Od grzejnika do komputera

… czyli krótki wstęp do programowania funkcyjnego

Tomasz Kulis

KSI[5]

2025-05-24

Czym jest funkcja?

Czym jest funkcja?

$$f(x) = x + 2$$

Czym jest funkcja?

$$f(x) = x + 2$$

Czym jest funkcja?

$$f(x) = x + 2$$

$$g(x) = x \cdot x$$

Czym jest funkcja?

$$f(x) = x + 2$$

$$g(x) = x \cdot x$$

$$h(x) = \begin{cases} \text{if } x \leq 1 \text{ then:} & x \\ \text{otherwise:} & x + \min(h(x-1), h(x-2)) \end{cases}$$

Dla ciekawskich: A002620

Czym jest funkcja?

$$f(x) = x + 2$$

$$g(x) = x \cdot x$$

$$h(x) = \begin{cases} \text{if } x \leq 1 \text{ then:} & x \\ \text{otherwise:} & x + \min(h(x-1), h(x-2)) \end{cases}$$

Dla ciekawskich: A002620

$$\ell(x) = g \circ f = g(f(x)) = f(x) \cdot f(x) = (x+2) \cdot (x+2)$$

Przejrzystość referencyjna

$$\ell(x) = g \circ f = g(f(x)) \overset{!!!}{=} f(x) \cdot f(x) = (x+2) \cdot (x+2)$$

Zauważmy, co tutaj zrobiliśmy: podstawiliśmy $f(x)$ pod obydwa wystąpienia argumentu $g(x)$!

Przejrzystość referencyjna

$$\ell(x) = g \circ f = g(f(x)) \overset{!!!}{=} f(x) \cdot f(x) = (x+2) \cdot (x+2)$$

Zauważmy, co tutaj zrobiliśmy: podstawiliśmy $f(x)$ pod obydwa wystąpienia argumentu $g(x)$!

Legalność takiego podstawienia nazywamy **przejrzystością referencyjną**.

English: referential transparency

Przejrzystość referencyjna

$$\ell(x) = g \circ f = g(f(x)) \overset{!!!}{=} f(x) \cdot f(x) = (x+2) \cdot (x+2)$$

Zauważmy, co tutaj zrobiliśmy: podstawiliśmy $f(x)$ pod obydwa wystąpienia argumentu $g(x)$!

Legalność takiego podstawienia nazywamy **przejrzystością referencyjną**.

English: referential transparency

Prowadzi to do jednej z dwóch głównych idei programowania funkcyjnego: chcemy zachować przejrzystość referencyjną.

Przejrzystość referencyjna

Konsekwencje przejrzystości referencyjnej są o wielu aspektach.

Przejrzystość referencyjna

Konsekwencje przejrzystości referencyjnej są o wielu aspektach.

Ewaluacja staje się wręcz trywialna (chociaż trywialny sposób nie zawsze jest efektywny).

Przejrzystość referencyjna

Konsekwencje przejrzystości referencyjnej są o wielu aspektach.

Ewaluacja staje się wręcz trywialna (chociaż trywialny sposób nie zawsze jest efektywny).

Debugowanie również staje prostsze - dużo “naturalnych” matematycznie własności staje się automatycznie prawdziwymi.

Przejrzystość referencyjna

Konsekwencje przejrzystości referencyjnej są o wielu aspektach.

Ewaluacja staje się wręcz trywialna (chociaż trywialny sposób nie zawsze jest efektywny).

Debugowanie również staje prostsze - dużo “naturalnych” matematycznie własności staje się automatycznie prawdziwymi.

Mamy więcej opcji optymalizacji, np. cacheowanie wyników funkcji.

```
g(f(x)) + h(f(x)) <===> let tmp = f(x) in g(tmp) + f(tmp)
```

Przejrzystość referencyjna

Konsekwencje przejrzystości referencyjnej są o wielu aspektach.

Ewaluacja staje się wręcz trywialna (chociaż trywialny sposób nie zawsze jest efektywny).

Debugowanie również staje prostsze - dużo “naturalnych” matematycznie własności staje się automatycznie prawdziwymi.

Mamy więcej opcji optymalizacji, np. cacheowanie wyników funkcji.

```
g(f(x)) + h(f(x)) <===> let tmp = f(x) in g(tmp) + f(tmp)
```

Przejrzystość implikuje również trywialne zrównoleglanie, bo...

Przejrzystość referencyjna

… wymaga ona, aby funkcje nie zależały od zewnętrznego, nie-stałego stanu – co więcej, nie mogą one go w żaden sposób zmieniać.

```
int f() { static int ret = 0; return ret++; } // !!!
```

Przejrzystość referencyjna

... wymaga ona, aby funkcje nie zależały od zewnętrznego, nie-stałego stanu – co więcej, nie mogą one go w żaden sposób zmieniać.

```
int f() { static int ret = 0; return ret++; } // !!!
```

Co z tego wynika? Między innymi to, że, nie możemy zbytnio korzystać ze *zmiennych* ani operacji *Input/Output*.

```
int g(int x) { if (x >= 10) x *= 2; return x; }
```

Przejrzystość referencyjna

... wymaga ona, aby funkcje nie zależały od zewnętrznego, nie-stałego stanu – co więcej, nie mogą one go w żaden sposób zmieniać.

```
int f() { static int ret = 0; return ret++; } // !!!
```

Co z tego wynika? Między innymi to, że, nie możemy zbytnio korzystać ze *zmiennych* ani operacji *Input/Output*.

```
int g(int x) { if (x >= 10) x *= 2; return x; }
```

```
int p(int val) { return val * val; }
int q() { printf("Hello! "); return 2; }
p(q()) /* Hello!*/; q() * q() /* Hello! Hello! */
```

Czym jest grzejnik?

Źródło: https://allegro.pl/oferta/grzejnik-elektryczny-olejowy-1500w-z-termostatem-13211862072

Czym jest grzejnik?

So [us] geeky Haskell guys have started with a completely useless language.
In the end a program with no effect, there's no point in running it, is there?
You know, you have this black box, and you press "go" and it says you know, it gets hot but there's no output – wait, why did you run the program?
The reason to run a program is to have an effect.

— Simon Peyton Jones, jeden z twórców Haskella

Co z tym zrobić?

Nudna opcja: nic

Znacząca większość języków funkcyjnych pozwala na efekty uboczne.

```
greet(male, Name) ->
  io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
  io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
  io:format("Hello, ~s!", [Name]).
```

Źródło: Learn You Some Erlang for great good!

Rozwiązujemy w ten sposób grzejnik *ale* tracimy zalety przejrzystości.

Ciekawa opcja: coś

Załóżmy jednak, że chcemy zachować przejrzystość referencyjną, jednocześnie zezwalając na sensowny system wykonywania efektów (m.in. operacji I/O).

Ciekawa opcja: coś

Załóżmy jednak, że chcemy zachować przejrzystość referencyjną, jednocześnie zezwalając na sensowny system wykonywania efektów (m.in. operacji I/O).

Całe szczęście nie jesteśmy jedyni i ktoś już rozwiązał ten problem za nas! Zobaczymy dzisiaj kilka takich rozwiązań, które występują w bardziej lub jeszcze bardziej niszowych językach programowania.

Ciekawa opcja: coś

Załóżmy jednak, że chcemy zachować przejrzystość referencyjną, jednocześnie zezwalając na sensowny system wykonywania efektów (m.in. operacji I/O).

Całe szczęście nie jesteśmy jedyni i ktoś już rozwiązał ten problem za nas! Zobaczymy dzisiaj kilka takich rozwiązań, które występują w bardziej lub jeszcze bardziej niszowych językach programowania.

Żeby jednak dojść do pierwszego z nich, wpierw będziemy musieli wprowadzić podstawę, na której opierać się będzie nasz system efektów.

MONADY

Monada

All told, a **monad** in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor.

— Saunders MacLane

...

Monada

All told, a **monad** in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor.

— Saunders MacLane

...

Chcielibyśmy zamiast tego wprowadzić intuicję, która wytłumaczy nam dlaczego ten obiekt matematyczny jest ciekawy dla naszych potrzeb.

Drugi fundament programowania funkcyjnego

Drugi fundament programowania funkcyjnego

Funkcje!

Funkcje

Poza przejrzystością referencyjną, w programowaniu funkcyjnym kluczowym jest, aby funkcje były *obywatelami pierwszej klasy*. W praktyce oznacza to, że możemy traktować je jak obiekty matematyczne analogicznie do każdego innego typu.

English: first class citizens

Funkcje

Poza przejrzystością referencyjną, w programowaniu funkcyjnym kluczowym jest, aby funkcje były *obywatelami pierwszej klasy*. W praktyce oznacza to, że możemy traktować je jak obiekty matematyczne analogicznie do każdego innego typu.

English: first class citizens

Jest to bardzo ważne pod tym względem, że jak zaraz się okaże będziemy znacząco wykorzystywać fakt, że możemy przekazywać i zwracać funkcje z innych funkcji (funkcje operujące na funkcjach zwykle nazywa się *funkcjami wyższego rzędu*).

Funkcje

Przypomnijmy jeszcze następujący fakt (który być może został wprowadzony przy teorii mnogości na 1 roku):

Istnieje naturalna bijekcja pomiędzy funkcjami:

$$f : X \times Y \to Z$$
$$f' : X \to Y \to Z$$

W sumie to dokładniej bijekcja jest między zbiorami $Z^{X \times Y}$ a $\left(Z^Y\right)^X$

Funkcje

Przypomnijmy jeszcze następujący fakt (który być może został wprowadzony przy teorii mnogości na 1 roku):

Istnieje naturalna bijekcja pomiędzy funkcjami:

$$f : X \times Y \to Z$$
$$f' : X \to Y \to Z$$

W sumie to dokładniej bijekcja jest między zbiorami $Z^{X \times Y}$ a $\left(Z^Y\right)^X$

Wykorzystamy to, i przez większość czasu będziemy operować na takich funkcjach unarnych.

Dygresja o składni

Aby uniknąć potrzeby wprowadzania składni i semantyki Haskella, na potrzeby prezentacji będzimemy posługiwać się językiem matematyki, z dodatkowymi rozszerzeniami.

Z owych rozszerzeń, pozwolimy na anotację parametryzacji typu przez $\langle$nawiasy$\rangle$ i na funkcje anonimowe w stylu JavaScripta: $x \Rightarrow x^2 + 2$.

Funktor

Funktor to interfejs nad typem generycznym (parametryzowanym), definiujący jedną funkcję “map”.

$$\mathrm{map}_F : (\alpha \to \beta) \to F\langle\alpha\rangle \to F\langle\beta\rangle$$

Przykłady funktorów: List

Typ listy jest funktorem, z klasyczną funkcją “map”, która aplikuje zadaną funkcję na każdym elemencie i zwraca nową listę.

$$\text{map} : (\alpha \to \beta) \to \text{List}\langle\alpha\rangle \to \text{List}\langle\beta\rangle$$

$$\text{map}\ (f)\ ([a_1, a_2, ...]) = [f(a_1), f(a_2), ...]$$

Przykłady funktorów: List

Typ listy jest funktorem, z klasyczną funkcją “map”, która aplikuje zadaną funkcję na każdym elemencie i zwraca nową listę.

$$\text{map} : (\alpha \to \beta) \to \text{List}\langle\alpha\rangle \to \text{List}\langle\beta\rangle$$

$$\text{map}\ (f)\ ([a_1, a_2, ...]) = [f(a_1), f(a_2), ...]$$

$$\text{map}(x \Rightarrow x \cdot 2)([1, 2, 3]) = [2, 4, 6]$$

$$\text{map}(x \Rightarrow x \cdot x - 1)([2, 4, 6]) = [3, 15, 35]$$

$$\text{map}(x \Rightarrow [x \cdot 2, x + 2])([1, 2, 3]) = [[2, 3], [4, 4], [6, 5]]$$

Przykłady funktorów: Option

Inny przykład funktora: typ `Option` (znany też jako `Optional` i `Maybe`).

Definiujemy go jako *typ sumy* z dwoma konstruktorami:

- Some(T) zawierającym pewną wartość typu `T`
- None, reprezentującym brak wartości (`null`)

Przykłady funktorów: Option

Inny przykład funktora: typ `Option` (znany też jako `Optional` i `Maybe`).

Definiujemy go jako *typ sumy* z dwoma konstruktorami:

- Some(T) zawierającym pewną wartość typu `T`
- None, reprezentującym brak wartości (`null`)

Definiujemy metodę "map" w następujący sposób:

$$\text{map} : (\alpha \to \beta) \to \text{Option}\langle\alpha\rangle \to \text{Option}\langle\beta\rangle$$

$$\text{map}(f)(a) = \begin{cases} \text{Some}(f(v)) & \text{gdy} \quad a = \text{Some}(v) \\ \text{None} & \text{gdy} \quad a = \text{None} \end{cases}$$

Przykłady funktorów: Option

$$\text{map} : (\alpha \to \beta) \to \text{Option}\langle\alpha\rangle \to \text{Option}\langle\beta\rangle$$

$$\text{map}(f)(a) = \begin{cases} \text{Some}(f(v)) & \text{gdy} \quad a = \text{Some}(v) \\ \text{None} & \text{gdy} \quad a = \text{None} \end{cases}$$

$$\text{map}(x \Rightarrow x + 2)(\text{Some}(4)) = \text{Some}(6) : \text{Option}\langle\text{Int}\rangle$$

$$\text{map}(x \Rightarrow x + 2)(\text{None}) = \text{None} \qquad : \text{Option}\langle\text{Int}\rangle$$

$$\text{map}(x \Rightarrow \text{None})(\text{Some}(7)) = \text{Some}(\text{None}) : \text{Option}\langle\text{Option}\langle\text{Int}\rangle\rangle$$

Przykłady funktorów: Result

Analogiczny funktor istnieje dla typu `Result<A,B>` z konstruktorami:

- $\mathrm{Err}(A)$ zawierający "błąd" typu pierwszego parametru A
- $\mathrm{Ok}(B)$ zawierający wartość typu drugiego parametru B

Przykłady funktorów: Result

Analogiczny funktor istnieje dla typu `Result<A,B>` z konstruktorami:

- $\mathrm{Err}(A)$ zawierający "błąd" typu pierwszego parametru A
- $\mathrm{Ok}(B)$ zawierający wartość typu drugiego parametru B

$$\mathrm{map} : (\alpha \to \beta) \to \mathrm{Result}\langle C, \alpha\rangle \to \mathrm{Result}\langle C, \beta\rangle$$

$$\mathrm{map}(f)(r) = \begin{cases} \mathrm{Ok}(f(b)) & \text{gdy } \; r = \mathrm{Ok}(b) \\ \mathrm{Err}(a) & \text{gdy } \; r = \mathrm{Err}(a) \end{cases}$$

Uwaga: powyżej zezwalamy na "częściową aplikację typu generycznego". Dokładniej, funktorem jest typ $R_C\langle T\rangle \equiv \mathrm{Result}\langle C, T\rangle$ dla każdego C.

Przykłady funktorów: Result

$$\mathrm{map} : (\alpha \to \beta) \to \mathrm{Result}\langle C, \alpha\rangle \to \mathrm{Result}\langle C, \beta\rangle$$

$$\mathrm{map}(f)(r) = \begin{cases} \mathrm{Ok}(f(b)) & \text{gdy } \; r = \mathrm{Ok}(b) \\ \mathrm{Err}(a) & \text{gdy } \; r = \mathrm{Err}(a) \end{cases}$$

Uwaga: powyżej zezwalamy na "częściową aplikację typu generycznego". Dokładniej, funktorem jest typ $R_C\langle T\rangle \equiv \mathrm{Result}\langle C, T\rangle$ dla każdego C.

$$\mathrm{map}(x \Rightarrow x + 2)(\mathrm{Ok}(10)) = \mathrm{Ok}(12) : \mathrm{Result}\langle C, \mathrm{Int}\rangle$$

$$\mathrm{map}(x \Rightarrow x + 2)(\mathrm{Err}(42)) = \mathrm{Err}(42) : \mathrm{Result}\langle \mathrm{Int}, \mathrm{Int}\rangle$$

Prawa funktorów

Aby dany typ generyczny był funktorem chcemy trochę więcej niż byle jaką definicję `map`. W szczególności aby spełniała następujące prawa:

$$\mathrm{map}(\mathrm{id}) = \mathrm{id}$$

$$\text{czyli } \mathrm{map}(x \Rightarrow x) = (x \Rightarrow x)$$

Prawa funktorów

Aby dany typ generyczny był funktorem chcemy trochę więcej niż byle jaką definicję `map`. W szczególności aby spełniała następujące prawa:

$$\mathrm{map}(\mathrm{id}) = \mathrm{id}$$

$$\text{czyli } \mathrm{map}(x \Rightarrow x) = (x \Rightarrow x)$$

$$\mathrm{map}\ (f \circ g) = \mathrm{map}(f) \circ \mathrm{map}(g)$$

Prawa funktorów

Aby dany typ generyczny był funktorem chcemy trochę więcej niż byle jaką definicję `map`. W szczególności aby spełniała następujące prawa:

$$\text{map}(\text{id}) = \text{id}$$

$$\text{czyli } \text{map}(x \Rightarrow x) = (x \Rightarrow x)$$

$$\text{map } (f \circ g) = \text{map}(f) \circ \text{map}(g)$$

Łatwo można zweryfikować, że powyżej zdefiniowane operacje dla typów `List, Option, Result` faktycznie spełniają owe założenia.

Monada

Monada to **funktor**, który dodatkowo definiuje dwie nowe metody:

$$\text{unit}_M : \alpha \to M\langle\alpha\rangle$$

$$\text{join}_M : M\langle M\langle\alpha\rangle\rangle \to M\langle\alpha\rangle$$

Monada

Monada to **funktor**, który dodatkowo definiuje dwie nowe metody:

$$\text{unit}_M : \alpha \to M\langle\alpha\rangle$$

$$\text{join}_M : M\langle M\langle\alpha\rangle\rangle \to M\langle\alpha\rangle$$

Konceptyjnie, monady możemy traktować jako *złączalny kontekst*, wewnątrz którego wykonujemy dane obliczenia – możemy opakować typ w kontekst korzystając z unit, lub złączyć dwa zagnieżdzone konteksty w jeden przez join.

Alternatywna definicja

Często spotyka się **równoważną** definicję monady, w której funkcja "join" zastąpiona jest funkcją znaną jako "bind" czy "flatMap":

$$\mathrm{flatMap}_M : (\alpha \to M\beta) \to M\langle\alpha\rangle \to M\langle\beta\rangle$$

$$\mathrm{flatMap}(f)(x) = \mathrm{join}(\mathrm{map}(f)(x))$$

Alternatywna definicja

Często spotyka się **równoważną** definicję monady, w której funkcja "join" zastąpiona jest funkcją znaną jako "bind" czy "flatMap":

$$\text{flatMap}_M : (\alpha \to M\beta) \to M\langle\alpha\rangle \to M\langle\beta\rangle$$

$$\text{flatMap}(f)(x) = \text{join}(\text{map}(f)(x))$$

Z równoważności, mając "flatMap" możemy otrzymać "map" i "join":

$$\text{map}(f)(x) = \text{flatMap}(\text{unit} \circ f)(x)$$

$$\text{join}(m) = \text{flatMap}(\text{id})(m)$$

Prawa monad

Podobnie do funktorów, definicja monady musi zachowywać kilka praw.

$$\mathrm{map}(f) \circ \mathrm{unit} = \mathrm{unit} \circ f$$

Prawa monad

Podobnie do funktorów, definicja monady musi zachowywać kilka praw.

$$\mathrm{map}(f) \circ \mathrm{unit} = \mathrm{unit} \circ f$$

$$\mathrm{join} \circ \mathrm{unit} = \mathrm{join} \circ \mathrm{map}(\mathrm{unit}) = \mathrm{id}$$

Prawa monad

Podobnie do funktorów, definicja monady musi zachowywać kilka praw.

$$\mathrm{map}(f) \circ \mathrm{unit} = \mathrm{unit} \circ f$$

$$\mathrm{join} \circ \mathrm{unit} = \mathrm{join} \circ \mathrm{map}(\mathrm{unit}) = \mathrm{id}$$

$$\mathrm{join} \circ \mathrm{join} = \mathrm{join} \circ \mathrm{map}(\mathrm{join})$$

Prawa monad

Podobnie do funktorów, definicja monady musi zachowywać kilka praw.

$$\mathrm{map}(f) \circ \mathrm{unit} = \mathrm{unit} \circ f$$

$$\mathrm{join} \circ \mathrm{unit} = \mathrm{join} \circ \mathrm{map}(\mathrm{unit}) = \mathrm{id}$$

$$\mathrm{join} \circ \mathrm{join} = \mathrm{join} \circ \mathrm{map}(\mathrm{join})$$

$$\mathrm{join} \circ \mathrm{map}(\mathrm{map}(g)) = \mathrm{map}(g) \circ \mathrm{join}$$

Prawa monad

Podobnie do funktorów, definicja monady musi zachowywać kilka praw.

$$\mathrm{map}(f) \circ \mathrm{unit} = \mathrm{unit} \circ f$$

$$\mathrm{join} \circ \mathrm{unit} = \mathrm{join} \circ \mathrm{map}(\mathrm{unit}) = \mathrm{id}$$

$$\mathrm{join} \circ \mathrm{join} = \mathrm{join} \circ \mathrm{map}(\mathrm{join})$$

$$\mathrm{join} \circ \mathrm{map}(\mathrm{map}(g)) = \mathrm{map}(g) \circ \mathrm{join}$$

Analogiczny zbiór praw istnieje dla definicji z “flatMap” (wynika z powyższych z naszych definicji – pominiemy go dla przejrzystości).

Po co nam te prawa?

Prawa te wynikają w dużej mierze z definicji monady jako tego dziwnego obiektu z teorii kategorii. Okazuje się jednak, że to te własności czynią ten interfejs tak ogólnym.

Po co nam te prawa?

Prawa te wynikają w dużej mierze z definicji monady jako tego dziwnego obiektu z teorii kategorii. Okazuje się jednak, że to te własności czynią ten interfejs tak ogólnym.

Kluczowy okazuje się właśnie fakt, że te prawa indukują *monoid* – to jest, łączność (operacji join i map) i element neutralny (unit). Zastanówmy się, czyżby był jakiś **potężny** konstrukt, który wspiera operacje łącznego składania, ma element neutralny i przewija się wszędzie w programowaniu (nie tylko funkcyjnym)?

Po co nam te prawa?

Prawa te wynikają w dużej mierze z definicji monady jako tego dziwnego obiektu z teorii kategorii. Okazuje się jednak, że to te własności czynią ten interfejs tak ogólnym.

Kluczowy okazuje się właśnie fakt, że te prawa indukują *monoid* – to jest, łączność (operacji join i map) i element neutralny (unit). Zastanówmy się, czyżby był jakiś **potężny** konstrukt, który wspiera operacje łącznego składania, ma element neutralny i przewija się wszędzie w programowaniu (nie tylko funkcyjnym)?

Czym bowiem jest programowanie jak nie **składaniem funkcji**?

Po co nam te prawa?

Prawa te sprawiają, że monady z operacją flatMap zachowują się jak *składanie funkcji*, z dodatkowym *kontekstem* obliczeń.

Po co nam te prawa?

Prawa te sprawiają, że monady z operacją flatMap zachowują się jak *składanie funkcji*, z dodatkowym *kontekstem* obliczeń.

Dokładniej każdy monad indukuje operator złożenia postaci:

$$\textbf{>=>} : (\alpha \to M\langle\beta\rangle) \to (\beta \to M\langle\gamma\rangle) \to (\alpha \to M\langle\gamma\rangle)$$

$$f \textbf{ >=> } g = (x \Rightarrow \mathrm{flatMap}(g)(f(x)))$$

Prawa dla tej definicji są najbardziej intuicyjne: wynika z nich, że operator ten jest łączny, a "unit" jest jego elementem neutralnym (tak, jak identyczność nad zwykłymi funkcjami).

Podsumowanie definicji

Podsumowując, monadę możemy zdefiniować przez aż trzy równoważne definicje:

- **join**, **unit** i **map** – potrafimy skonstruować kontekst, łączyć konteksty i zaaplikować funkcję pod kontekstem
- **unit** i **flatMap** – potrafimy skonstruować kontekst i zaaplikować funkcję, jednocześnie łącząc konteksty
- **unit** i **>=>** – potrafimy operować na funkcjach z kontekstem jak na zwykłych funkcjach ze złożeniem i elementem neutralnym.

Podczas reszty wykładu będziemy trochę żonglować tymi definicjami.

Przykłady monad: Identity

Weźmy typ Identity$\langle T \rangle$ z jednym konstruktorem Ident(T), będącym "opakowaniem" wartości typu T. Zauważmy, że jest ona monadą:

Przykłady monad: Identity

Weźmy typ $\mathrm{Identity}\langle T \rangle$ z jednym konstruktorem $\mathrm{Ident}(T)$, będącym "opakowaniem" wartości typu T. Zauważmy, że jest ona monadą:

$$\mathrm{map} : (\alpha \to \beta) \to \mathrm{Identity}\langle \alpha \rangle \to \mathrm{Identity}\langle \beta \rangle$$

$$\mathrm{map}(f)(\mathrm{Ident}(x)) = \mathrm{Ident}(f(x))$$

Przykłady monad: Identity

Weźmy typ $\mathrm{Identity}\langle T \rangle$ z jednym konstruktorem $\mathrm{Ident}(T)$, będącym "opakowaniem" wartości typu T. Zauważmy, że jest ona monadą:

$$\mathrm{map} : (\alpha \to \beta) \to \mathrm{Identity}\langle \alpha \rangle \to \mathrm{Identity}\langle \beta \rangle$$

$$\mathrm{map}(f)(\mathrm{Ident}(x)) = \mathrm{Ident}(f(x))$$

$$\mathrm{unit} : \alpha \to \mathrm{Identity}\langle \alpha \rangle$$

$$\mathrm{unit}(x) = \mathrm{Ident}(x)$$

Przykłady monad: Identity

Weźmy typ $\text{Identity}\langle T \rangle$ z jednym konstruktorem $\text{Ident}(T)$, będącym "opakowaniem" wartości typu T. Zauważmy, że jest ona monadą:

$$\text{map} : (\alpha \to \beta) \to \text{Identity}\langle \alpha \rangle \to \text{Identity}\langle \beta \rangle$$

$$\text{map}(f)(\text{Ident}(x)) = \text{Ident}(f(x))$$

$$\text{unit} : \alpha \to \text{Identity}\langle \alpha \rangle$$

$$\text{unit}(x) = \text{Ident}(x)$$

$$\text{join} : \text{Identity}\langle \text{Identity}\langle \alpha \rangle \rangle \to \text{Identity}\langle \alpha \rangle$$

$$\text{join}(\text{Ident}(\text{Ident}(a))) = \text{Ident}(a)$$

Przykłady monad: Identity

Weźmy typ $\text{Identity}\langle T \rangle$ z jednym konstruktorem $\text{Ident}(T)$, będącym "opakowaniem" wartości typu T. Zauważmy, że jest ona monadą:

$$\text{unit} : \alpha \to \text{Identity}\langle \alpha \rangle$$

$$\text{unit}(x) = \text{Ident}(x)$$

$$\text{join} : \text{Identity}\langle \text{Identity}\langle \alpha \rangle \rangle \to \text{Identity}\langle \alpha \rangle$$

$$\text{join}(\text{Ident}(\text{Ident}(a))) = \text{Ident}(a)$$

$$\texttt{>=>} : (\alpha \to \text{Identity}\langle \beta \rangle) \to (\beta \to \text{Identity}\langle \gamma \rangle) \to (\alpha \to \text{Identity}\langle \gamma \rangle)$$

$$\texttt{>=>} \equiv \circ$$

Przykłady monad: Lista

Podane wcześniej funktory mają naturalne rozszerzenia do monady.

$$\text{unit}(x) = [x]$$

$$\text{join}([l_1, l_2, ...]) = l_1 + l_2 + ...$$

$$\text{join}([[a_1, a_2, ..., a_k], [b_1, b_2, ..., b_l], ...]) = [a_1, a_2, ..., a_k, b_1, b_2, ..., b_l, ...]$$

Przykłady monad: Lista

Podane wcześniej funktory mają naturalne rozszerzenia do monady.

$$\text{unit}(x) = [x]$$

$$\text{join}([l_1, l_2, ...]) = l_1 + l_2 + ...$$

$$\text{join}([[a_1, a_2, ..., a_k], [b_1, b_2, ..., b_l], ...]) = [a_1, a_2, ..., a_k, b_1, b_2, ..., b_l, ...]$$

$$\texttt{sqrt} : \mathbb{C} \to [\mathbb{C}]$$

$$(\text{flatMap}(\texttt{sqrt}) \circ \text{flatMap}(\texttt{sqrt}))(16) = [2, -2, 2i, -2i]$$

Przykłady monad: Lista

Podane wcześniej funktory mają naturalne rozszerzenia do monady.

$$\text{unit}(x) = [x]$$

$$\text{join}([l_1, l_2, ...]) = l_1 + l_2 + ...$$

$$\text{join}([[a_1, a_2, ..., a_k], [b_1, b_2, ..., b_l], ...]) = [a_1, a_2, ..., a_k, b_1, b_2, ..., b_l, ...]$$

$$\texttt{sqrt} : \mathbb{C} \to [\mathbb{C}]$$

$$(\text{flatMap}(\texttt{sqrt}) \circ \text{flatMap}(\texttt{sqrt}))(16) = [2, -2, 2i, -2i]$$

Koncepcyjnie jest to monad *wyboru* / *niederministycznych obliczeń*.

Przykłady monad: Option

Naturalna definicja i intepretacja dla `Option` – monady *braku/błędu*:

$$\mathrm{unit}(x) = \mathrm{Some}(x)$$

$$\mathrm{join}(x) = \begin{cases} \mathrm{Some}(v) & \text{gdy} \quad x = \mathrm{Some}(\mathrm{Some}(v)) \\ \mathrm{None} & \text{gdy} \quad x \in \{\mathrm{None}, \mathrm{Some}(\mathrm{None})\} \end{cases}$$

Przykłady monad: Option

Naturalna definicja i intepretacja dla `Option` – monady *braku/błędu*:

$$\text{unit}(x) = \text{Some}(x)$$

$$\text{join}(x) = \begin{cases} \text{Some}(v) & \text{gdy} \quad x = \text{Some}(\text{Some}(v)) \\ \text{None} & \text{gdy} \quad x \in \{\text{None}, \text{Some}(\text{None})\} \end{cases}$$

$$\texttt{sqrt} : \text{Int} \to \text{Option}\langle\text{Int}\rangle$$

$$\text{flatMap}(\text{sqrt})(\text{Some}(-2)) = \text{None}$$

$$(\text{flatMap}(\text{sqrt}) \circ \text{flatMap}(\text{sqrt}))(\text{Some}(16)) = \text{Some}(2)$$

Przykłady monad: Result

Analogiczną definicję możemy utworzyć dla `Result`, czyli kontekstu *błędu z wiadomością*. Owa monada zwraca wynik lub pierwszy napotkany błąd.

$$\mathrm{unit}(x) = \mathrm{Ok}(x)$$

$$\mathrm{join}(x) = \begin{cases} \mathrm{Err}(a) & \text{gdy } x = \mathrm{Err}(a) \\ \mathrm{Err}(b) & \text{gdy } x = \mathrm{Ok}(\mathrm{Err}(b)) \\ \mathrm{Ok}(v) & \text{gdy } x = \mathrm{Ok}(\mathrm{Ok}(v)) \end{cases}$$

Przykłady monad: Result

$$\mathrm{unit}(x) = \mathrm{Ok}(x)$$

$$\mathrm{join}(x) = \begin{cases} \mathrm{Err}(a) & \text{gdy } x = \mathrm{Err}(a) \\ \mathrm{Err}(b) & \text{gdy } x = \mathrm{Ok}(\mathrm{Err}(b)) \\ \mathrm{Ok}(v) & \text{gdy } x = \mathrm{Ok}(\mathrm{Ok}(v)) \end{cases}$$

$$\texttt{sqrt} : \mathrm{Int} \to \mathrm{Result}\langle \mathrm{String}, \mathrm{Int} \rangle$$

$$(\mathrm{flatMap}(\mathrm{sqrt}) \circ \mathrm{flatMap}(\mathrm{sqrt}))(-9) = \mathrm{Err}(\text{"negative argument"})$$

$$(\mathrm{flatMap}(\mathrm{sqrt}) \circ \mathrm{flatMap}(\mathrm{sqrt}))(9) = \mathrm{Err}(\text{"non-integer result"})$$

$$(\mathrm{flatMap}(\mathrm{sqrt}) \circ \mathrm{flatMap}(\mathrm{sqrt}))(81) = \mathrm{Ok}(3)$$

To co z tymi efektami?

Zdefiniowaliśmy jakieś śmieszne matematyczne obiekty matematyczne, mamy aż trzy sposoby patrzenia na monady, ale niekoniecznie jeszcze widać jak ma to nam umożliwić wykonywanie efektów w sposób czysto funkcyjny.

To co z tymi efektami?

Zdefiniowaliśmy jakieś śmieszne matematyczne obiekty matematyczne, mamy aż trzy sposoby patrzenia na monady, ale niekoniecznie jeszcze widać jak ma to nam umożliwić wykonywanie efektów w sposób czysto funkcyjny.

Wprowadzimy teraz kilka magicznych monad, które krok po kroku pokażą nam, jak możemy modelować operacje na zewnętrznym świecie (tj. oczekiwane efekty).

Reader

Weźmy typ funkcji postaci $R \to A$ dla ustalonego (stałego) R.
Spróbujemy zdefiniować sobie jakąś instancję monady dla tego typu.

Reader

Weźmy typ funkcji postaci $R \to A$ dla ustalonego (stałego) R.
Spróbujemy zdefiniować sobie jakąś instancję monady dla tego typu.

Definicję "map" możemy wywnioskować wprost z oczekiwanego typu:

$$\text{map} : (\alpha \to \beta) \to (R \to \alpha) \to (R \to \beta)$$

$$\text{map}(f)(r) = f \circ r$$

Reader

Weźmy typ funkcji postaci $R \to A$ dla ustalonego (stałego) R.
Spróbujemy zdefiniować sobie jakąś instancję monady dla tego typu.

Definicję "map" możemy wywnioskować wprost z oczekiwanego typu:

$$\text{map} : (\alpha \to \beta) \to (R \to \alpha) \to (R \to \beta)$$

$$\text{map}(f)(r) = f \circ r$$

Za "unit" przyjmiemy funkcję zwracającą funkcję stałą:

$$\text{unit}(a) = (x \Rightarrow a)$$

Reader

Brakuje nam jeszcze jednej z funkcji ze zbioru join, flatMap, **>=>**.

Reader

Brakuje nam jeszcze jednej z funkcji ze zbioru join, flatMap, **>=>**.

Rzućmy okiem na typ join

$$\text{join} : (R \to (R \to \alpha)) \to (R \to \alpha)$$

Reader

Brakuje nam jeszcze jednej z funkcji ze zbioru join, flatMap, **>=>**.

Rzućmy okiem na typ join

$$\text{join} : (R \to (R \to \alpha)) \to (R \to \alpha)$$

Po przypatrzeniu się co możemy zrobić, możemy wymyślić definicję:

$$\text{join}(f) = (r \Rightarrow f(r)(r))$$

Reader

Brakuje nam jeszcze jednej z funkcji ze zbioru join, flatMap, **>=>**.

Rzućmy okiem na typ join

$$\text{join} : (R \to (R \to \alpha)) \to (R \to \alpha)$$

Po przypatrzeniu się co możemy zrobić, możemy wymyślić definicję:

$$\text{join}(f) = (r \Rightarrow f(r)(r))$$

Rozszerzając definicję rzućmy też okiem na flatMap:

$$\text{flatMap}(m)(f) = (r \Rightarrow m(f(r))(r))$$

O co więc chodzi w tej konstrukcji?

Reader

Monada $\mathrm{Reader}\langle R, A\rangle \equiv (R \to A)$ reprezentuje obliczenie, które korzysta (na wielu krokach) z pewnej podanej "konfiguracji" R.

Łączenie operacji przekazuje ją dalej, żeby każde z obliczeń mogło ją wykorzystać.

Reader

Monada $\mathrm{Reader}\langle R, A\rangle \equiv (R \to A)$ reprezentuje obliczenie, które korzysta (na wielu krokach) z pewnej podanej "konfiguracji" R.

Łączenie operacji przekazuje ją dalej, żeby każde z obliczeń mogło ją wykorzystać.

Aby dostać dostęp do konfiguracji, możemy użyć fajnego tricku korzystając z funkcji id:

$$\mathrm{ask} : \mathrm{Reader}\langle R, R\rangle \equiv (R \to R)$$

Reader

$$\text{ask} : \text{Reader}\langle R, R\rangle \equiv (R \to R)$$

W ten sposób unikamy zagłębiania się w definicję monady, a operujemy jedynie w świecie abstrakcyjnych operacji.

Obliczenia nie korzystające z R możemy opakować wprost składając z "unit". Natomiast obliczenia zależne $f : R \to A$ opakowujemy przez:

$$\text{map}(f)(\text{ask}) : \text{Reader}\langle R, A\rangle$$

W teorii f już samo w sobie jest Readerem – w praktyce typ funkcji zwykle jest ukryty w ekstra opakowaniu jak w Identity, stąd wynika kontrukcja.

Składnia “do”

To jest chyba dobry moment, aby wprowadzić uproszczoną składnię “do”, występującą w wielu językach programowania funkcyjnego i znaną też jako “for comprehension”, “computation expressions”, etc.

Składnia “do”

$$\text{do} \begin{cases} x_1 \leftarrow e_1 \\ x_2 \leftarrow e_2 \\ \ldots \\ x_n \leftarrow e_n \\ e_{n+1} \end{cases}$$

Taką składnię tłumaczymy na:

$$\text{flatMap}(x_1 \Rightarrow \text{flatMap}(x_2 \Rightarrow \ldots \Rightarrow \text{flatMap}(x_n \Rightarrow e_{n+1})(e_n)\ \ldots)(e_2))(e_1))$$

Składnia “do”

$$\text{flatMap}(x_1 \Rightarrow \text{flatMap}(x_2 \Rightarrow ... \Rightarrow \text{flatMap}(x_n \Rightarrow e_{n+1})(e_n)\ ...)(e_2))(e_1))$$

Definiując rekurencyjnie:

$$\text{do}\ \{\text{expr} \equiv \text{expr}$$

$$\text{do}\begin{cases}\text{val} \leftarrow \text{expr}\\ \text{rest}\\ ...\end{cases} \equiv \text{flatMap}\left(\text{val} \Rightarrow \left(\text{do}\begin{cases}\text{rest}\\ ...\end{cases}\right)\right)(\text{expr})$$

Składnia “do”

$$\text{flatMap}(x_1 \Rightarrow \text{flatMap}(x_2 \Rightarrow ... \Rightarrow \text{flatMap}(x_n \Rightarrow e_{n+1})(e_n)\ ...)(e_2))(e_1))$$

Definiując rekurencyjnie:

$$\text{do}\ \{\text{expr} \equiv \text{expr}$$

$$\text{do}\begin{cases}\text{val} \leftarrow \text{expr}\\ \text{rest}\\ ...\end{cases} \equiv \text{flatMap}\left(\text{val} \Rightarrow \left(\text{do}\begin{cases}\text{rest}\\ ...\end{cases}\right)\right)(\text{expr})$$

O co w tym biega?

Składnia “do”

Składnia “do” daje nam wygodny sposób na podnoszenie klasycznych obliczeń do kontekstu *dowolnej* monady, bez korzystania bezpośrednio z operatorów join/flatMap/**>=>**.

Składnia “do”

Składnia “do” daje nam wygodny sposób na podnoszenie klasycznych obliczeń do kontekstu *dowolnej* monady, bez korzystania bezpośrednio z operatorów join/flatMap/**>=>**.

Zauważmy, że sam interfejs monady nie daje nam żadnego sposobu, aby wyciągnąć wartość z kontekstu – sposób jak wykorzystać monadę wynika z dokładnej struktury typu.

Składnia “do”

Składnia “do” daje nam wygodny sposób na podnoszenie klasycznych obliczeń do kontekstu *dowolnej* monady, bez korzystania bezpośrednio z operatorów join/flatMap/**>=>**.

Zauważmy, że sam interfejs monady nie daje nam żadnego sposobu, aby wyciągnąć wartość z kontekstu – sposób jak wykorzystać monadę wynika z dokładnej struktury typu.

Korzystając ze składni “do” zapominamy o tym problemie – wynika to z faktu, że z każdego kontekstu możemy wyciągnąć wartość danego typu do dalszych obliczeń *o ile zostajemy w jej kontekście.*

Przykłady składni “do”

$$\text{do} \begin{cases} a \leftarrow [1,2,3] \\ b \leftarrow [1,2,3] \\ \text{unit } (a \cdot b) \end{cases} = [1,2,3,2,4,6,3,6,9]$$

Przykłady składni “do”

$$\text{do} \begin{cases} a \leftarrow [1,2,3] \\ b \leftarrow [1,2,3] \\ \text{unit } (a \cdot b) \end{cases} = [1,2,3,2,4,6,3,6,9]$$

$$\text{do} \begin{cases} \text{config} \leftarrow \text{ask} \\ \text{unit config.time} \end{cases} \equiv \text{flatMap}(\text{config} \Rightarrow \text{unit config.time})(\text{ask})$$

$$\equiv \text{map}(\text{config} \Rightarrow \text{config.time})(\text{ask}) \Rightarrow \text{Reader}(\text{config} \Rightarrow \text{config.time})$$

Przykłady składni “do”

$$\text{do} \begin{cases} a \leftarrow [1,2,3] \\ b \leftarrow [1,2,3] \\ \text{unit } (a \cdot b) \end{cases} = [1,2,3,2,4,6,3,6,9]$$

$$\text{do} \begin{cases} \text{config} \leftarrow \text{ask} \\ \text{unit config.time} \end{cases} \equiv \text{flatMap}(\text{config} \Rightarrow \text{unit config.time})(\text{ask})$$

$$\equiv \text{map}(\text{config} \Rightarrow \text{config.time})(\text{ask}) \Rightarrow \text{Reader}(\text{config} \Rightarrow \text{config.time})$$

$$\text{do} \begin{cases} \text{time} \leftarrow \text{getTime} \\ \text{unit } (\text{time} + 10) \end{cases} \equiv (r \Rightarrow (r.\text{time}) + 10)$$

Writer

Umiemy czytać – a co z pisaniem?

Typ $W\langle R, A\rangle = A \to R$ nie jest funktorem w naszym znaczeniu słowa (jest on *kontrawariantny* w typie A).

Innymi słowy, nie ma sensownej definicji dla
$\mathrm{map}_W : (\alpha \to \beta) \to (\alpha \to R) \to (\beta \to R)$

ale za to jest naturalna definicja dla
$\mathrm{contramap}_W : (\beta \to \alpha) \to (\alpha \to R) \to (\beta \to R)$

$$\mathrm{contramap}(f)(g) = g \circ f$$

Writer

Umiemy czytać – a co z pisaniem?

Zamiast tego weźmiemy typ $\mathrm{Writer}\langle S, A\rangle \equiv (S, A)$.
Wtedy instancja funktora przychodzi bardzo naturalnie:

$$\mathrm{map} : (\alpha \to \beta) \to (S, \alpha) \to (S, \beta)$$

$$\mathrm{map}(f)((s, a)) = (s, f(a))$$

Writer

Umiemy czytać – a co z pisaniem?

Zamiast tego weźmiemy typ $\mathrm{Writer}\langle S, A\rangle \equiv (S, A)$.
Wtedy instancja funktora przychodzi bardzo naturalnie:

$$\mathrm{map} : (\alpha \to \beta) \to (S, \alpha) \to (S, \beta)$$

$$\mathrm{map}(f)((s, a)) = (s, f(a))$$

Problem dostajemy, gdy próbujemy definiować unit i join.

W związku z tym dorzucimy dodatkowy warunek – będziemy chceli, aby nasz typ S był **monoidem**.

Writer a Monoid

Monoidem nazywamy typ S wraz z dwiema wartościami:

$$\mathbb{E} : S$$
$$\diamond : S \to S \to S$$

Gdzie $\diamond$ jest łączny: $(a \diamond b) \diamond c = a \diamond (b \diamond c)$, a $\mathbb{E}$ to jego element neutralny: $a \diamond \mathbb{E} = \mathbb{E} \diamond a = a$. Przykłady monoidów to:

Lista z $\diamond = +, \mathbb{E} = []$ String z $\diamond = +, \mathbb{E} =$ ""

Int z dodawaniem i $\mathbb{E} = 0$ Int z mnożeniem i $\mathbb{E} = 1$

Writer

Wracając do Writer'a, mając założenie, że S to monoid możemy zdefiniować monadę:

$$\text{unit}(x) = (\mathbb{E}, s)$$

$$\text{join}((s_1, (s_2, a))) = (s_1 \diamond s_2, a)$$

Własności łączności monoidu przenoszą się w ten sposób na łączność operacji naszej monady.

Writer

Wracając do Writer'a, mając założenie, że S to monoid możemy zdefiniować monadę:

$$\text{unit}(x) = (\mathbb{E}, s)$$

$$\text{join}((s_1, (s_2, a))) = (s_1 \diamond s_2, a)$$

Własności łączności monoidu przenoszą się w ten sposób na łączność operacji naszej monady.

Po co to nam? Otóż możemy interpretować `Writer` m.in jako kontekst *logów / strumienia* obliczeń.

State

Przejdźmy teraz jeszcze do jednej z najpotężniejszych monad, czyli monady stanu.

State

Przejdźmy teraz jeszcze do jednej z najpotężniejszych monad, czyli monady stanu.

Najpotężniejsza jest monada kontynuacji, która może symulować każdą inną monadę. Brakuje nam czasu, więc nie opowiemy o niej tym razem.

State

Przejdźmy teraz jeszcze do jednej z najpotężniejszych monad, czyli monady stanu.

Najpotężniejsza jest monada kontynuacji, która może symulować każdą inną monadę. Brakuje nam czasu, więc nie opowiemy o niej tym razem.

Definiujemy typ State $\langle S, A \rangle \equiv (S \to (A, S))$. Koncepcyjnie, monada ta będzie kontekstem nad pewnym *stanem* typu S, który odpowiednio możemy modyfikować (i czytać, i pisać).

Monada stanu jest więc uogólnieniem monady `Reader` i `Writer`.

State

Jak zdefiniować operacje dla `State`?

Podobnie jak wcześniej, definicja funktora jest bardzo naturalna.

$$\text{map} : (\alpha \to \beta) \to (S \to (\alpha, S)) \to (S \to (\beta, S))$$

$$\text{map}(f)(g) = ((a, s) \Rightarrow (g(a), s)) \circ g$$

Definicja dla "unit" też przychodzi w miarę naturalnie – chcemy zwrócić operator, który zwraca wartość nie zmieniając stanu:

$$\text{unit} : \alpha \to (S \to (\alpha, S))$$

$$\text{unit}(x) = (s \Rightarrow (x, s))$$

State

$$\text{map}(f)(g) = ((a, s) \Rightarrow (f(a), s)) \circ g$$

$$\text{unit}(x) = (s \Rightarrow (x, s))$$

Zostaje nam operacja złożenia.

$$\text{join} : (S \to (S \to (\alpha, S), S)) \to (S \to (\alpha, S))$$

$$\text{join}(f) = (s \Rightarrow \text{let } (g, s') = f(s) \text{ in } g(s'))$$

State

$$\text{map}(f)(g) = ((a, s) \Rightarrow (f(a), s)) \circ g$$

$$\text{unit}(x) = (s \Rightarrow (x, s))$$

Zostaje nam operacja złożenia.

$$\text{join} : (S \to (S \to (\alpha, S), S)) \to (S \to (\alpha, S))$$

$$\text{join}(f) = (s \Rightarrow \text{let } (g, s') = f(s) \text{ in } g(s'))$$

Być może lepiej widać co się dzieje w definicji przez “flatMap”:

$$\text{flatMap} : (\alpha \to (S \to (\beta, S))) \to (S \to (\alpha, S)) \to (S \to (\beta, S))$$

$$\text{flatMap}(g)(h) = (s \Rightarrow \text{let } (a, s') = h(s) \text{ in } g(a)(s'))$$

State

Aby operować na stanie zdefiniujemy dwie funkcje pomocnicze:

$$\text{get} : \text{State}\ \langle \beta, \beta \rangle$$

$$\text{get} = (s \Rightarrow (s, s))$$

$$\text{put} : \beta \to \text{State}\ \langle \beta, \mathbb{1} \rangle$$

$$\text{put}(x) = (s \Rightarrow (\mathbb{1}, x))$$

State

Aby operować na stanie zdefiniujemy dwie funkcje pomocnicze:

$$\text{get} : \text{State}\ \langle \beta, \beta \rangle$$

$$\text{get} = (s \Rightarrow (s, s))$$

$$\text{put} : \beta \to \text{State}\ \langle \beta, \mathbb{1} \rangle$$

$$\text{put}(x) = (s \Rightarrow (\mathbb{1}, x))$$

Teraz, jak w poprzednich przypadkach, możemy operować na stanie nie wychodząc nigdy z poziomu abstrakcji oferowanego przez monadę.

$$\text{incrState} = \text{do} \begin{cases} s \leftarrow \text{get} \\ \text{put}\ (s{+}1) \end{cases}$$

IO

Dochodzimy do gwoździa programu: monady `IO`, która – jak nazwa sugeruje – pozwoli nam na wykonywanie operacji Input/Output.

IO

Dochodzimy do gwoździa programu: monady `IO`, która – jak nazwa sugeruje – pozwoli nam na wykonywanie operacji Input/Output.

Definicja tej monady jest wręcz uroczo prosta:

$$\mathrm{IO}\langle A\rangle = \mathrm{State}\ \langle \mathrm{RealWorld}, A\rangle \equiv (\mathrm{RealWorld}) \to (A, \mathrm{RealWorld})$$

IO

Dochodzimy do gwoździa programu: monady `IO`, która – jak nazwa sugeruje – pozwoli nam na wykonywanie operacji Input/Output.

Definicja tej monady jest wręcz uroczo prosta:

$$\mathrm{IO}\langle A\rangle = \mathrm{State}\ \langle \mathrm{RealWorld}, A\rangle \equiv (\mathrm{RealWorld}) \to (A, \mathrm{RealWorld})$$

Definicję i działanie `State` widzieliśmy już wcześniej. Warto zaznaczyć, że oczywiście nie możemy trzymać stanu całego świata poza nami.

W praktyce w IO możemy opakować dowolną procedurę, która niekoniecznie zachowuje przejrzystość referencyjną.

Przykład IO w Haskellu

```haskell
getLine :: IO String
putStrLn :: String -> IO ()
main :: IO ()
main = do
  putStrLn "Hello. What is your name?"
  name <- getLine
  putStrLn ("Hello there " <> name <> "!")
```

```
$ runghc Hello.hs
Hello! What is your name?
Dude
Hello there, Dude!
```

Więcej o monadach

Oczywiście monady nie kończą się na `IO`!

Jest wiele ciekawych monad i rozszerzeń ich idei, które pozwalają na jeszcze ciekawsze i mocniejsze modele obliczeń – w szczególności **transformery monad** pozwalają na łączenie różnych efektów.

Niestety, ponieważ (w momencie pisania tego) wykład ma już 60 slajdów, nie ma szans aby zagłębić się w nie dokładniej.

Wracając do samych efektów

Na koniec wspomnimy jeszcze o dwóch alternatywnych do monadów sposobach modelowania efektów, z zachowaniem przejrzystości referencyjnej.

Typy unikalne

Idea: zamiast opakowywać stan całego świata do monady IO, przekazujemy część stanu jako argument do funkcji.

Aby uniknąć problemu z przejrzystością dodajemy dodatkowe ograniczenia: typy wyrażeń (i argumentów) mogą zostać dodatkowo oznaczone jako **unikalne**.

Typy unikalne narzucają dodatkowy warunek dotyczący ich wykorzystania – mogą zostać przekazane do innej funkcji *maksymalnie raz*. W szczególności, funkcja przyjmująca taki argument wie, że jest to jedyna dostępna kopia tego unikalnego obiektu w całym programie.

Typy unikalne

Najlepiej chyba zobaczyć na przykładzie, o co chodzi.

```
File* OpenFile(String path);
File* WriteFile(File* file, String data);
void CloseFile(File* file);

// OK
f = OpenFile("/dev/tty0");
g = WriteFile(f, "Hello, terminal user!")
CloseFile(g)
```

Typy unikalne

```
// ERROR: UNIQUENESS VIOLATED
f = OpenFile("/dev/tty0")
WriteFile(f, "Hello, friend!")
CloseFile(f)
```

Chyba Jedynym jakkolwiek-znanym językiem z takimi typami to Clean.

Typy unikalne zezwalają też na modelowanie ciekawszych interakcji np. efektywne implementacje obiektów o zmiennym stanie – możemy utrzymywać gdzieś jakąś strukturę, a operować na niej jedynie przez *unikalne* pointery do tej struktury, zachowując przejrzystość.

Efekty algebraiczne

Względnie nową alternatywę oferują tzw. efekty algebraiczne i ich "handlery".

Po krótce, idea handlerów jest naturalnym rozszerzeniem operacji `throw` i `catch` obecnych w wielu językach programowania.

W dużym uproszczeniu, definiujemy funkcje:

$$\text{perform}_E : X \to Y$$

$$\text{handle}_E : (X, (Y \to Z)) \to Z$$

$$\text{return}_E : R \to Z$$

Efekty algebraiczne

```
perform (Print "Hello, ") ;;

handle
    perform (Print "World!")
with
| effect (Print msg) k ->
    perform (Print ("you got trolled, I will print this
message instead!\n"))
;;
```

Efekty algebraiczne

```
perform (Print "Hello, ") ;;

handle
    perform (Print "World!")
with
| effect (Print msg) k ->
    perform (Print ("you got trolled, I will print this
message instead!\n"))
;;
```

```
> Hello, you got trolled, I will print this message instead!
```

Efekty algebraiczne

Efekty algebraiczne wpierw zostały zaimplementowane w Haskellu, jako lekki hack na możliwościach jego systemu typów – obecnie istnieje kilka języków, które mają pierwszorzędne wsparcie dla nich, w tym język badawczy “Eff” (pokazany wcześniej) oraz komercyjny “Unison”.

Efekty algebraiczne

Efekty algebraiczne wpierw zostały zaimplementowane w Haskellu, jako lekki hack na możliwościach jego systemu typów – obecnie istnieje kilka języków, które mają pierwszorzędne wsparcie dla nich, w tym język badawczy “Eff” (pokazany wcześniej) oraz komercyjny “Unison”.

Zaletami efektów są m.in. łatwość podmienienia działania danego efektu (wystarczy nam użyć innego handlera), co zastępuje metodę “dependency injection” często używaną w programowaniu obiektowym oraz usunięcie problemu “quadratic instances”, który występuje przy transformerach monad (o których niestety nie zdążyliśmy powiedzieć).

Dziękuję za uwagę!