Programiranje z monadami

V prejšnji lekciji smo ugotovili, da lahko z monadami predstavimo računske učinke. Tokrat si bomo ogledali, kako lahko monade uporabimo kot programerji. Razne programerske tehnike lahko namreč izrazimo z monadami.

Še prej pa ponovimo, kaj je monada.

Definicija monade

Sestavni deli **monade** so:

1. *Operacija M iz tipov v tipe.* Za vsak tip τ si mislimo, da je $M\ \tau$ tip *izačunov*, ki (morda) uporabljajo računski učinek in vrnejo rezultat tipa τ.

2. *Funkcija* $\mathtt{return} : \tau \to M\ \tau$, ki pretvori vrednost tipa τ v izračun (ki ne uporablja nobenih učinkov).

3. *Funkcija* $\mathtt{bind} : M\ \rho \to (\rho \to M\ \tau) \to M\ \tau$, ki sestavi izračun tipa $M\ \rho$ in izračun tipa $M\ \tau$, pri čemer le ta pričakuje kot vhodni podatek vrednost tipa ρ.

Veljati morajo naslednje enačbe:

1. `return` je leva enota za `bind`: `bind (return v) q ≡ q v`
2. `return` je desna enota za `bind`: `bind p return ≡ p`
3. `bind` je asociativna opreacija: `bind p (λ v . bind (q v) r) ≡ bind (bind p q) r`

V Haskellovi notaciji:

1. `(return v) >>= f ≡ f v`
2. `p >>= return ≡ p`
3. `p >>= (λ x . q x >>= q) ≡ (p >>= q) >>= r`

Monade v Haskellu

Programski jezik Haskell je *čist*, s čimer želimo povedati, da je treba vse računske učinke narediti z monadami. (Tudi v ostalih programskih jezikih se v ozadju skriva monada računskih učinkov, a je ne opazimo, ker ni izražena s pomočjo tipov.)

Monade so v Haskellu **razred tipov**, zato najprej povejmo nekaj o tem konceptu.

Razredi tipov

Razredi tipov (angl. type classes) so mehanizem, s katerim lahko organiziramo večje programske enote. Namenjeni so podobnemu namenu kot razredi v objektnem programiranju in strukture v SML, a se od obojih razlikujejo.

Razred tipov `C` definiramo takole (poenostavljena različica):

```
class C a where
    v₁ :: τ₁(a)
    ...
    vᵢ :: τᵢ(a)
```

Simbol `a` označuje poljuben tip. Zgornja definicija je neke vrste vmesnik ali signatura, ki predpisuje, da je razred tipov `C` podan s tipom `σ` in vrednostmi `v₁`, ..., `vᵢ` tipov `τ₁(σ)`, ..., `τᵢ(σ)`.

Primerek ali **instanca** (angl. instance) razreda tipov `C` je definiran z

```
instance C σ where
    v₁ = ...
    ...
    vᵢ = ...
```

Najbolje bo, da si ogledamo primer.

Razred tipov `Num`

Vsak programski jezik ima **numerične tipe**. To so tisti tipi, katerih vrednosti lahko seštevamo, odštevamo, množicmo itd. V Haskellu je v ta namen definiran razred tipov Num.

Če v Haskellu definiramo funkcijo, ki uporablja aritmetične operacije, kakšen je njen tip?

```
λ> let f x y = 2 * x * x + 3 * y + 7
λ> :t f
f :: Num a => a -> a -> a
```

Haskell odgovori, da ima `f` tip `a -> a -> a`, kjer je `a` poljuben tip, ki mora biti iz razreda `Num`. To je oblika *ad-hoc polimorfizma*: `f` ima več tipov (polimorfizem), vendar se ne obnaša enakomerno za vse tipe, ampak je njeno obnašanje pri tipu `a` določeno s tem, kako so definirane aritmetične operacije na `a` (ad-hoc obnašanje).

Programer lahko dodaja svoje numerične tipe. Dodajmo kompleksna števila:

```
data Complex = C { re :: Float, im :: Float }
             deriving (Show, Eq)
```

Zgornja definicija pove, da kompleksno število `x + y · i` zapišemo `C x y`. Če je `z` kompleksno število, dobimo njegovi komponenti z `re z` in `im z`. Določilo `deriving` je navodilu Haskellu, naj za `Complex` sam generira še instance za razreda tipov `Show` in `Eq`.

Podatkovni tip `Complex` opremimo s strukturo numeričnega tipa:

```
instance Num Complex where
    z + w  = C (re z + re w) (im z + im w)
    z * w  = C (re z * re w - im w * im w) (re z * im w + im z * re w)
    abs z  = C (sqrt (re z * re z + im z * im z)) 0
    signum z = let r = sqrt (re z * re z + im z * im z) in C (re z / r)
    (im z / r)
    fromInteger k = C (fromInteger k) 0
    negate z = C (negate (re z)) (negate (im z))
```

Sedaj lahko računamo s kompleksnimi števili in jih uporabljamo povsod, kjer Haskell pričakuje numerični tip:

```
λ> C 1 2 + C 3 4
C {re = 4.0, im = 6.0}
λ> C 1 2 * C 3 4
C {re = -13.0, im = 10.0}
λ> C 0 1 * C 0 1
C {re = -1.0, im = 0.0}
λ> abs (C 1 1)
C {re = 1.4142135, im = 0.0}
λ> f (C 4 6) (C (-2) 3)
C {re = -156.0, im = -111.0}
```

Razreda tipov `Functor`, `Applicative`

Haskellova standard knjižnica zahteva, da mora vsak tip, ki je monada (primerek razreda tipov `Monad`) biti hkrati še `Applicative`. Vsak primerek `Applicative` pa mora biti `Functor`. Te zahteve izhajajo iz matematične definicije monade.

Oglejmo si, kaj sta Functor in Applicative.

Razred `Functor` zahteva, da opremimo tip s funkcijo `fmap`, ki se obnaša tako kot `map` na seznamih. Torej je tip primerek razreda `Functor`, če je neke vrste "zbirka" elementov, ki jo lahko preslikamo v drugo "zbirko". Dvojiška drevesa so `Functor`:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
            deriving Show

instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

Razred `Applicative` opisuje zbirke, pri kateri lahko zbirko funkcij kombiniramo z zbirko argumentov in dobimo zbirko rezultatov.

Monada `Maybe a`

Sedaj se lahko posvetimo nekaterim monadam. Poglejmo si monado za tip Maybe. To je monada, ki pove, kako računamo z *opcijskimi vrednostmi*.

Ali vas na kaj spominja?

Monada `[a]`

Seznami tvorijo monado, ker si lahko seznam vrednosti tipa `a` predstavljamo kot **nedeterministični izračun**, ki vrne poljubno število rezultatov. To pride prav, kadar računamo rešitve problema, ki ima lahko nič ali več rešitev, saj jih preprosto zložimo v seznam.

Primer: problem šahovskih dam

Na vajah boste rešili problem šahovskih dam na klasičen način in "na roke" izračunali boste seznam vseh rešitev. Na predavanjih poglejmo, kako lahko isti problem rešimo z monado.

Najprej poglejmo primer uporabe. Radi izračunali vse možne vsote, ko vržemo dve kocki:

1. Brez monade: `[x + y | x <- [1..6], y <- [1..6]]`
2. Z monado:

```
do x <- [1..6]
   y <- [1..6]
   return (x + y)
```

Še en primer: funkcija, reši kvadratno enačbo.

Iskanje v globino

Nazadnje si oglejmo monado `Search` za splošno iskanje rešitve problema.

Predstavljajmo si, da poskušamo izračunati rezultat tipa `a`, pri čemer moramo v postopku računanja izbirati med možnostmi tipa `r`. Nekatere možnosti uspešno vodijo do rezultatov, druge pa ne. Radi bi uspešno izračunali rezultat in zaporedje možnosti, ki je pripeljalo do njega. (Še bolje, radi bi seznam vseh rezultatov in zaporedij množnosti, ki so pripeljali do njega.)

Na primer, v labirintu iščemo lonec zlata (tip `a` je "zlato"), na vsakem razpotju pa se moramo odločiti, kam bomo zavili (tip `r` je "smer").

To klasično nalogo bi lahko rešili s *sestopanjem*: vsakič, ko imamo na voljo več možnosti, po vrsti preizkusimo vsako (sestopimo). Če možnost ne deluje, preizkusimo naslednjo.