

Wstęp do Programowania potok funkcyjny

Marcin Kubica

2016/2017

Outline

- 1 Moduły i bariery abstrakcji
 - Moduły = Struktury + Sygnatury
 - Jak wyodrębniać moduły?
 - Przykład: Pakiet liczb wymiernych

Moduły — co to jest i po co to jest?

- Duży system dzielimy na mniejsze, łatwiejsze do ogarnięcia składowe — *moduły*.
- Zależności między modułami powinny być ograniczone do minimum.
- System może być zbyt złożony, żeby ogarnąć go całego. Możemy jednak ogarnąć jeden moduł i to od czego on zleży.
- *Interfejs* modułu, to zestaw pojęć programistycznych, które moduł realizuje.
- Jedne moduły mogą korzystać z pojęć zaimplementowanych przez inne.
- Moduł to fragment programu polegający na wykonaniu wyodrębnionego zadania programistycznego.
- Sformułowaniu tego zadania programistycznego towarzyszy (mniej lub bardziej formalna) specyfikacja pojęć implementowanych przez moduł.

Niezależność modułów

- Po podzieleniu programu na moduły i wyspecyfikowaniu interfejsów, można je niezależnie implementować.
- W Ocamlu osobno definiujemy interfejsy i implementacje modułów.
Interfejsy nazywamy *sygnaturami*, a implementacje *strukturami*.
- Sygnatury i struktury są bytami niezależnymi.
Można je niezależnie kompilować.
- Moduły można niezależnie kompilować.
Wszystkie informacje konieczne do skompilowania modułu są zawarte w interfejsach wykorzystywanych przez niego modułów.

Moduły na poziomie plików

Definition (Struktura)

TODO:

struct ... end .

- Implementacja (struktura): definicje umieszczone w pliku *.ml
- Interfejs (sygnatura): deklaracje umieszczone w pliku *.mli

Struktury

Definition (Struktura)

Struktura to nazwany zestaw definicji pojęć otoczonych słowami struct ... end .

$$\begin{aligned}\langle \text{definicja} \rangle &::= \text{module } \langle \text{Identyfikator} \rangle \equiv \langle \text{struktura} \rangle \\ \langle \text{struktura} \rangle &::= \text{struct } \{ \langle \text{definicja} \rangle \}^* \text{end}\end{aligned}$$

Wszystkie zdefiniowane wewnątrz pojęcia są w pełni widoczne.

Struktury

Definition (Nazwy kwalifikowane)

Do wnętrza struktury możemy dostać się stosując nazwy kwalifikowane:

$$\langle \text{identyfikator} \rangle ::= \langle \text{Identyfikator} \rangle . \langle \text{identyfikator} \rangle$$

Można też „otworzyć” strukturę — wyłuskać z niej wszystkie udostępniane przez nią pojęcia tak, aby były dostępne bez kwalifikowania:

$$\langle \text{jednostka kompilacji} \rangle ::= \underline{\text{open}} \langle \text{Identyfikator} \rangle$$

Struktury

Example

```
module Modulik =  
  struct  
    type typik = int list  
    let lista = [2; 3; 7]  
    let rec prod l =  
      if l = [] then 1 else hd l * prod (tl l)  
    end;;  
  
  Modulik.prod Modulik.lista;;  
  
  open Modulik;;  
  prod lista;;  
  
module Pusty = struct end;;
```


Struktury

Example

Struktury mogą zawierać wewnątrz struktury lokalne.
(Sensowne zastosowania zobaczymy później.)

```
module M =  
  struct  
    module A =  
      struct  
        let a = 27  
      end  
    module B =  
      struct  
        let b = 15  
      end  
  end;;  
M.A.a + M.B.b;;
```

Struktury

Definition

Struktury można rozszerzać.

Definiując jedną strukturę można „wciągnąć” do niej zawartość innej struktury.

$$\langle \text{definicja} \rangle ::= \text{include } \langle \text{Identyfikator} \rangle$$

Example

```
module Mod =  
  struct  
    include Modulik  
    let n = List.length lista  
  end;;
```

Struktury

Definition

Struktury można rozszerzać.

Definiując jedną strukturę można „wciągnąć” do niej zawartość innej struktury.

$$\langle \text{definicja} \rangle ::= \quad \text{include} \quad \langle \text{Identyfikator} \rangle$$

Example

```
module Mod =  
  struct  
    include Modulik  
    let n = List.length lista  
  end;;
```

Sygnatury

- Sygnatura określa, które elementy struktury mają być widoczne na zewnątrz.
- Wszystko czego nie widać w sygnaturze jest ukryte.
- Sygnatura może zawierać **deklaracje**:
 - wartości, z podaniem typu wartości,
 - typu wraz z jego definicją,
 - typu abstrakcyjnego (bez podania definicji),
 - sygnatury lokalnej
 - wyjątków.

Sygnatury

Definition

Składnia definicji sygnatur:

```
⟨definicja⟩ ::= module type ⟨Identyfikator⟩ = ⟨sygnatura⟩  
⟨sygnatura⟩ ::= sig { ⟨deklaracja⟩ } * end  
⟨deklaracja⟩ ::= type { ⟨parametr typowy⟩ } * ⟨identyfikator⟩  
                                                         [ = ⟨typ⟩ ] |  
               val ⟨identyfikator⟩ : ⟨typ⟩ |  
               module type ⟨Identyfikator⟩ = ⟨sygnatura⟩ |  
               exception ⟨wariant⟩
```

Sygnatury

Example

Przykład sygnatury:

```
module type S =  
  sig  
    type abstrakcyjny  
    type konkretny = int * float  
    val x : abstrakcyjny * konkretny  
    module type Pusty = sig end  
    exception Wyjatek of abstrakcyjny  
  end;;
```

Sygnatury

Example

Sygnatura kolejek FIFO:

```
module type FIFO =  
  sig  
    exception EmptyQueue  
    type 'a queue  
    val empty : 'a queue  
    val insert : 'a queue -> 'a -> 'a queue  
    val front : 'a queue -> 'a  
    val remove : 'a queue -> 'a queue  
  end;;
```

Sygnatury

Definition

Sygnatury można rozszerzać.

Definiując jedną sygnaturę można „wciągnąć” do niej zawartość innej sygnatury.

$$\langle \text{deklaracja} \rangle ::= \text{include } \langle \text{Identyfikator} \rangle$$

Sygnatury

Example

Sygnatura kolejek dwustronnych:

```
module type QUEUE =  
  sig  
    include FIFO  
    val back : 'a queue -> 'a  
    val insert_front : 'a queue -> 'a -> 'a queue  
    val remove_back : 'a queue -> 'a queue  
  end;;
```

Sygnatury

Definition

Sygnatury można ukonkretniać.

Jeżeli sygnatura zawiera typ abstrakcyjny, to można podać jaka ma być jego implementacja.

$$\langle \text{sygnatura} \rangle ::= \langle \text{sygnatura} \rangle \text{ with } \underline{\text{type}} \\ \{ \langle \text{parametr typowy} \rangle \}^* \langle \text{identyfikator} \rangle \equiv \langle \text{typ} \rangle \\ | \dots$$

Example

```
module type LIST = FIFO with type 'a queue = 'a list;;
```

Łączenie struktur i sygnatur

Definition

Podając sygnaturę dla struktury ograniczamy wgląd do jej środka. Można to zrobić na kilka sposobów:

```

$$\langle \text{definicja} \rangle ::= \text{module } \langle \text{Identyfikator} \rangle [ \_ : \langle \text{sygnatura} \rangle ] \equiv \langle \text{struktura} \rangle$$

$$\langle \text{struktura} \rangle ::= \langle \text{struktura} \rangle \_ : \langle \text{sygnatura} \rangle$$

$$\langle \text{struktura} \rangle ::= \langle \text{Identyfikator} \rangle \mid \dots$$

$$\langle \text{sygnatura} \rangle ::= \langle \text{Identyfikator} \rangle \mid \dots$$

```

Łączenie struktur i sygnatur

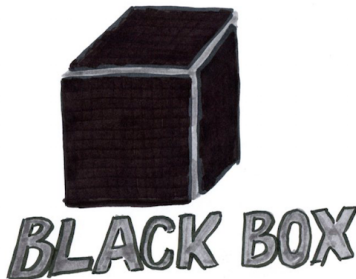
Example

Różne sposoby łączenia struktury i sygnatury:

```
module Fifo_implementation =  
  struct  
    exception EmptyQueue  
    type 'a queue = 'a list  
    let empty = []  
    let insert q x = q @ [x]  
    let front q = ...  
    let remove q = ...  
  end;;  
  
module Fifo : FIFO = Fifo_implementation;;  
module Fifo = (Fifo_implementation : FIFO);;  
module Fifo : sig ... end = struct ...end;;
```

Moduły — zasady tworzenia

- *Black-box approach* — moduł ma charakter czarnej skrzynki, na zewnątrz widoczne jest tylko to, co tworzy interfejs.
- *Information hiding* — sposób implementacji elementów interfejsu, jak i ew. pojęcia pomocnicze są ukryte wewnątrz modułu.



Moduły — zasady tworzenia

- *Separation of concerns* — specyfikacja nie powinna odwoływać się do sposobu implementacji modułu, a jedynie do takich jego właściwości, które może zaobserwować użytkownik. Podobnie, użytkownik nie powinien zakładać nic, co nie wynika ze specyfikacji.
- *Sekret modułu* — to coś, czego sposób implementacji jest ukryty wewnątrz modułu.
- Często sekretem modułu jest struktura danych wraz z operacjami na niej.

Moduły — zasady tworzenia

- Jak sprawdzić czy podział na moduły jest właściwy?
- Eksperyment myślowy:
 - potencjalne zmiany w programie,
 - jakich modułów dotyczy każda zmiana,
 - bardziej lokalne zmiany, to lepszy podział na moduły.

Wartości abstrakcyjne i konkretne

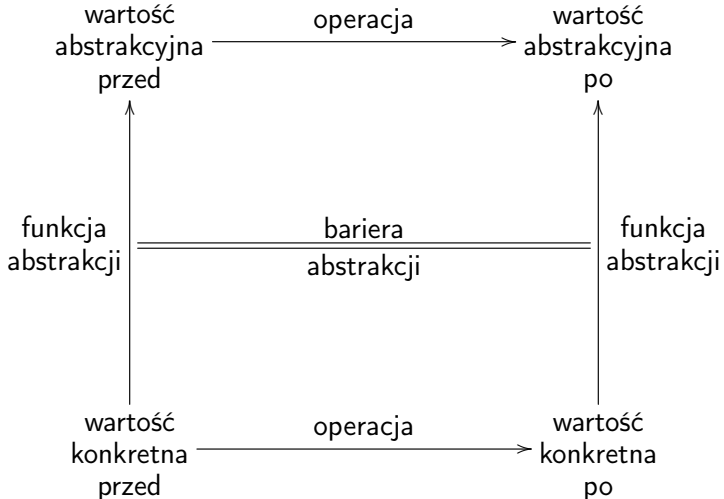


Wartości abstrakcyjne i konkretne

Tworząc strukturę danych musimy podjąć kilka decyzji projektowych:

- interfejs — operacje udostępniane użytkownikowi struktury danych,
- wartości abstrakcyjne — zbiór wartości, które chcemy reprezentować,
- wartości konkretne — struktura danych odpowiednia do reprezentowania wartości abstrakcyjnych:
 - typ danych,
 - niezmiennik struktury danych — wartości, które są „poprawne”,
- funkcja abstrakcji — przekształca wartości konkretne w odpowiadające wartości abstrakcyjne.

Wartości abstrakcyjne i konkretne



Konstruktory, selektory, modyfikatory

- Operacje tworzące interfejs możemy podzielić na trzy kategorie:
 - konstruktory — tworzą złożone wartości z prostszych,
 - selektory — wyłuskują elementy lub badają cechy złożonych wartości,
 - modyfikatory — przekształcają złożone wartości.
- Dodatkowa bariera abstrakcji — odpowiednio dobrane konstruktory i selektory,
- Implementacja modyfikatorów nie zależy od implementacji struktury danych.
- Kryterium lokalności zmian — im mniejsze fragmenty kodu są wrażliwe na potencjalne zmiany, tym lepiej.

Bariery abstrakcji

Bariery abstrakcji oddzielają różne poziomy abstrakcji:

programy korzystające ze
złożonej struktury danych

implementacja modyfikatorów

implementacja struktury danych

realizacja typów w języku programowania i
wykorzystywane struktury danych

modyfikatory

konstruktory i selektory

prostsze typy i
operacje na nich

Przykład: Pakiet liczb wymiernych

Example

```
module type ULAMKI =  
  sig  
    type t  
    (* typ abstrakcyjny reprezentujący ułamki *)  
  
    val ulamek : int -> int -> t  
    (* konstruktor, ulamek / m tworzy ułamek  $\frac{l}{m}$   
       przy założeniu, że  $m \neq 0$  *)  
  
    val licznik : t -> int  
    (* selektor, zwraca licznik ułamka *)  
  
    val mianownik : t -> int  
    (* selektor, zwraca mianownik ułamka *)  
  end;;
```

Przykład: Pakiet liczb wymiernych

Example

Specyfikacja:

$$\forall_{l,m \in \mathbb{N}, m \neq 0} \quad \frac{\text{licznik (ulamek l m)}}{\text{mianownik (ulamek l m)}} = \frac{l}{m}$$

Implementację struktury danych odkładamy na chwilę

```
module Ulamki : ULAMKI = struct
  :
end
```

Przykład: Pakiet liczb wymiernych

Example

```
module type RAT =  
  sig  
    include ULAMKI  
    val rowne : t -> t -> bool      (* porównanie *)  
    val plus : t -> t -> t          (* suma *)  
    val minus : t -> t -> t         (* różnica *)  
    val razy : t -> t -> t          (* iloczyn *)  
    val podziel : t -> t -> t       (* iloraz *)  
  end;;
```

Przykład: Pakiet liczb wymiernych

Example

Modyfikatory i porównywanie możemy zaimplementować korzystając z następujących tożsamości:

$$\frac{l_1}{m_1} + \frac{l_2}{m_2} = \frac{l_1 m_2 + l_2 m_1}{m_1 m_2} \quad \frac{l_1}{m_1} - \frac{l_2}{m_2} = \frac{l_1 m_2 - l_2 m_1}{m_1 m_2}$$

$$\frac{l_1}{m_1} \cdot \frac{l_2}{m_2} = \frac{l_1 l_2}{m_1 m_2} \quad \frac{l_1/m_1}{l_2/m_2} = \frac{l_1 m_2}{l_2 m_1} \quad \text{dla } l_2 \neq 0$$

$$\frac{l_1}{m_1} = \frac{l_2}{m_2} \Leftrightarrow l_1 \cdot m_2 = l_2 \cdot m_1 \quad \text{dla } m_1 \neq 0 \text{ i } m_2 \neq 0$$

Przykład: Pakiet liczb wymiernych

Example

Implementacja modyfikatorów:

```
module Rat : RAT = struct
  include Ułamki
  let plus x y = ...
  let minus x y = ...
  let razy x y = ...
  let podziel x y = ...
  let rowne x y = ...
end;;
```

Nie zależy od implementacji struktury danych.

Przykład: Pakiet liczb wymiernych

Example

Implementacja struktury danych:

```
module Ułamki : ULAMKI = struct
  type t = int * int
  let ulamek l m = (l, m)
  let licznik (l, _) = l
  let mianownik (_, m) = m
end;;
```

- Niezmiennik struktury danych (l, m) : $m \neq 0$,
- Funkcja abstrakcji $f(l, m) = \frac{l}{m}$.

Przykład: Pakiet liczb wymiernych

Example

Zmiana reprezentacji liczb wymiernych:

- Ułamki pamiętamy w postaci skróconych par (licznik, mianownik),
- Nie wymagamy normalizacji znaku (mianownika),
- Niezmiennik danych (l, m) : $m \neq 0 \wedge \text{nwd}(l, m) = 1$,
- Funkcja abstrakcji — bez zmian (nadal nie jest różnowartościowa).

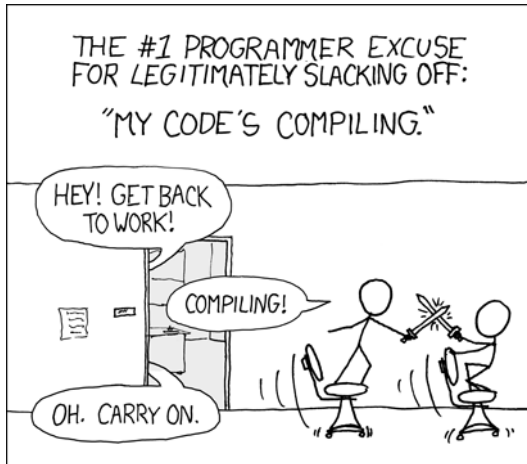
Przykład: Pakiet liczb wymiernych

Example

```
module Ułamki : ULAMKI = struct
  type t = int * int
  let ulamek l m =
    let r = nwd l m
    in ((l / r), (m / r))
  let licznik (l, _) = l
  let mianownik (_, m) = m
end;;
```

Modyfikatory nie wymagają żadnych zmian.

Deser



<http://xkcd.com/303/>