References

"Effectful property abstraction"

Paradigmák

Imperatív

- Ciklus
- "Változók"
- Implicit mellékhatások
- Globális állapot

Funkcionális

- Rekurzió
- Kifejezések
- Explicit mellékhatások
- Hivatkozási helyfüggetlenség
- Kompozícionalitás
- Magasabbrendű függvények

Emlékeztető

```haskell
data Maybe a = Nothing | Just a
```

```haskell
data Rec = R
           { f1 :: String
           , f2 :: Int -> Int
           }
```

```haskell
f1 :: Rec -> String
f1 (R s _) = s

f2 :: Rec -> (Int -> Int)
f2 (R _ f) = f
```

Monádok

1105 What is a monad?

548 A monad is just a monoid in the category of endofunctors, what's the problem?

Egy példa ...

```
data Employee = Boss
                  { name :: String
                  , salary :: Int
                  , subordinates :: [Employee]
                  }
                | Subordinate
                  { name :: String
                  , salary :: Int
                  , isIntern :: Bool
                  , workDays :: [Day]
                  }

data Day = M | Tu | W | Th | F | Sa | Su
```

Állapot módosítás

Adat elérése mintaillesztéssel → Módosítás → "Visszapakolás"

```
promoteIntern1 :: Employee -> Employee
promoteIntern1 (Subordinate n s _ ds) = Subordinate n s False ds
promoteIntern1 x = x
```

Állapot módosítás

(Adat elérése mintaillesztéssel) → Módosítás → ("Visszapakolás")

```
promoteIntern2 :: Employee -> Employee
promoteIntern2 Subordinate{..} = Subordinate{isIntern = False, ..}
promoteIntern2 x = x
```

Állapot módosítás

Útvonal definiálása → Módosító fv. definiálása

```
promoteIntern3 :: Employee -> Employee
promoteIntern3 = isIntern .= False
```

Referenciák definiálása

Írási szemantika
Olvasási szemantika
Kezdő állapot
Módosított állapot
Elérendő érték
Módosított Érték

```
data Reference w r s t a b
  = Reference { refGet    :: forall x . (a -> r x) -> s -> r x
              , refSet    :: b -> s -> w t
              , refUpdate :: (a -> w b) -> s -> w t
              }
```

s állapotból elérem az a elemet úgy, hogy azt b-re változtatva, az állapot t-re változik

```haskell
just :: Reference w r (Maybe a) (Maybe b) a b

_2 :: Reference w r (u,v) (u,v') v v'

headElem :: Referece w r [a] [a] a a
```

Monadic Unit

```
data MU a = MU
```

Referencia típusok

```
type Getter r s t a b = Reference MU r s t a b
type Setter w s t a b = Reference w MU s t a b

type Lens s t a b
  = RefMonads w r => Reference w r s t a b

type Partial s t a b = (RefMonads w r, MonadPlus r, Morph Maybe r)
     => Reference w r s t a b

type Traversal s t a b = (RefMonads w r, MonadPlus r, Morph [] r)
     => Reference w r s t a b
```

Morph típusosztály

```haskell
class Morph m1 m2 where
  morph :: m1 a -> m2 a
```

```haskell
instance Morph Maybe [] where
  morph Nothing  = []
  morph (Just a) = [a]
```

```haskell
instance Morph Identity m where
  morph (Identity a) = return a
```

```haskell
instance Morph m MU where
  morph _ = MU
```

Operátorok

Tiszta

```
(^.) ::
  s -> Getter Identity s t a b -> a

(.=) :: Setter Identity s t a b ->
        b -> s -> t

(.-) :: Setter Identity s t a b ->
       (a -> b) -> s -> t
```

Monadikus

```
(^?) :: Monad m =>
  s -> Getter m s t a b -> m a

(!=) :: Setter m s t a b ->
        b -> s -> m t

(!~) :: Setter m s t a b ->
       (a -> m b) -> s -> m t
```

Referenciák kombinálása

```
(&) :: (Monad w, Monad r)
 =>
  Reference w r s t c d ->
  Reference w r c d a b ->
  Reference w r s t a b
```

- Nagyon hasonló a függvény kompozícióhoz.
- Asszociatív

```
(&+&) :: (RefMonads w r,
  MonadPlus r, Morph [] r) =>
  Reference w r s s a a ->
  Reference w r s s a a ->
  Reference w r s s a a
```

- Kommutatív, ha nem használják egymás eredményeit, és nem vesszük figyelembe a monadikus mellékhatásokat
- Figyeljünk egy adat esetleges többszöri elérésre

A monádok meghatározása

Mi lesz a következő kifejezések a típusa?

```
[1,2]  ^? headElem
Just 5 ^? just
```

Az olvasási szemantika a használat helyénél dől el.

A monád meghatározása típus-inferenciával, és típusszintű számítások segítségével történik.

A valódi probléma

Miért is szükséges a típusszintű számítás?

```
instance (Morph a b, Morph b c) => Morph a c where
  morph = (morph :: b x -> c x) . (morph :: a x -> b x)
```

Ehhez sajnos nem elég kifonomult a GHC contraint resolver-e.

A számítás menete

1. Mindenhol csupán megszorításokat teszünk a típusváltozóinkra (`w` és `r`)
2. A típus-inferencia a használat helyénél meghatározza a legáltalánosabb típust
3. A típusszintű Dijkstra algoritmus meghatározza a legrövidebb utat a minimálisan megkötött, és a célmonád között
4. Az út ellenőrzése (avagy megnézzük, hogy csakis a megfelelő típus konstruktorokat tartalmaz-e az út)
5. Típusosztályok segítségével mintákat illesztünk az útra, és elkészítjük az értékszintű reprezentációt

Lehetséges alkalmazások

- Részleges lockolása komplex adatszerkezetnek (implicit módon)
- Kényelmes fájlkezelés (pl.: a kontextus lehet az elérési útvonal, az adat lehet egy string)
- Egy adat különböző nézeteinek előállításai izomorfizmusokkal (pl.: listák kiegyensúlyozott fákként, és fordítva)

Köszönöm a figyelmet!