Paradigmas de Programação

Fabrício Olivetti de França

24 de Julho de 2018

Tautologia

Para esse projeto crie um novo projeto usando o *stack* com o nome **tautologia**.
O conteúdo inicial de *Main.hs* deve ser:

```
module Main where

data Prop = Const Bool          -- constante
          | Var Char            -- variável
          | Not Prop            -- Não
          | And Prop Prop       -- E
          | Imply Prop Prop     -- Se-Então

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']

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

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

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

p4 :: Prop
p4 = ((Var 'A') `And` ((Var 'A') `Imply` (Var 'B'))) `Imply` (Var 'B')

main :: IO ()
main = do
```

```
print $ isTaut p1
print $ isTaut p2
print $ isTaut p3
print $ isTaut p4
```

Uma proposição lógica é definida por constantes booleanas (Verdadeiro ou Falso), variáveis lógicas (A, B, ..., Z) e os operadores Não, E, Se-Então (implicação). Uma tautologia é uma proposição que sempre retorna Verdadeiro não importando os valores das variáveis. Considere as proposições: $(A \wedge B) \implies A$ e $A \implies (A \wedge B)$. As tabelas verdades ficam:

A	B	$(A \wedge B) \implies A$
F	F	V
F	V	V
V	F	V
V	V	V

A	B	$A \implies (A \wedge B)$
F	F	V
F	V	V
V	F	F
V	V	V

Verificamos que a primeira proposição é uma tautologia enquanto a segunda não. Para avaliar se uma proposição é uma tautologia, precisamos primeiro avaliar uma proposição ao atribuir valores para as variáveis.

O tipo `Subst` mantém uma lista associativa entre o nome da variável (um `Char`) e seu valor (`True` ou `False`). O primeiro passo é, dada uma lista de associação (ex.: `[('A', True), ('B', False)]`), avaliar a proposição:

```
-- retorna o resultado ao substituir as variáveis de uma proposição por valores booleanos.
-- crie um pattern matchin para cada possível valor de Prop
avalia :: Subst -> Prop -> Bool
```

Em seguida, crie uma função que retorna a lista de variáveis em uma proposição:

```
vars :: Prop -> [Char]

-- remove as variáveis duplicadas da lista
uniquevars = nub vars
```

Agora precisamos de uma função que gera todas as combinações de valores `True` e `False` para um certo número de variáveis. Ou seja:

```
bools 3
[[False, False, False],
 [False, False, True],
 [False, True, False],
 [False, True, True],
 [True, False, False],
 [True, False, True],
 [True, True, False],
 [True, True, True],
```

Observe que a lista `bools 3` possui duas cópias da lista `bools 2`, uma precedida de `False` e outra de `True`. Ou seja, `bools n` pode ser definida recursivamente acrescentando `False` em cada lista de `bools (n-1)`, acrescentando `True` em cada lista de `bools (n-1)` e concatenando os dois resultados:

```
bools :: Int -> [[Bool]]
```

Agora basta criar todos os mapas de substituição utilizando as listas de substituições de `bools`:

```
-- Exemplo: substs (Var 'A') `And` (Var 'B') deve gerar
-- [[('A', False), ('B', False)], [('A', False), ('B', True)],
--  [('A', True), ('B', False)], [('A', True), ('B', True)]]
substs :: Prop -> [Subst]
```

Finalmente, basta definir a função que verifica se a proposição é uma tautologia:

```
isTaut :: Prop -> Bool
```