Kontynuacje (i inne abstrakcje)

Tomasz Kulis

2024-05-11

KSI[4]

Wprowadzenie

Funkcje i składnia

```
times2 x = x * 2
```

```
(define (times2 arg)
  (* arg 2))
```

Funkcje i składnia

```
times2 x = x * 2
```

```
(define (times2 arg)
  (* arg 2))
```

```
factorial 0 = 1
factorial n = n * (factorial (n - 1))
```

```
(define (factorial n)
  (if (eq? n 0) 1 (* n (factorial (- n 1)))))
```

Funkcje i składnia

```haskell
times2 x = x * 2
```

```lisp
(define (times2 arg)
  (* arg 2))
```

```haskell
factorial 0 = 1
factorial n = n * (factorial (n - 1))
```

```lisp
(define (factorial n)
  (if (eq? n 0) 1 (* n (factorial (- n 1)))))
```

```haskell
map (\x -> x * x) [1, 2, 3] ≡ [1, 4, 9]
```

```lisp
(map (lambda (x) (* x x)) '(1 4 9)) ≡ '(1 4 9)
```

Kontynuacje

```
(* (+ 1 2) (- 3 7) 5)
```

Kontynuacje

```
(* (+ 1 2) (- 3 7) 5)

((lambda (x)
         (* x (- 3 7) 5))
         (+ 1 2))
```

Kontynuacje

```
(* (+ 1 2) (- 3 7) 5)

((lambda (x)
         (* x (- 3 7) 5))
         (+ 1 2))
```

Kontynuacje

```
(* (+ 1 2) (- 3 7) 5)
```

```
((lambda (x)
          (* x (- 3 7) 5))
          (+ 1 2))
```

```
(* □ (- 3 7) 5) ;; kontynuacja
```

Kontynuacja to abstrakcyjne pojęcie opisujące aktualny stan przebiegu programu, w szczególności nadchodzącej ewaluacji.

Kontynuacje

Przykład zastosowania kontynuacji poza informatyką można znaleźć np. w analizie – obliczając pochodną złożenia rozbijamy funkcję na najbardziej zewnętrzną kontynuację.

$$
\begin{aligned}
&(\sqrt{\cos(x)})' \\
&(\sqrt{\cdot}(\cos(x)))' \\
&(\sqrt{\cdot})'(\cos(x)) \cdot (\cos(x))' \\
&\frac{1}{2\sqrt{\cos(x)}} \cdot (-\sin(x))
\end{aligned}
$$

Ale dlaczego?

```
(define (factorial n)
  (if (eq? n 0) 1 (* n (factorial (- n 1)))))

(factorial 4)
```

Ale dlaczego?

```
(define (factorial n)
  (if (eq? n 0) 1 (* n (factorial (- n 1)))))

(factorial 4)

(* 4 (factorial 3))
(* 4 (* 3 (factorial 2)))
(* 4 (* 3 (* 2 (factorial 1))))
(* 4 (* 3 (* 2 (* 1 (factorial 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

Ale dlaczego?

```
(define (factorial2 a n)
  (if (eq? n 0) a (factorial2 (* n a) (- n 1))))

(factorial2 4)
```

Ale dlaczego?

```
(define (factorial2 a n)
  (if (eq? n 0) a (factorial2 (* n a) (- n 1))))

(factorial2 4)

(factorial2 1 4)
(factorial2 (* 1 4) 3)
(factorial2 (* 4 3) 2)
(factorial2 (* 12 2) 1)
(factorial2 (* 24 1) 0)
24
```

CPS

Continuation Passing Style to paradygmat programowania, w którym każda (nieprymitywna) funkcja ma typ postaci `... -> (b -> r) -> r`, gdzie ostatni argument to funkcja reprezentująca kontynuację obliczeń.

CPS

Continuation Passing Style to paradygmat programowania, w którym każda (nieprymitywna) funkcja ma typ postaci `... -> (b -> r) -> r`, gdzie ostatni argument to funkcja reprezentująca kontynuację obliczeń.

```
(define (*# x y cont) (cont (* x y)))
(define (-# x y cont) (cont (- x y)))
(define (eq?# x y cont) (cont (eq? x y)))

(define (factorial# n acc cont)
  (eq?# n 0 (lambda (b)
    (if b
      (cont acc)
        (*# n acc (lambda (acc2)
          (-# n 1 (lambda (n2)
            (factorial# n2 acc2 cont)))))))))
```

CPS

Specyfika ewaluacji CPS sprawia, że każda ewaluacja następuje ogonowo rekurencyjnie.

CPS wykorzystywany jest np. w kompilatorach i interpreterach języków – rozbicie wyrażenia na CPS pozwala na jego efektywną i schematyczną ewaluację.

`call/cc`, czyli `goto` języków funkcyjnych

`call-with-current-continuation`

Funkcja `call-with-current-continuation` (zazwyczaj nazywana call/cc w związku z jej aliasem w językach Scheme) przyjmuje jako argument inną funkcję jednoargumentową, i przekazuje jej „funkcję zwrotu do obecnej kontynuacji”. Co to oznacza?

call-with-current-continuation

Funkcja `call-with-current-continuation` (zazwyczaj nazywana call/cc w związku z jej aliasem w językach Scheme) przyjmuje jako argument inną funkcję jednoargumentową, i przekazuje jej „funkcję zwrotu do obecnej kontynuacji". Co to oznacza?

```
(define (call/cc-example)
  (call/cc (lambda (cont) (+ 1 2 (cont 3)))))
```

call-with-current-continuation

Funkcja `call-with-current-continuation` (zazwyczaj nazywana call/cc w związku z jej aliasem w językach Scheme) przyjmuje jako argument inną funkcję jednoargumentową, i przekazuje jej „funkcję zwrotu do obecnej kontynuacji". Co to oznacza?

```
(define (call/cc-example)
  (call/cc (lambda (cont) (+ 1 2 (cont 3)))))
```

call/cc przekazuje swojemu argumentowi funkcję podobną do `goto` lecz przyjmującą wartość, i na jej wykonaniu skacze do miejsca wywołania call/cc, przekazując otrzymaną wartość jako wartość wywołania call/cc

Pętle przez call/cc

```
(define loop-cont #f)
(define (example-loop)
  (match (call/cc (lambda (cont)
                    (set! loop-cont cont)
                    0))
    [5 'done]
    [x (display x)
       (loop-cont (+ x 1))]))
```

Pętle przez call/cc

```
(define loop-cont #f)
(define (example-loop)
  (match (call/cc (lambda (cont)
                    (set! loop-cont cont)
                    0))
    [5 'done]
    [x (display x)
       (loop-cont (+ x 1))]))
(define (example-break)
  (call/cc (lambda (break)
    (for-each (lambda (x)
      (if (> x 3)
        (break x)
        (display x)))
      '(1 2 3 4 5)))))
```

Wyjątki przez `call/cc`

Uogólniając poprzedni przykład z `break`, możemy wykorzystać `call/cc` do zaimplementowania wyjątków.

```
(define (div# fail a b)
    (if (eq? b 0)
        (fail 'divby0)
        (/ a b)))

(define (example-nothrow)
    (call/cc (lambda (throw)
       (map (div# throw 10) '(1 2 3)))))

(define (example-throws)
    (call/cc (lambda (throw)
       (map (div# throw 10) '(4 1 0)))))
```

Wyjątki przez `call/cc`

```
(define (catch (handler block cont))
    (cont
        (match (call/cc (lambda (back)
            (block (lambda (x) (list 'error x)))))
            [(list 'error err) (handler err)]
            [x x])))
```

Korutyny przez call/cc

```
(define (printer other)
    (display "Example1")
    (set! other (call/cc other))
    (display "Example2")
    (set! other (call/cc other))
    (display "Example3"))

(define (sleeper other)
    (sleep 5)
    (set! other (call/cc other))
    (sleep 10)
    (set! other (call/cc other))
    (sleep 15)zz
    (set! other (call/cc other)))
```

Delimited Continuations

Istnieje również wariant `call/cc` o bardziej lokalnym działaniu – tzw „delimited continuations”.

Delimited Continuations

Istnieje również wariant `call/cc` o bardziej lokalnym działaniu – tzw „delimited continuations”.

```
(reset (* 2 (shift b (- 7 (b (* 3 4))))))
```

Delimited Continuations

Istnieje również wariant `call/cc` o bardziej lokalnym działaniu – tzw „delimited continuations”.

```
(reset (* 2 (shift b (- 7 (b (* 3 4))))))
```

```
(reset (* 2 (shift b (- 7 (b 12)))))
(- 7 (* 2 12))
-17
```

Delimited Continuations

Istnieje również wariant `call/cc` o bardziej lokalnym działaniu – tzw „delimited continuations”.

```
(reset (* 2 (shift b (- 7 (b (* 3 4))))))
```

```
(reset (* 2 (shift b (- 7 (b 12)))))
(- 7 (* 2 12))
-17
```

```
(reset ((lambda (val) cont...)
    (shift k expr...)))
(reset ((lambda (k) expr...)
    (lambda (val) (reset cont...))))
```

Delimited Continuations

Istnieje również wariant `call/cc` o bardziej lokalnym działaniu – tzw „delimited continuations”.

```
(reset (* 2 (shift b (- 7 (b (* 3 4))))))
```

```
(reset (* 2 (shift b (- 7 (b 12)))))
(- 7 (* 2 12))
-17
```

```
(reset ((lambda (val) cont...)
    (shift k expr...)))
(reset ((lambda (k) expr...)
    (lambda (val) (reset cont...))))
```

`reset` w tym przypadku rozdziela kontynuację `shift` od reszty programu – stąd wynika ich nazwa. Warto zaznaczyć, że można zaimplementować ten rodzaj kontynuacji korzystając z `call/cc`.

`catch/throw` jako Delimited Continuations

Okazuje się, że jest wiele operacji które można zaimplementować jako delimited kontynuacje – przykładem jest np. para `catch/throw`:

```
(catch (lambda (x) x + 2)
    (+ 1 2 3 (throw 14) 5))
```

`catch/throw` jako Delimited Continuations

Okazuje się, że jest wiele operacji które można zaimplementować jako delimited kontynuacje – przykładem jest np. para `catch/throw`:

```
(catch (lambda (x) x + 2)
    (+ 1 2 3 (throw 14) 5))
```

Można zaimplementować te procedury np. przez `reset/shift`:

```
(define (catch handler body)
        (match (reset (body))
               [(cons 'thrown error) (handler error)]
               [val val]))

(define (throw value)
        (shift ignored
               (cons 'thrown value)))
```

Generatory przez Delimited Continuations

```
(define (append value) (shift cont (cons x (cont '()))))
(reset ((lambda ()
    (append 1)
    (append 2)
    (append 3)
    '())))
```

Generatory przez Delimited Continuations

```
(define (append value) (shift cont (cons x (cont '()))))
(reset ((lambda ()
    (append 1)
    (append 2)
    (append 3)
    '())))
```

Ogólniej:

```
(define (yield value) (shift cont (pair value cont)))
```

dynamic-wind

Scheme udostępnia dodatkową funkcję przydatną do korzystania z kontynuacji: `dynamic-wind`.

```
(dynamic-wind
        enter
        body
        leave)
```

dynamic-wind

Scheme udostępnia dodatkową funkcję przydatną do korzystania z kontynuacji: `dynamic-wind`.

```
(dynamic-wind
        enter
        body
        leave)
```

```
(define inwind #f)
(define inwind #f)
(dynamic-wind
 (lambda () (displayln "entry"))
 (lambda () (call/cc (lambda (cont) (set! inwind cont))))
 (lambda () (displayln "exit")))
```

`(call/cc (lambda (return) ...))`

... czyli dygresja o monadach

Funkcje kontratakują

Zastanówmy się nad matematycznym obiektem funkcji:

$$\exp : \mathbb{R} \ni x \mapsto e^x \in \mathbb{R}$$

Zauważmy, że każdemu elementowi dziedziny odpowiada dokładnie jeden – co więcej, w każdym momencie zachodzi $exp(x) = exp(x)$ – wartość funkcji nie zależy od żadnego stanu poza jej argumentami.

Funkcje kontratakują

Zastanówmy się nad matematycznym obiektem funkcji:

$$\exp : \mathbb{R} \ni x \mapsto e^x \in \mathbb{R}$$

Zauważmy, że każdemu elementowi dziedziny odpowiada dokładnie jeden – co więcej, w każdym momencie zachodzi $exp(x) = exp(x)$ – wartość funkcji nie zależy od żadnego stanu poza jej argumentami.

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

Funkcje kontratakują

Zastanówmy się nad matematycznym obiektem funkcji:

$$\exp : \mathbb{R} \ni x \mapsto e^x \in \mathbb{R}$$

Zauważmy, że każdemu elementowi dziedziny odpowiada dokładnie jeden – co więcej, w każdym momencie zachodzi $exp(x) = exp(x)$ – wartość funkcji nie zależy od żadnego stanu poza jej argumentami.

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

1. Jej wynik wykonania wynosi tyle samo dla takich samych argumentów

Funkcje kontratakują

Zastanówmy się nad matematycznym obiektem funkcji:

$$\exp : \mathbb{R} \ni x \mapsto e^x \in \mathbb{R}$$

Zauważmy, że każdemu elementowi dziedziny odpowiada dokładnie jeden – co więcej, w każdym momencie zachodzi $exp(x) = exp(x)$ – wartość funkcji nie zależy od żadnego stanu poza jej argumentami.

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

1. Jej wynik wykonania wynosi tyle samo dla takich samych argumentów
2. Nie wpływa na zewnętrzny stan programu – nie ma „efektów ubocznych”

Funkcje kontratakują

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

1. Jej wynik wykonania wynosi tyle samo dla takich samych argumentów
2. Nie wpływa na zewnętrzny stan programu – nie ma „efektów ubocznych”

Operowanie tylko na takich funkcjach ma wiele zalet:

Funkcje kontratakują

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

1. Jej wynik wykonania wynosi tyle samo dla takich samych argumentów
2. Nie wpływa na zewnętrzny stan programu – nie ma „efektów ubocznych”

Operowanie tylko na takich funkcjach ma wiele zalet:

1. Podczas ewaluacji możemy podstawić $f(x, y, z)$ za jego wartość bez martwienia się kolejnością ewaluacji

Funkcje kontratakują

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

1. Jej wynik wykonania wynosi tyle samo dla takich samych argumentów
2. Nie wpływa na zewnętrzny stan programu – nie ma „efektów ubocznych”

Operowanie tylko na takich funkcjach ma wiele zalet:

1. Podczas ewaluacji możemy podstawić $f(x, y, z)$ za jego wartość bez martwienia się kolejnością ewaluacji
2. Możemy nie wywoływać funkcji – jeżeli nie jest to wymagane, możemy nie obliczać wartości funkcji.

Funkcje kontratakują

Funkcję (procedurę) nazywamy czysto funkcyjną (ang. *purely functional*) gdy:

1. Jej wynik wykonania wynosi tyle samo dla takich samych argumentów
2. Nie wpływa na zewnętrzny stan programu – nie ma „efektów ubocznych”

Operowanie tylko na takich funkcjach ma wiele zalet:

1. Podczas ewaluacji możemy podstawić $f(x, y, z)$ za jego wartość bez martwienia się kolejnością ewaluacji
2. Możemy nie wywoływać funkcji – jeżeli nie jest to wymagane, możemy nie obliczać wartości funkcji.
3. Możemy bardzo łatwo zrównoleglić wykonania funkcji – brak zmian stanu implikuje brak problemów z synchronizacją

Funkcje kontratakują

Oczywiście nie jest taka interpretacja bez wad – główne z nich to:

Funkcje kontratakują

Oczywiście nie jest taka interpretacja bez wad – główne z nich to:

1. Dla każdej zmiany stanu potrzebujemy go skopiować (nie możemy zmodyfikować wartości argumentu)

Funkcje kontratakują

Oczywiście nie jest taka interpretacja bez wad – główne z nich to:

1. Dla każdej zmiany stanu potrzebujemy go skopiować (nie możemy zmodyfikować wartości argumentu)
2. Brak możliwości komunikacji z programem – Input/Output nie spełnia założeń bycia czysto funkcyjnym, więc zamiast działającego komputera mamy grzejnik

Funkcje kontratakują

Oczywiście nie jest taka interpretacja bez wad – główne z nich to:

1. Dla każdej zmiany stanu potrzebujemy go skopiować (nie możemy zmodyfikować wartości argumentu)
2. Brak możliwości komunikacji z programem – Input/Output nie spełnia założeń bycia czysto funkcyjnym, więc zamiast działającego komputera mamy grzejnik

Nie będziemy przejmować się tymi problemami na tym wykładzie – od tego momentu poprzez „funkcja” mamy na myśli procedurę czysto funkcyjną.

Funkcje kontratakują

Załóżmy, że chcemy dodać do języka
funkcję obliczającą odwrotność liczby k w grupie $\mathbb{Z}_n$. Będzie to funkcja rodzaju `Int -> Int -> ...`. Jaki typ powinna zwrócić ta funkcja?

Funkcje kontratakują

Załóżmy, że chcemy dodać do języka
funkcję obliczającą odwrotność liczby k w grupie $\mathbb{Z}_n$. Będzie to funkcja rodzaju `Int -> Int -> ...`. Jaki typ powinna zwrócić ta funkcja?

Jeżeli zwrócimy Int to nastąpi problem gdy liczba nie jest względnie pierwsza z n – wtedy nie istnieje taka odwrotność.

Funkcje kontratakują

Załóżmy, że chcemy dodać do języka
funkcję obliczającą odwrotność liczby k w grupie $\mathbb{Z}_n$. Będzie to funkcja rodzaju `Int -> Int -> ...`. Jaki typ powinna zwrócić ta funkcja?

Jeżeli zwrócimy Int to nastąpi problem gdy liczba nie jest względnie pierwsza z n – wtedy nie istnieje taka odwrotność.

Oczywiście moglibyśmy wykorzystać wcześniej wspomniane `catch/throw` – jednak jako że chcemy, aby funkcja nie miała efektów ubocznych możemy podejść do tego problemu inaczej...

Maybe/Optional

Zdefiniujmy typ:

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

Nasz typ może posiadać dwa rodzaje wartości – `Just a` oznaczający jakiś wynik lub `Nothing` oznaczający brak wartości. Zauważmy, że możemy teraz zdefiniować naszą funkcję jako zwracającą typ Maybe zamiast zwykłego Inta:

```
inv a n = if gcd a n == 1
            then Just (calcInv a n)
            else Nothing
```

Maybe/Optional

Zdefiniujmy typ:

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

Nasz typ może posiadać dwa rodzaje wartości – `Just a` oznaczający jakiś wynik lub `Nothing` oznaczający brak wartości. Zauważmy, że możemy teraz zdefiniować naszą funkcję jako zwracającą typ Maybe zamiast zwykłego Inta:

```
inv a n = if gcd a n == 1
            then Just (calcInv a n)
            else Nothing
```

Ale co jeżeli chcemy coś zrobić z wynikiem funkcji – np. dodać 2 – mamy do wyboru dwie opcje:

- Wyjść z kontekstu `Maybe`, przyjmując jakąś wartość domyślną gdy wynik pierwotnej funkcji to `Nothing`
- Pozostać w kontekście `Maybe`...

Czym jest funktor?

Funktorem nazywamy typ danych parametryzowany jednym typem wraz z definiowaną funkcją

```
map :: (a -> b) -> (Functor a -> Functor b)
```

spełniającą dodatkowe dwa założenia:

Czym jest funktor?

Funktorem nazywamy typ danych parametryzowany jednym typem wraz z definiowaną funkcją

```
map :: (a -> b) -> (Functor a -> Functor b)
```

spełniającą dodatkowe dwa założenia:

1. `map (\x -> x) x = x` – identyczność jest zachowana

Czym jest funktor?

Funktorem nazywamy typ danych parametryzowany jednym typem wraz z definiowaną funkcją

```
map :: (a -> b) -> (Functor a -> Functor b)
```

spełniającą dodatkowe dwa założenia:

1. `map (\x -> x) x = x` – identyczność jest zachowana
2. `map f (map g x) = map (\a -> f (g a)) x` – operacja jest łączna względem złożenia funkcji

Czym jest funktor?

Funktorem nazywamy typ danych parametryzowany jednym typem wraz z definiowaną funkcją

```
map :: (a -> b) -> (Functor a -> Functor b)
```

spełniającą dodatkowe dwa założenia:

1. `map (\x -> x) x = x` – identyczność jest zachowana
2. `map f (map g x) = map (\a -> f (g a)) x` – operacja jest łączna względem złożenia funkcji

Dobrze znanym przykładem funktora jest lista wraz ze standardową operacją mapy po elementach.

Funktor Maybe

Dla typu `Maybe` możemy zdefiniować funktor w następujący sposób:

```
map f Nothing = Nothing
map f (Just a) = Just (f a)
```

Łatwo można potwierdzić, że powyższa definicja spełnia prawa funktorów. Rozwiązuje to wcześniej napotkany problem wychodzenia z kontekstu – korzystając z map możemy obliczyć funkcję *f* pozostając w kontekście typu `Maybe`:
`f x = map (\x -> x + 2) (inv x)`.

Funktor Maybe

Dla typu `Maybe` możemy zdefiniować funktor w następujący sposób:

```
map f Nothing = Nothing
map f (Just a) = Just (f a)
```

Łatwo można potwierdzić, że powyższa definicja spełnia prawa funktorów. Rozwiązuje to wcześniej napotkany problem wychodzenia z kontekstu – korzystając z map możemy obliczyć funkcję f pozostając w kontekście typu `Maybe`:
`f x = map (\x -> x + 2) (inv x)`.

Ale co jeżeli chcielibyśmy obliczyć odwrotność tej nowej liczby? Zauważmy, że aplikując ponownie `map` otrzymamy element typu `Maybe (Maybe x)`. Czy mamy jak pozbyć się jednej z warstw `Maybe`?

Czym jest monada?

Monada to koncept często powtarzający się w programowaniu funkcyjnym, pozwalającym opisywać standardowo stanowe operacje bez wykonywania ich efektów ubocznych.

Czym jest monada?

Monada to koncept często powtarzający się w programowaniu funkcyjnym, pozwalającym opisywać standardowo stanowe operacje bez wykonywania ich efektów ubocznych.

Formalniej, monadą nazywamy funktor dla którego zdefiniowane są dodatkowe operacje:

```
unit :: a -> Monad a
join :: Monad (Monad a) -> Monad a
```

Czym jest monada?

Monada to koncept często powtarzający się w programowaniu funkcyjnym, pozwalającym opisywać standardowo stanowe operacje bez wykonywania ich efektów ubocznych.

Formalniej, monadą nazywamy funktor dla którego zdefiniowane są dodatkowe operacje:

```
unit :: a -> Monad a
join :: Monad (Monad a) -> Monad a
```

W moim przekonaniu najłatwiej myśleć o nich jako o kontekście nad którym możemy wykonywać obliczenia: wtedy `map` to dekorator na funkcji pozwalający jej pracować nad danym kontekstem, `unit` do nadanie wartości „domyślnego” kontekstu, a „join” pozwala na połączenie kontekstów dwóch wcześniej wykonanych operacji. warstwy kontekstu.

Monada Maybe

Wracając do `Maybe` możemy zdefiniować funkcję monadyczne w następujący sposób:

```
map func (Just a) = Just (func a)
map Nothing = Nothing

unit value = Just value

join (Just val) = val
join Nothing = Nothing
```

Monada Maybe

Wracając do `Maybe` możemy zdefiniować funkcję monadyczne w następujący sposób:

```
map func (Just a) = Just (func a)
map Nothing = Nothing

unit value = Just value

join (Just val) = val
join Nothing = Nothing

map (\x -> x + 2) (inv 2 7) == Just 6
map (\x -> x + 2) (inv 3 9) == Nothing
```

Prawa monad

Monady muszą spełniać następujące prawa:

1. `map (\a -> a) x == x` – identyczność to el. neutralny `map`, przeniesione z funktora

Prawa monad

Monady muszą spełniać następujące prawa:

1. `map (\a -> a) x == x` – identyczność to el. neutralny `map`, przeniesione z funktora
2. `map f (map g x) == map (\e -> f (g e)) x` – łączność `map` względem siebie, jak w funktorze

Prawa monad

Monady muszą spełniać następujące prawa:

1. `map (\a -> a) x == x` – identyczność to el. neutralny `map`, przeniesione z funktora
2. `map f (map g x) == map (\e -> f (g e)) x` – łączność `map` względem siebie, jak w funktorze
3. `join (unit x) == x` – `unit` to el. neutralny `join`

Prawa monad

Monady muszą spełniać następujące prawa:

1. `map (\a -> a) x == x` – identyczność to el. neutralny `map`, przeniesione z funktora
2. `map f (map g x) == map (\e -> f (g e)) x` – łączność `map` względem siebie, jak w funktorze
3. `join (unit x) == x` – `unit` to el. neutralny `join`
4. `join (map join x) == join (join x)` – łączność `join` względem siebie

Prawa monad

Monady muszą spełniać następujące prawa:

1. `map (\a -> a) x == x` – identyczność to el. neutralny `map`, przeniesione z funktora
2. `map f (map g x) == map (\e -> f (g e)) x` – łączność `map` względem siebie, jak w funktorze
3. `join (unit x) == x` – `unit` to el. neutralny `join`
4. `join (map join x) == join (join x)` – łączność `join` względem siebie
5. `join (map (map f x)) == map f (join x)` – relacja na map i join

Monada – równoważne definicje

W programowaniu często korzysta się z alternatywnej definicji monady, gdzie `join` zastąpione jest przez operator >>= (tzw. bind).

Operator ten o typie `Monad a -> (a -> Monad b) -> Monad b` można zdefiniować jako
`monad >>= func = join (map func monad)`. Zachodzi wtedy `join m == (m >>= (\x -> x))`. Z wcześniej opisanych praw wynikają prawa o el. neutralnym i łączności >>= – w związku z tym jest on bardzo przydatny w programowaniu, ponieważ pozwala on „łączyć” kolejne operacje w pewnym kontekście.

Monada – równoważne definicje

W programowaniu często korzysta się z alternatywnej definicji monady, gdzie `join` zastąpione jest przez operator >>= (tzw. bind).

Operator ten o typie `Monad a -> (a -> Monad b) -> Monad b` można zdefiniować jako `monad >>= func = join (map func monad)`. Zachodzi wtedy `join m == (m >>= (\x -> x))`. Z wcześniej opisanych praw wynikają prawa o el. neutralnym i łączności >>= – w związku z tym jest on bardzo przydatny w programowaniu, ponieważ pozwala on „łączyć" kolejne operacje w pewnym kontekście.

Jeszcze inną równoważną definicją jest operator „kompozycji"

```
(>=>) :: (a -> Monad b) -> (b -> Monad c) -> (a -> Monad c)
```

– podczas gdy jest on również bardzo interesujący, nie będziemy się w niego zagłębiać.

Either

```haskell
data Either l r = Left l | Right r

map :: (a -> b) -> (Either l a -> Either l b)
map func (Right val) = Right (func val)
map func (Left e) = (Left e)

unit value = Right value

join (Right value) = value
join (Left value) = Left value
```

List

```haskell
data List x = Nil | Cons x (List x)

concat :: List x -> List x -> List x
concat Nil other = other
concat (Cons x rest) other = Cons x (concat rest other)

map :: (a -> b) -> (List a -> List b)
map func Nil = Nil
map func (Cons a rest) = Cons (func a) (map func rest)

join Nil = Nil
join (Cons list rest) = concat list (join rest)
```

Identity

```haskell
data Identity a = Identity a
map func (Identity a) = Identity (f a)

unit a = Identity a

join (Identity (Identity a)) = Identity a
```

```
(return '())
```

... czyli co to ma do kontynuacji

Kontynuacja jako monada

Zastanowimy się, czy nie da się zdefiniować operacji dla funkcji przyjmujących kontynuacje `(a -> r) -> r` względem typu `a`.

Kontynuacja jako monada

Zastanowimy się, czy nie da się zdefiniować operacji dla funkcji przyjmujących kontynuacje `(a -> r) -> r` względem typu `a`. Definicja `unit` wynika naturalnie z dostępnych nam wartości:

```
unit :: a -> (a -> r) -> r
unit a = \cont -> cont a
```

Kontynuacja jako monada

Zastanowimy się, czy nie da się zdefiniować operacji dla funkcji przyjmujących kontynuacje `(a -> r) -> r` względem typu `a`. Definicja `unit` wynika naturalnie z dostępnych nam wartości:

```
unit :: a -> (a -> r) -> r
unit a = \cont -> cont a
```

Definicja `map` również przychodzi wraz z analizą dostępnych nam typów:

```
map :: (a -> b) -> ((a -> r) -> r) -> ((b -> r) -> r)
map func ext = \cont -> ext (\val -> cont (func val))
```

Ostatnia definicja – `join` – również powstaje z rozumowania na typach:

```
join :: (((a -> r) -> r) -> r) -> r
join cont = \c -> cont (\a -> a c)
```

Kontynuacja jako uniwersalna monada

Łatwo można zweryfikować, że dane operacje spełniają prawa monad. Ciekawsze jest zachowanie operatora >>=. Po rozpisaniu definicji otrzymujemy

```
val >>= fun = \cont -> val $ \res -> (fun res) cont.
```

Okazuje się, że mając „wartość” w kontekście kontynuacji `(a -> r) -> r` i `a -> (b -> r) -> r`, to operator >>= równoważny jest złożenia funkcji stylu CPS.

Kontynuacja jako uniwersalna monada

Łatwo można zweryfikować, że dane operacje spełniają prawa monad. Ciekawsze jest zachowanie operatora >>=. Po rozpisaniu definicji otrzymujemy

```
val >>= fun = \cont -> val $ \res -> (fun res) cont.
```

Okazuje się, że mając „wartość” w kontekście kontynuacji `(a -> r) -> r` i `a -> (b -> r) -> r`, to operator >>= równoważny jest złożenia funkcji stylu CPS.

W 1994 roku Andrzej Filinski wykazał, że każda monada o obliczalnych funkcjach może zostać zaimplementowana korzystając z monadu kontynuacji w języku z leniwą ewaluacją.

Maybe jako Kontynuacje

```haskell
nothing :: (a -> Maybe r) -> Maybe r
nothing = (\x -> Nothing)

just :: a -> (a -> Maybe r) -> Maybe r
just a = (\c -> c a)
-- czyli just a = unit a
```

Maybe jako Kontynuacje

```
nothing :: (a -> Maybe r) -> Maybe r
nothing = (\x -> Nothing)


just :: a -> (a -> Maybe r) -> Maybe r
just a = (\c -> c a)
-- czyli just a = unit a

join (just nothing) == \c -> (\x -> x nothing) (\a -> a c)
    == \c -> (\a -> a c) nothing == \c -> (nothing c)
    == \c -> Nothing == nothing
join (nothing) == \c -> (\x -> Nothing) (\a -> a c)
    == \c -> Nothing == nothing
join (just (just v)) == \c -> (\x -> x (\y -> y v)) (\a -> a c)
    == \c -> (\a -> a c) (\y -> y v) == \c -> (\y -> y v) c
    == \c -> (c v) = just v
```

Kontynuacja tematu

- `http://hjemmesider.diku.dk/~andrzej/papers/RM-abstract.html` – oryginalna publikacja dotycząca monadów i kontynuacji
- `https://www.youtube.com/watch?v=TE48LsgVlIU` – wykład o kontynuacjach, w szczególności o delimited continuations
- `https://wiki.haskell.org/All_About_Monads` – fragment wiki Haskella opisujący najczęściej używane monady