0)

> ((run-state (state/sum 2)) 3)
(5 . 3)
```

Using this, we can return a tree consisting of partial sums of the elements, and in which the state contains the final sum of the tree.

```
> (define traverse-state/sum
    (traverse inj-state bind-state state/sum))

> ((run-state (traverse-state/sum '((1 . 2) . (3 . (4 . 5))))) 0)
(15 . ((0 . 1) 3 6 . 10))
```

Brainteaser: Continuation monad

Take a look in the monads.rkt file for the definition of the continuation monad.

For more examples using the `Cont` monad, see pp. 16-18 of A Schemer's View of Monads.

CPS Monad Interpreter

The following interpreter is a direct style interpreter resembling what Dan wrote in class:

```
(define value-of
  (lambda (expr env)
    (match expr
      [(? number?) expr]
      [(? boolean?) expr]
      [(? symbol?) (apply-env env expr)]
      [`(* ,x1 ,x2) (* (value-of x1 env) (value-of x2 env))]
      [`(sub1 ,x) (sub1 (value-of x env))]
      [`(zero? ,x) (zero? (value-of x env))]
      [`(if ,test ,conseq ,alt) (if (value-of test env)
                                     (value-of conseq env)
                                     (value-of alt env))]
      [`(capture ,k-id ,body) (callcc (lambda (k)
                                           (value-of body (extend-env k-id k env))))]
      [`(return ,k-exp ,v-exp) ((value-of k-exp env) (value-of v-exp env))]
      [`(lambda (,id) ,body) (closure id body env)]
      [`(,rator ,rand) (apply-proc (value-of rator env) (value-of rand env))])))
```

Use the `Cont` monad to create a monadic `value-of`, and call it `value-of-cps`. Provide your own `empty-env`, `apply-env`, `extend-env`, `closure`, and `apply-proc` in representations of your choice. Most of the same helpers should work for both `value-of` and `value-of-cps`. Here are some tests your interpreter should pass:

```
> (define fact-5
    '((lambda (f)
        ((f f) 5))
      (lambda (f)
        (lambda (n)
          (if (zero? n)
            1
            (* n ((f f) (sub1 n))))))))


> ((run-cont (value-of-cps fact-5 (empty-env))) (lambda (v) v))
120

> (define capture-fun
    '(* 3 (capture q (* 2 (return q 4)))))

> ((run-cont (value-of-cps capture-fun (empty-env))) (lambda (v) v))
12
```

Just Dessert

Here is the Reverse State monad [https://lukepalmer.wordpress.com/2008/08/10/mindfuck-the-reverse-state-monad/]. Do something interesting with it. You might have to use Lazy Racket [http://docs.racket-lang.org/lazy/index.html?q=lazy]. Since it might be in a different language, you should leave your answer commented out at the bottom of your file. Of course, if you want to go forward **and** backward through time, you'd use a Tardis [https://hackage.haskell.org/package/tardis-0.3.0.0/docs/Control-Monad-Tardis.html].

monads.txt · Last modified: 2019/04/11 09:11 by mvc