Metody programowania 2017

Lista zadań nr 14

Na zajęcia 13 i 14 czerwca 2017

Niech

```
class Monad m ⇒ Failable m where
   failure :: m a
   handle :: m a → a → a

instance Failable Maybe where
   failure = Nothing
   (Just x) `handle` _ = x
   Nothing `handle` x = x

class Monad (m s) ⇒ Stateful m s where
   update :: (s → s) → m s s
   getState :: m s s
   getState = update id
   putState :: s → m s ()
   putState s = update (const s) >> return ()

newtype SC s a = SC { exec :: s → (a,s) }

instance Monad (SC s) where
   t >>= f = SC $ uncurry (exec . f) . exec t
   return = SC . (,)

newtype Id a = Id { unId :: a }

newtype Writer a =
   Writer { unWriter :: (a, String) }

class (Monad m, Monad (t m)) ⇒ MonadTrans t m
   where lift :: m a → t m a
```

Zadanie 1 (2 pkt). Zainstaluj typy `Id`, `Writer` i `(s→)` w klasie `Monad`. Zdefiniuj typy

```
IdT, WriterT, ReaderT, MaybeT, ListT,
   StateT s :: (* → *) → * → *
```

będące transformatorami monad odpowiadającymi monadom `Id`, `Writer`, `(s→)`, `Maybe`, `[]` i `StateComp s`. Dla dowolnej monady `m` zainstaluj typy

```
IdT m, WriterT m, ReaderT s m, MaybeT m, ListT m,
   StateT s m :: * → *
```

w klasie `Monad`. *Uwaga:* nie zawsze powyższe typy będą spełniać aksjomaty monad! Zainstaluj typy `IdT`, `WriterT`, `(s→)`, `MaybeT`, `ListT`, `StateT s` w klasie `MonadTrans`. Dla dowolnej monady `m` zainstaluj typ `MaybeT m` w klasie `Failable`, typ `StateT s m` w klasie `Stateful`, a typ `ListT m` w klasie `MonadPlus`.

Zadanie 2 (1 pkt). Jak działa typ `MaybeT Id`? W jakim sensie jest on izomorficzny z `Maybe`? Rozważ typy

```
MaybeT (StateComp s)
   StateT s Maybe
MaybeT (StateT s Id)
StateT s (MaybeT Id)
```

W jakim sensie są one izomorficzne?

Zadanie 3 (1 pkt). Oto parser

```
newtype Parser token m value =
   Parser ([token] → m ([token],value))
```

tj. monada stanowa, w której stanem obliczeń jest lista tokenów (typu `token`), a dostarczaniem wyników parsowania zajmuje się monada `m` (np. lista, jeśli chcemy mieć parser z nawracaniem lub `Maybe`, jeśli wolimy parser deterministyczny).

Zainstaluj typ `Parser token m` w klasie `MonadPlus` i zaprogramuj biblioteczkę następujących kombinatorów parsujących:

```
parse :: Monad m ⇒ Parser token m value
            → [token] → m value
isElem :: (Eq token, MonadPlus m) ⇒ [token]
            → Parser token m token
isEmpty :: MonadPlus m ⇒ Parser token m ()
many :: MonadPlus m ⇒ Parser token m value
          → Parser token m [value]
many1 :: MonadPlus m ⇒ Parser token m value
            → Parser token m [value]
option :: MonadPlus m ⇒ Parser token m value
            → Parser token m (Maybe value)
```

Zadanie 4 (1 pkt). Do strumienia tokenów dodajemy dodatkowy stan `st`:

```
newtype Parser token st m value
   = Parser (([token],st)
      → m ([token],st,value))
```

Rozwiąż poprzednie zadanie dla tej implementacji.

Zadanie 5 (1 pkt). Rozważmy wyrażenia złożone z identyfikatorów (ciągi małych i wielkich liter oraz cyfr zaczynające się literą), literałów całkowitoliczbowych (niepuste ciągi cyfr) oraz operatorów `+`, `-`, `*`, `/`, `^`. Ostatni operator (potęgowanie) wiąże najsilniej i w prawo, pozostałe operatory — w lewo, przy czym `+` i `-` słabiej, niż `*` i `/`. W wyrażeniach można ponadto używać nawiasów i konstrukcji x`=`e`:`e', która wiąże najsłabiej i oznacza związanie wartości wyrażenia e z identyfikatorem x w wyrażeniu e', np. wyrażenie

```
x=1+2*3:2*x^2
```

ma wartość 98.

Uczyń stanem obliczeń prasera z poprzedniego zadania słownik odwzorowujący nazwy identyfikatorów w liczby całkowite (tak by nie trzeba go było jawnie przekazywać do i z funkcji parsującej) i napisz kalkulator wyznaczający wartość opisanych wyżej wyrażeń.

Metody typu należącego do

```
class Monoid t where
   mzero :: t
   mplus :: t → t → t
```

powinny spełniać równości

```
mzero `mplus` x  =  x
x `mplus` mzero  =  x
(x `mplus` y) `mplus` z  =
   x `mplus` (y `mplus` z)
```

Metoda typu należącego do

```
class Functor m where
   fmap :: (a → b) → (m a → m b)
```

powinna spełniać równości

```
fmap id  =  id
fmap (f . g)  =  fmap f . fmap g
```

Metody typu należącego do

```
class Functor m ⇒ Applicative m where
   pure :: a → m a
   (<*>) :: m (a → b) → m a → m b
```

powinny spełniać równości

```
pure id <*> v  =  v
((pure (.) <*> u) <*> v) <*> w  =
   u <*> (v <*> w)
pure f <*> pure x  =  pure (f x)
u <*> pure y  =  pure ($ y) <*> u
```

Metody typu należącego do

```
class Applicative m ⇒ Monad m where
   (>>=) :: m a → (a → m b) → m b
   return :: a → m a
```

powinny spełniać równości

```
return a >>= f  =  f a
m >>= return  =  m
(m >>= (λ a → n)) >>= p  =
 m >>= (λ a → (n >>= p))
```

Pomiędzy metodami funktora, funktora aplikatywnego i monady powinny zachodzić ponadto zależności:

```
(1)   fmap f x  =  pure f <*> x
(2)   fmap f x  =  x >>= return . f
(3)   pure  =  return
(4)   (<*>)  =  ap
```

gdzie

```
m `ap` n = m >>= (λf → n >>= (λx → return (f x)))
```

Metody typu należącego do

```
class (Monoid m a, Monad m) ⇒ MonadPlus m
```

powinny ponadto spełniać równości

```
mzero >>= f  =  mzero
m >>= (λ a → mzero)  =  mzero
(m `mplus` n) >>= k  =
   (m >>= k) `mplus` (n >>= k)
```

Monada jest *przemienna*, jeżeli

```
m >>= (λ a → (n >>= (λ b → p)))  =
n >>= (λ b → (m >>= (λ a → p)))
```

tj. gdy

```
do { a ← m; b ← n; p }  =  do { b ← n; a ← m; p }
```

(To jest ważne pojęcie, gdyż `listT m` jest monadą wtedy i tylko wtedy, gdy `m` jest monadą przemienną).

Zadanie 6 (1 pkt). Udowodnij, że funkcja `fmap` zdefiniowana dla funktora aplikatywnego równością `(1)` spełnia prawa funktora. Udowodnij, że funkcja `fmap` zdefiniowana dla monady równością `(2)` spełnia prawa funktora. Udowodnij, że funkcje `pure` i `<*>` zdefiniowane dla monady równościami `(3)` i `(4)` spełniają prawa funktora aplikatywnego.

Zadanie 7 (1 pkt). Udowodnij, że `[]` i `Maybe` zainstalowane w klasie `MonadPlus` rzeczywiście są monadami addytywnymi, tj. spełniają odpowiednie prawa. Udowodnij, że typ

```
newtype StateComp s a =
   StateComp { exec :: s → (a,s) }
```

zainstalowany na wykładzie w klasie `Monad` rzeczywiście jest monadą, tj. spełnia odpowiednie prawa.

Zadanie 8 (1 pkt). Niech

```
newtype Id a = Id { unId :: a }
newtype Writer a =
   Writer { unWriter :: (a, String) }
```

Uczyń typy `Id`, `Writer` i `(s →)` monadami, gdzie `(s →)` oznacza `(→) s`, tj. polimorficzny typ funkcji o dziedzinie `s`, czyli `(s →) t` jest typem `s → t`. Uczyń te typy, które można, monadami addytywnymi. Udowodnij, że Twoje implementacje spełniają niezbędne prawa równościowe.

Zadanie 9 (1 pkt). Udowodnij, że `Id` oraz `Maybe` są monadami przemiennymi. Czy `[]`, `(s →)`, `StateComp s` i `Writer` są monadami przemiennymi?