

# Programowanie Funkcyjne 2018

Lista zadań nr 11: Przygody w krainie typów.

Część 2: GADT, typy i klasy fantomowe, typy egzystencjalne i wyższych rzędów

Na zajęcia 10 stycznia 2019. Wersja z komentarzami

Dla osób znudzonych czytaniem długich komentarzy oferujemy na stronie zajęć także wersję TL;DR zawierającą wyłącznie polecenia do wykonania.

**Zadanie 1 (2 pkt).** *Języki dedykowane (Domain-Specific Languages, DSL)* są, w odróżnieniu od języków ogólnego przeznaczenia, używane do rozwiązywania problemów w konkretnych, wąskich dziedzinach. Prominentnymi przykładami takich języków są SQL i HTML, które są powszechnie wykorzystywane w oprogramowaniu serwerów WWW. Oprogramowanie serwera jest zwykle napisane w języku ogólnego przeznaczenia (PHP, Ruby, Python, Java itp.), a fragmenty kodu w językach SQL i HTML osadza się w nim w postaci zwykłych napisów. Nie tylko wykonanie, ale również analiza składniowa i kompilacja takich wstawek odbywa się dynamicznie podczas wykonania programu serwera, a nie statycznie podczas jego kompilacji. Jedynym sposobem sprawdzenia poprawności tych wstawek jest testowanie — kompilator nie wspiera nawet sprawdzenia ich poprawności składniowej, kontroli typów itp.

Języki o dużej sile wyrazu, w szczególności języki funkcyjne, pozwalają na osadzanie DSL-i w postaci wyrażeń określonego typu, kompilowanych i sprawdzanych razem z kodem gospodarza. Jednym z najstarszych przykładów tego typu są *kombinatory parsujące*, które pozwalają na zapisanie w języku funkcyjnym gramatyki bezkontekstowej i tym samym osadzenie w kodzie programu dowolnego parsera. Takie rozwiązanie jest znacznie wygodniejsze niż generowanie parsera za pomocą osobnego programu, takiego jak Bison. W Haskellu kombinatory parsujące są dostępne w module Parsec.

W tym zadaniu osadzimy w Haskellu bardzo prosty DSL — mały kalkulator. Wyrażenia naszego kalkulatora są opisane następującą składnią konkretną:

```
⟨expression⟩ ::= ( ⟨expression⟩ )
               | ⟨integer literal⟩
               | True
               | False
               | ( ⟨expression⟩ , ⟨expression⟩ )
               | ⟨unary operator⟩ ⟨expression⟩
               | ⟨expression⟩ ⟨binary operator⟩ ⟨expression⟩
               | ⟨expression⟩ ? ⟨expression⟩ : ⟨expression⟩
⟨unary operator⟩ ::= ! | fst | snd
⟨binary operator⟩ ::= + | - | * | / | < | <= | > | >= | != | == | && | || | +
```

Operatory binarne łączą w lewo. Operator / wyznacza parę złożoną z części całkowitej ilorazu i reszty z dzielenia podanych argumentów. Operatory uszeregowane według priorytetów: operatory unarne !, fst, snd, operatory multiplikatywne \*, /, operatory addytywne + i -, operatory relacyjne <, <=, >, >=, !=, ==, operator &&, operator ||, operator ternarny ?: . Reguły typowania wyrażeń są przedstawione na Rysunku 1.

Moglibyśmy z łatwością napisać interpreter wyrażeń całkowitoliczbowych:

```
interpreter :: String -> Integer
```

i umieszczać w programie Haskellowym wstawki w języku kalkulatora np. tak:

```
interpreter "2*7 > fst(6/3) && 3 < 5 : snd(15/12)+3 ? 7*8"
```

$\frac{n \text{ — integer literal}}{n : \text{Integer}}$	$\frac{}{\text{True} : \text{Bool}}$	$\frac{}{\text{False} : \text{Bool}}$
$\frac{e_1 : \sigma_1 \quad e_2 : \sigma_2}{(e_1, e_2) : (\sigma_1, \sigma_2)}$	$\frac{p : (\sigma_1, \sigma_2)}{\text{fst } p : \sigma_1}$	$\frac{p : (\sigma_1, \sigma_2)}{\text{snd } p : \sigma_2}$
$\frac{e_1 : \text{Integer} \quad e_2 : \text{Integer}}{e_1 \oplus e_2 : \text{Integer}}$	$\frac{e_1 : \text{Integer} \quad e_2 : \text{Integer}}{e_1 \odot e_2 : \text{Bool}}$	$\frac{e_1 : \text{Integer} \quad e_2 : \text{Integer}}{e_1 / e_2 : (\text{Integer}, \text{Integer})}$
$\frac{b : \text{Bool}}{! b : \text{Bool}}$	$\frac{b_1 : \text{Bool} \quad b_2 : \text{Bool}}{b_1 \otimes b_2 : \text{Bool}}$	$\frac{b : \text{Bool} \quad e_1 : \sigma \quad e_2 : \sigma}{b ? e_1 : e_2 : \sigma}$

Symbol  $\oplus$  oznacza dowolny z operatorów  $+$ ,  $-$ ,  $*$ , symbol  $\odot$  oznacza dowolny z operatorów  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $!=$ ,  $==$ , a symbol  $\otimes$  oznacza dowolny z operatorów  $\&\&$  i  $||$ .

Rysunek 1: Reguły typowania wyrażeń kalkulatora

(widać, że język należałoby znacznie rozbudować, żeby był użyteczny, ale nie o użyteczność, tylko o ideę tu chodzi).

W powyższym przykładzie wyrażenie w języku kalkulatora jest wstawione do programu Haskellowego w postaci napisu. Kompilator zaakceptuje dowolny napis, również zawierający błędy składniowe lub typowe, które zostaną wykryte dopiero przez funkcję interpreter podczas biegu programu. Lepszym rozwiązaniem byłoby zastąpienie w programie Haskellowym składni konkretnej wyrażeń kalkulatora składnią abstrakcyjną, np. tak:

```
infixl 6 :*
infixl 5 :+, :-
data Expr = C Integer | (:+) Expr Expr | (-) Expr Expr | (*) Expr Expr | ...
```

Zamiast `"2 + 3 * 7" :: String` moglibyśmy napisać

```
C 2 :+: C 3 :* C 7 :: Expr
```

Składnia niewiele straciła na czytelności, a kompilator Haskell'a sprawdza teraz poprawność składniową wyrażenia. Niestety w powyższej składni abstrakcyjnej wszystkie wyrażenia, niezależnie od swojego typu, są reprezentowane przez wartości typu `Expr`, nie jest więc możliwa kontrola poprawności typowej. Aby odróżnić skończoną liczbę typów moglibyśmy zdefiniować osobne typy danych, np. `BoolExpr`, `IntegerExpr` itd. Niestety w języku kalkulatora typów jest nieskończenie wiele. Możemy sparametryzować typ `Expr` typem wyrażenia. Chcielibyśmy np. żeby `C True :: Expr Bool`, a

```
(:+) :: Expr Integer -> Expr Integer -> Expr Integer
```

Argumenty i wynik konstruktora `(: +)` mają zawężony typ. Definicja typu `Expr` a będzie więc zawierać rekursję niejednorodną, którą rozważaliśmy w zadaniach z poprzedniej listy. Tym razem jednak niejednorodność dotyczy nie tylko argumentów, ale też *wyniku* konstruktora `(: +)`. Używana przez nas do tej pory deklaracja `data` nie pozwala na taką niejednorodność. Haskell został zatem rozszerzony o dodatkową, ogólniejszą deklarację `data`, w której wymienia się nie same typy argumentów, tylko kompletne typy definiowanych konstruktorów:

```
data Expr a where
  C :: a -> Expr a
  P :: (Expr a, Expr b) -> Expr (a,b)
  Not :: Expr Bool -> Expr Bool
  (:+), (: -), (:*) :: Expr Integer -> Expr Integer -> Expr Integer
  (: / ) :: Expr Integer -> Expr Integer -> Expr (Integer,Integer)
  (<), (>), (<=), (>=), (!=), (==)
    :: Expr Integer -> Expr Integer -> Expr Bool
  (:&&), (||) :: Expr Bool -> Expr Bool -> Expr Bool
```

```
(:?) :: Expr Bool -> Expr a -> Expr a -> Expr a
Fst  :: Expr (a,b) -> Expr a
Snd  :: Expr (a,b) -> Expr b
```

Przypomnijmy, że identyfikatory symboliczne będące nazwami konstruktorów muszą rozpoczynać się znakiem `:`. W zapisie wyrażeń zawierających trójargumentowy operator `:?` możemy się posłużyć operatorem `$` w celu oddzielenia trzeciego argumentu, pisząc np. `b :? e1 $ e2`. Daje to w Haskellu namiastkę ternarnego operatora miksfiksowego. Należy jeszcze dodać dyrektywy ustalające priorytety operatorów:

```
infixl 6 :*, :/
infixl 5 :+, :-
infixl 4 :<, :>, :<=, :>=, :!=, :==
infixl 3 :&&
infixl 2 :||
infixl 1 :?
```

Wyrażenie z wcześniejszego przykładu zapisane w naszym osadzonym DSL-u wygląda następująco:

```
C 2 :* C 7 :> Fst (C 6 :/ C 3) :&& C 3 :< C 5 :?
  Snd (C 15 :/ C 12) :+ C 3 $ C 7 :* C 8
```

i ma typ `Expr Integer`.

Ponieważ typy wprowadzone deklaracją `data` nazywa się *algebraicznymi typami danych* (bo ich deklaracja, to w istocie definicja algebry wolnej o podanej sygnaturze), więc typy wprowadzone powyższą deklaracją nazwano *uogólnionymi algebraicznymi typami danych* (*Generalized Algebraic Data Types, GADT*).

Zaprogramuj ewaluator wyrażeń języka kalkulatora, tj. funkcję

```
eval :: Expr a -> a
```

*Uwaga:* zaprogramowanie funkcji `eval` nie wymaga rozwiązania jakichkolwiek problemów, jest śmiesznie łatwe, oczywiście, wręcz nudne. Prawdziwym celem zadania nie jest rozwój umiejętności programistycznych, tylko dostarczenie Rozwiązującemu ciekawego przeżycia emocjonalnego — podczas zapisywania funkcji `eval` (która jest w istocie izomorfizmem pomiędzy naszym językiem kalkulatora i niewielkim podzbiorem Haskellu) cały czas ma się wrażenie, że ta funkcja nie ma prawa się skompilować, gdyż zawiera błędy typowe — prawe strony różnych klauzul są różnych, niezgadnialnych typów. GADT, wbrew pozorom, są potężnym rozszerzeniem możliwości języka!

**Zadanie 2 (13 pkt).** W zadaniach z poprzedniej listy wyrażaliśmy niezmienniki strukturalne typów danych za pomocą typów niejednorodnych (pierwszego bądź wyższego rzędu). W wielu przypadkach zastosowanie tej techniki wiąże się ze sporymi komplikacjami. Przypomnijmy, że drzewa czerwono-czarne (rozważane już w zadaniach z listy 8), to binarne drzewa poszukiwań (*BST*) spełniające następujące niezmienniki strukturalne:

1. wierzchołki wewnętrzne są oznaczone kolorami czerwonym i czarnym,
2. każdy czerwony wierzchołek ma czarnego ojca,
3. na każdej ścieżce od korzenia do liścia jest tyle samo czarnych wierzchołków.

Z warunku 2 wynika, że na każdej ścieżce od korzenia do liścia jest nie więcej czerwonych wierzchołków niż czarnych, co w połączeniu z warunkiem 3 gwarantuje, że najdłuższa ścieżka w drzewie czerwono-czarnym jest co najwyżej dwa razy dłuższa od najkrótszej, a więc wysokość drzewa (długość najdłuższej ścieżki) jest logarytmiczna względem liczby wierzchołków.

Drzewa czerwono-czarne można traktować jak pełne drzewa binarne złożone z czarnych wierzchołków, w których pod niektórymi czarnymi wierzchołkami znajdują się pośrednie czerwone wierzchołki. Przypomnijmy, że pełne drzewa binarne reprezentowaliśmy na poprzedniej liście zadań następująco:

```
data Empty a = Empty
data Fork t a = Fork (t a) a (t a)
```

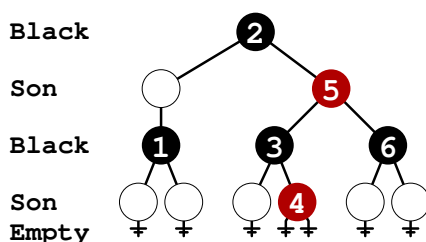
Aby ukryć rozmiar drzewa dołączaliśmy do niego „korzeń palowy” — ciąg konstruktorów Node długości równej wysokości drzewa i zakończony konstruktorem Leaf:

```
data Tree t a = Node (Tree (Fork t) a) | Leaf
```

Było to wygodne, gdy używaliśmy takich drzew w reprezentacjach numerycznych, ale jest niepraktyczne, jeśli pojedyncze drzewo stanowi kompletną strukturę danych. Wykorzystując technikę z poprzedniej listy moglibyśmy zdefiniować drzewa czerwono-czarne następująco:

```
data Empty a = Empty
data Black s a = Black (Son s a) a (Son s a)
data Son t a = Son (t a) | Red (t a) a (t a)
data Tree t a = Node (Tree (Black t) a) | Leaf (t a)
newtype RedBlackTree a = Tree Empty a
```

Przykładowe drzewo w tej reprezentacji wygląda tak:



a w notacji Haskellowej:

```
t = Black (Son $ Black (Son Empty) 1 (Son Empty))
      2
      (Red (Black (Son Empty) 3 (Red Empty 4 Empty))
        5
        (Black (Son Empty) 6 (Son Empty)))
  :: Black (Black Empty) Integer
```

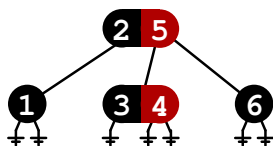
```
Node $ Node $ Leaf $ t :: Tree Empty Integer
```

```
RedBlackTree $ Node $ Node $ Leaf $ t :: RedBlackTree Integer
```

Drzewo, którego czarna wysokość (liczba czarnych wierzchołków na dowolnej ścieżce od korzenia do liścia) wynosi  $h$  ma w tej reprezentacji  $2h + 1$  poziomów. Możemy usunąć pośrednie wierzchołki typu Son łącząc konstruktory Son i Red z konstruktorem Black:

```
data Empty a = Empty
data Black s a = BlackBlack (s a) a (s a)
                  | RedBlack (s a) a (s a) a (s a)
                  | BlackRed (s a) a (s a) a (s a)
                  | RedRed (s a) a (s a) a (s a) a (s a)
```

Drzewo z poprzedniego przykładu zakodujemy używając tego typu następująco:



Zauważmy, że otrzymaliśmy w istocie 2-3-4-drzewa — przekształciliśmy drzewa czerwono-czarne w drzewa, w których wszystkie ścieżki od korzenia do liścia są tej samej długości. Za pomocą typów niejednorodnych potrafimy bowiem wyrażać tylko takie niezmienniki strukturalne. Informacja o wysokości drzewa

jest zakodowana wprost w definiowanym typie. Wygodniej byłoby zdefiniować osobny typ opisujący wysokość drzewa i sparametryzować nim typ drzew (moglibyśmy wówczas łatwo odróżnić czarne i czerwone wierzchołki).

Potrzebujemy nieskończenie wielu typów reprezentujących poszczególne liczby naturalne, ale nie zamierzamy korzystać z wartości tych typów. Typy te mogą nawet nie posiadać żadnych wartości poza  $\perp$ . Haskell pozwala na definiowanie takich typów — wystarczy pominąć w ich definicji znak równości i definicje konstruktorów:

```
data Zero :: *
data Succ :: * -> *
```

Typy nieposiadające innych wartości poza  $\perp$  nazywamy *typami fantomowymi*. Nie opisują one wprost żadnych danych, tylko służą jako parametry dla innych typów. Mamy typy: Zero, Succ Zero, Succ (Succ Zero) itd. Typ Zero reprezentuje wysokość drzewa pustego, a konstruktor typu Succ :: \* -> \* pozwala, dla ustalonej wysokości, wyznaczyć typ reprezentujący wysokość o jeden większą.

Rozważmy najpierw niezmiennik czarnej wysokości. Typ drzewa sparametryzujemy nie tylko typem etykiety, ale też typem fantomowym oznaczającym czarną wysokość drzewa. Do typu Tree h a powinny należeć drzewa elementów typu a o wysokości h. Konstruktor Black powinien mieć typ

```
Black :: Tree h a -> a -> Tree h a -> Tree (Succ h) a
```

Dodanie czarnego wierzchołka zwiększa czarną wysokość o 1, co jest wyrażone za pomocą typu Succ h w wyniku konstruktora Black. Parametr h w definicji typu Tree jest niejednorodny, a skoro występuje niejednorodnie także w wyniku konstruktora, to potrzebujemy GADT:

```
data Tree h a where
  Black :: Tree h a -> a -> Tree h a -> Tree (Succ h) a
  Red   :: Tree h a -> a -> Tree h a -> Tree h a
  Empty :: Tree Zero a
```

Aby wyrazić warunek, że synami czerwonego wierzchołka nie są czerwone wierzchołki wprowadzimy dwa dodatkowe typy fantomowe:

```
data Red :: *
data Black :: *
```

i sparametryzujemy dodatkowo typ Tree typem oznaczającym kolor korzenia. Drzewa typu Tree Red h a to drzewa elementów typu a o wysokości h zawierające w korzeniu czerwony wierzchołek. Drzewa typu Tree Black h a zawierają w korzeniu czarny wierzchołek lub są puste. Definicja typu Tree przyjmuje ostatecznie postać

```
data Tree :: * -> * -> * -> * where
  Empty :: Tree Black Zero a
  Black :: Tree c1 h a -> a -> Tree c2 h a -> Tree Black (Succ h) a
  Red   :: Tree Black h a -> a -> Tree Black h a -> Tree Red h a
```

Aby poprawnie skompilować powyższe definicje, trzeba włączyć następujące rozszerzenia standardu Haskell'a:

```
{-# LANGUAGE KindSignatures, GADTs #-}
```

Drzewo z poprzednich przykładów (nazwijmy je t) zapiszemy teraz jak zwykłe drzewo czerwono-czarne:

```
Black (Black Empty 1 Empty)
      2
      (Red (Black Empty 3 (Red Empty 4 Empty)) 5 (Black Empty 6 Empty))
```

Ma ono typ

```
Tree Black (Succ (Succ Zero)) Integer
```

Parametr Black informuje, że korzeń tego drzewa nie jest czerwony, a parametr Succ (Succ Zero) — że czarna wysokość tego drzewa jest równa 2.

Wszystkie zmienne typowe występujące wolno w typach argumentów konstruktorów w klasycznej deklaracji data muszą wystąpić jako parametry definiowanego typu. W przeciwnym przypadku, jeśli używamy klasycznego algorytmu rekonstrukcji typów, można by napisać funkcję identyczności typu  $a \rightarrow b$ , tj. załamać system typów:

```
data T = C a
id a = unC (C a) where unC (C a) = a
```

W klasycznym algorytmie rekonstrukcji typów wzorce w definicjach funkcji są typowane tak samo, jak zwykłe wyrażenia (co jest nienaturalne biorąc pod uwagę, że opisują one nie wynik, tylko argument funkcji, a więc występują na pozycji negatywnej). Jeśli we wzorcach występują konstruktory polimorficzne, prowadzi to do problemów. Algorytm rekonstrukcji typów w Haskellu działa w takich przypadkach znacznie bardziej naturalnie i nie wymaga takich ograniczeń. Najlepiej korzystać z notacji GADT, w której nie nakłada się żadnych ograniczeń na zmienne występujące w typach konstruktorów (zmienne którymi jest sparametryzowany konstruktor typu po słowie kluczowym data wskazują jedynie rodzaj typu i najlepiej je pominąć podając rodzaj typu *explicite*). W Haskellu możemy napisać:

```
data T where C :: a -> T
```

Oczywiście próba zdefiniowania funkcji unC skończy się błędem. Rozważmy typ

```
data T where C :: [a] -> T
```

Możemy zdefiniować

```
len :: T -> Int
len (C xs) = length xs
```

ale próba ujawnienia elementu listy opakowanej konstruktorem C, np. tak:

```
hd :: T -> a
hd (C xs) = head xs
```

zakończy się błędem typowym.

Płytkie kwantyfikatory ogólne zwykle się w Haskellu opuszcza pisząc po prostu  $C :: [a] \rightarrow T$ , ale faktycznie typem konstruktora C jest  $\forall a. [a] \rightarrow T$ . Ponieważ dla dowolnych formuł  $\phi$  i  $\psi$  w logice intuicjonistycznej zachodzi równoważność  $(\forall x. (\phi \rightarrow \psi)) \Leftrightarrow ((\exists x. \phi) \rightarrow \psi)$ , to typ ten jest izomorficzny z typem  $C :: (\exists a. [a]) \rightarrow T$ . Wyrażenie typowe  $\exists a. [a]$  oznacza „listę elementów pewnego typu”. Dla porównania  $[] :: \forall a. [a]$ , a zatem typ konstruktora  $[]$  oznacza „listę elementów dowolnego typu”. Gdyby konstruktor C miał typ  $(\forall a. [a]) \rightarrow T$ , to jedynym poprawnym typowo wyrażeniem (poza, oczywiście,  $C \perp$ ) byłoby  $C []$ . Typy uniwersalne służą do wyrażania polimorfizmu, typy egzystencjalne zaś służą do definiowania abstrakcyjnych typów danych, tj. ukrywania (pewnych) detali implementacyjnych — w powyższym przykładzie wiemy, że C xs jest listą, ale nie wiemy jakich elementów. Możemy na tej liście operować tak długo, jak długo nie żądamy ujawnienia jej elementów.

Typów egzystencjalnych możemy użyć do ukrycia wysokości drzewa:

```
data RedBlackTree :: * -> * where
  RedBlackTree :: Tree Black h a -> RedBlackTree a
```

Mamy teraz:

```
RedBlackTree t :: RedBlackTree Integer
```

W definicjach konstruktorów `Red`, `Black` i `Empty` próbowaliśmy odtworzyć warunek 2 drzew czerwono-czarnych, ale faktycznie zakodowaliśmy słabszy warunek: synem czerwonego wierzchołka jest wierzchołek nieczerwony. Z oryginalnego warunku wynika dodatkowo, że korzeń całego drzewa jest czarny. Wyraziliśmy ten dodatkowy warunek w typie konstruktora `RedBlackTree`.

Typy egzystencjalne pojawiły się już w chwili definiowania konstruktora `Black` (co wówczas dyskretnie przemilczeliśmy) w postaci zmiennych typowych `c1` i `c2` oznaczających kolory poddrzew. Ponieważ kolory te mogą być dowolne, zignorowaliśmy je.

Zaprogramuj funkcje:

```
find :: Ord a => a -> RedBlackTree a -> Bool
insert :: Ord a => a -> RedBlackTree a -> RedBlackTree a
delete :: Ord a => a -> RedBlackTree a -> RedBlackTree a
flatten :: RedBlackTree a -> [a]
```

Przyjmij, że elementy są wstawiane do drzewa bez powtórzeń.

**Zadanie 3 (bonus 15 pkt).** Niezmienniki strukturalne drzew czerwono-czarnych możemy wyrazić używając klas typów zamiast typów niejednorodnych. Podobnie jak w zadaniach z poprzedniej listy możemy spróbować zdefiniować trzy nierekurencyjne typy danych opisujące czarny, czerwony i pusty wierzchołek:

```
data Black :: (* -> *) -> (* -> *) -> * -> * where
  Black :: l a -> a -> r a -> Black l r a
data Red   :: (* -> *) -> (* -> *) -> * -> * where
  Red     :: l a -> a -> r a -> Red l r a
data Empty :: * -> * where
  Empty :: Empty a
```

(typy `Black` i `Red` odpowiadają typowi `Fork` z poprzedniej listy). Notacja GADT została użyta powyżej wyłącznie dla czytelności, choć nie korzystamy z żadnych rozszerzeń przez nią wprowadzonych. Aby drzewo `Black l r a` było poprawnie zbudowanym drzewem czerwono-czarnym, drzewa `l` i `r` powinny być poprawnie zbudowanymi drzewami o tej samej czarnej wysokości. Możemy więc zdefiniować binarną klasę typów:

```
class SameHeight (t1 :: * -> *) (t2 :: * -> *)
instance SameHeight Empty Empty
instance SameHeight Empty (Red Empty Empty)
instance (SameHeight l1 l2, SameHeight r1 r2, SameHeight l1 r1) =>
  SameHeight (Black l1 r1) (Black l2 r2)
instance (SameHeight (Black l1 r1) l2, SameHeight l2 r2) =>
  SameHeight (Black l1 r1) (Red l2 r2)
instance (SameHeight l1 r1, SameHeight t l1) => SameHeight (Red l1 r1) t
```

Podane definicje instancji tej klasy wyglądają jak program w Prologu, który sprawdza, czy podane drzewa są tej samej wysokości. W odróżnieniu od Prologu głowy klauzul powinny być rozłączne (inaczej spowodujemy błąd *Overlapping instances*), dlatego musimy rozpatrzyć aż pięć przypadków.

Zdefiniowane powyżej klasy typów są czystymi zbiorami typów — nie posiadają żadnych metod. Takie klasy, analogicznie do typów, nazywamy *klasami fantomowymi*.

Możemy następnie zdefiniować klasy typów `Tree` i `NotRed`, do których będą należeć, odpowiednio, wszystkie drzewa i drzewa nieposiadające w korzeniu czerwonego wierzchołka:

```
class Tree (t :: * -> *)
instance Tree (Red l r)
instance Tree (Black l r)
instance Tree Empty

class NotRed (t :: * -> *)
instance NotRed (Black l r)
instance NotRed Empty
```

Do definicji typów `Black`, `Red` i `Empty` powinniśmy dodać więzy określające przynależność poddrzew do odpowiednich klas:

```
data Black :: (* -> *) -> (* -> *) -> * -> * where
  Black :: (Tree l, Tree r, SameHeight l r) => l a -> a -> r a -> Black l r a
data Red :: (* -> *) -> (* -> *) -> * -> * where
  Red :: (NotRed l, NotRed r, SameHeight l r) => l a -> a -> r a -> Red l r a
data Empty :: * -> * where
  Empty :: Empty a
```

Podobnie jak w poprzednim zadaniu używając typów egzystencjalnych możemy zataić informacje o szczegółach budowy drzewa:

```
data RedBlackTree :: * -> * where
  RedBlackTree :: NotRed t => t a -> RedBlackTree a
```

Zaprogramuj funkcje `find`, `insert`, `delete` i `flatten` dla typu `RedBlackTree`.

**Zadanie 4 (5 pkt).** Pierwsza lista zadań na pracownię z bieżącego przedmiotu dla wybranych grup zawierała polecenie zaprogramowania w (ohydny) języku Ocaml operacji arytmetycznych na tzw. *liczebnikach Churcha*, tj. funkcjach postaci  $\lambda f x \rightarrow f^n x$ . Łatwo pokazać, że w języku typu *strict* liczebniki Churcha wyczerpują zbiór wartości typu  $(a \rightarrow a) \rightarrow (a \rightarrow a)$  (zbiór wartości tego typu w języku *non-strict*, takim jak Haskell, jest obszerniejszy). Typ  $(a \rightarrow a) \rightarrow (a \rightarrow a)$  jest więc (w języku *strict*) izomorficzny ze zbiorem liczb naturalnych. Możemy z łatwością zaprogramować operacje arytmetyczne na wartościach tego typu:

```
zero = \ f x -> x
one = \ f x -> f x
two = \ f x -> f (f x)
succ n = \ f -> f . n f
add n m = \ f -> n f . m f
mul n m = n . m
```

Zauważmy, że `one = id` a `id` jest elementem neutralnym składania funkcji (zatem `one` jest elementem neutralnym mnożenia). Podobnie `zero = flip const`, a `flip const f . g = g` (zatem `zero` jest elementem neutralnym dodawania i pochłaniającym mnożenia).

Aby móc zainstalować typ  $(a \rightarrow a) \rightarrow (a \rightarrow a)$  w różnych klasach, moglibyśmy dorobić mu „czapkę”:

```
newtype Church a = Church ((a -> a) -> (a -> a))
```

i dodać odpowiednie sygnatury:

```
zero :: Church a
zero = Church $ \ f x -> x
one :: Church a
one = Church $ \ f x -> f x
two :: Church a
two = Church $ \ f x -> f (f x)
succ :: Church a -> Church a
succ (Church n) = Church $ \ f -> f . n f
add :: Church a -> Church a -> Church a
add (Church n) (Church m) = Church $ \ f -> n f . m f
mul :: Church a -> Church a -> Church a
mul (Church n) (Church m) = Church $ n . m
```



Niestety szczęście kończy się szybko. Próba zdefiniowania operacji potęgowania:

```
exp :: Church a -> Church a -> Church a
exp (Church n) (Church m) = Church $ m n
```

(zauważmy, że  $\text{mul} = (.)$ , zaś  $\text{exp} = \text{flip } (\$)$ ) kończy się błędem typowym. Najogólniejszym typem funkcji  $\lambda n m \rightarrow m n$  jest  $a \rightarrow (a \rightarrow b) \rightarrow b$ , który nie jest unifikowalny z typem  $(c \rightarrow c) \rightarrow (c \rightarrow c)$ . Ostatnia deklaracja daje się skompilować, jeśli nadamy jej sygnaturę

```
exp :: Church a -> Church (a -> a) -> Church a
```

ale w ten sposób tylko odwlekamy, a nie rozwiązujemy problem. Nie da się np. zdefiniować funkcji  $n \rightarrow n^n$ , musielibyśmy bowiem przypisać typ funkcji  $\lambda n \rightarrow n n$ . Zauważmy, że nie ma problemu z otypowaniem wyrażenia

```
let f x = x in f f
```

gdyż w chwili typowania wyrażenia  $f f$  zmienna  $f$  ma przypisany typ  $\forall a. a \rightarrow a$ . Ten rodzaj polimorfizmu nosi z resztą nazwę let-polimorfizmu. W klasycznych językach jest to jedyny dostępny rodzaj polimorfizmu parametrycznego. Wystąpienia wszystkich zmiennych z wyjątkiem tych, które są związane deklaracją `let` lub jej równoważną są typowane monomorficznie. Na poprzedniej liście rozważaliśmy rekursję polimorficzną, w której funkcja definiowana rekurencyjnie jest typowana polimorficznie. Warto teraz rozważyć, czy moglibyśmy polimorficznie typować argumenty funkcji. To jest tzw. *polimorfizm wyższego rzędu*. Rekonstrukcja typów dla wszystkich wspomnianych rozszerzeń jest nierozstrzygalna, dlatego zawsze w takich przypadkach należy jawnie podawać sygnatury zmiennych.

Jeśli tylko włączymy rozszerzenie

```
{-# LANGUAGE Rank2Types #-}
```

to deklaracja

```
omega :: (forall a. a) -> b
omega f = f f
```

daje się poprawnie skompilować. (Jeśli kwantyfikator ogólny nie występuje preneksowo w argumentach funkcji tylko jest głębiej zagnieżdżony, potrzeba włączyć rozszerzenie `RankNTypes`.)

Możemy teraz zdefiniować liczebniki Churcha następująco:

```
newtype Church = Church (forall a. (a -> a) -> (a -> a))
```

Zauważmy, że w odróżnieniu od poprzedniej definicji argumentem konstruktora `Church` może być tylko liczebnik Churcha, tj. funkcja posiadająca typ  $(a \rightarrow a) \rightarrow (a \rightarrow a)$ . Poprzednio np. typem wyrażenia `Church (\ f x -> x ++ g x)` był typ `Church [a]`. Obecnie powyższe wyrażenie jest niepoprawne typowo.

Zainstaluj typ `Church` w klasach `Eq`, `Ord`, `Show` i `Num`. W ostatnim przypadku przyjmij, że  $n - m = 0$  jeśli  $n \leq m$ .

Zauważ, że zainstalowanie typu `Church` w klasach `Num` i `Show` pozwala używać liczebników Churcha tak jak zwykłych liczb:

```
*Main> 2 + 3 * 4 :: Church
14
```