Podstawy Programowania Funkcyjnego

Chlebowski Sz. Gajda A.

Zakład Logiki i Kognitywistyki
Instytut Psychologii
Uniwersytet im. Adama Mickiewicza

26 kwietnia 2023

1. Type, Data, Newtype
2. Typy rekurencyjne
3. Automatyczne dowodzenie w logice klasycznej
4. Zadania

Type, Data, Newtype

- Mechanizm *type* służy do wprowadzania skrótów dla wcześniej zdefiniowanych typów

```
type String = [Char]

type Pos = (Int,Int)

type Trans = Pos -> Pos

type Pair a = (a,a)
```

- Mechanizm *data* służy do definiowania nowych typów

```
data Bool = False | True

data Move = North | South | East | West

data Shape = Circle Float | Rect Float Float

data Maybe a = Nothing | Just a
```

- Mechanizm *newtype* jest podobny do Data, jednak może zawierać tylko jeden konstruktor

```
newtype Nat = N Int

newtype Point = P (Int, Int)
```

- Liczby naturalne

```
data Nat = Zero | Succ Nat

nat2int :: Nat -> Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

- Dodawanie w liczbach naturalnych

```
add :: Nat -> Nat -> Nat
add Zero n      = n
add (Succ m) n = Succ (add m n)
```

- Typ listy

```haskell
data List a = Nil | Cons a (List a)

len :: List a -> Int
len Nil          = 0
len (Cons _ xs) = 1 + len xs
```

- Drzewa binarne

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

t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
(Node (Leaf 6) 7 (Leaf 9))

occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y)      = x==y
occurs x (Node l y r) = x==y
                        || occurs x l
                        || occurs x r
```

- Inne popularne rodzaje drzew

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

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

data Tree a b = Leaf a
              | Node (Tree a b) b (Tree a b)

data Tree a = Node a [Tree a]
```

Automatyczne dowodzenie w logice klasycznej

- Definicja formuł

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop

p1 :: Prop
p1 = And (Var 'A') (Not (Var 'A'))

p2 :: Prop
p2 = Imply (And (Var 'A') (Var 'B')) (Var 'A')

p3 :: Prop
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))
```

- Podstawienia

```
type Subst = Assoc Char Bool

type Assoc k v = [(k,v)]


find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k == k']
```

- Funkcja obliczająca wartość logiczną formuł (przy danym wartościowaniu/podstawieniu)

```
eval :: Subst -> Prop -> Bool
eval _ (Const b)    = b
eval s (Var x)      = find x s
eval s (Not p)      = not (eval s p)
eval s (And p q)    = eval s p && eval s q
eval s (Imply p q) = eval s p <= eval s q
```

Zmienne formuły

- Lista zmiennych w danej formule

```
vars :: Prop -> [Char]
vars (Const _)   = []
vars (Var x)     = [x]
vars (Not p)     = vars p
vars (And p q)   = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q
```

- Tabelka prawdziwościowa dla *n* zmiennych

```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False:) bss ++ map (True:) bss
     where bss = bools (n-1)
```

Tabelka

- Tabelka prawdziwościowa dla *n* zmiennych

```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False:) bss ++ map (True:) bss
     where bss = bools (n-1)
```

- Funkcja usuwająca powtórzenia

```
rmdups :: Eq a => [a] -> [a]
rmdups []     = []
rmdups (x:xs) = x : filter (/= x) (rmdups xs)
```

Możliwe podstawienia

- Generujemy listę wszystkich podstawień

```
substs :: Prop -> [Subst]
substs p = map (zip vs) (bools (length vs))
    where vs = rmdups (vars p)
```

Możliwe podstawienia

- Generujemy listę wszystkich podstawień

```
substs :: Prop -> [Subst]
substs p = map (zip vs) (bools (length vs))
    where vs = rmdups (vars p)
```

- Sprawdzamy, czy formuła jest tautologią

```
isTaut :: Prop -> Bool
isTaut p = and [eval s p | s <- substs p]
```

Zadania

Zadania 1

1. Zdefiniuj funkcję, która dla danej formuły zwróci listę wszystkich wartościowań, przy których jest ona fałszywa/prawdziwa. Użyj testów, żeby przekonać się, czy implementacja jest poprawna.
2. Rozszerz program sprawdzający tautologiczność tak, aby akceptował również altenatywę i równoważność. Użyj testów.
3. Rozważ następujący typ reprezentujący drzewa binarne:

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

   Powiemy, że drzewo jest w równowadze, jeśli liczba liści w lewej gałęzi różni się do liczby liści na prawej gałęzi o co najwyżej 1. Zdefiniuj funkcję *balanced*, która sprawdza, czy drzewo jest w równowadze.

4. Zdefiniuj funkcję `balance :: [Int] -> Tree`, która przekształca niepustą listę liczb w drzewo w równowadze, które zawiera wszystkie elementy z zadanej listy. Zdefiniuj funkcje pomocniczą, która dzieli listę na dwie części, których długość różni się co najwyżej o 1.
5. Zdefiniuj funkcję `treemap`, która bierze drzewo zawierające elementy typu `a` (`Tree a`) oraz funkcję `f :: a -> b` i zwraca drzewo zawierające elementy typu `b` (`Tree b`).
6. Zdefiniuj funkcję która przekształca dowolne drzewo określone typem:

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

na drzewo określone typem:

```
data RoseTree a = Node a [RoseTree a]
```

Czy konwersja odwrotna jest możliwa?

Zadania 3

7. Zaimplementuj *tautology-checker* dla wybranej logiki wielowartościowej.