

Lógica

BCC22 Programação Funcional — Aula Prática 05

Lucília Camarão de Figueiredo

A prática deve ser realizada antes da aula prevista para a discussão da mesma. Verifique a data prevista para a discussão de cada aula prática no calendário do curso disponível em www.dcc.ufmg.br/~lucilia/cursos/func. Traga para a aula de discussão o código que você implementou. A discussão não será eficaz se você não tentar fazer os exercícios previamente.

Você pode trabalhar em conjunto com outros colegas, mas deve entender o que está fazendo, pois não irá contar com a ajuda de colegas durante os exames.

A realização das práticas e presença nas aulas são obrigatórias e valem pontos na disciplina.

1 Lógica

Nesta prática você vai implementar Lógica Proposicional em Haskell. No arquivo `pratica05.hs` você verá as seguintes declarações de tipo:

```
type Name = String

data Prop = Var Name          -- variável
          | F                  -- falso
          | T                  -- verdadeiro
          | Not Prop           -- negação
          | Prop :&: Prop       -- conjunção
          | Prop :|: Prop       -- disjunção
```

O tipo `Prop` representa fórmulas da Lógica Proposicional. Variáveis de proposição, tais como P e Q , podem ser representadas como `Var "P"` e `Var "Q"`. Além disso, temos as constantes booleanas `T` e `F` para ‘verdadeiro’ e ‘falso’, o predicado unário `Not` para negação (não confundir com a função `not :: Bool -> Bool`), e os predicados binários `:|:` e `:&:` para disjunção (\vee) e conjunção (\wedge).

Outro tipo definido no arquivo `pratica05.hs` é

```
type Env = [(Name, Bool)]
```

O tipo `Env` é usado como um ‘ambiente’ (*environment*) no qual são avaliadas proposições: ele consiste em uma lista de atribuição de valores-verdade a variáveis proposicionais. Usando esses tipos, são definidas no arquivo `pratica05.hs` as seguintes funções:

- `satisfiable :: Prop -> Bool`, que verifica se uma fórmula é satisfazível, isto é, se existe alguma possível atribuição de valores-verdade às variáveis da fórmula de modo que a fórmula seja verdadeira.

```
Main> satisfiable (Var "P" :&: Not (Var "P"))
False
Main> satisfiable ((Var "P" :&: Not (Var "Q")) :&: (Var "Q" :|: Var "P"))
True
```

- `eval :: Env -> Prop -> Bool`, que avalia uma dada proposição no ambiente dado. Por exemplo:

```
Main> eval [("P", True), ("Q", False)] (Var "P" :|: Var "Q")
True
```

- `showProp :: Prop -> String`, que converte uma proposição para uma notação mais próxima da notação matemática usual. Por exemplo:

```
Main> showProp (Not (Var "P") :&: Var "Q")
"((~P)& Q)"
```

- `names :: Prop -> Names`, que retorna todas os nomes de variáveis que ocorrem em uma proposição. Exemplo:

```
Main> names (Not (Var "P") :&: Var "Q")
["P", "Q"]
```

- `envs :: Names -> [Env]` que gera a lista de todas as possíveis atribuições de valores-verdade para a lista de variáveis dada. Exemplo:

```
Main> envs ["P", "Q"]
[("P",False),("Q",False)],
[("P",False),("Q",True)],
[("P",True),("Q",False)],
[("P",True),("Q",True)] ]
```

- `table :: Prop -> IO ()`, que imprime a tabela-verdade de uma fórmula.

```
Main> table ((Var "P":&: Not (Var "Q")) :&: (Var "Q":|: Var "P"))
P Q | ((P&(~Q))&(Q|P))
- - | -----
F F |          F
F T |          F
T F |          T
T T |          F
```

- `fullTable :: Prop -> IO ()`, que imprime a tabela-verdade de uma fórmula incluindo os valores verdade de todas as suas subfórmulas. (*Nota:* `fullTable` usa a função `subformulas`, que você irá definir no exercício 5, portanto, ela não funciona ainda.)

```
Main> fullTable ((Var "P":&: Not (Var "Q")) :&: (Var "Q":|: Var "P"))
P Q | ((P&(~Q))&(Q|P)) (P&(~Q)) (~Q) (Q|P)
- - | -----
F F |          F          F          T          F
F T |          F          F          F          T
T F |          T          T          T          T
T T |          F          F          F          T
```

Exercícios

1. Escreva as seguintes fórmulas como Props (chame-as `p1`, `p2` e `p3` respectivamente, no seu programa):

- (a) $((P \vee Q) \wedge (P \vee Q))$
 (b) $((P \vee Q) \wedge ((\neg P) \wedge (\neg Q)))$
 (c) $((P \wedge (Q \vee R)) \wedge (((\neg P) \vee (\neg Q)) \wedge ((\neg P) \vee (\neg R))))$
2. (a) Usando `names`, `envs` e `eval`, escreva a função `tautology :: Prop -> Bool`, que verifica se uma proposição é uma tautologia. Teste essa função nos exemplos do exercício (1) e nas negações de cada uma dessas fórmulas.
- (b) Defina dois testes QuickCheck para verificar que sua função `tautology` funciona corretamente. Use como base os seguintes fatos:

Para qualquer fórmula proposicional P ,

- i. ou P é uma tautologia ou $\neg P$ é satisfazível,
- ii. ou P não é satisfazível ou $\neg P$ não é uma tautologia.

Nota: cuidado para não confundir a negação para valores do tipo `Boolean` (`not`) e a negação para proposições, de tipo `Prop` (`Not`).

3. Vamos agora estender a representação de proposições, assim como algumas funções definidas anteriormente, de maneira a considerar proposições formadas com os conectivos lógico de implicação (\rightarrow) e de bi-implicação (\leftrightarrow). Depois que você tiver implementado essas extensões, as tabelas-verdade para esses conectivos devem ser as seguintes:

Main> table (Var "P" :->: Var "Q")		
P	Q	(P → Q)
-	-	-----
F	F	T
F	T	T
T	F	F
T	T	T

Main> table (Var "P" <->: Var "Q")		
P	Q	(P ↔ Q)
-	-	-----
F	F	T
F	T	F
T	F	F
T	T	T

- (a) Estenda a definição do tipo de dado `Prop` de maneira a incluir os conectivos de implicação e bi-implicação.
- (b) Modifique as definições das funções `showProp`, `names` e `eval` de maneira a considerar proposições construídas com esses conectivos. Teste suas definições imprimindo as tabelas-verdade acima.
- (c) Defina cada uma das proposições a seguir como `Props` (respectivamente, proposições `p4`, `p5` e `p6` no seu programa). Verifique se cada uma é ou não satisfazível e imprima a tabela-verdade de cada uma delas.
- (a) $((P \rightarrow Q) \wedge (P \leftrightarrow Q))$
 (b) $((P \rightarrow Q) \wedge (P \wedge (\neg Q)))$
 (c) $((P \leftrightarrow Q) \wedge ((P \wedge (\neg Q)) \vee ((\neg P) \wedge Q)))$
- (d) Abaixo da seção ‘`exercises`’ da `pratica05.hs`, na seção intitulada ‘`for QuickCheck`’, você vai encontrar uma declaração que começa com:

```
instance Arbitrary Prop where
```

Isso informa ao QuickCheck como gerar valores arbitrários do tipo `Prop` de modo a realizar testes. Para fazer com que o QuickCheck use os novos construtores, remova o comentário das duas linhas do meio dessa definição:

```
-- , liftM2 (:->:) subform subform
-- , liftM2 (<->:) subform' subform'
```

Agora você pode testar as propriedades do exercício (2b) novamente.

4. Duas fórmulas são equivalentes se elas têm sempre o mesmo valor verdade, para cada uma das possíveis atribuições de valores-verdade às suas variáveis (e, é claro, se possuem o mesmo conjunto de variáveis). Em outras palavras, duas fórmulas são equivalentes se, em qualquer *environment*, elas são ambas verdadeiras, ou são ambas falsas.

- (a) Escreva a função `equivalent :: Prop -> Prop -> Bool`, que retorna `True` apenas se as duas proposições dadas como argumento são equivalentes. Por exemplo:

```
Main> equivalent (Var "P":& Var "Q") (Not (Not (Var "P") :|: Not (Var "Q")))
True
Main> equivalent (Var "P") (Var "Q")
False
Main> equivalent (Var "R":|: Not (Var "R")) (Var "Q":|: Not (Var "Q"))
True
```

Você pode usar as funções `names` e `envs` para gerar os *environments* relevantes, e usar `eval` para avaliar as duas proposições em cada *environment*.

- (b) Escreva outra versão da função `equivalent`, agora combinando os dois argumentos em uma nova proposição e usando a função `tautology` ou a função `satisfiable`.
- (c) Escreva um teste QuickCheck para verificar que as suas duas implementações são equivalentes.

As *subformulas* de uma proposição podem ser definidas recursivamente do seguinte modo:

- Uma variável proposicional P tem como subfórmula apenas ela própria. Idem para as constantes T e F .
- Uma proposição da forma $\neg P$ tem como subfórmulas ela própria e todas as subfórmulas de P .
- Uma proposição da forma $P \wedge Q$, $P \vee Q$, $P \rightarrow Q$, ou $P \leftrightarrow Q$ tem como subfórmulas ela própria e todas as subfórmulas de P e de Q .

A função `fullTable :: Prop -> IO ()`, já definida no arquivo `pratica05.hs`, imprime a tabela-verdade de uma fórmula, incluindo uma coluna para cada uma de suas subfórmulas não triviais.

Exercícios

3. Defina a função `subformulas :: Prop -> [Prop]`, que retorna a lista de todas as subfórmulas de uma fórmula. Por exemplo:

```
Main> map showProp (subformulas p2)
["((P|Q)&((~P)&(~Q)))", "(P|Q)", "P", "Q", "((~P)&(~Q))", "(~P)", "(~Q)"]
```

(É necessário usar `map showProp` para converter proposição em um string; caso contrário seria mais difícil visualizar o resultado.)

Teste as funções `subformulas` e `fullTable` para cada uma das proposições definidas anteriormente (p1 a p6).

2 Exercícios adicionais

2.1 Formas Normais

Nesta parte do trabalho vamos obter diferentes *formas normais* para uma proposição. Primeiramente, vamos lidar com a *forma normal negativa*. Uma fórmula está na forma normal negativa se ela consiste apenas dos conectivos \vee, \wedge , letras proposicionais P , ou letras proposicionais negadas $\neg P$, além das constantes t e f . Portanto, a negação apenas é aplicada a variáveis proposicionais. Para transformar uma fórmula para a *forma normal negativa*, você pode usar as seguintes equivalências:

$$\begin{aligned}\neg(P \wedge Q) &\Leftrightarrow (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) &\Leftrightarrow (\neg P) \wedge (\neg Q) \\ (P \rightarrow Q) &\Leftrightarrow (\neg P) \vee Q \\ (P \leftrightarrow Q) &\Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P) \\ \neg(\neg P) &\Leftrightarrow P\end{aligned}$$

Exercícios

7. Defina a função `toNNF :: Prop -> Prop` que transforma uma proposição para a forma normal negativa.
8. Defina a função `isNNF :: Prop -> Bool` que determina se uma proposição está na forma normal negativa. Use as funções de teste `prop_NNF1` e `prop_NNF2` para verificar se suas definições estão corretas.

Agora vamos transformar uma fórmula para a *forma normal conjuntiva*. Isso significa que a fórmula é uma conjunção de cláusulas, onde cada cláusula é uma disjunção de variáveis proposicionais (possivelmente negadas).

Você precisará ter atenção especial ao caso das constantes t e f : elas próprias são consideradas como fórmulas em forma normal conjuntiva, mas não devem ocorrer em fórmulas mais complexas em forma normal conjuntiva. Essas constantes podem ser eliminadas usando-se as seguintes equivalências:

$$\begin{aligned}(P \wedge t) &\Leftrightarrow (t \wedge P) \Leftrightarrow P \\ (P \wedge f) &\Leftrightarrow (f \wedge P) \Leftrightarrow f \\ (P \vee t) &\Leftrightarrow (t \vee P) \Leftrightarrow t \\ (P \vee f) &\Leftrightarrow (f \vee P) \Leftrightarrow P\end{aligned}$$

Exercícios

9. Defina a função `isCNF :: LProp -> Bool` que determina se uma proposição está na forma normal conjuntiva.

Uma maneira comum de representar fórmulas na *forma normal conjuntiva* é como uma lista de listas. Assim:

$$((A \vee B) \wedge ((C \vee D) \vee E)) \wedge G \Leftrightarrow [[A, B], [C, D, E], [G]]$$

10. Pense como as constantes t e f podem ser representadas como listas.
Dica: uma fórmula em forma normal conjuntiva é verdadeira quando todas as suas cláusulas são verdadeiras. Uma cláusula é verdadeira se qualquer dos seus átomos é verdadeiro.
11. Escreva a função `listsToCNF` que traduz uma lista de lista de `Props` (que você deve supor que sejam variáveis ou negações de variáveis) como uma `Prop` em forma normal conjuntiva.
12. Escreva a função `listsFromCNF` que converte uma fórmula em forma normal conjuntiva para sua representação como uma lista de listas de átomos.

Finalmente, você vai converter uma fórmula **Prop** arbitrária em uma lista de listas. Você pode usar a seguinte lei de distributividade (verifique que essa equivalência é válida usando funções que você definiu anteriormente):

$$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$$

Ou, em uma versão mais geral:

$$\begin{array}{c} (P_1 \wedge P_2 \wedge \dots \wedge P_m) \vee (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n) \\ \Updownarrow \\ (P_1 \vee Q_1) \wedge (P_1 \vee Q_2) \wedge (P_1 \vee Q_3) \wedge \dots \wedge (P_1 \vee Q_n) \wedge \\ (P_2 \vee Q_1) \wedge (P_2 \vee Q_2) \wedge (P_2 \vee Q_3) \wedge \dots \wedge (P_2 \vee Q_n) \wedge \\ \dots \\ (P_m \vee Q_1) \wedge (P_m \vee Q_2) \wedge (P_m \vee Q_3) \wedge \dots \wedge (P_m \vee Q_n) \end{array}$$

Exercícios

13. Escreva a função **toCNFList** que converte uma **Prop** em uma lista de listas, representando a fórmula em forma normal conjuntiva.

Nota: transformar uma fórmula para a forma normal conjuntiva tem custo computacional muito alto, especialmente para fórmulas que envolvem grande número de bi-implicações (\leftrightarrow). Ao testar suas funções, escolha como dados de teste fórmulas não muito complexas. Depois, teste sua função usando **QuickChek**, pela propriedade **prop_CNF**.