Login

Assignment 12: Introduction to Monads

> A monad is just a monoid in the category of endofunctors. What's the problem?

("A Brief, Incomplete, and Mostly Wrong History of Programming Languages" [http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html])

> "Of course!" Joe thinks. "It's all so simple now. The key to understanding monads is that they are Like Burritos."

(Abstraction, intuition, and the "monad tutorial fallacy" [https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/])

> Our biggest mistake: using the scary term "monad" rather than "warm fuzzy thing".

Assignment

Please complete the problems below. When you are finished, place all of your code in a file named `a12.rkt` and submit it to Oncourse.

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]

Note: In Cameron's Notes and in monads.rkt, Return is equivalent to Unit from class.

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 a12-student-tests.rkt test file.
- Use monadic style.
- Do not use `set!` or any another Racket procedures that perform side effects.
- Unless otherwise specified, you may use `do` to write any program below.

Maybe Monad

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

1. The function `assv` takes an association list and a value to look up. In our implementation, we will either return a value if one is found or `(Nothing)` if no match is found in the list.

```
> (assv-maybe 'c '((a . 1) (b . 2) (c . 3)))
(Just 3)
> (assv-maybe 'd '((a . 1) (b . 2) (c . 3)))
(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 first position and the values that do in the second position. Implement this using the `writer` monad. *Do not use `do` syntax for this problem.*

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

> (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]
      [(= n 1) 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.

```
> (powerXpartials 2 6)
(64 . (2 4 8))

> (powerXpartials 3 5)
(243 . (3 9 81))

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

State Monad

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

4. The `abc` game is a game in which a list is read and a score is computed. For each `a` seen, the score is `score + 1`. For each `b` seen, the score is `score - 1`. For each `c` seen, the score is unchanged. Implement `abc-game`, using the `state` of the state monad to keep score.

```
> ((abc-game '(a b c c b a)) 0)
(__ . 0)

> ((abc-game '(a b c c b a a)) 0)
(__ . 1)

> ((abc-game '(a a a)) 0)
(__ . 3)
```

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 (return bind f)
      (letrec
        ((trav
           (lambda (tree)
             (cond
               [(pair? tree)
                (do bind
                  (a <- (trav (car tree)))
                  (d <- (trav (cdr tree)))
                  (return (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 return-maybe 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).

```
> (halve 6)
(3 . ())

> (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 return-writer bind-writer halve))

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

```
(((1 . 1) . (3 . (2 . 5))) . (1 3 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.

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

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

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

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 return-state bind-state state/sum))

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

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 the direct style interpreter from Assignment 7: Continuations and Representation Independence:

```
(define value-of
  (lambda (expr env)
    (pmatch expr
      [`,n (guard (or (number? n) (boolean? n))) n]
      [`,x (guard (symbol? x)) (apply-env env x)]
      [`(* ,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) (call/cc (lambda (k)
                                         (value-of body (extend-env k-id k env))))]
      [`(return ,v-exp ,k-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))))))))
```

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

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

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

Just Dessert: The Fringe

Like baking soda, the `traverse` function above has many applications. Here's one more. We're going to use the `Cont` monad to achieve a kind of trampolining or multitasking.

The *fringe* of a tree is the list of its leaves in left-to-right order. For example, the fringe of

```
(("Time" . "flies") . ("like" . ("an" . "arrow")))
```

is the list of strings

```
("Time" "flies" "like" "an" "arrow")
```

Given two trees, the `driver` function compares their fringes while ignoring any differences in capitalization. If the two fringes are the same, then `driver` returns the two trees (in a list) with their leaves swapped. If the two fringes have bigger differences than mere capitalization, then `driver` returns `#f`.

For example, here are two tests:

```
> (driver '(("Time" . "flies") . ("like" . ("an" . "arrow")))
          '("time" . ("FLIES" . (("like" . "an") . "aRrOw"))))
((("time" . "FLIES") . ("like" . ("an" . "aRrOw")))
 ("Time" . ("flies" . (("like" . "an") . "arrow"))))

> (driver '(("Time" . "flies") . ("like" . ("arrow" . "an")))
          '("time" . ("FLIES" . (("like" . "an") . "aRrOw"))))
#f
```

Below is most of the definition of `driver`. **Your mission is to complete the definition by defining `yield-cont`.**

```
> (define exchange
    (lambda ($1 $2)
      (pmatch `(,$1 ,$2)
        [`((stopped ,tree1) (stopped ,tree2))
         `(,tree1 ,tree2)]
        [`((yielding ,x1 ,k1) (yielding ,x2 ,k2))
         (guard (string-ci=? x1 x2))
         (exchange (k1 x2) (k2 x1))]
        [else #f])))

> (define tree->stream
    (lambda (tree)
      (((traverse return-cont bind-cont yield-cont)
        tree)
       (lambda (tree) `(stopped ,tree)))))

> (define driver
    (lambda (tree1 tree2)
      (exchange (tree->stream tree1) (tree->stream tree2))))
```

monads.txt · Last modified: 2014/11/20 18:23 by joshcox