Programação Funcional
Aula 14 — Um verificador de tautologias

Pedro Vasconcelos
DCC/FCUP

2021

Proposições lógicas

Uma *proposição lógica* é construida apartir de:

constantes V, F (*verdade* e *falsidade*)

variáveis $a, b, \ldots\ldots$

conectivas lógicas $\wedge, \vee, \neg, \implies$

parêntesis $(,)$

Exemplos:

$$x \wedge \neg y$$

$$x \wedge ((\neg x) \implies F)$$

$$(\neg(a \vee b)) \implies ((\neg a) \wedge (\neg b))$$

Tabelas de verdade das conectivas

a	b	$a \wedge b$
F	F	F
V	F	F
F	V	F
V	V	V

a	b	$a \vee b$
F	F	F
V	F	V
F	V	V
V	V	V

a	$\neg a$
F	V
V	F

a	b	$a \Longrightarrow b$
F	F	V
V	F	F
F	V	V
V	V	V

Tautologias

Uma tautologia é uma proposição cujo valor é *verdade* para qualquer atribuição de valores às variáveis.

Exemplo:

a	$\neg a$	$a \vee (\neg a)$
F	V	V
V	F	V

Conclusão: $a \vee (\neg a)$ é uma tautologia.

Representação de proposições

Vamos definir um tipo recursivo para representar proposições.

```
data Prop = Const Bool      -- constante
          | Var Char        -- variável
          | Neg Prop        -- negação
          | Conj Prop Prop  -- conjunção
          | Disj Prop Prop  -- disjunção
          | Impl Prop Prop  -- implicação
          deriving (Eq,Show)
```

Representação de proposições (cont.)

Exemplo: a proposição

$$a \implies ((\neg a) \implies F)$$

será representada por

```
Impl (Var 'a') (Impl (Neg (Var 'a')) (Const False))
```

Associação de valores a variáveis

Para atribuir valores de verdade às variáveis vamos usar uma *lista de associações*.

Exemplo: a atribuição

$$\begin{cases} a = V \\ b = F \\ c = V \end{cases}$$

será representada pela lista

```
[('a',True),('b',False),('c',True)]
```

Associação de valores a variáveis (cont.)

Definimos:

- *listas de associações* entre *chaves* e *valores*

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

- uma função para procurar o valor associado a uma chave

```
find :: Eq k => k -> Assoc k v -> v
find k assocs = head [v | (k',v)<-assocs, k==k']
```

`find` é uma função parcial: dá um erro `empty list` se não encontrar a chave.

Calcular o valor duma proposição

Vamos definir o *valor de verdade* de uma proposição por recursão.

O primeiro argumento é uma *atribuição de valores* às variáveis.

```
type Atrib = Assoc Char Bool

valor :: Atrib -> Prop -> Bool
valor s (Const b)  = b
valor s (Var x)    = find x s
valor s (Neg p)    = not (valor s p)
valor s (Conj p q) = valor s p && valor s q
valor s (Disj p q) = valor s p || valor s q
valor s (Impl p q) = not (valor s p) || valor s q
```

Gerar atribuições às variáveis

- Com 1 variável a tabela tem 2 linhas
- Com 2 variáveis a tabela tem 4 linhas
- Com 3 variáveis a tabela tem 8 linhas
- Com n variáveis a tabela tem 2^n linhas

Para obter todas as atribuições de forma sistemática vamos definir uma função auxiliar para gerar as sequências de n valores boleanos:

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

Gerar atribuições às variáveis (cont.)

Exemplo, as sequências de comprimento 3 (três variáveis):

False	*False*	*False*
False	*False*	*True*
False	*True*	*False*
False	*True*	*True*
True	*False*	*False*
True	*False*	*True*
True	*True*	*False*
True	*True*	*True*

} *bits* 3

Gerar atribuições às variáveis (cont.)

Podemos decompor em duas cópias da tabela para 2 variáveis com uma coluna extra:

False	*False*	*False*	} *bits 2*
False	*False*	*True*	
False	*True*	*False*	
False	*True*	*True*	
True	*False*	*False*	} *bits 2*
True	*False*	*True*	
True	*True*	*False*	
True	*True*	*True*	

Gerar atribuições às variáveis (cont.)

Generalização: podemos escrever gerar as sequências de forma recursiva.

```
bits :: Int -> [[Bool]]
bits 0 = [[]]
bits n = [b:bs | bs<-bits (n-1), b<-[False,True]]
```

Gerar atribuições às variáveis (cont.)

Para gerar atribuições começamos por listar todas as variáveis numa proposição.

```
vars :: Prop -> [Char]
vars (Const _)   =  []
vars (Var x)     =  [x]
vars (Neg p)     =  vars p
vars (Conj p q)  =  vars p ++ vars q
vars (Disj p q)  =  vars p ++ vars q
vars (Impl p q)  =  vars p ++ vars q
```

Gerar atribuições às variáveis (cont.)

A função seguinte gera todas as as atribuições de variáveis duma proposição:

```
atribs :: Prop -> [Atrib]
atribs p = map (zip vs) (bits (length vs))
     where vs = nub (vars p)
```

(A função *nub* do módulo *Data.List* remove variáveis repetidas.)

Verificar tautologias

Uma proposição é tautologia se e só se o seu valor for `True` para todas as atribuições de variáveis.

```
tautologia :: Prop -> Bool
tautologia p = and [valor s p | s<-atribs p]
```

Verificar tautologias (cont.)

Alguns exemplos:

```
> tautologia (Var 'a')
False

> tautologia (Impl (Var 'p') (Var 'p'))
True

> tautologia (Disj (Var 'a') (Neg (Var 'a')))
True
```

Extras

- Escrever uma função que calcula a lista das atribuições que tornam uma proposição *falsa* (i.e. uma lista de contra-exemplos)
- Escrever um programa para imprimir a tabela de verdade duma proposição