Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 6

Do zgłoszenia w SKOS-ie do 17 kwietnia 2020

Zadania różne

Zadanie 1 (2 pkt). Zaprogramuj zwięźle listę

```
natPairs :: [(Integer,Integer)]
```

zawierającą wszystkie pary nieujemnych liczb całkowitych w porządku przekątniowym Cantora:

$$[\quad (0,0),\quad (0,1),(1,0),\quad (0,2),(1,1),(2,0),\quad (0,3),(1,2),(2,1),(3,0),\quad \ldots]$$

Wskazówka: użyj *list comprehensions*. Zaprogramuj następnie funkcję

```
(><) :: [a] -> [b] -> [(a,b)]
```

wyznaczającą produkt dwóch list w porządku przekątniowym Cantora, tj.

$$[x_1, x_2, x_3, \ldots] \texttt{ >< } [y_1, y_2, y_3, \ldots] \quad = \quad [(x_1,y_1),(x_1,y_2),(x_2,y_1),(x_1,y_3),(x_2,y_2),(x_3,y_1),\ldots]$$

Zauważ, że listy mogą być skończone, bądź nie, np. możemy napisać

```
natPairs' :: [(Integer,Integer)]
natPairs' = [0..] >< [0..]
```

ale też np.

```
xs = [1..13] >< [1..42]
```

Postaraj się, by swoje rozwiązanie napisać pięknie.

Polecenie 1. Przeczytaj podrozdział *3.15 Datatypes with Field Labels* oraz punkt *Labelled Fields* z podrozdziału *4.2.1 Algebraic Datatype Declarations* definicji języka *Haskell 2010 Language Report*. Odpowiedz następnie na pytanie, które z poniższych deklaracji są poprawne. Jakie jest ich znaczenie?

```
data C = F {f1,f2 :: Int, f3 :: Bool}
f (F {f1=m, f2=n, f3=True}) = m+n
g (F n m True) = m+n
h (F {f3=False, f1=m}) = m
data D = G {f2 :: Int} | H {f1 :: Bool, f2 :: Int}
data E = J | K Int Int Int
k (K {}) = 1
x = H {f1=False}
y = x {f2=5}
```

Przejrzyj następnie podrozdział *9.5 Extensions to the record system* dokumentacji kompilatora *GHC User's Guide Documentation*.

Zadanie 2 (1 pkt). W zadaniu 6 z listy 3 rozważaliśmy klasę zbiorów:

```
class Set s where
   empty  :: s a
   search :: Ord a => a -> s a -> Maybe a
   insert :: Ord a => a -> s a -> s a
   delMax :: Ord a => s a -> Maybe (a, s a)
   delete :: Ord a => a -> s a -> s a
```

Definiując instancje tej klasy zwracaliśmy uwagę (nie wszyscy zwracali!), że metoda `(<=)` klasy `Ord` może być tylko *pra*-porządkiem. Dlatego metoda `search` zwracała wyszukany, a nie wyszukiwany element, a metoda `insert` aktualizowała element, jeśli już znajdował się w zbiorze. Takie zbiory możemy wykorzystać do zaprogramowania *słowników* (map):

```
class Dictionary d where
   emptyD  :: d k v
   searchD :: Ord k => k -> d k v -> Maybe v
   insertD :: Ord k => k -> v -> d k v -> d k v
   deleteD :: Ord k => k -> d k v -> d k v
```

Możemy bowiem przechowywać w słowniku pary:

```
data KeyValue key value = KeyValue { key :: key, value :: value }
```

i zrobić odwzorowanie, które zamienia zbiór w słownik:

```
newtype SetToDict s k v = SetToDict (s (KeyValue k v))
```

Dokończ definicję tej transformacji zbiorów w słowniki, tj. dokończ definicję instancji

```
instance Set s => Dictionary (SetToDict s) where ...
```

Pamiętaj przy tym o eleganckim wykorzystaniu różnych wariantów składni dla nazwanych pól rekordów!

Schemat rekursji prostej

Rozważmy zasadę indukcji znaną z kursu *Logiki dla Informatyków*:

$$\frac{\Phi(0) \quad \forall n \in \mathbb{N}.\Phi(n) \Rightarrow \Phi(n+1)}{\forall n \in \mathbb{N}.\Phi(n)}$$

Towarzyszy jej *twierdzenie o definiowaniu przez indukcję*:

> Niech $g : \mathbb{N}^m \to \mathbb{N}$ i $h : \mathbb{N}^{m+2} \to \mathbb{N}$ będą dowolnymi funkcjami całkowitymi. Istnieje wówczas dokładnie jedna funkcja całkowita $f : \mathbb{N}^{m+1} \to \mathbb{N}$ spełniająca równości:
>
> $$\begin{aligned} f(0, x_1, \ldots, x_m) &= g(x_1, \ldots, x_m), && \text{dla } x_1, \ldots, x_m \in \mathbb{N} \\ f(n+1, x_1, \ldots, x_m) &= h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m), && \text{dla } n, x_1, \ldots, x_m \in \mathbb{N} \end{aligned}$$

Twierdzenie to jest dosyć stare, pochodzi od Richarda Dedekinda (1888). Możemy zatem rozważyć klasę funkcji liczbowych zdefiniowanych konstruktywnie w następujący sposób:

- wybrać pewne proste funkcje jako przypadki bazowe indukcji,
- umieć je dowolnie składać,
- móc budować nowe funkcje za pomocą schematu rekursji.

Tak dochodzimy do definicji *funkcji pierwotnie rekurencyjnych*. Niech $\mathcal{PR}^m \subseteq \mathbb{N}^{\mathbb{N}^m}$ oznacza zbiór m-argumentowych funkcji pierwotnie rekurencyjnych. Niech:

- $Z, S \in \mathcal{PR}^1$, gdzie $Z(n) = 0$, $S(n) = n + 1$ dla $n \in \mathbb{N}$

- $P_i^n \in \mathcal{PR}^n$ dla $1 \leq i \leq n$, gdzie $P_i^n(x_1, \ldots, x_i, \ldots, x_n) = x_i$, $n \in \mathbb{N}$
- $C_m^n[f, g_1, \ldots, g_m] \in \mathcal{PR}^n$ dla $f \in \mathcal{PR}^m$ oraz $g_1, \ldots, g_m \in \mathcal{PR}^n$, gdzie
  $C_m^n[f, g_1, \ldots, g_m](x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ dla $x_1, \ldots, x_n \in \mathbb{N}$.
- $R[g, h] \in \mathcal{PR}^{m+1}$ dla $g \in \mathcal{PR}^m$ oraz $h \in \mathcal{PR}^{m+2}$, gdzie
  $R[g, h](0, x_1, \ldots, x_m) = g(x_1, \ldots, x_m)$ dla $x_1, \ldots, x_m \in \mathbb{N}$ oraz
  $R[g, h](n+1, x_1, \ldots, x_m) = h(n, R[g, h](n, x_1, \ldots, x_m), x_1, \ldots, x_m)$ dla $n, x_1, \ldots, x_m \in \mathbb{N}$.

Na mocy twierdzenia o definiowaniu przez indukcję powyższa definicja jest poprawna, a dla dowolnego $m > 0$ rodzina $\mathcal{PR}^m$ jest zbiorem pewnych całkowitych funkcji m-argumentowych.

Polecenie 2. Odpowiedz na pytanie: czym jest funkcja $R[P_1^1, C_1^3[S, P_2^3]]$?

Na początku XX wieku wydawało się, że funkcje pierwotnie rekurencyjne są dobrą formalizacją pojęcia funkcji obliczalnych. Jednak w 1928 Wilhelm Ackermann zdefiniował funkcję $\mathfrak{A} : \mathbb{N} \to \mathbb{N}$, całkowitą i w oczywisty sposób obliczalną, która nie należy do $\mathcal{PR}^1$, tj. nie jest pierwotnie rekurencyjna. Funkcja ta jest zdefiniowana następująco:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m+1, 0) &= A(m, 1) \\ A(m+1, n+1) &= A(m, A(m+1, n)) \end{aligned}$$

$$\mathfrak{A}(m) = A(m, m)$$

dla $m, n \in \mathbb{N}$. Widać tu wyraźnie metodę przekątniową: aby obliczyć $\mathfrak{A}(m)$, trzeba m-krotnie zastosować schemat rekursji prostej. Nie przedstawimy więc tej funkcji za pomocą złożenia skończonej liczby schematów rekursji prostej.
Niech wartości typu

```
data PrimRec = Zero | Succ | Proj Int Int
             | Comb PrimRec [PrimRec] | Rec PrimRec PrimRec
```

przedstawiają funkcje pierwotnie rekurencyjne. Oczywiście typ ten nie jest wolny, gdyż wiele jego wartości nie opisuje poprawnie zbudowanych funkcji pierwotnie rekurencyjnych.

Zadanie 3 (1 pkt). Zaprogramuj funkcję

```
arityCheck :: PrimRec -> Maybe Int
```

sprawdzającą, czy wyrażenie jest poprawnie zbudowane i zwracającą arność danej funkcji pierwotnie rekurencyjnej. Zauważ, że `Proj 2 1` też nie jest poprawne.

Rodzinę funkcji pierwotnie rekurencyjnych rozbiliśmy na nieskończony ciąg zbiorów funkcji o ustalonej arności. Aby móc za pomocą systemu typów kontrolować poprawność budowy wyrażeń opisujących funkcje pierwotnie rekurencyjne, musielibyśmy zdefiniować typ `PrimRec :: Nat -> Type`, tj. użyć rozszerzenia `DataKinds`. Być może zrobimy to na kolejnych listach.

Zadanie 4 (1 pkt). Zaprogramuj funkcję

```
evalPrimRec :: PrimRec -> [Integer] -> Integer
```

Obliczenie powinno zakończyć się błędem, jeśli wyrażenie opisujące funkcję pierwotnie rekurencyjną nie jest poprawne, jego arność jest inna niż długość listy liczb lub gdy wśród tych liczb jest liczba ujemna.

Kurt Gödel postanowił połączyć schemat rekursji prostej z rachunkiem funkcji wyższych rzędów (Rachunkiem Lambda), tworząc System T. Składnia typów i wyrażeń tego systemu jest następująca:

$$\begin{aligned} \sigma &::= \texttt{Nat} \mid \sigma_1 \to \sigma_2 \\ e &::= \texttt{Z} \mid \texttt{S} \mid \texttt{rec} \mid e_1\, e_2 \mid \lambda x.e \end{aligned}$$

Mamy funkcje zera i następnika, projekcje i złożenia załatwia sam rachunek lambda, a dla schematu rekursji prostej mamy specjalną stałą. Reguły typowania:

$$\begin{aligned} \texttt{Z} &:: \texttt{Nat} \\ \texttt{S} &:: \texttt{Nat} \to \texttt{Nat} \\ \texttt{rec} &:: (\texttt{Nat} \to \sigma \to \sigma) \to \sigma \to \texttt{Nat} \to \sigma \end{aligned}$$

$$\frac{e_1 :: \sigma \to \tau \quad e_2 :: \sigma}{e_1\, e_2 :: \tau}$$

$$\frac{\boxed{\begin{array}{c}\text{zał. } x :: \sigma \\ \vdots \\ e :: \tau\end{array}}}{\lambda x.e :: \sigma \to \tau}$$

Reguły redukcji:

$$\begin{aligned} (\lambda x.e_1)\, e_2 &\to e_1[x/e_2] \\ \lambda x.(e\, x) &\to e, \quad \text{jeśli } x \notin \mathrm{FV}(e) \\ \texttt{rec}\, f\, g\, \texttt{Z} &\to g \\ \texttt{rec}\, f\, g\, (\texttt{S}\, \texttt{e}) &\to h\, e\, (\texttt{rec}\, f\, g\, e) \end{aligned}$$

Polecenie 3. Jaką funkcję definiuje wyrażenie $\texttt{rec}\,(\lambda x.\texttt{S})$?

System T Gödla jest uogólnieniem funkcji rekurencyjnych na funkcje wyższych rzędów. Jest to znacznie szersza klasa funkcji definiowalnych! Faktycznie trudno znaleźć „naturalną" funkcję obliczalną, która nie należałaby do tego systemu. Znane przykłady funkcji spoza Systemu T są bardzo sztuczne i skomplikowane, zwykle zbudowane za pomocą paradoksu kłamcy („interpreter systemu T nie jest funkcją należącą do T"). Dla przykładu funkcja Ackermanna należy do Systemu T. Mamy:

$$\begin{aligned} A_0(n) &= n+1 \\ A_{m+1}(0) &= A_m(1) \\ A_{m+1}(n+1) &= A_m(A_{m+1}(n)) \end{aligned}$$

A zatem:

$$\begin{aligned} A_0 &= \texttt{S} \\ A_{m+1} &= \texttt{rec}\,(\lambda x.A_m)\,(A_m(\texttt{S}\,\texttt{Z})) = (\lambda a.\lambda b.\texttt{rec}\,(\lambda x.b)\,(b(\texttt{S}\,\texttt{Z})))\,m\,(A_m) \end{aligned}$$

Stąd:

$$\mathfrak{A} = \texttt{rec}\,(\lambda a.\lambda b.\texttt{rec}\,(\lambda x.\lambda y.b(y))\,(b\,(\texttt{S}\,\texttt{Z})))\,\texttt{S}$$

System T daje się naturalnie wyrazić w Haskellu:

```
data Nat = S Nat | Z
rec :: (Nat -> a -> a) -> a -> Nat -> a
rec _ g Z = g
rec f g (S n) = f n (rec f g n)

add :: Nat -> Nat -> Nat
add = rec (const S)

mult :: Nat -> Nat -> Nat
mult n = rec (const (add n)) Z

pred :: Nat -> Nat
pred = rec const Z

ackermann :: Nat -> Nat -> Nat
ackermann = rec (\ _ b -> rec (const b) (b (S Z))) S
```

Schemat rekursji prostej realizuje tu nie specjalna konstrukcja języka, tylko zwykła funkcja wyższego rzędu, zwana *rekursorem*. Udowodniono, że funkcja całkowita jest definiowalna w Systemie T wtedy i tylko wtedy, gdy można o niej w arytmetyce drugiego rzędu udowodnić, że jest całkowita. To jest bardzo szeroka klasa funkcji. Innymi słowy rekursor jest potężnym narzędziem do definiowania funkcji.

Parametr `n` w definicji rekursora jest często zbędny. Możemy go opuścić, zamieniając definicję rekursora w definicję *iteratora*:

```
iter :: (a -> a) -> a -> Nat -> a
iter _ g Z = g
iter f g (S n) = f (iter f g n)
```

Zauważmy, że `iter` $f\, g\, n = f^n\, g$. Innymi słowy w wyrażeniu $n =$ `S`^{n} `Z` iterator wymienia wszystkie wystąpienia `S` na funkcję f, zaś wystąpienie `Z` na g. Nie jest to przypadek — iteratory dla wszystkich typów tak robią.

Dodawanie, mnożenie i funkcję Ackermanna można łatwo zdefiniować za pomocą iteratora, gdyż w ich definicjach za pomocą rekursora i tak pomijaliśmy dodatkowy parametr `n`. Mamy:

```
add = iter S
mult n = iter (add n) Z
ackermann = iter (\ b -> iter b (b (S Z))) S
```

Zdefiniowanie poprzednika:

```
pred = rec const Z
```

za pomocą iteratora wymaga pewnej pomysłowości — obliczenie prowadzimy na parach liczb:

```
pred = fst . iter (\ (x,y) -> (y, S y)) (Z,Z)
```

Zadanie 5 (1 pkt). Zdefiniuj rekursor za pomocą iteratora (i oczywiście bez jawnej rekursji).

Zatem iteratory definiują tę samą klasę funkcji, co rekursory, niekiedy jedynie mniej wygodnie.

Niech X będzie dowolnym zbiorem, a $\mathcal{F} = \{F_n\}_{n\in\mathbb{N}}$, gdzie

- $F_0 \subseteq X$,
- $F_n \subseteq X^{X^n}$ dla $n \in \mathbb{N}^+$.

Zbiór $Y \subseteq X$ jest *indukcyjny względem* $\mathcal{F}$, jeśli:

- $F_0 \subseteq Y$,
- dla dowolnych $n \in \mathbb{N}^+$, $f \in F_n$ i $y_1, \ldots, y_n \in Y$ zachodzi $f(y_1, \ldots, y_n) \in Y$.

Podzbiorem generowanym w X przez $\mathcal{F}$ (ozn. $\mathcal{G}(\mathcal{F})$) nazywamy najmniejszy podzbiór X indukcyjny względem $\mathcal{F}$. Ta definicja jest poprawna: przekrój niepustej rodziny podzbiorów X indukcyjnych względem $\mathcal{F}$ jest indukcyjny. Rodzina wszystkich podzbiorów X indukcyjnych względem $\mathcal{F}$ jest niepusta, bo należy do niej X. Dla podzbiorów generowanych słuszna jest następująca *Zasada indukcji strukturalnej:*

> Jeśli zbiór $Y \subseteq X$ jest indukcyjny względem $\mathcal{F}$, to $\mathcal{G}(\mathcal{F}) \subseteq Y$.

Dowód: z definicji $\mathcal{G}(\mathcal{F})$ jest najmniejszym zbiorem indukcyjnym, zawiera się zatem w każdym zbiorze indukcyjnym.

Niech Σ_n będzie zbiorem symboli funkcyjnych o arności n, zaś $\mathcal{X}$ będzie zbiorem zmiennych. Zbiór $\mathcal{T}(\Sigma, \mathcal{X})$ *termów* nad sygnaturą $\Sigma = \{\Sigma_n\}_{n\in\mathbb{N}}$ ze zmiennymi ze zbioru $\mathcal{X}$, to najmniejszy zbiór spełniający warunki:

- $\Sigma_0 \subseteq \mathcal{T}(\Sigma, \mathcal{X})$,
- dla dowolnego $n \in \mathbb{N}^+$, dowolnego symbolu $f \in \Sigma_n$ i dowolnych $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X})$ zachodzi $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$.

Dla termów mamy w szczególności następującą zasadę indukcji strukturalnej:

$$\frac{\begin{array}{l}\forall x \in \mathcal{X}.\Phi(x) \\ \forall c \in \Sigma_0.\Phi(c) \quad \forall n \in \mathbb{N}^+.\forall f \in \Sigma_n.\forall t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X}).(\Phi(t_1) \wedge \ldots \wedge \Phi(t_n) \Rightarrow \Phi(f(t_1, \ldots, t_n)))\end{array}}{\forall t \in \mathcal{T}(\Sigma, \mathcal{X}).\Phi(t)}$$

Deklaracja

```
data Nat = S Nat | Z
```

definiuje algebrę termów stałych nad sygnaturą $\Sigma_0 = \{\texttt{Z}\}$ i $\Sigma_1 = \{\texttt{S}\}$, mamy dla nich zatem zasadę indukcji:

$$\frac{\Phi(\texttt{Z}) \quad \forall n \in \texttt{Nat}.(\Phi(n) \Rightarrow \Phi(\texttt{S}(n)))}{\forall n \in \texttt{Nat}.\Phi(n)}$$

Mamy też naturalnie (dowodliwe przez indukcję strukturalną) twierdzenie o definiowaniu przez indukcję dla skończonych wartości typu `Nat`:

> Dla dowolnej funkcji $g :: \texttt{Nat} \to \sigma \to \sigma$ określonej dla skończonych wartości typu `Nat` i stałej $c :: \sigma$ istnieje dokładnie jedna funkcja $f :: \texttt{Nat} \to \sigma$ określona dla skończonych wartości typu `Nat` spełniająca równości:
>
> $$\begin{aligned} f(\texttt{Z}) &= c \\ f(\texttt{S}(n)) &= h\, n\, (f(n)) \end{aligned}$$

Uwaga: typ `Nat` ma w Haskellu poza skończonymi wartościami, także wartości częściowe i jedną wartość nieskończoną! Rozważmy teraz Haskellowe listy:

```
data [a] = (:) a [a] | []
```

Mamy:

```
(x:) :: [a] -> [a], dla x :: a
 []  :: [a]
```

zatem $\Sigma_0 = \{\texttt{[]}\}$ i $\Sigma_1 = \{(x\texttt{:}) \mid x :: a\}$, a stąd mamy zasadę indukcji strukturalnej dla list (skończonych):

$$\frac{\Phi(\texttt{[]}) \quad \forall x :: a.\forall xs :: \texttt{[}a\texttt{]}.\Phi(xs) \Rightarrow \Phi(x\texttt{:}xs)}{\forall xs :: \texttt{[}a\texttt{]}.\Phi(xs)}$$

Iterator dla list jest bardzo znajomą funkcją:

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

Wiele funkcji w naturalny sposób wyrażamy za pomocą iteratora, np.:

```
(++) :: [a] -> [a] -> [a]
(++) = flip $ foldr (:)

length :: [a] -> Int
length = foldr (const (+1)) 0

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\ x xs -> if p x then x:xs else xs) []

map :: (a -> b) -> ([a] -> [b])
map f = foldr ((:) . f) []

sum :: [Integer] -> Integer
sum = foldr (+) 0
```

Nie jest to przypadkiem — iterator dla list jest równie potężny, jak iterator w Systemie T. Trudno podać przykład obliczalnej funkcji całkowitej działającej na listach, której nie dałoby się zaprogramować za pomocą `foldr`, choć wiele funkcji ma nieczytelne definicje, a niekiedy dochodzi też do pogorszenia złożoności obliczeniowej.

Zadanie 6 (1 pkt). Zaprogramuj za pomocą `foldr` funkcje `tail`, `reverse` i `zip`.

Iterator zamienia w wyrażeniu konstruktory na podane funkcje. Rozważmy definicję `foldr`, w której funkcja, będąca argumentem `foldr`, jest przedstawiona w postaci operatora infiksowego łączącego w prawo:

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

Mamy zatem:

$$\texttt{foldr}\ (\otimes)\ c\ (x_1\ \texttt{:}\ \ldots\ \texttt{:}\ x_n\ \texttt{:}\ \texttt{[]}) \quad = \quad x_1 \otimes \ldots \otimes x_n \otimes c$$

gdzie zarówno `:` jak i $\otimes$ łączą w prawo. Możemy też rozważyć wersję dla operatora $\oplus$ łączącego w lewo:

$$\texttt{foldl}\ (\oplus)\ c\ (x_1\ \texttt{:}\ \ldots\ \texttt{:}\ x_n\ \texttt{:}\ \texttt{[]}) \quad = \quad c \oplus x_1 \oplus \ldots \oplus x_n$$

Zatem:

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

W obu przypadkach widać, że iterator zamienia w strukturze danych wszystkie wystąpienia konstruktora `(:)` na podaną funkcję, a wystąpienie `[]` na podaną stałą i przedstawia tak zmodyfikowane wyrażenie do dalszego obliczania.

Z powodu leniwości Haskella w każdym typie danych mamy poza wartościami skończonymi także $\bot$ i inne wartości częściowe oraz wartości nieskończone (będące kresami górnymi częściowych skończonych aproksymacji). Jak zatem dowodzić twierdzeń przez indukcję strukturalną dla *wszystkich*, nie tylko skończonych wartości danego typu? Możemy dodać $\bot$ jako dodatkową podstawę indukcji. Na przykład dla list możemy przyjąć $\Sigma_0 = \{\texttt{[]}, \bot\}$ i $\Sigma_1 = \{(x\texttt{:}) \mid x :: a\}$. Otrzymamy wówczas zbiór wszystkich list skończonych i częściowych. Obowiązuje dla niego następująca zasada indukcji strukturalnej:

$$\frac{\Phi(\texttt{[]}) \quad \Phi(\bot) \quad \forall x :: a.\forall xs :: \texttt{[}a\texttt{]}.\Phi(xs) \Rightarrow \Phi(x\texttt{:}xs)}{\forall xs :: \texttt{[}a\texttt{]}.\Phi(xs)}$$

A co z wartościami nieskończonymi? Okazuje się, że wiele rodzajów twierdzeń, w tym równości skwantyfikowane uniwersalnie, ma własność *ciągłości* (zachowują kresy): jeśli zachodzą dla wszystkich wartości skończonych i częściowych, to zachodzą też dla wartości nieskończonych. Powyższa zasada indukcji pozwala więc dowodzić takich twierdzeń o *wszystkich* listach w Haskellu.

Polecenie 4. Przyjmując, że

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

udowodnij, że `(++)` jest łączne, tj. dla wszystkich list xs, ys i zs tego samego typu zachodzi:

$$(xs\ \texttt{++}\ ys)\ \texttt{++}\ zs \quad = \quad xs\ \texttt{++}\ (ys\ \texttt{++}\ zs)$$

Udowodnij następnie, że dla wszystkich *skończonych* list `xs` zachodzi

$$(\texttt{reverse . reverse})\ xs \quad = \quad xs$$

Pokaż, że mimo to

$$\texttt{reverse . reverse} \quad \neq \quad \texttt{id}$$

Wskaż miejsce, w którym załamuje się dowód przez indukcję. (Zauważ, że poza zasadą indukcji musimy tu też skorzystać z zasady ekstensjonalności).

Nieregularne typy danych

W zadaniu 8 z poprzedniej listy zaimplementowaliśmy listy o dostępie swobodnym:

```
data LTree a = LTree a :/\: LTree a | LLeaf a
data Digit a = Zero | One (LTree a)
newtype RAList a = RAList { fromRAList :: [Digit a] }
```

Niestety nie jest to *typ wolny*, bo zawiera *junk* — wartości, które nie są legalne. Drzewa mogą nie być pełne i nie muszą być uporządkowane ściśle według wysokości. Powyższa definicja nie narzuca żadnych niezmienników strukturalnych na wartości typu `LTree a`. W szczególności poddrzewa `l` i `r` drzewa `l :/\: r` mogą mieć różną liczbę elementów. Byłoby wygodnie, gdyby informacja o rozmiarze drzewa była zawarta w jego typie. Wówczas warunek, że `l` i `r` są tego samego typu implikowałby, że są tego samego rozmiaru. Ponieważ drzewo `l :/\: r` ma dwa razy tyle elementów, co poddrzewa `l` i `r`, to powinien być innego typu niż `l` i `r`. W takiej implementacji więc konstruktor `(:/\:)` ma typ $\tau_1 \to \tau_1 \to \tau_2$, przy czym typy drzew τ_1 i τ_2 są różne. Prowadzi to do rekurencyjnej definicji typu polimorficznego, w której definiowany typ występuje w swojej definicji po lewej i prawej stronie znaku równości z różnymi parametrami typowymi. Taką rekursję nazywamy *niejednorodną* (*non-uniform*), a tak zdefiniowany typ — typem *niejednorodnym* lub *zagnieżdżonym* (*nested*). Najprostszym przykładem takiego typu są doskonałe drzewa binarne o etykietowanych liściach:

```
data PTree a = PNode (PTree (a,a)) | PLeaf a
```

Funkcje rekurencyjne działające na wartościach typu niejednorodnego wykorzystują *rekursję polimorficzną* — argument wywołania rekurencyjnego jest innego typu niż argument definiowanej funkcji, np. w ostatnim wierszu definicji

```
size :: PTree a -> Int
size (PLeaf _) = 1
size (PNode t) = 2 * size t
```

typem argumentu funkcji `size` po lewej stronie znaku równości jest `PTree a`, po prawej zaś `PTree(a,a)`. Kompilator nie potrafi samodzielnie rekonstruować typów funkcji definiowanych za pomocą rekursji polimorficznej, dlatego definicję takiej funkcji należy *koniecznie* poprzedzić jej sygnaturą. Dotyczy to także funkcji lokalnych, np. deklarowanych po słowie kluczowym `where`.

Drzewa typu `PTree` spełniają niezmienniki strukturalne nakładane przez definicję kopca (są doskonałe, tj. mają po 2^k elementów), ale umieszczenie ich na zwykłej liście spowodowałoby, że nie kontrolowalibyśmy zależności pomiędzy ich wysokościami. Zauważmy, że każda wartość typu `PTree a` ma postać $\texttt{PNode}^k(\texttt{PLeaf}\ p)$ gdzie p jest parą par par ... par elementów typu `a`, więc konstruktor `PNode` można przerobić na konstruktory `Zero` i `One` tworzące polimorficzną listę drzew:

```
data RAList a = RAZero (RAList (a,a)) | RAOne a (RAList (a,a)) | RANil
```

Zadanie 7 (2 pkt). Zainstaluj typ `RAList` w klasie

```
class ListView t where
   viewList :: t a -> List t a
   toList :: t a -> [a]
   cons :: a -> t a -> t a
   nil :: t a
data List t a = Cons a (t a) | Nil
```

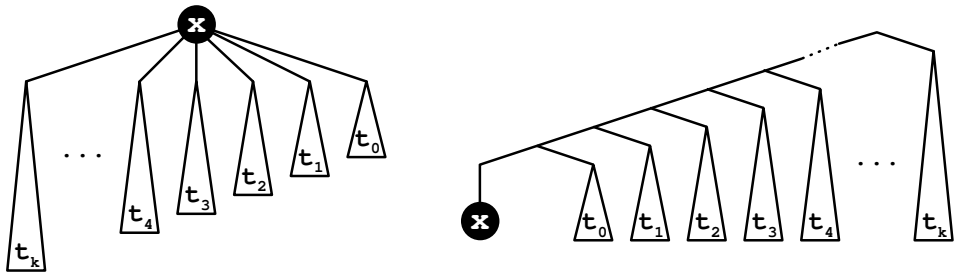
z poprzedniej listy.

Poniżej będziemy implementować kolejki priorytetowe, opisane następującą specyfikacją, w której metoda `single` tworzy kolejkę jednoelementową zawierającą podany element:

```
class Ord a => Prioq t a where
   empty      :: t a
   isEmpty    :: t a -> Bool
   single     :: a -> t a
   insert     :: a -> t a -> t a
   merge      :: t a -> t a -> t a
   extractMin :: t a -> (a, t a)
   findMin    :: t a -> a
   deleteMin  :: t a -> t a
   fromList   :: [a] -> t a
   toList     :: t a -> [a]
   insert = merge . single
   single =  flip insert empty
   extractMin t = (findMin t, deleteMin t)
   findMin = fst . extractMin
   deleteMin = snd . extractMin
   fromList = foldr insert empty
   toList = unfoldr (\ t -> if isEmpty t then Nothing else Just (extractMin t))
```

Dla niektórych funkcji podano domyślne implementacje. W każdej instancji należy zdefiniować co najmniej jedną z funkcji `insert` i `single` oraz funkcję `extractMin` lub obie funkcje `findMin` i `deleteMin`. Pozostałe funkcje będą miały domyślną implementację, choć *możemy* zadeklarować własną, być może bardziej efektywną.

Drzewa binarne o etykietowanych liściach są izomorficzne z drzewami dwumianowymi i mogą służyć do ich reprezentowania w komputerze. Sposób reprezentacji objaśnia poniższy rysunek:



Kopce dwumianowe możemy więc reprezentować w postaci typu strukturalnie identycznego z typem `RAList`:

```
data BHeap a = BZero (BHeap (a,a)) | BOne a (BHeap (a,a)) | BNil
```

Zadanie 8 (2 pkt). Zainstaluj typ `BHeap` w klasie `Prioq`.