Wykład 5

Ewaluacja gorliwa i leniwa

Strategie ewaluacji

Ewaluacja leniwa w językach gorliwych

Listy leniwe w języku OCaml

Moduł Lazy, wyrażenia i wzorce leniwe

Przekazywanie argumentów do funkcji

Argumenty pozycyjne, nazwane i domyślne

Ewaluacja gorliwa i leniwa

Najważniejsze strategie ewaluacji (wartościowania, obliczania), stosowane w językach programowania to ewaluacja gorliwa i leniwa.

- *Ewaluacja gorliwa (rygorystyczna, sterowana danymi, sterowana podażą)* (ang. eager evaluation, strict evaluation, data-driven evaluation, supply-driven evaluation). Jest stosowana najczęściej.

- *Ewaluacja leniwa (nierygorystyczna, sterowana popytem)* (ang. lazy evaluation, non-strict evaluation, demand-driven evaluation).

Stosowana strategia ewaluacji stanowi jedną z najważniejszych charakterystyk języka programowania. Zwykle obok głównej strategii, języki programowania udostępniają mechanizmy, umożliwiające lokalne stosowanie drugiej strategii.

OCaml, Scheme, Java, C++ (i większość języków programowania) wykorzystują ewaluację gorliwą, natomiast Haskell – leniwą.

Poniżej idea obu strategii zostanie wyjaśniona na przykładzie definiowania zmiennej. Precyzyjna definicja strategii wartościowania wymagałaby użycia pewnych formalizmów, np. rachunku lambda.

Definicja zmiennej

Definicja zmiennej składa się z trzech faz:

(1) deklaracja zmiennej
(2) zdefiniowanie wyrażenia, z wartością którego zmienna ma być związana
(3) ewaluacja wyrażenia i związanie zmiennej z obliczoną wartością

Powyższe fazy mogą być wykonywane jednocześnie lub kolejno.
We wszystkich językach funkcyjnych (1) i (2) **muszą** być wykonane jednocześnie.

- W językach funkcyjnych z ewaluacją gorliwą (OCaml, SML, Scheme, …) wykonywana jest od razu faza (3) i zmienna jest wiązana z wartością.
- W językach funkcyjnych z ewaluacją leniwą (Haskell, …) faza (3) jest wykonywana dopiero wtedy, kiedy wartość zmiennej jest potrzebna.

Analogicznie:

- W językach funkcyjnych z ewaluacją gorliwą argument funkcji jest obliczany zawsze, a do funkcji jest przekazywana jego wartość.
- W językach funkcyjnych z ewaluacją leniwą argument funkcji jest przekazywany do funkcji jako wyrażenie, którego wartość jest obliczana dopiero wtedy, kiedy jest potrzebna (być może nigdy).

Leniwa ewaluacja koniunkcji i alternatywy

Większość współczesnych języków programowania, m.in. Scala, OCaml, Java, C++, ewaluuje koniunkcję i alternatywę leniwie.

---------- OCaml

```
# true || failwith "error";;
- : bool = true
# false && failwith "error";;
- : bool = false
```

---------- Scala

```
true || (throw new Exception("error"))
res0: Boolean = true
false && (throw new Exception("error"))
res1: Boolean = false
```

Ewaluacja leniwa w językach gorliwych

Najprostszym sposobem zachowania kontroli nad momentem ewaluacji wyrażenia w językach gorliwych jest utworzenie z niego funkcji. Funkcja (dokładniej: jej domknięcie) jest wartością. Wyrażenie, stanowiące treść funkcji, będzie obliczane dopiero po zaaplikowaniu funkcji do argumentu.

Jaki powinien być argument takiej funkcji? We współczesnych językach funkcyjnych z typizacją statyczną służy do tego typ unit, którego jedyną wartością jest ().

```
function () -> expr        (* abstrakcja funkcyjna, wartość *)

(function () -> expr)()    (* ewaluacja expr *)
```

Domknięcie funkcji jest wartością

```
-------------------------------------------------------
# let f = function () -> 2+5;;
(* lub z lukrem syntaktycznym: let f() = 2+5  *)

val f : unit -> int = <fun>
# f();;
- : int = 7
```

Listy leniwe (1)

Łącząc abstrakcję funkcyjną z abstrakcją danych i wykorzystując fakt, że funkcja (dokładniej – jej domknięcie) jest wartością, możemy zdefiniować nieskończone listy czy drzewa w językach z ewaluacją gorliwą.

Typ `llist` reprezentuje listy leniwe (skończone i nieskończone).

```
# type 'a llist = LNil | LCons of 'a * (unit -> 'a llist);;
type 'a llist = LNil | LCons of 'a * (unit -> 'a llist)

# let lhd = function
    LNil -> failwith "lhd"
  | LCons (x, _) -> x
;;
val lhd : 'a llist -> 'a = <fun>

# let ltl = function
    LNil -> failwith "ltl"
  | LCons (_, xf) -> xf()
;;
val ltl : 'a llist -> 'a llist = <fun>
```

Listy leniwe (2)

Funkcja `lfrom` generuje ciąg rosnących liczb całkowitych zaczynający się od *k*.

```
# let rec lfrom k = LCons (k, function () -> lfrom (k+1));;
val lfrom : int -> int llist = <fun>
```

Wywołanie `ltake(n,xl)` zwraca pierwszych *n* elementów listy leniwej *xl* w postaci zwykłej listy.

```
# let rec ltake = function
    (0, _)            -> []
  | (_, LNil)         -> []
  | (n, LCons(x,xf)) -> x::ltake(n-1, xf())
;;
val ltake : int * 'a llist -> 'a list = <fun>
```

Np.

```
# ltake (5,lfrom 30);;
- : int list = [30; 31; 32; 33; 34]
```

Listy leniwe (3)

Funkcja `toLazyList: 'a list -> 'a llist` ze zwykłej listy tworzy listę leniwą.

```
let rec toLazyList = function
    [] -> LNil
  | x::xs -> LCons(x, function () -> toLazyList xs);;
```

Powyższe funkcje są bardzo pomocne w testowaniu funkcji działających na listach leniwych.

Listy leniwe (OCaml) - ewaluacja

Obliczenie `ltake(2,lfrom 30)` przebiega następująco:

```
ltake(2,lfrom 30)
=> ltake(2,LCons(30, function () -> lfrom (30+1)))
=> 30::ltake(1,lfrom (30+1))
=> 30::ltake(1,LCons(31, function () -> lfrom (31+1)))
=> 30::31::ltake(0,lfrom (31+1))
=> 30::31::ltake(0,LCons(32, function () -> lfrom (32+1)))
=> 30::31::[]
≡ [30;31]
```

Należy pamiętać, że **literał funkcyjny** (`function` x → e) **jest wartością** (patrz wykład 1, reguła *Fun*), więc wyrażenie e nie jest wartościowane do momentu aplikacji funkcji do argumentu.

Funkcjonały dla list leniwych: konkatenacja

Funkcja (@$) konkatenuje dwie listy leniwe (odpowiednik funkcji @ dla zwykłych list). Jeśli *ll1* jest nieskończona to wynikiem *ll1*@$ *ll2* ma być *ll1*.

```
# let rec (@$) ll1 ll2 =
  match ll1 with
      LNil -> ll2
    | LCons(x, xf) -> LCons(x, function () -> (xf()) @$ ll2);;
val ( @$ ) : 'a llist -> 'a llist -> 'a llist = <fun>

# let ll1 = LCons(2,function ()->LCons(1,function ()->LNil));;
val ll1 : int llist = LCons (2, <fun>)
# let ll2 = lfrom 3;;
val ll2 : int llist = LCons (3, <fun>)
# ltake (10, ll1 @$ ll2);;
- : int list = [2; 1; 3; 4; 5; 6; 7; 8; 9; 10]
```

Funkcjonały dla list leniwych: lmap

Dla list leniwych można zdefiniować funkcjonały podobne do tych, jakie zdefiniowaliśmy dla zwykłych list, np. `lmap`.

```
# let rec lmap f = function
    LNil -> LNil
  | LCons(x,xf) -> LCons(f x, function () -> lmap f (xf()) )
;;
val lmap : ('a -> 'b) -> 'a llist -> 'b llist = <fun>

# let sqr_llist = lmap (function x -> x*x);;
val sqr_llist : int llist -> int llist = <fun>

# ltake (6, sqr_llist (lfrom 3));;
- : int list = [9; 16; 25; 36; 49; 64]
```

Funkcjonały dla list leniwych: lfilter, liter

Funkcja `lfilter` wywoływana jest rekurencyjnie (rekursja ogonowa) tak długo, dopóki nie zostanie znaleziony element, spełniający podany predykat. Jeśli taki element nie istnieje, wartościowanie funkcji się nie skończy (dla nieskończonych list leniwych).

```
# let rec lfilter pred = function
    LNil -> LNil
  | LCons(x,xf) -> if pred x
                   then LCons(x, function () -> lfilter pred (xf()) )
                   else lfilter pred (xf())
;;
val lfilter : ('a -> bool) -> 'a llist -> 'a llist = <fun>
```

Funkcja `lfrom` jest szczególnym przypadkiem funkcji `liter`, która generuje ciąg:
`[x, f(x), f(f(x)), … f`^{k}`(x), …]`.

```
# let rec liter f x = LCons(x, function () -> liter f (f x));;
val liter : ('a -> 'a) -> 'a -> 'a llist = <fun>
```

Generowanie liczb pierwszych metodą sita Eratostenesa

Utwórz ciąg nieskończony [2,3,4,5,6, ...]. Dwa jest liczbą pierwszą. Usuń z ciągu wszystkie wielokrotności dwójki, ponieważ nie mogą być liczbami pierwszymi. Pozostaje ciąg [3,5,7,9,11, ...]. Trzy jest liczbą pierwszą. Usuń z ciągu wszystkie wielokrotności trójki, ponieważ nie mogą być liczbami pierwszymi. Pozostaje ciąg [5,7,11,13,17, ...]. Pięć jest liczbą pierwszą itd.

Na każdym etapie ciąg zawiera liczby, które nie są podzielne przez żadną z wygenerowanych do tej pory liczb pierwszych, więc pierwsza liczba tego ciągu jest liczbą pierwszą. Opisane kroki można powtarzać w nieskończoność.

```
#let primes =
   let rec sieve = function
       LCons(p,nf) -> LCons( p,
                             function () -> sieve
                                (lfilter (function n -> n mod p <> 0)
                                         (nf())
                                )
                           )
     | LNil -> failwith "Impossible! Internal error."
    in sieve (lfrom 2)
;;
val primes : int llist = LCons (2, <fun>)
```

Np.

```
# ltake (6,primes);;
- : int list = [2; 3; 5; 7; 11; 13]
```

Generowanie liczb pierwszych „na zamówienie”

Znalezienie kolejnej liczby pierwszej wymagałoby jednak powtórzenia wszystkich obliczeń od początku. Można temu zapobiec, modyfikując funkcję `ltake` w taki sposób, żeby oprócz skończonej listy początkowych elementów listy leniwej zwracała ogon listy leniwej.

```
# let rec ltakeWithTail = function
    (0, xq) -> ([],xq)
  | (_, LNil) -> ([],LNil)
  | (n, LCons(x,xf)) ->
      let (l,tail)=ltakeWithTail(n-1, xf())
      in (x::l,tail);;
val ltakeWithTail : int * 'a llist -> 'a list * 'a llist = <fun>
```

Teraz kolejne liczby pierwsze można otrzymywać w miarę potrzeby, wykorzystując zapamiętany ogon listy leniwej.

```
# let (p1,t) = ltakeWithTail (3, primes);;
val p1 : int list = [2; 3; 5]
val t : int llist = LCons (7, <fun>)
# let (p2,t) = ltakeWithTail (3, t);;
val p2 : int list = [7; 11; 13]
val t : int llist = LCons (17, <fun>)
# let (p3,t) = ltakeWithTail (5, t);;
val p3 : int list = [17; 19; 23; 29; 31]
val t : int llist = LCons (37, <fun>)
# let (p4,t) = ltakeWithTail (4, t);;
val p4 : int list = [37; 41; 43; 47]
val t : int llist = LCons (53, <fun>)
```

Moduł Lazy (1)

W języku OCaml istnieje moduł `Lazy`, który umożliwia kontrolę nad momentem ewaluacji wyrażenia. Wykorzystując słowo kluczowe `lazy` możemy utworzyć wartość typu `'a lazy_t`, zawierającą „zamrożone” wartościowanie, które możemy „odmrozić” (tj. wymusić ewaluację) za pomocą funkcji `Lazy.force : 'a Lazy.t -> 'a.`
Typ `'a t` jest zdefiniowany w module `Lazy` jako: `type 'a t = 'a lazy_t`. Np.

```
# let x = lazy (true||false, 3*4);;
val x : (bool * int) lazy_t = <lazy>
# Lazy.force x;;
-: bool * int = (true, 12)
```

„Zamrożone” wartościowanie zostało wykonane i obliczona wartość jest zapamiętana, dzięki czemu ponowna ewaluacja nie będzie potrzebna. Są tu wykorzystane imperatywne cechy języka, o których będziemy mówili później.

```
# x;;
- : (bool * int) lazy_t = lazy (true, 12)
```

Moduł Lazy (2)

Oto inny przykład.

```
# let f x = "OK";;
val f : 'a -> string = <fun>
# f(lazy(1/0));;
- : string = "OK"
```

Argument jest ewaluowany leniwie, a ponieważ nie jest wykorzystywany w funkcji oznacza to, że nigdy nie jest wartościowany.

```
# f(1/0);;
Exception: Division_by_zero.
```

Tutaj argument jest ewaluowany gorliwie.

Leniwe wzorce

Do wzorca

lazy *pat*

dopasowuje się wartość *v* typu Lazy.t, jeśli wynik wymuszenia Lazy. force *v* dopasowuje się do wzorca *pat*. Słowo lazy może wystąpić we wzorcu wszędzie tam, gdzie można umieścić zwykły konstruktor wartości.

Wykorzystując ten mechanizm można zaimplementować funkcję biblioteczną Lazy.force:

```
# let force (lazy v) = v;;
val force : 'a lazy_t -> 'a = <fun>
```

Wykorzystując moduł Lazy i leniwe wzorce można definiować struktury nieskończone bez jawnego użycia funkcji dla opóźnienia ewaluacji wyrażenia. Poniżej zostanie to zilustrowane na przykładzie list leniwych.

Wzorce leniwe i listy leniwe (1)

```
# type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t;;
type 'a llist = LNil | LCons of 'a * 'a llist Lazy.t

# let lhd = function
     LNil -> failwith "lhd"
   | LCons (x, _) -> x;;
val lhd : 'a llist -> 'a = <fun>
# let ltl = function
    LNil -> failwith "ltl"
  | LCons (_, lazy t) -> t;;
val ltl : 'a llist -> 'a llist = <fun>

# let rec lfrom k = LCons (k, lazy (lfrom (k+1)));;
val lfrom : int -> int llist = <fun>
# let rec ltake = function
    (0, _) -> []
  | (_, LNil) -> []
  | (n, LCons(x,lazy xs)) -> x::ltake(n-1,xs);;
val ltake : int * 'a llist -> 'a list = <fun>
```

Wzorce leniwe i listy leniwe (2)

```
# let rec lsquares = function
    LNil -> LNil
  | LCons(x,lazy xs) -> LCons(x*x, lazy(lsquares xs));;
val lsquares : int llist -> int llist = <fun>

# ltake (6, lsquares(lfrom 3));;
- : int list = [9; 16; 25; 36; 49; 64]

# let rec lmap f = function
    LNil -> LNil
  | LCons(x,lazy xs) -> LCons(f x, lazy(lmap f xs));;
val lmap : ('a -> 'b) -> 'a llist -> 'b llist = <fun>

# let sqr_llist = lmap (function x -> x*x);;
val sqr_llist : int llist -> int llist = <fun>

# ltake (6, sqr_llist (lfrom 3));;
- : int list = [9; 16; 25; 36; 49; 64]
```

Wzorce leniwe i listy leniwe (3)

```
# let rec lzip pll =
  match pll with
    (LCons(x, lazy xs), LCons(y, lazy ys)) -> LCons((x,y), lazy (lzip (xs, ys)))
  |                          _                          -> LNil;;
      val lzip : 'a llist * 'b llist -> ('a * 'b) llist = <fun>
# let llp = lzip(lfrom 1, lfrom 10);;
val llp : (int * int) llist = LCons ((1, 10), <lazy>)
# ltake(5, llp);;
- : (int * int) list = [(1, 10); (2, 11); (3, 12); (4, 13); (5, 14)]
# let rec lunzip ll =
  match ll with
  | LCons((h1,h2), lazy ps) -> (LCons(h1, lazy (fst(lunzip ps))),  LCons(h2, lazy (snd(lunzip ps))))
  | LNil                            -> (LNil,LNil);;
      val lunzip : ('a * 'b) llist -> 'a llist * 'b llist = <fun>
# let (ll1, ll2)=lunzip llp in (ltake(7, ll1),ltake(5, ll2));;
- : int list * int list = ([1; 2; 3; 4; 5; 6; 7], [10; 11; 12; 13; 14])
```

Parametry a argumeny funkcji

W wywołaniu funkcji (aplikacji funkcji do argumentu) przekazywane wyrażenia są *argumentami* funkcji.

Argumenty są wykorzystywane do inicjowania *parametrów* funkcji, czyli:

parametry = argumenty formalne (Stroustrup) lub

argumenty = parametry aktualne.

To rozróżnienie jest szczególnie ważne we współczesnym C++, ponieważ parametry są l-wartościami (ang. l-values), ale argumenty, którymi są inicjowane mogą być l-wartościami lub r-wartościami.

Przekazywanie argumentów do funkcji

Mechanizmy gorliwe

- przez wartość (ang. call by value)
- przez referencję (ang. call by reference)
- przez przenoszenie w C++ (ang. move semantics)

Mechanizmy leniwe

- przez nazwę (ang. call by name)
- przez potrzebę (ang. call by need)

W językach z ewaluacją gorliwą najczęściej stosowany jest mechanizm przekazywania argumentów do funkcji przez wartość.

Przekazywanie argumentów do funkcji

Wywołanie przez wartość (ang. call by value)
Argument aktualny jest ewaluowany zawsze. Do funkcji jest przekazywana jego wartość lub odwołanie do wartości, ale funkcja nie może zmieniać tej wartości w środowisku wywołującym.
Mechanizm wykorzystywany w językach OCaml, Java, C; domyślnie w językach Scala, C++, C#, Pascal.

Wywołanie przez referencję (ang. call by reference)
Argument aktualny jest ewaluowany zawsze. Do funkcji jest przekazywane odwołanie (referencja) do obliczonej wartości. Zmiana wartości argumentu wewnątrz funkcji powoduje jej zmianę w środowisku wywołującym.
Mechanizm opcjonalny w językach C++ (argument przekazywany przez referencję jest poprzedzany symbolem `&`), Pascal (argument przekazywany przez referencję jest poprzedzany słowem kluczowym `var`), C# (słowo kluczowe `ref`).

Wywołanie przez nazwę (ang. call by name)
Argument aktualny nie jest ewaluowany. Do funkcji jest przekazywane jest jego domknięcie (ang. thunk). Obliczanie wartości domknięcia jest *wymuszane* (ang. forced) tylko wtedy, kiedy wartość argumentu jest potrzebna (za każdym razem).
Mechanizm opcjonalny w języku Scala.

Wywołanie przez potrzebę (ang. call by need)
Jest to odmiana wywołania przez nazwę. Po pierwszym wymuszeniu obliczania wartości domknięcia argumentu aktualnego, wartość ta jest zapamiętywana i wykorzystywana w dalszych obliczeniach. Mechanizm wykorzystywany w języku Haskell.

Argumenty pozycyjne, nazwane i domyślne

Poniższa klasyfikacja przekazywania argumentów do funkcji jest bardziej związana z udogodnieniami składniowymi, niż z mechanizmami semantycznymi.

Argumenty pozycyjne

Jest to najczęściej stosowany sposób. Polega on na podawaniu wartości argumentów w kolejności ich występowania na liście deklaracji parametrów formalnych w definicji funkcji.

Argumenty nazwane (etykietowane)

W niektórych językach programowania argumenty można identyfikować za pomocą nazwy parametru formalnego (np. Scala) lub dodatkowej etykiety (np. OCaml). Nie trzeba wtedy przestrzegać kolejności argumentów, ustalonej w definicji funkcji. Szczegóły zależą od języka programowania.

Argumenty domyślne (parametry opcjonalne)

Deklarując parametry funkcji, można podać ich domyślne wartości (zwykle po znaku równości).

Argumenty nazwane w języku OCaml

W OCamlu parametry etykietowane są deklarowane za pomocą składni *~label: pattern*. Składnia argumentów etykietowanych jest podobna *~label: expression.*

```
# let minusRazy ~m_n:(m, n)  ~razy:razy = (m – n)*razy;;
val minusRazy : m_n:int *int -> razy:int -> int = <fun>
# minusRazy ~razy:2 ~m_n:(5, 2);;
- : int = 6
# minusRazy ~razy:2;;
- : m_n:int*int -> int = <fun>
```

Jeśli etykiety są takie same jak nazwy parametrów, to można użyć skrótu *~label*.

```
# let minusRazy ~m_n:(m, n)  ~razy = (m – n)*razy;;
val minusRazy : m_n:int *int -> razy:int -> int = <fun>
# let razy = 2;;
val m : int = 2
# minusRazy ~razy ~m_n:(1+4, 2);;
- : int = 6
# minusRazy (5, 2) 2;;
- : int = 6
```

Argumenty nazwane i domyślne w języku OCaml

Argumenty nazwane (etykietowane)

```
# minusRazy ~m_n:(5, 2)  2;;
Characters 22-23:
  minusRazy ~m_n:(5, 2)  2;;
                         ^
Error: The function applied to this argument has type razy:int -> int
This argument cannot be applied without label
```

Argumenty domyślne

Parametry opcjonalne mają składnię ?(*label* = *expression*). Parametr opcjonalny nie może być ostatnim parametrem.

```
# let minus ?(m=5) n = m - n;;
val minus : ?m:int -> int -> int = <fun>
# minus 2;;
- : int = 3
# minus ~m:15 2;;
- : int = 13
```

Funkcje wariadyczne

Wiele języków programowania pozwala definiować *funkcje wariadyczne* (ang. variadic functions), które przyjmują zmienną liczbę argumentów. W zależności od języka programowania kompilator tworzy z takich argumentów listę lub tablicę i przekazuje ją do funkcji.

OCaml ani Haskell nie udostępniają specjalnej notacji dla funkcji wariadycznych, ale taka notacja istnieje w wielu językach, np. Scheme, Scala, Java.

Zadania kontrolne

1. Zdefiniuj funkcję, która dla danej nieujemnej liczby całkowitej *k* i listy leniwej [x_0, x_1, x_2, ...] zwraca listę leniwą, w której każdy element jest powtórzony *k* razy, np. dla *k*=3:
[x_0, x_0, x_0, x_1, x_1, x_1, x_2, x_2, x_2, ...].

2. Zdefiniuj leniwą listę liczb Fibonacciego *lfib* : `int llist`,

3. Polimorficzne leniwe drzewa binarne można zdefiniować następująco:

```
type 'a lBT = LEmpty
            | LNode of 'a * (unit ->'a lBT) * (unit -> 'a lBT);;
```

a) Napisz funkcję *itr*, która dla zadanej liczby całkowitej *n* konstruuje nieskończone leniwe drzewo binarne z korzeniem o etykiecie *n* i z dwoma poddrzewami itr(2**n*) i itr(2**n*+1). To drzewo jest przydatne do testowania funkcji z następnego podpunktu.

b) Napisz funkcję, tworzącą leniwą listę, zawierającą wszystkie etykiety leniwego drzewa binarnego.
Wskazówka: zastosuj obejście drzewa wszerz, reprezentując kolejkę jako zwykłą listę.