Programowanie 2009 Programming 2009

Lista zadań nr 6 Problem set no. 6

Na zajęcia 7–8 kwietnia 2009 Due April 8, 2009

Grupy zasadnicze

Zadanie 1 (1 pkt). Złożenie funkcji zdefiniowano w Haskellu (podobnie jak w całej matematyce) następująco:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

Ponadto w Haskellu mamy operator aplikacji:

```
($) :: (a -> b) -> (a -> b)
f $ x = f x
```

Oba operatory łączą w prawo, „`.`" ma priorytet 9 (najwyższy), zaś „`$`" — 0 (najniższy). Przypuśćmy, że `h x y = f(g x y)`. Które z poniższych równości są prawdziwe?

```
h = f . g
h = f $ g
h x = f . (g x)
h x y = (f . g) x y
h x y = f $ g $ x y
h x y = f $ g x $ y
h x y = f $ g x y
h (f x) (f y) = h . f x $ f y
```

Przepisz poniższe wyrażenia nie używając operatorów „`.`" i „`$`":

```
u . v . w $ x . y $ z
u $ v . w $ x . y $ z
```

Zadanie 2 (1 pkt). Pitagoras rozważał trójkąty prostokątne o całkowitych długościach boków. Gdyby znał Haskella, byłoby mu łatwiej. Zdefiniuj w Haskellu funkcję

```
f :: Integer -> [(Integer, Integer, Integer)]
```

która dla podanego parametru n `:: Integer` tworzy listę trójek całkowitych liczb (a, b, c), takich, że $1 \leq a \leq b \leq c \leq n$ i a, b, c są długościami boków trójkąta prostokątnego (jedyna trudność w tym zadaniu polega na przypomnieniu sobie, że liczby a, b i c są długościami boków trójkąta prostokątnego wtedy i tylko wtedy, gdy $a^2 + b^2 = c^2$).

Basic groups

Problem 1 (1 p). The function composition has been defined in Haskell (as in the whole mathematics) in the following way:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(f . g) x = f (g x)
```

Besides, we have in Haskell the application operator:

```
($) :: (a -> b) -> (a -> b)
f $ x = f x
```

Both operators associate to the right, "`.`" has priority 9 (the biggest), but "`$`" — 0 (the smallest). Let `h x y = f(g x y)`. Which of the following equalities are true?

```
h = f . g
h = f $ g
h x = f . (g x)
h x y = (f . g) x y
h x y = f $ g $ x y
h x y = f $ g x $ y
h x y = f $ g x y
h (f x) (f y) = h . f x $ f y
```

Rewrite the expressions below not using the operators "`.`" and "`$`":

```
u . v . w $ x . y $ z
u $ v . w $ x . y $ z
```

Problem 2 (1 p). Pythagoras considered right triangles with integer sides. If he had known Haskell, it would have been easier for him. Define in Haskell a function

```
f :: Integer -> [(Integer, Integer, Integer)]
```

which, for a given parameter n `:: Integer`, creates a list of triples (a, b, c) of integers, such that $1 \leq a \leq b \leq c \leq n$, and a, b, c are the lengths of the sides of a right triangle (the only difficulty in this problem boils down to recalling that numbers a, b and c are the sides of a right triangle if and only if $a^2 + b^2 = c^2$).

Zadanie 3 (1 pkt). Zaprogramuj w Haskellu funkcję

Problem 3 (1 p). Define in Haskell a function

```
f :: Integer -> [(Integer, Integer, Integer, Integer)]
```

która dla podanego parametru n `::` `Integer` tworzy listę wszystkich czwórek (a, b, c, d), takich, że $0 < a, b, c, d \leq n$ i $a^2 + b^2 = c^2 + d^2$.

which, for a given parameter n `::` `Integer`, creates a list of all quadruples (a, b, c, d) such that $0 < a, b, c, d \leq n$ and $a^2 + b^2 = c^2 + d^2$.

Zadanie 4 (1 pkt). Funkcje `foldr` i `foldl` zdefiniowano w Haskellu następująco:

Problem 4 (1 p). The functions `foldr` and `foldl` have been defined in Haskell as follows:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ c [] = c
foldr (+) c (x:xs) = x + foldr (+) c xs

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ c [] = c
foldl (+) c (x:xs) = foldl (+) (c + x) xs
```

Używając wyłącznie funkcji `foldr` i `foldl` (w szczególności nie używając wyrażeń listowych „`[ | ]`" ani pomocniczych funkcji zdefiniowanych rekurencyjnie) zapisz wyrażenia równe następującym funkcjom standardowym (definicje funkcji standardowych znajdziesz w opisie preludium standardowego):

Using exclusively the functions `foldr` and `foldl` (in particular using neither the list comprehensions "`[ | ]`", nor auxiliary functions defined recursively) write expressions equal to the following standard functions (the definitions of the standard functions can be found in the description of the standard prelude):

```
map :: (a -> b) -> [a] -> [b]
(++) :: [a] -> [a] -> [a]
unzip :: [(a,b)] -> ([a],[b])
filter :: (a -> Bool) -> [a] -> [a]
concat :: [[a]] -> [a]
length :: [a] -> Int
```

Zadanie 5 (1 pkt). Wykonaj polecenie z poprzedniego zadania dla funkcji:

Problem 5 (1 p). Do the same as in the previous problem for the functions:

```
head :: [a] -> a
tail :: [a] -> [a]
last :: [a] -> a
init :: [a] -> [a]
null :: [a] -> Bool
```

Uwaga: to zadanie jest poniekąd perwersyjne — te funkcje jest wygodniej i znacznie efektywniej zaprogramować wprost.

Remark: this problem is somehow perverse — those functions can be more easily and much more efficiently defined directly.

Zadanie 6 (1 pkt). Niech

Problem 6 (1 p). Let

```
type Var = String
data Expr = Const Integer | Var Var | Neg Expr |
            (:+:) Expr Expr | (:-:) Expr Expr |
            (:*:) Expr Expr | (:/:) Expr Expr |
            Sin Expr | Cos Expr | Ln Expr
infixl 7 :*:, :/:
infixl 6 :+:, :-:
```

Aby móc czytelnie wypisywać takie wyrażenia uczyń typ `Expr` instancją klasy `Show` (zauważ, że domyślna metoda `show` nie robiłaby tego, co chcemy).

To be able to output such expressions in a readable form, make the type `Expr` an instance of the class `Show` (note that the default method `show` would not work as expected).

Zadanie 7 (1 pkt). Zaprogramuj funkcję

Problem 7 (1 p). Define a function

```
eval :: [(Var, Double)] -> Expr -> Double
```

obliczającą wartość podanego wyrażenia. Pierwszym parametrem funkcji `eval` jest lista asocjacji zmiennych i ich wartości liczbowych.

that evaluates a given expression. The first parameter of the function `eval` is an association list of variables and their numerical values.

Grupy rozszerzone

Extended groups

Zadanie 1 (1 pkt). Zaprogramuj funkcję

Problem 1 (1 p). Define a function

```
diff :: Expr -> Var -> Expr
infix 5 `diff`
```

która wyznacza pochodną podanej funkcji względem podanej zmiennej, np.

which computes a derivative of a given function with respect to a given variable, *e.g.*

```
Sin (Const 2 :*: Var"x") `diff` "x"  ⟹
Const 2 :*: Cos (Const 2 :*: Var"x")
```

W powyższym przykładzie popełniłem małe oszustwo: jeśli zaprogramujesz reguły obliczania pochodnych zgodnie z tym, czego nauczyłeś się na zajęciach z analizy matematycznej, to wynik nie będzie tak ładny. Nie poprawiaj go jednak wewnątrz funkcji `diff`. Zamiast tego rozwiąż kolejne zadanie.

In the example above I cheated a little: if you program the rules of differentiation according to what you have learned during a course of mathematical analysis, then the result will not be so elegant. However do not improve it inside your `diff` function. Solve the next problem instead.

Zadanie 2 (1 pkt). Zaprogramuj funkcję

Problem 2 (1 p). Define a function

```
simplify :: Expr -> Expr
```

upraszczającą wyrażenia. Wybierz pewien rozsądny zbiór reguł przepisywania, np.

that simplifies expressions. Choose a decent set of rewriting rules, *e.g.*

```
Const 0 :*: x  ⟹  Const 0
Const 1 :*: x  ⟹  x
Const 0 :+: x  ⟹  x
```

Niestety jeżeli wyrażenia zawierają funkcje przestępne, to nie istnieje dla nich rekurencyjna postać kanoniczna, a sprawdzanie równości dwóch wyrażeń jest nierozstrzygalne. Nie ma więc algorytmu, który wyznaczałby np. najkrótsze wyrażenie równe podanemu. Można więc wykonywać tylko *pewne* uproszczenia, licząc na to, że wynik będzie *dostatecznie* czytelny.

Unfortunately if expressions contain transcendental functions, then there is no recursive canonical form for them and checking for equality of two expressions is unsolvable. There is no, for example, algorithm that finds the shortest expression equal to a given one. One can only to perform *some* simplifications, hoping that the result will be *sufficiently* readable.

Zadanie 3 (1 pkt). *Drzewa czerwono-czarne*, to binarne drzewa poszukiwań, których każdy wierzchołek wewnętrzny jest pomalowany na czerwono lub na czarno:

Problem 3 (1 p). *The red-black trees* are binary search trees with nodes painted red and black:

```
data RBT a = Red (RBT a) a (RBT a) | Black (RBT a) a (RBT a) | Leaf
```

Niezmienniki drzew czerwono-czarnych są następujące:

1. Każdy wierzchołek spełnia następującą własność: etykieta każdego jego lewego potomka jest nie większa, zaś prawego — większa od etykiety tego wierzchołka.
2. Korzeń drzewa jest czarny.
3. Na każdej ścieżce od korzenia do liścia występuje tyle samo czarnych wierzchołków.
4. Żaden czerwony wierzchołek nie ma czerwonego syna.

Zaprogramuj funkcję

The invariants of the red-black trees are the following:

1. Every node has the following property: the label of any of its left descendant is not greater, and of any of its right descendant is greater than the label of this node.
2. The root of the tree is black.
3. All paths from the root to leaves contain the same number of black nodes.
4. No red node has a red son.

Define a function

```
insert :: Ord a => a -> RBT a -> RBT a
```

wstawiającą element do drzewa czerwono-czarnego w taki sposób, by były zachowane wszystkie cztery niezmienniki. (*Wskazówka:* wstaw element tak, jak do zwykłego drzewa binarnych poszukiwań i pomaluj go na czerwono. Mogłeś zepsuć warunki 2 i 4. Warunek 4 przywróć przebudowując odpowiednio drzewo, np.

that inserts an element to a red-black tree preserving all the invariants. (*Hint:* insert an element as to an ordinary binary search tree and paint it red. Doing this you could spoil the conditions 2 and 4. Restore the condition 4 properly rebuilding the tree, *e.g.*

```
Black (Red (Red t1 x t2) y t3) z t4
```

zastąp przez

replace with

```
Red (Black t1 x t2) y (Black t3 z t4)
```

Taką przebudowę drzewa nazywa się *rotacją*. Przywrócenie warunku 4 nie narusza warunków 1 i 3. Na koniec w trywialny sposób przywróć warunek 2. Szczegóły znajdziesz np. w Cormenie.)

Such a rebuilding is called *a rotation*. Restoring condition 4 does not violate the conditions 1 and 3. Finally restore in a trivial way the condition 2.

Zadanie 4 (1 pkt). Udowodnij, że drzewo czerwono-czarne o n etykietach ma wysokość $O(\log n)$ (*wskazówka:* zauważ, że najdłuższa ścieżka w drzewie jest co najwyżej dwa razy dłuższa od najkrótszej). Pokaż przykład, że wstawienie do zwykłego drzewa poszukiwań n elementów może trwać $\Omega(n^2)$ kroków. Pokaż, że wstawienie n elementów do drzewa czerwono-czarnego trwa $O(n \cdot \log n)$ kroków.

Problem 4 (1 p). Prove that a red-black tree with n labels has the hight of $O(\log n)$ (*hint:* note that the longest path in the tree is no more that twice as long as the shortest one). Show an example that inserting n elements to an ordinary binary search tree can last for $\Omega(n^2)$ steps. Show that inserting n elements to a red-black tree lasts for $O(n \cdot \log n)$ steps.

Zadanie 5 (1 pkt). Termy pierwszego rzędu reprezentujemy w Haskellu w postaci typu

Problem 5 (1 p). In Haskell we represent the first order terms in the form of a type

```
data Term = Var Variable | Term Symbol [Term var]
type Variable = String
type Symbol = String
```

Uczyń typ `Term` instancją klasy `Show` i klasy `Read`, tak, by można było wypisywać i wczytywać termy w notacji zbliżonej do prologowej.
Niech

Make the type `Term` an instance of the classes `Show` and `Read` so the terms can be read adn wtritten in a notation similar to used in Prolog.
Let

```
type Substitution = [(Variable, Term)]
```

Zaprogramuj funkcję

Define a function

```
unify :: (Term, Term) -> Maybe Substitution
```

która wyznacza najogólniejszy unifikator pary termów.

that computes the most general unifier of a pair of terms.

Zadanie 6 (1 pkt). Klauzule reprezentujemy w postaci termów zbudowanych przy użyciu symboli `\+/1` (negacja) oraz `;/2` (alternatywa). Dana jest lista klauzul. Zaprogramuj funkcję, która wyznacza listę wszystkich rezolwent klauzul z podanej listy.

Problem 6 (1 p). We represent clauses in the form of terms built with the use of symbols `\+/1` (negation) and `;/2` (disjunction). Given a list of clauses. Define a function that computes a list of all resolvents of clauses from the given list.

Zadanie 7 (1 pkt). Wyrażenia w notacji postfiksowej to ciągi tokenów. Tokenami są liczby i operatory arytmetyczne:

Problem 7 (1 p). Expressions in postfix notation are the sequences of tokens. Tokens are numbers and arithmetic operators:

```
data Token = Number Integer | Plus | Minus | Times | Div
type PostfixExpression = [Token]
```

Wartość wyrażenia w notacji postfiksowej oblicza się przy użyciu stosu:

The value of an expression in the infix notation can be computed using a stack:

```
calc :: PostfixExpression -> Integer
calc = fst . pop . foldl eval empty where
   eval :: IntegerStack -> Token -> IntegerStack
   eval s (Number n) = push (n, s)
   eval s binop = push ((case binop of
                           Plus -> (+)
                           Minus -> (-)
                           Times -> (*)
                           Div -> div) n2 n1, s2) where
                              (n1, s1) = pop s
                              (n2, s2) = pop s1
```

Stos możemy zaimplementować za pomocą listy:

We can implement stacks using lists:

```
type IntegerStack = [Integer]
empty :: [Integer]
empty = []
push :: (Integer, IntegerStack) -> IntegerStack
push (a, s) = a : s
pop :: IntegerStack -> (Integer, IntegerStack)
pop (a : s) = (a, s)
```

choć lepiej sparametryzować algorytm obliczania wartości wyrażeń zmiennym typem należącym do odpowiednio dobranej klasy specyfikującej (polimorficzny) stos. Zaprogramuj taką klasę i dostosuj do niej przedstawiony wyżej algorytm.

but a better solution is to parametrise the algorithm for computing a value with a variable type belonging to some suitably chosen class that specifies a (polymorphic) stack. Define such a class and adjust to it the algorithm presented above.

Grupa zaawansowana

Zadanie 1 (2 pkt). Rozważmy fragment języka Haskell związany z deklaracjami klas i ich instancji:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid T\,\vec{\alpha} \\
\sigma & ::= & (C_1\tau_1, \ldots, C_n\tau_n) \Rightarrow \tau \\
\delta & ::= & \texttt{class}\; C\,\alpha\; \texttt{where}\; m_1 :: \tau_1; \ldots m_k :: \tau_k \\
 & \mid & \texttt{instance}\; C\,(T\,\vec{\tau})\; \texttt{where}\; m_1 = e_1; \ldots m_k = e_k \\
 & \mid & \texttt{data}\; T\,\vec{\alpha} = D_1\,\vec{\tau_1}\; \texttt{|}\; \ldots\; \texttt{|}\; D_k\,\vec{\tau_k} \\
 & \mid & f :: \tau;\; f\,p_1^1 \ldots p_1^k = e_1; \ldots f\,p_m^1 \ldots p_m^k = e_m \\
e & ::= & x \mid D \mid e_1 e_2 \\
p & ::= & x \mid D\,p_1 \ldots p_k
\end{array}
$$

Dany jest program (ciąg deklaracji δ). Zakładamy, że program jest poprawny pod względem typów. Zdefiniuj zachowującą znaczenie translację tego programu do podzbioru języka nie zawierającego klas:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid T\,\vec{\alpha} \\
\delta & ::= & \texttt{data}\; T\,\vec{\alpha} = D_1\,\vec{\tau_1}\; \texttt{|}\; \ldots\; \texttt{|}\; D_k\,\vec{\tau_k} \\
 & \mid & f :: \tau;\; f\,p_1^1 \ldots p_1^k = e_1; \ldots f\,p_m^1 \ldots p_m^k = e_m \\
e & ::= & x \mid D \mid e_1 e_2 \\
p & ::= & x \mid D\,p_1 \ldots p_k
\end{array}
$$

Advanced group

Problem 1 (2 p). Consider a fragment of the Haskell language with class and instance declarations:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid T\,\vec{\alpha} \\
\sigma & ::= & (C_1\tau_1, \ldots, C_n\tau_n) \Rightarrow \tau \\
\delta & ::= & \texttt{class}\; C\,\alpha\; \texttt{where}\; m_1 :: \tau_1; \ldots m_k :: \tau_k \\
 & \mid & \texttt{instance}\; C\,(T\,\vec{\tau})\; \texttt{where}\; m_1 = e_1; \ldots m_k = e_k \\
 & \mid & \texttt{data}\; T\,\vec{\alpha} = D_1\,\vec{\tau_1}\; \texttt{|}\; \ldots\; \texttt{|}\; D_k\,\vec{\tau_k} \\
 & \mid & f :: \tau;\; f\,p_1^1 \ldots p_1^k = e_1; \ldots f\,p_m^1 \ldots p_m^k = e_m \\
e & ::= & x \mid D \mid e_1 e_2 \\
p & ::= & x \mid D\,p_1 \ldots p_k
\end{array}
$$

Given a program (a sequence of declarations δ). We assume that the program is type correct. Define a meaning preserving translation of this program to a subset of the language that does not contain classes:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid T\,\vec{\alpha} \\
\delta & ::= & \texttt{data}\; T\,\vec{\alpha} = D_1\,\vec{\tau_1}\; \texttt{|}\; \ldots\; \texttt{|}\; D_k\,\vec{\tau_k} \\
 & \mid & f :: \tau;\; f\,p_1^1 \ldots p_1^k = e_1; \ldots f\,p_m^1 \ldots p_m^k = e_m \\
e & ::= & x \mid D \mid e_1 e_2 \\
p & ::= & x \mid D\,p_1 \ldots p_k
\end{array}
$$