Lista zadań nr 14

"Monad is a monoid in the category of endofunctors."

Monada obliczeniowa dana jest przez polimorficzny typ `(M 'a)` oraz dwie operacje

```
returnM : ('a -> (M 'a))
bindM   : (((M 'a) * ('a -> (M 'b))) -> (M 'b))
```

spełniające następujące równoważności

```
(bindM (returnM a) f) ≡ (f a)
    (bindM c returnM) ≡ c
(bindM (bindM c f) g) ≡ (bindM c (λ (a) (bindM (f a) g)))
```

W pliku `monadic-eval.rkt` znajduje się generyczny evaluator napisany w stylu monadycznym, sparametryzowany monadą `M`, która dla kompletności kodu została ustalona jako *monada identycznościowa*, gdzie

```
(define-type-alias (M 'a) 'a)

(define (returnM [v : 'a]) : (M 'a) v)
(define (bindM [c : (M 'a)] [f : ('a -> (M 'b))]) : (M 'b) (f c))
```

Monada identycznościowa służy do opisu obliczeń, bez efektów obliczeniowych, w związku z czym błędy ewaluacji obsługujemy przy pomocy procedury `error` z Plaita (zobacz definicję funkcji `errorM`). Procedurą udostępniającą użytkownikowi wynik obliczeń w postaci wartości typu `String` jest `showM`.

Zadanie 1. (2 pkt)

Przeanalizuj kod z pliku `monadic-eval.rkt`, a następnie:

1. Przeprowadź *inlining* definicji operacji monadycznych w ewaluatorze, czyli zastąp identyfikatory `returnM`, `bindM`, `errorM` i `showM` ich definicjami, upraszczając kod gdzie to możliwe i sprawdź jaki ewluator otrzymasz.

2. Zapisz ewaluator z użyciem notacji `do` zdefiniowanej jako makro (wykład 11, plik `error-ans-monad-macros.rkt`).

3. Udowodnij, że definicje `returnM` i `bindM` spełniają wymagane prawa równościowe.

W pozostałych zadaniach należy zmodyfikować definicje z pliku `monadic-eval.rkt` tak, by wyposażyć definiowany język w zadane efekty obliczeniowe. W każdym z przypadków należy upewnić się, że rzeczywiście mamy do czynienia z monadą, czyli, że spełnione są wymagane prawa.

Zadanie 2. (2 pkt)

Monada błędów/wyjątków (widzieliśmy ją już na wykładzie 11, np. plik `error-ans-monad.rkt`):

```
(define-type (M 'a)
  (valM [val : 'a])
  (errM (l : Symbol) (m : String)))
```

W tym ewaluatorze wszystkie błędy obliczeń powinny zostać odnotowane jako wartość typu `(M Value)`. Jeśli masz ochotę, możesz dodać do definiowanego języka konstrukcję zgłaszającą błąd.

Przykładowe dane testowe i oczekiwane wyniki:

```
(module+ test
  (test (run `{+ {* 2 3} {+ 5 8}})
        "value: 19")
  (test (run `{{lambda {x} {+ x 1}} 5})
        "value: 6")
  (test (run `{lambda {x} {+ x 1}})
        "value: #<procedure>")
  (test (run `{1 2})
        "error in apply: not a function")
  (test (run `x)
        "error in lookup-env: unbound variable")
  (test (run `{+ 1 {lambda {x} x}})
        "error in prim-op: not a number"))
```

Zadanie 3. (2 pkt)

Monada stanu (tę też widzieliśmy na wykładzie):

```
(define-type-alias State Number)
(define-type-alias (M 'a) (State -> ('a * State)))
```

W tym ewaluatorze chcemy wyposażyć język w pojedynczą modyfikowalą komórkę pamięci potrafiącą przechowywać wartości liczbowe. W komórce tej chcemy trzymać liczbę do tej pory wykonanych operacji prymitywnych i aplikacji funkcji. Trzeba zatem zmodyfikować ewaluator tak by każde wywołanie operacji prymitywnej lub funkcji zwiększało licznik – warto zdefiniować sobie w tym celu operację `tickM : (M Void)`, którą będzie można składać przy użyciu `bindM` z pozostałymi obliczeniami. Do samego języka dodajemy konstrukcję `count`, która udostępnia programiście wartość licznika. Początkową wartość licznika powinna ustalać funkcja `showM`.

Przykładowe dane testowe i oczekiwane wyniki:

```
(module+ test
  (test (run `2)
        "value: 2, state: 0")
  (test (run `{+ {* 2 3} {+ 5 8}})
        "value: 19, state: 3")
  (test (run `{{lambda {x} {+ x 1}} 5})
        "value: 6, state: 2")
  (test/exn (run `{1 2})
            "not a function")
  (test (run `{+ {+ count count}
                 {+ count count}})
        "value: 2, state: 3"))
```

Zadanie 4. (2 pkt)

Błędy i stan razem. Monadę błędów

```
(define-type (E 'a)
  (valE [val : 'a])
  (errE (l : Symbol) (m : String)))
```

można złożyć z monadą stanu na dwa sposoby. Albo tak, by wystąpienie błędu nie pozwalało zajrzeć do pamięci:

```
(define-type-alias (M 'a) (State -> (E ('a * State))))
```

i wówczas spodziewamy się takich testów

```
(module+ test
  (test (run `{+ {* 2 3} {+ 5 8}})
        "value: 19, state: 3")
  (test (run `{+ {+ 1 2} {lambda {x} x}})
        "error in prim-op: not a number")
  (test (run `{+ {+ count count}
                 {+ count count}})
        "value: 2, state: 3"))
```

Albo też zachowujemy stan pamięci z chwili wystąpienia błędu:

```
(define-type-alias (M 'a) (State -> ((E 'a) * State)))
```

i wówczas spodziewamy się takich testów

```
(module+ test
  (test (run `{+ {* 2 3} {+ 5 8}})
        "value: 19, state: 3")
  (test (run `{+ {+ 1 2} {lambda {x} x}})
        "error in prim-op: not a number, state: 1"))
```

Zrealizuj jeden z tych wariantów.

Zadanie 5. (2 pkt)

Monada wyjścia:

```
(define-type-alias Output (Listof Number))
(define-type-alias (M 'a) ('a * Output))
```

w której modelujemy operację `(write e)`, wypisującą wartość (wyłącznie liczbową) na wyjście reprezentowane jako lista (typ `Output`). Do języka warto dodać jeszcze sekwencjonowanie wyrażeń `(begin e1 e2)`, a do gramatyki wartości `voidV`.

Przykładowe dane testowe i oczekiwane wyniki (listy wartości wysłanych na wyjście wypisywane ze spacją jako separatorem – możesz to dowolnie zmienić):

```
(module+ test
  (test (run `{+ {* 2 3} {+ 5 8}})
        "value: 19, output: ")
  (test/exn (run `{1 2})
            "not a function")
  (test (run `{write {{lambda {x} {begin {write x} {+ x 1}}} 5}})
        "value: #<void>, output: 5 6 "))
```

Zadanie 6. (2 pkt)

Niedeterminizm:

```
(define-type-alias (M 'a) (Listof 'a))
```

Rozszerzamy język o operacje niedeterministycznego wyboru `(amb e1 e2)` oraz porażki `fail`. Na poziomie semantyki, operacja `amb` uruchamia oba swoje argumenty, a następnie zbiera wyniki zwrócone przez nie (konkatenuje listy). Operacja `fail` z kolei nie generuje żadnej wartości (co odpowiada liście pustej).

Przykładowe dane testowe i oczekiwane wyniki (listy wartości są wypisywane ze średnikiem jako separatorem – możesz to dowolnie zmienić):

```
(module+ test
  (test (run `{+ {* 2 3} {+ 5 8}})
        "19 ; ")
  (test (run `{{lambda {x} {+ x 1}} 5})
        "6 ; ")
  (test/exn (run `{1 2})
            "not a function")
  (test (run `{amb 1 2})
        "1 ; 2 ; ")
  (test (run `{amb 1 fail})
        "1 ; ")
  (test (run `{+ {amb 1 10} {amb 100 1000}})
        "101 ; 1001 ; 110 ; 1010 ; ")
  (test (run `{+ {amb 1 fail} {amb 100 1000}})
        "101 ; 1001 ; "))
```