Zippers

Opis problemu oraz programu

Grzegorz Łach
Paula Chajduła

07.02.2020

Co to są te Zippery?

Zipper to idiom wykorzystujący ideę „kontekstu” do manipulowania lokalizacjami w strukturze danych. Zipper monada to monada, która implementuje zamek błyskawiczny dla drzew binarnych.

Jeśli mamy drzewo pełne piątki (może piątki, a może?) I chcemy zmienić jedną z nich w szóstkę, musimy mieć sposób, aby wiedzieć dokładnie, którą piątkę z naszego drzewa chcemy zmienić. Musimy wiedzieć, gdzie jest w naszym drzewie. W nieczystych językach moglibyśmy po prostu zauważyć, gdzie w naszej pamięci znajduje się pięć i to zmienić. Ale w Haskell jedna piątka jest tak dobra jak inna, więc nie możemy dyskryminować ze względu na to, gdzie są w naszej pamięci. Nie możemy też tak naprawdę niczego zmienić ; kiedy mówimy, że zmieniamy drzewo, w rzeczywistości mamy na myśli, że bierzemy drzewo i zwracamy nowe, które jest podobne do oryginalnego drzewa, ale nieco inne.

Przykład

Istnieje wiele różnych rodzajów drzew, więc wybierzmy ziarno, którego użyjemy do sadzenia naszego. Oto on

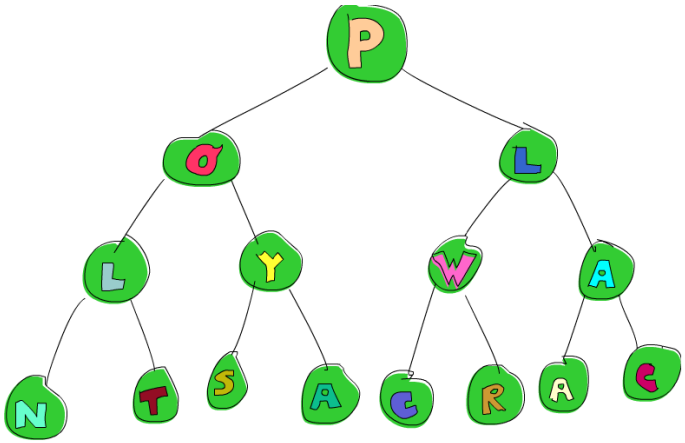
```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Więc nasze drzewo jest puste lub jest węzłem, który ma element i dwa podgrzewa. Oto świetny przykład takiego drzewa

Przykład cd..

```
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

Przykład cd..



Zauważyłeś że W na drzewie? Chcemy zmienić je na P . Jak moglibyśmy to zrobić? Jednym ze sposobów byłoby wzorowanie dopasowania na naszym drzewie, dopóki nie znajdziemy elementu, który jest zlokalizowany, najpierw przesuwając się w prawo, a następnie w lewo i zmieniając ten element. Oto kod tego

Przykład cd..

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Czy jest na to lepszy sposób? Może sprawimy, że nasza funkcja zajmie drzewo wraz z listą wskazówek. Kierunki będą L lub R , reprezentujące odpowiednio lewą i prawą, a my zmienimy element, do którego dochodzimy, jeśli będziemy postępować zgodnie z podanymi wskazówkami. Oto on

Przykład cd..

```haskell
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions-> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Aby uniknąć drukowania całego drzewa, utwórzmy funkcję, która pobiera listę wskazówek i mówi nam, jaki jest element w miejscu docelowym

```haskell
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

Przykład cd..

Tutaj zmieniamy „W” na „P” i sprawdzamy, czy zmiana w naszym nowym drzewie się trzyma

```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

W tych funkcjach lista kierunków działa jako swego rodzaju fokus , ponieważ wskazuje dokładnie jedno sub-drzewo z naszego drzewa. Chociaż ta technika może wydawać się fajna, może być raczej nieefektywna, szczególnie jeśli chcemy wielokrotnie zmieniać elementy

Breadcrumbs

Ok, więc aby skupić się na sub-drzewie, chcemy czegoś lepszego niż tylko lista kierunków, które zawsze podążamy od nasady naszego drzewa. To znaczy, kiedy idziemy w lewo, pamiętamy, że poszliśmy w lewo, a kiedy idziemy w prawo, pamiętamy, że poszliśmy w prawo. Aby reprezentować naszą bułkę tartą, użyjemy również listy Kierunków (która jest albo L albo R), tylko zamiast nazywać ją Kierunkami , nazwiemy ją Breadcrumbs

```
typ Breadcrumbs  =  [ Kierunek ]
```

Breadcrumbs cd..

Oto funkcja, która pobiera drzewo i Breadcrumbs i przesuwa się do lewego sub-drzewa, dodając L do nagłówka listy reprezentującej nasze bułki tartej:

```
goLeft :: ( Tree a, Breadcrumbs ) -> ( Tree a, Breadcrumbs )
goLeft ( Węzeł _ l _, bs) = (l, L : bs)
```

Ignorujemy element w katalogu głównym i prawym sub-drzewie i po prostu zwracamy lewe sub-drzewo wraz ze starym Breadcrumbs z L jako głową. Oto funkcja, aby przejść w prawo:

```
goRight :: ( Tree a, Breadcrumbs ) -> ( Tree a, Breadcrumbs )
goRight ( Węzeł _ _ r, bs) = (r, R : bs)
```

Breadcrumbs cd..

Działa w ten sam sposób. Użyjmy tych funkcji, aby wziąć nasze bezpłatne drzewo i iść w prawo, a następnie w lewo:

```
ghci>  goLeft (goRight (freeTree, []))
( Węzeł „W”  ( Węzeł „C” Pusty Pusty ) ( Węzeł „R” Pusty Pusty ), [ L , R ])
```

Okej, więc teraz mamy drzewo, które ma „W” w swoim katalogu głównym i „C” w katalogu głównym lewego sub-drzewa i „R” w katalogu głównym swojego sub-drzewa.Breadcrumbs to [L, R] , ponieważ najpierw poszliśmy w prawo, a potem w lewo.Aby uczynić spacer po naszym drzewie wyraźniejszym, możemy użyć funkcji - :, którą tak zdefiniowaliśmy:

Breadcrumbs cd..

Aby uczynić spacer po naszym drzewie wyraźniejszym, możemy użyć funkcji -", którą tak zdefiniowaliśmy

```
x -: f  =  f x
```

Co pozwala nam zastosować funkcje do wartości, najpierw pisząc wartość, a następnie pisząc : : a potem funkcję. Zamiast goRight (freeTree, []) , możemy napisać (freeTree, []) -: goRight .

```
ghci>  (freeTree, []) -: goRight -: goLeft
( Węzeł „W”  ( Węzeł „C” Pusty Pusty ) ( Węzeł „R” Pusty Pusty ), [ L , R ])
```

Manipulowanie drzewami w centrum uwagi

Teraz, gdy możemy poruszać się w górę i w dół, utwórzmy funkcję, która modyfikuje element w katalogu głównym poddrzewa, na którym koncentruje się suwak:

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Jeśli skupiamy się na węźle, modyfikujemy jego element główny za pomocą funkcji f. Teraz możemy zacząć od drzewa, przenieść się w dowolne miejsce i zmodyfikować element, jednocześnie utrzymując skupienie na tym elemencie np.

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

Manipulowanie drzewami w centrum uwagi cd..

Idziemy w lewo, potem w prawo, a następnie modyfikujemy element główny, zastępując go literą „P" . To brzmi jeszcze lepiej, jeśli użyjemy - ::

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

Możemy wtedy przejść w górę, jeśli chcemy, i zastąpić element tajemniczym „X" :

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

Lub jeśli pisaliśmy go -: :

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Manipulowanie drzewami w centrum uwagi cd..

Każdy węzeł ma dwa podgrzewa, nawet jeśli są to puste drzewa. Jeśli więc skupiamy się na pustym poddrzewie, jedyne, co możemy zrobić, to zastąpić go niepustym poddrzewem, przyłączając w ten sposób drzewo do węzła liścia.

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

Bierzemy drzewo i zipper i zwracamy zipper , którego ostrość została zastąpiona dostarczonym drzewem.

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

Manipulowanie drzewami w centrum uwagi cd..

W ten sposób możemy nie tylko przedłużać drzewa, zastępując puste podgrzewa nowymi drzewami, ale także zastępować całe istniejące podgrzewa. Dołączmy drzewo po lewej stronie naszego darmowego drzewa :

```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

newFocus koncentruje się teraz na drzewie, które właśnie przyczepiliśmy, a reszta drzewa leży odwrócona w bułce tartej. Gdybyśmy użyli goUp, aby przejść do szczytu drzewa, byłoby to to samo drzewo co FreeTree, ale z dodatkowym „Z" po jego lewej stronie.

Koncentrowanie się na listach

Zippery mogą być używane z dowolną strukturą danych, więc nic dziwnego, że można je wykorzystać do skoncentrowania się na podlistach list. W końcu listy są bardzo podobne do drzew, tylko tam, gdzie węzeł w drzewie ma element (lub nie) i kilka poddrzewa, węzeł na liście ma element i tylko jedną podlistę. Kiedy wdrożyliśmy własne listy , zdefiniowaliśmy nasz typ danych w następujący sposób:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Porównaj to z naszą definicją naszego drzewa binarnego, a łatwo jest zobaczyć, jak listy mogą być przeglądane jako drzewa, w których każdy węzeł ma tylko jedno poddrzewo.

Koncentrowanie się na listach cd..

Lista taka jak [1,2,3] może być zapisana jako 1: 2: 3: [] . Składa się z nagłówka listy, którym jest 1, a następnie ogona listy, który wynosi 2: 3: [] . Z kolei 2: 3: [] ma także głowę, która jest 2 i ogon, który jest 3: [] . Z 3: [] The 3 jest głową i ogonem jest pustym, [] . Listy są prostsze niż drzewa, więc nie musimy pamiętać, czy poszliśmy w lewo, czy w prawo, ponieważ jest tylko jeden sposób, aby wejść głębiej w listę.
Ponieważ pojedynczy element nawigacyjny jest tutaj tylko elementem, tak naprawdę nie musimy umieszczać go w typie danych, tak jak zrobiliśmy to, gdy stworzyliśmy typ danych Crumb dla suwaków drzewa:

```
type ListZipper a = ([a],[a])
```

Koncentrowanie się na listach cd..

Pierwsza lista reprezentuje listę, na której się skupiamy, a druga lista to bułka tarta. Zróbmy funkcje, które przechodzą do przodu i do tyłu na listy:

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Kiedy idziemy do przodu, skupiamy się na ogonie bieżącej listy i pozostawiamy element head jako bułkę tartą. Kiedy cofamy się, bierzemy najnowszy Breadcrumbs i umieszczamy ją na początku listy.

Koncentrowanie się na listach cd..

Oto te dwie funkcje w akcji:

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs,[])
([2,3,4],[1])
ghci> goForward ([2,3,4],[1])
([3,4],[2,1])
ghci> goForward ([3,4],[2,1])
([4],[3,2,1])
ghci> goBack ([4],[3,2,1])
([3,4],[2,1])
```

Widzimy, że Breadcrumbs w przypadku list jest tylko odwróconą częścią naszej listy. Element, od którego się oddalamy, zawsze trafia do głowy Breadcrumbs, więc łatwo jest cofnąć się, po prostu biorąc ten element z głowy bułki tartej i czyniąc go głównym celem naszej uwagi.

Bardzo prosty system plików

Teraz, gdy wiemy, jak działają zamki błyskawiczne, użyjmy drzew do reprezentowania bardzo prostego systemu plików, a następnie utwórz zamek błyskawiczny dla tego systemu plików, który pozwoli nam poruszać się między folderami, tak jak to zwykle robimy podczas przeskakiwania po naszym systemie plików.

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

Plik zawiera dwa ciągi, które reprezentują jego nazwę i przechowywane przez niego dane. Folder zawiera ciąg znaków, który jest jego nazwą i listą elementów. Jeśli ta lista jest pusta, mamy pusty folder.

Bardzo prosty system plików cd..

Oto folder z niektórymi plikami i podfolderami:

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl_bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
        ]
```

Zippery dla naszego systemu plików

Jeśli skupimy się na folderze „root”, a następnie na pliku , zastanowimy się jak powinien wyglądać pozostawiony przez nas plik nawigacyjny. Powinien zawierać nazwę swojego folderu nadrzędnego wraz z elementami znajdującymi się przed plikiem, na którym się skupiamy, oraz elementami, które następują po nim. Potrzebujemy więc tylko nazwy i dwóch list przedmiotów.

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

A oto synonim typu dla naszego zamka błyskawicznego:

```
type FSZipper = (FSItem, [FSCrumb])
```

Zippery dla naszego systemu plików cd..

Powrót do hierarchii jest bardzo prosty. Po prostu bierzemy najnowszą bułkę tartą i gromadzimy nowy fokus z bieżącej fokusu i bułki tartej. Tak jak:

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

Ponieważ nasz breadcrumb wiedział, jak nazywa się folder nadrzędny, a także elementy, które pojawiły się przed naszym elementem skupionym w folderze (to jest ls) i te, które pojawiły się po nim (to jest rs), przejście w górę było łatwe.

Zippery dla naszego systemu plików cd..

Oto funkcja, która podając nazwę, skupia się na pliku folderu, który znajduje się w bieżącym folderze aktywnym:

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

fsTo pobiera Nazwę i FSZipper i zwraca nowy FSZipper, który skupia się na pliku o podanej nazwie. Ten plik musi znajdować się w bieżącym folderze aktywnym. Ta funkcja nie wyszukuje w dowolnym miejscu, po prostu przegląda bieżący folder.

Zippery dla naszego systemu plików cd..

Teraz możemy poruszać się w górę i w dół naszego systemu plików. Zacznijmy od katalogu głównego i przejdźmy do pliku $\text{skull}_{m}an(scary).bmp$:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

newFocus jest teraz zamkiem, który koncentruje się na pliku $\text{skull}_{m}an(scary).bmp$.

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

Zippery dla naszego systemu plików cd..

Przejdźmy w górę, a następnie skupmy się na sąsiednim pliku „watermelon_smash.gif”

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

Manipulowanie naszym systemem plików

Teraz, gdy wiemy, jak poruszać się po naszym systemie plików, manipulowanie nim jest łatwe. Oto funkcja, która zmienia nazwę aktualnie zaznaczonego pliku lub folderu:

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

Teraz możemy zmienić nazwę naszego folderu „pics” na „cspi” :

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

Zeszliśmy do folderu „pics” , zmieniliśmy jego nazwę, a następnie wróciliśmy do góry.

Manipulowanie naszym systemem plików cd..

Nowy element w bieżącym folderze

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

Dodajmy plik do naszego folderu „pics”, a następnie wróćmy do katalogu głównego:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "lol") -: fsUp
```

Bezpieczeństwo

Jak dotąd, podczas przeglądania naszych struktur danych, niezależnie od tego, czy były to drzewa binarne, listy czy systemy plików, tak naprawdę nie przejmowaliśmy się tym, że posunęliśmy się o krok za daleko i spadliśmy. Na przykład nasza funkcja goLeft pobiera zamek błyskawiczny z drzewa binarnego i przenosi fokus do lewego pod-drzewa:

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Ale co, jeśli drzewo, z którego zejdziemy, jest pustym drzewem? To znaczy, co jeśli nie jest to Węzeł , ale Pusty ? W takim przypadku wystąpiłby błąd czasu wykonywania

Bezpieczeństwo cd..

Do tej pory niepowodzenie funkcji, które mogły zawieść, zawsze było odzwierciedlone w ich wyniku i tym razem nie jest inaczej. Oto więc goLeft i goRight z dodatkową możliwością niepowodzenia:

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

Robimy krok w lewo

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

Bezpieczeństwo cd..

To jest funkcja goUp, która generuje błąd, jeśli nie trzymamy się granicy naszego drzewa:

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Teraz zmodyfikujmy go tak, aby z wdziękiem zawodził:

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

Bezpieczeństwo cd..

Jeśli mamy breadcrumbs, wszystko jest w porządku i przywracamy nowe, udane podejście, ale jeśli nie, to zwracamy błąd.
Wcześniej te funkcje pobierały Zippery i zwracały Zippery, co oznaczało, że mogliśmy je połączyć w następujący sposób:

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

Bezpieczeństwo cd..

Możemy ponownie połączyć nasze funkcje

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

Użyliśmy return, aby umieścić zamek błyskawiczny w Just, a następnie użyliśmy » =, aby nakarmić to naszą funkcją goRight . Najpierw stworzyliśmy drzewo, które ma po lewej puste pod-drzewo, a po prawej węzeł, który ma dwa puste podgrzewa. Teraz wyposażyliśmy nasze drzewa w siatkę bezpieczeństwa, która złapie nas w razie upadku.

Koniec

Prezentacje Przygotowali:
Grzegorz Łach
Paula Chajduła