Paradygmaty Programowania

Ćwiczenia

Rozwiązania zadań teoretycznych

Lista 1./Zadanie 7.

Rozwiąż układ równań rekurencyjnych (zakładając, że N jest potęgą dwójki):

$$\begin{cases} T(1) = 1 \\ T(N) = c \log_2 N + T\left(\frac{N}{2}\right) \qquad dla\ N \geq 2 \end{cases}$$

Wykorzystaj technikę zilustrowaną na wykładzie 1 (Dodatek: Złożoność obliczeniowa. Podstawowe pojęcia).

Rozwiązanie:

Rozpiszmy kilka kolejnych wyrazów ciągu, aby zobaczyć jego właściwości a następnie zapiszmy to wyrażenie przy pomocy operatora sumy:

$$T(N) = c \log_2 N + c \log_2 \frac{N}{2} + c \log_2 \frac{N}{4} + c \log_2 \frac{N}{8} + \cdots + c \log_2 8 + c \log_2 4 + c \log_2 2 + 1$$

$$T(N) = c \cdot \left(\log_2 N + \log_2 \frac{N}{2} + \log_2 \frac{N}{4} + \log_2 \frac{N}{8} + \cdots + \log_2 8 + \log_2 4 + \log_2 2 \right) + 1$$

$$T(N) = 1 + c \cdot \sum_{i=0}^{\frac{N}{2}} \log_2 \left(\frac{N}{2^i}\right) = 1 + c \cdot \sum_{i=0}^{k-1} \log_2 \left(\frac{2^k}{2^i}\right) = 1 + c \cdot \sum_{i=0}^{k-1} \log_2 2^{k-i} = 1 + c \cdot \sum_{i=0}^{k-1} (k-i)$$

Zauważmy, że jest to ciąg arytmetyczny, w którym $a_1 = \log_2 2 = 1$, $a_n = \log_2 N = k$ i $n = k - 1$. Korzystając ze wzoru:

$$S_n = \frac{a_1 + a_n}{2} \cdot n$$

Sumę można zapisać w następujący sposób:

$$T(N) = 1 + c \cdot k \cdot \frac{1+k}{2} = 1 + \frac{c}{2} \cdot (k + k^2) = 1 + \frac{c}{2} \cdot [\log_2 N + (\log_2 N)^2] = \Theta((\log_2 N)^2)$$

Lista 2./Zadanie 1.

Jaka będzie głębokość stosu (i dlaczego) w Scali, a jaka w OCamlu dla wywołania `evenR(3)` (funkcja zdefiniowana na wykładzie)?

Przypomnienie funkcji:

a) Scala

```
def evenR(n: Int): Boolean =
    if n == 0 then true else oddR(n-1)

def oddR(n: Int): Boolean =
    if n == 0 then false else evenR(n-1)
```

b) OCaml

```
let rec evenR n =
    if n = 0 then true else oddR(n-1)

let rec oddR n =
    if n = 0 then false else evenR(n-1)
```

Rozwiązanie:

Głębokość stosu dla wywołania `evenR(3)` w Scali będzie wynosiła 4 (+1 dla każdego `n`). Wynika to z faktu, że kompilator nie wykonuje domyślnie optymalizacji dla rekurencji ogonowej. Potrzebna do tego jest adnotacja `@tailrec`, jednakże w tym wypadku nie zadziałałaby, gdyż funkcja ta wywołuje inną funkcję (funkcje wywołują się wzajemnie).

W OCaml optymalizacja rekurencji ogonowej zawsze wykonywana jest automatycznie, jednak w tym wypadku również mamy do czynienia z funkcjami wywołującymi się wzajemnie, przez co głębokość stosu również wyniesie 4.

Lista 3./Zadanie 1.

Podaj (i uzasadnij!) najogólniejsze typy poniższych funkcji (samodzielnie, bez pomocy kompilatora OCamla!):

a) `let f1 x = x 2 2;;`

b) `let f2 x y z = x (y ^ z);;`

Rozwiązania:

a) Najogólniejszy typ funkcji `f1`:

`x` musi być funkcją przyjmującą dwa argumenty typu `int`, a zatem typy funkcji `x`: `int -> int -> a'` (nie wiadomo, jaki typ ma zwracać ta funkcja).

Podsumowując:

`f1: (int -> int -> a') -> a'`

b) Najogólniejszy typ funkcji `f2`:
argumenty `y` i `z` muszą być typu `string`, ponieważ jak można zauważyć, w definicji pomiędzy nimi występuje operator konkatenacji `^`. `x` musi być zatem funkcją przyjmującą argument typu `string` (wynik konkatenacji `y` i `z`). Zatem typ funkcji `x: string -> a'` (nie wiadomo, jaki typ ma zwracać ta funkcja).
Podsumowując:
`f2: (string -> a') -> string -> string -> a'`

Lista 3./Zadanie 4.

Poniższe dwie wersje funkcji `quicksort` działają niepoprawnie. Dlaczego?

a)

```
let rec quicksort = function
      [] -> []
    | [x] -> [x]
    | xs -> let small = List.filter (fun y -> y < List.hd xs) xs
            and large = List.filter (fun y -> y >= List.hd xs) xs
            in quicksort small @ quicksort large;;
```

b)

```
let rec quicksort' = function
      [] -> []
    | x::xs -> let small = List.filter (fun y -> y < x) xs
               and large = List.filter (fun y -> y > x) xs
               in quicksort' small @ (x::quicksort' large)
```

Rozwiązanie:

a) Wersja `quicksort` błędnie wybiera pivot'a. Jest on dwukrotnie wybierany za pomocą `List.hd xs`, lecz pozostaje w samym `xs` przy wywołaniu `List.filter`, co może prowadzić do powielenia go lub usunięcia.

b) Wersja `quicksort'` działa lepiej, natomiast brakuje w niej warunku odpowiedniego rozdzielenia elementów (tzn. np. `y >= x`), przez co elementy równe wybranemu pivot'owi nie są obecne w posortowanej liście wynikowej.

Lista 4./Zadanie 1.

Podaj (i wyjaśnij!) typy poniższych funkcji (samodzielnie, bez pomocy kompilatora OCamla!):

a) `let f1 x y z = x y z;;`

b) `let f2 x y = function z -> x::y;;`

Rozwiązania:

a) `x` musi być funkcją przyjmującą dwa argumenty nieznanego typu oraz zwracać również typ, który nie jest sprecyzowany. Typ funkcji `x: a' -> b' -> c'`.
Podsumowując:
`f1: (a' -> b' -> c') -> a' -> b' -> c'`

b) Zauważmy, że y musi być listą elementów typu argumentu `x`. Przykładowo `x: a'` oraz `y: a' list`. Argument z nie jest używany, a więc może być dowolnego typu.
Podsumowując:
`f2: a' -> a' list -> b' -> a' list`

Lista 6./Zadanie 3.

Co i dlaczego wydrukuje poniższy program w Javie?

```
public class isEqual {
    static boolean isEqual1(int m, int n) {return m==n;}
    static boolean isEqual2(Integer m, Integer n) {return m==n;}
    public static void main(String[] args) {
        System.out.println(isEqual1(500,500));
        System.out.println(isEqual1(500,500));
    }
}
```

Rozwiązanie:

Program ten wydrukuje kolejno `true` oraz `true`. Wynika to z tego, że operator == dla typów prymitywnych (jak `int`) porównuje ich wartość, zamiast sprawdzania, czy obiekty zajmują ten sam adres pamięci (i są *de facto* tym samym obiektem). W tym wypadku mają one obie wartość `500`, a więc są równe. Z kolei dla typów obiektowych (*Object Types*), operator ten porównuje zajmowany obszar w pamięci (adres), czyli sprawdza, czy obie zmienne są referencjami do jednej instancji danego obiektu. W tym wypadku oba obiekty typu `Integer` mają wartość 500 i nie są tym samym obiektem. **Jednakże** w tym konkretnym przykładzie, gdy wartości podawane są bezpośrednio jako argumenty metody (*inline*), w niektórych kompilatorach lub środowiskach JVM dochodzi do optymalizacji, w której argumenty o tej samej wartości redukowane są do jednego obiektu `Integer`, a w innych nie. Zwykle wartości typu `Integer` z przedziału poza $[-128, 127]$ nie są cache'owane a zastosowanie polecenia `Integer.valueOf(500)` zwróci dwie różne instancje. Możliwa jest zatem również odpowiedź `true` oraz `false`.

Lista 6./Zadanie 4.

Co i dlaczego wydrukuje poniższy program w Javie?

```
public class Porównanie {
    public static void main(String[] args) {
        String s1 = "foo";
        String s2 = "foo";
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
        String s3 = new String("foo");
        System.out.println(s1 == s3);
        System.out.println(s1.equals(s3));
    }
}
```

Rozwiązanie:

Program ten wydrukuje:

```
true
true
false
true
```

Wynika to z tego, że operator == porównuje zmienne jako referencje do obiektów, tzn. sprawdza, czy wskazują na ten sam adres w pamięci urządzenia. Metoda `equals()` z kolei porównuje zawartość obiektów, czyli wartości wszystkich ich pól. Naturalnie pomyśleć można by, że `s1 == s2` zwróci wartość `false` (ponieważ to dwa różne obiekty typu `String`), jednakże Java korzysta z pewnej optymalizacji w postaci tzw. *String pool*. Jest to miejsce w pamięci, gdzie przechowywane są wszystkie dotychczas zainicjalizowane wartości zmiennych typu `String` używane w danym programie. Optymalizacja polega na tym, że nie są tworzone dwa różne obiekty `String` o wartości `"foo"`, a zamiast tego tylko jeden, na który wskazują wszystkie inne zmienne typu `String` z tą samą przypisaną wartością. Dopiero jawne zadeklarowanie i zdefiniowanie nowego obiektu `String` (tutaj `s3`) sprawia, że faktycznie tworzona jest nowa instancja `String` (poza *String pool*), stąd `s1 == s3` zwróci `false`.

Lista 6./Zadanie 5.

Co i dlaczego wydrukuje poniższy program w Javie?

```
public class Aliasy {
    public static void main(String[] args) {
        int[] ints = {1,2,3};
        for (int i : ints) {
            System.out.println(i); i = 0;
        }
        for (int i : ints) {
            System.out.println(i);
        }
        int[] ints2 = ints;
        for (int i = 0; i < ints2.length; i++) {
            System.out.println(ints2[i]); ints[i] = -1;
        }
        for (int i : ints) System.out.println(i);
    }
}
```

Rozwiązanie:

Program ten wydrukuje:

```
1
2
3
1
2
3
1
2
3
-1
-1
-1
```

Wynika to z faktu, że pętla typu `for-each` tworzy kopię elementu kolekcji, po której iteruje, dlatego każda zmiana (np. `i = 0`) w tej pętli ma zasięg jedynie lokalny i nie jest utrzymywana poza pętlą, a kopia elementu zostaje usunięta po zakończeniu tej pętli. Zmiany zachodzą dopiero przy wykorzystaniu zwykłej pętli `for`, gdyż wówczas operuje ona na konkretnej kolekcji samej w sobie i jej elementach, a nie ich kopii, przez co operacja `ints[i] = 0` jest możliwa i trwała.

Lista 8./Zadanie 2.

Przeanalizuj następujący program w Javie. Czy ten program się skompiluje? Jeśli nie, to dlaczego i jak go poprawić (bez zmieniania argumentów metod)?

```
public class Test {
    int zawartość = 0;
    static void argNiemodyfikowalny(final Test zmienna) {
        zmienna.zawartość = 1;
        zmienna = null;
    }
    static void argModyfikowalny(Test zmienna) {
        zmienna.zawartość = 1;
        zmienna = null;
    }
    public static void main(String[] args) {
        Test modyfikowalna = new Test();
        final Test niemodyfikowalna = new Test();
    // tutaj wstaw instrukcje
    }
}
```

Co i dlaczego zostanie wyświetlone, jeśli wiersz „`// tutaj wstaw instrukcje`" zastąpimy następującymi instrukcjami:

a)
```
argNiemodyfikowalny(modyfikowalna);
System.out.println(modyfikowalna.zawartość);
```
b)
```
argNiemodyfikowalny(niemodyfikowalna);
System.out.println(niemodyfikowalna.zawartość);
```
c)
```
argModyfikowalny(modyfikowalna);
System.out.println(modyfikowalna.zawartość);
```
d)
```
argModyfikowalny(niemodyfikowalna);
System.out.println(niemodyfikowalna.zawartość);
```

Rozwiązanie:

Program ten nie skompiluje się, ponieważ w metodzie `argNiemodyfikowalny()` występuje pewien istotny błąd. Przekazywana zmienna `Test` musi być `final`, czyli referencja do niej nie może być modyfikowana, jednakże, w ciele metody jest ona modyfikowana, co powoduje błąd: `zmienna = null`. Operacja `zmienna.zawartość = 1` jest jednak możliwa, gdyż modyfikuje właściwość obiektu, a nie zmienia referencji.

W celu poprawy działania programu należy więc usunąć z metody linijkę modyfikującą referencję.

a) Program wykona się prawidłowo i wyświetlone zostanie `1`, ponieważ mimo, że przekazywana zmienna jest teoretycznie modyfikowalna, to nie jest ona modyfikowana w ciele metody. Istotne jest to, że sam parametr metody jest final, sama faktyczna zmienna taka być nie musi, ponieważ metoda operuje jedynie na kopii referencji.

b) Program wykona się prawidłowo i wyświetlone zostanie `1`. Przekazywana zmienna (referencja) jest niemodyfikowalna w `main()`, jej kopia również i jako taka jest traktowana w ciele metody.

c) Program wykona się prawidłowo i wyświetlone zostanie `1`. Przekazywana zmienna (referencja) jest modyfikowalna w `main()` i jej kopia (również modyfikowalna) jest modyfikowana w ciele metody.

d) Program wykona się prawidłowo i wyświetlone zostanie `1`. Mimo że przekazywana zmienna (referencja) jest oznaczona jako `final`, to jest ona taka jedynie w metodzie `main()`; sam parametr metody `argModyfikowalny()` taki już nie jest a metoda operuje na kopii referencji do przekazywanej zmiennej .

Lista 10./Zadanie 1.

Klasa `GenericCellmm` kompiluje się jako klasa inwariantna i kowariantna.

```
scala> class GenericCellmm[T](val x: T)
// defined class GenericCellmm

scala> class GenericCellmm[+T](val x: T)
// defined class GenericCellmm
```

Natomiast klasa `GenericCellMut` kompiluje się tylko jako klasa inwariantna.

```
scala> class GenericCellMut[T](var x: T)
// defined class GenericCellMut

scala> class GenericCellMut[+T](var x: T)
-- Error:
1 |class GenericCellMut[+T](var x: T)
  |                               ^
  |covariant type T occurs in contravariant position in type T of parameter x_ =
```

a) Wyjaśnij powód tego błędu (należy dokładnie wyjaśnić powyższy komunikat).
b) Czy można się pozbyć tego błędu? Uzasadnij swoją odpowiedź.
c) Czy wersja kontrawariantna skompiluje się? Uzasadnij swoją odpowiedź.
`class GenericCellMut[-T](private var x: T)`

Rozwiązanie:

a) Powodem tego błędu jest fakt, że argument kowariantny nie może być użyty w roli argumentu kontrawariantnego. Dokładnie chodzi o to, że `val` umożliwia jedynie

odczytywanie wartości zmiennej, `var` jednakże umożliwia jej zapisywanie, co jest typowo kontrawariantną cechą (argument funkcji w pozycji kontrawariantnej).

b) Tego błędu można pozbyć się zmieniając typ kowariantny `+T` na inwariantny `T` lub zmieniając `val` na `var`.

c) Nie, wersja kontrawariantna nie skompiluje się, ponieważ `var` dalej umożliwia odczytywanie zmiennej (czyli cecha kowariantna), a wymagane jest występowanie jedynie możliwości zapisu.

Lista 10./Zadanie 2.

Poniższa definicja powoduje błąd kompilacji.

```
scala> abstract class Sequence[+A]:
       |   def append(x: Sequence[A]): Sequence[A]
       |
-- Error:
2 |def append(x: Sequence[A]): Sequence[A]
  |               ^^^^^^^^^^^
  |covariant type A occurs in contravariant position in type Sequence[A] of
parameter x
```

Wyjaśnij przyczynę tego błędu. Czy można się go pozbyć?

Rozwiązanie:

Przyczyną tego błędu jest występowanie argumentu kowariantnego w roli argumentu kontrawariantnego. Metoda `append()` ma za zadanie „dopisywanie" (kontrawariantna pozycja argumentu funkcji) sekwencji elementów danego typu `A` (oraz zwracanie sekwencji elementów typu `A`), tylko że sekwencja, do której dopisywane są elementy może składać się z elementów typu `+A`, co oznacza, że np. jeśli mamy typy `Dog <: Animal` oraz `A = Dog` (czyli `Sequence[+Dog]` oraz `Sequence[Dog] <: Sequence[Animal]`), to teoretycznie możliwe jest dopisanie `Sequence[Animal]` do `Sequence[Dog]`, a także w przypadku, `Cat <: Animal` oznaczało by to również możliwość dopisania `Sequence[Cat]` do `Sequence[Dog]`.

Tego błędu można pozbyć się na dwa sposoby. Pierwszy z nich to zmiana typu generycznego z `+A` na `A` (z kowariantnego na inwariantny). Umożliwi to operowanie na sekwencji elementów dokładnie typu `A`. Drugi z nich to posłużenie się kolejnym podtypem `B >: A` (`def append[B >: A](x: Sequence [B]): Sequence [B]`). Zagwarantuje to używanie argumentu funkcji, który będzie zawsze typem lub nadtypem `A` i nie spowoduje błędu kompilacji, gdyż w sytuacji `Dog <: Animal` oraz `A = Dog` i `B = Animal`, możliwe jest dodawanie elementów `Sequence[Dog]` do `Sequence[Animal]`.