Krótko o wolnych monadach

Łukasz Dąbek

15 października 2013

Aby z funktora `m` uczynić monadę należy napisać parę funkcji:

- `return :: a -> m a`,
- `(>>=) :: m a -> (a -> m b) -> m b`.

Korzystając z faktu, że mamy do dyspozycji funkcję `fmap` możemy równoważnie napisać inną parę funkcji:

- `return :: a -> m a`,
- `join :: m (m a) -> m a`.

Wolna monada to taka monada, która wykonuje minimalną pracę przy `join`ie.

Zaczniemy od definicji typu danych. Dla dowolnego funktora `f` definiujemy:

```
data Free f a = Pure a | Free (f (Free f a))
```

W ramach rozgrzewki możemy napisać instancję funktora:

```
instance Functor f => Functor (Free f) where
    fmap f (Pure a) = Pure (f a)
    fmap f (Free c) = Free (fmap (fmap f) c)
```

Zanim przejdziemy do implementacji interfejsu monadycznego rozważmy „kształt" danych typu `Free f a` dla przykładowych funktorów:

- `f = Maybe`: w tym wypadku struktura wygląda jak lista „bez danych" na „consach".
- `f = []`: w tym wypadku struktura wygląda jak drzewo.
- `f = (a,)`: w tym wypadku struktura wygląda jak regularna lista.

Z powyższych obserwacji możemy opisać kształt typu `Free f a` na potrzeby naszych intuicji:

> Wolna monada nad funktorem f jest drzewem, które zawiera dane typu polimorficznego w liściach i w każdym wierzchołku wewnętrznym zawiera pojemnik (funktor) dzieci.

Przykładowo struktura typu `Free [] Int` jest drzewem przechowującym w liściach liczby całkowite.

Uzbrojeni w nowe intuicje możemy zinterpretować typ funkcji `join` dla wolnych monad: jest to funkcja, która przetwarza drzewo drzew na normalne drzewo. Jak to robi? Podczepiając w miejsce liści wspomniane drzewa!

Zobaczmy implementację interfejsu monadycznego dla wolnych monad:

```
instance Functor f => Monad (Free f) where
    return = Pure
    join (Pure a) = a
    join (Free c) = Free (fmap join c)
```

Zauważmy, że przy implementacji funkcji `join` tylko pierwsze równanie wykonuje jakąkolwiek realną pracę: drugie równanie wywołuje funkcję `join` na swoich dzieciach.

Możemy jeszcze zinterpretować typ funkcji `(>>=)` w terminach drzew. Przypomnijmy ten typ: `Free f a -> (a -> Free f b) -> Free f b`. Przekładając na język drzew: funkcja `bind` bierze drzewo z liśćmi typu `a` i produkuje drzewo z liśćmi typu `b`. Robi to przy pomocy funkcji podanej w drugim argumencie: jest to funkcja, która dla każdego z liści wstawia na jego miejsce nowe drzewo.

Piotrek Polesiuk i Wojtek Jedynak wspomnieli mi, że wolne monady wyglądają bardzo podobnie do monady podstawiającej zmienne. Faktycznie tak jest, a po szczegóły odsyłam do Dana Piponiego[2].

Więcej o wolnych monadach można poczytać na blogu Gabriela Gonzalesa[1].

Literatura

[1] Gabriel Gonzalez, *Why free monads matter*. `http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html`.

[2] Dan Piponi, *Variable substitution gives a....* `http://blog.sigfpe.com/2006/11/variable-substitution-gives.html`.