Programowanie Funkcyjne 2018

Lista zadań nr 10: Przygody w krainie typów. Część 1: Typy niejednorodne

Na zajęcia 3 stycznia 2019

Uwaga: w różnych zadaniach z bieżącej listy używamy tych samych identyfikatorów do nazwania różnych wartości i typów. Aby uniknąć błędów kompilacji rozwiązanie każdego zadania najlepiej umieścić w osobnym pliku.

Zadanie 1 (3 pkt). W zadaniach z poprzedniej listy wykorzystywaliśmy dwa rodzaje drzew zawierających dokładnie 2^k elementów (tzw. *drzew doskonałych*) do zaprogramowania kolejek priorytetowych. Wadą tych implementacji był brak kontroli nad tym, czy budowane drzewa faktycznie mają 2^k elementów. Rozważmy definicję drzew binarnych o etykietowanych liściach:

```
data BTree a = Node (BTree a) (BTree a) | Leaf a
```

Powyższa definicja nie narzuca żadnych niezmienników strukturalnych na wartości typu `BTree a`. W szczególności poddrzewa `l` i `r` drzewa `Node 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 `Node 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 `Node` 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:

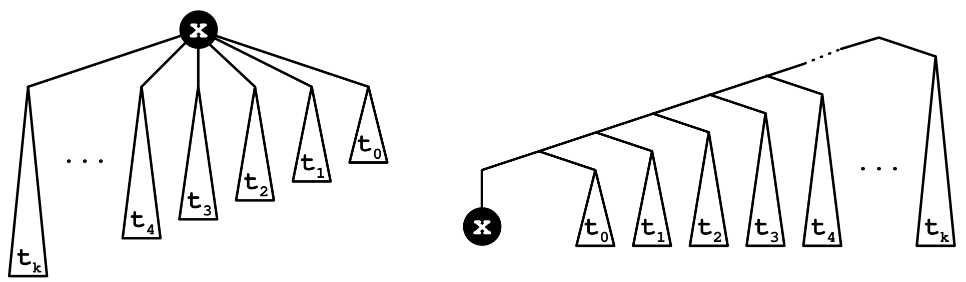
```
data Tree a = Node (Tree (a,a)) | Leaf a
```

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 :: Tree a -> Int
size (Leaf _) = 1
size (Node t) = 2 * size t
```

typem argumentu funkcji `size` po lewej stronie znaku równości jest `Tree a`, po prawej zaś `Tree (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 binarne o etykietowanych liściach są izomorficzne z drzewami dwumianowymi i mogą służyć do ich reprezentowania w komputerze. Sposób reprezentacji objaśnia poniższy rysunek:



Zainstaluj typ `Tree` w klasie `Base2Tree` z poprzedniej listy tak, by wartości tego typu reprezentowały drzewa dwumianowe.

Zadanie 2 (2 pkt). Drzewa z poprzedniego zadania spełniają niezmienniki strukturalne nakładane przez definicję kopca (mają po 2^k elementów), ale implementacje kopców z poprzedniej listy nie kontrolują, czy w danym miejscu listy drzew występuje drzewo o właściwym rozmiarze. Zauważmy, że każda wartość typu `Tree a` ma postać $\texttt{Fork}^k(\texttt{Leaf}\ p)$ gdzie p jest parą par par … par elementów typu a, więc konstruktor `Fork` można przerobić na konstruktory `Zero` i `One` tworzące polimorficzną listę drzew:

```
data Heap a = Nil | Zero (Heap (a,a)) | One a (Heap (a,a))
```

W implementacji gęstych kopców z poprzedniej listy konstruktor `Zero` odpowiada wyrażeniu `(Nothing :)`, konstruktor `One` — wyrażeniu `(:) . Just`, zaś konstruktor `Nil` — konstruktorowi `[]`. Takie kopce spełniają wszystkie niezmienniki strukturalne. Zainstaluj typ `Heap` w klasie `Prioq` z poprzedniej listy.

Zadanie 3 (3 pkt). Drzewa w zadaniu 1 są utworzone nie za pomocą konstruktorów typu `Tree` tylko za pomocą konstruktora pary. Stanie się to lepiej widoczne, jeśli zastąpimy standardowy typ

```
data (,) a b = (,) a b
```

(w którym konstruktory typu i wartości celowo zapisaliśmy prefiksowo) przez dedykowany typ par elementów tego samego typu:

```
data Pair a = Pair a a
```

albo lepiej, skoro konstruktor tego typu ma służyć do reprezentowania węzłów drzewa:

```
data Fork a = Fork a a
```

Drzewo elementów typu a wysokości k ma typ $\texttt{Fork}^k$ a. Jeśli w definicji typu `Tree` wymienimy typ `(,)` na typ `Fork`, to otrzymamy typ

```
data PTree t = Node (PTree (Fork t)) | Leaf t
```

Konstruktor `Node` ma typ

```
Node :: Tree (Fork t) -> Tree t
```

zaś konstruktor `Leaf` ma typ

```
Leaf :: t -> Tree t
```

zatem $\texttt{Node}^j(\texttt{Leaf}\ p)$ ma typ $\texttt{Tree}(\texttt{Fork}^{k-j}$ a) jeśli p ma typ $\texttt{Fork}^k$ a. Na przykład mamy

```
Fork (Fork 1 2) (Fork 3 4) :: Fork (Fork Integer)
Leaf $ Fork (Fork 1 2) (Fork 3 4) :: Tree (Fork (Fork Integer))
Node $ Node $ Leaf $ Fork (Fork 1 2) (Fork 3 4)) :: Tree Integer
```

Typ `Fork` jest nierekurencyjny. Pełne drzewa binarne o różnej wysokości mają różne typy (dzięki temu ich typ przenosi informację o ich wysokości). Aby móc skorzystać z takich drzew w programach które przetwarzają drzewa dowolnej wysokości, musimy „ukryć" informację o ich wysokości. Do tego właśnie celu służą konstruktory `Node` i `Leaf`.

Aby zbudować pełne drzewa binarne o etykietowanych wierzchołkach wewnętrznych trzeba parę w definicji typu z zadania 1 zastąpić trójką złożoną z dwóch drzew i etykiety. Najlepiej użyć do tego celu dedykowanego typu danych:

```
data Fork t a = Fork t a t
```

Drzewa definiujemy teraz bardzo podobnie jak w zadaniu 1:

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

Na przykład drzewo 3-elementowe:

```
Node $ Node $ Leaf $ Fork (Fork () 1 ()) 2 (Fork () 3 ())
```

ma typ

```
Tree () Integer
```

W rzeczywistości drzewa `Tree t a` to pełne drzewa binarne o wierzchołkach wewnętrznych etykietowanych elementami typu a i liściach etykietowanych elementami typu `t`. W naszych zastosowaniach etykietami liści mogłyby być wartości `()`. Wygodniej jednak zdefiniować własny typ danych, izomorficzny z typem `()`:

```
data Empty = Empty
```

Mamy teraz na przykład:

```
Empty :: Empty
Fork Empty 1 Empty :: Fork Empty Integer
Fork (Fork Empty 1 Empty) 2 (Fork Empty 3 Empty)
   :: Fork (Fork Empty Integer) Integer
Leaf $ Fork (Fork Empty 1 Empty) 2 (Fork Empty 3 Empty)
   :: Tree (Fork (Fork Empty Integer) Integer) Integer
Node $ Node $ Leaf $ Fork (Fork Empty 1 Empty) 2 (Fork Empty 3 Empty)
   :: Tree Empty Integer
```

Możemy już zaprogramować proporczyki:

```
data Pennant a = Pennant a (Tree Empty a)
```

Zainstaluj typ `Pennant` w klasie `Base2Tree` z poprzedniej listy.

Zadanie 4 (3 pkt). Przez analogię do zadania 2 moglibyśmy teraz zaprogramować kopce proporczykowe wykorzystujące drzewa z poprzedniego zadania, a następnie powtórzyć całą procedurę dla drzew i kopców dwumianowych. Aby nie wydało się to nudne, zmienimy jednak nieco sposób definiowania drzew: użyjemy rekursji nieregularnej wyższego rzędu. Ma ona miejsce wówczas, gdy nieregularny parametr typu ma rodzaj (*kind*) różny od $*$. Rozważmy takie drzewa:

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

Parametr `t` typu `Tree` ma rodzaj `* -> *`. Definicja typu `Tree` będzie rekurencyjna właśnie względem niego. Drzewo

```
Fork (Fork Empty 1 Empty) 2 (Fork Empty 3 Empty)
```

(które, swoją drogą, wygląda jak *zwykłe* drzewo binarne i jego zapis jest identyczny z zapisem drzewa z poprzedniego zadania) ma teraz typ

```
Fork (Fork Empty) Integer
```

Drzewo elementów typu a wysokości k ma typ `Fork`^{k} `Empty` a.

Aby móc skorzystać z takich drzew w implementacjach kopców z poprzedniej listy musimy na powrót zakryć różnice pomiędzy drzewami, np. tak:

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

Pomijanie informacji o rozmiarze drzewa jest oczywiście dosyć sztuczne, ale może posłużyć jako wprawka przed następnym zadaniem. Zainstaluj typ `Pennant` w klasie `Base2Tree`.

Zadanie 5 (3 pkt). Zauważmy, że konstruktory `Node` z poprzedniego zadania możemy zastąpić konstruktorami tworzącymi listę proporczyków dokładnie w taki sam sposób jak w zadaniu 2, co prowadzi do następującej definicji:

```
data Empty a = Empty
data Fork t a = Fork (t a) a (t a)
data Pennant t a = Pennant a (t a)
data List t a = Nil | Zero (List (Fork t) a) | One (Pennant t a) (List (Fork t) a)
newtype Heap a = Heap (List Empty a)
```

Zainstaluj typ `Heap` w klasie `Prioq`.

Zadanie 6 (3 pkt). Drzewa dwumianowe z nieregularnością wyższego rzędu możemy zaprogramować w dokładnie taki sam sposób, jak drzewa binarne, zastępując parę dzieci przez polimorficzną listę dzieci:

```
data Empty a = Empty
data Cons l a = Cons (Fork l a) (l a)
data Fork l a = Fork a (l a)
```

Podobnie jak w zadaniu 4 możemy teraz poćwiczyć operowanie na takich drzewach ukrywając informację o ich stopniu:

```
data Tree l a = Node (Tree (Cons l) a) | Leaf (Fork l a)
newtype Binomial a = Binomial (Tree Empty a)
```

Zainstaluj typ `Binomial` w klasie `Base2Tree`.

Zadanie 7 (3 pkt). Poprzednie zadanie było tylko wprawką przed prawdziwą implementacją kopców dwumianowych których typy gwarantują wszystkie niezmienniki strukturalne:

```
data Empty a = Empty
data Cons l a = Cons (Fork l a) (l a)
data Fork l a = Fork a (l a)
data List l a = Nil | Zero (List (Cons l) a) | One (Fork l a) (List (Cons l) a)
newtype Heap a = Heap (List Empty a)
```

Zainstaluj typ `Heap` w klasie `Prioq`.

Wersja z poprawkami z 28 grudnia 2018 15:54.