Exame Especial de Programação Funcional

ATENÇÃO

- A interpretação dos enunciados faz parte da avaliação.
- A avaliação deve ser resolvida INDIVIDUALMENTE. Qualquer tipo de plágio será punido de acordo com as regras vigentes na universidade.
- Você deverá incluir uma foto especificando o tipo de avaliação escolhida. A foto deve possuir o nome MATRICULA.JPG, em que MATRICULA é o número de matrícula do aluno. Por exemplo, se o aluno possui a matrícula 20.1.1010 o arquivo deve ser 2011010.jpg. A opção de prova a ser resolvida deve ser uma das seguintes:

1. Substitutiva da avaliação 1. Neste caso você deve resolver as questões 1, 2 e 3. Você só poderá escolher essa opção se não tiver feito a avaliação 1.
2. Substitutiva da avaliação 2. Neste caso você deve resolver as questões 1, 2, 3, 4 e 5. Você só poderá escolher essa opção se não tiver feito a avaliação 2.
3. Substitutiva da avaliação 3. Neste caso você deve resolver as questões 6 e 7. Você só poderá escolher essa opção se não tiver feito a avaliação 3.
4. Exame especial total: Você deverá resolver todas as questões desta avaliação.

Caso o aluno não especifique qual opção de prova, será considerado que ele optou por realizar o exame especial total. Se o aluno faltou a mais de uma avaliação, ele só poderá escolher o exame total.

- Se você utilizar recursos disponíveis na internet e que não fazem parte da bibliografia, você deverá explicitamente citar a fonte apresentando o link pertinente como um comentário em seu código.
- Todo código produzido por você deve ser acompanhado por um texto explicando a estratégia usada para a solução. Lembre-se: meramente parafrasear o código não é considerado uma explicação!
- Não é permitido modificar a seção Setup inicial do código, seja por incluir bibliotecas ou por eliminar a diretiva de compilação -Wall.
- Seu código deve ser compilado sem erros e warnings de compilação. A presença de erros acarretará em uma penalidade de 20% para cada erro de compilação e de 10% para cada warning. Esses valores serão descontados sobre a nota final obtida pelo aluno.
- Todo o código a ser produzido por você está marcado usando a função "undefined". Sua solução deverá substituir a chamada a undefined por uma implementação apropriada.
- Sobre a entrega da solução:

1. A entrega da solução da avaliação deve ser feita como um único arquivo .zip contendo todo o projeto stack usado.
2. O arquivo .zip a ser entregue deve usar a seguinte convenção de nome: MATRÍCULA.zip, em que matrícula é a sua matrícula. Exemplo: Se sua matrícula for 20.1.2020 então o arquivo entregue deve ser 2012020.zip. A não observância ao critério de nome e formato da solução receberá uma penalidade de 20% sobre a nota obtida na avaliação.
3. O arquivo de solução deverá ser entregue usando a atividade "Entrega da Exame Especial" no Moodle dentro do prazo estabelecido.
4. É de responsabilidade do aluno a entrega da solução dentro deste prazo.
5. Sob NENHUMA hipótese serão aceitas soluções fora do prazo ou entregues usando outra ferramenta que não a plataforma Moodle.

Setup Inicial

```haskell
{-# OPTIONS_GHC -Wall #-}

module Main where

main :: IO ()
main = return ()
```

Validando dados

Um problema central na computação é o de validação de dados em software. Dados utilizados por programas vem das mais diferentes fontes: arquivos, via rede, interação com usuários, entre muitos outros. Nesta avaliação, você deverá construir uma linguagem de domínio específico (DSL - Domain Specific Language) para validação de dados utilizando a linguagem Haskell.

Introdução

O problema central a ser enfrentado nessa avaliação consiste na definição de uma linguagem para validação de dados. Em Haskell, podemos representar linguagens por tipos de dados algébricos e funções sobre estes. Para nossos objetivos, vamos considerar o tipo `Constr a` que representa uma *restrição* sobre valores de tipo `a`.

```haskell
type Label = String

data Constr a
  = Atom Label (a -> Bool)
  | (Constr a) :&: (Constr a)
  | (Constr a) :|: (Constr a)
  | Not (Constr a)
```

O significado dos construtores de `Constr a` é como se segue: o construtor `Atom` representa uma condição atômica, que é associada a um rótulo (codificado como uma string). Restrições podem ser combinadas utilizando os construtores `:&:`, que representa a conjunção ("e" entre valores lógicos); `:|:`, que representa a disjunção ("ou" entre valores lógicos) e `Not` que denota a negação booleana.

Com base no apresentado, desenvolva as questões a seguir.

Parte I: Funções de ordem superior e tipos de dados algébricos

Questão 1. Desenvolva a função

```
foldConstr :: (Label -> (a -> Bool) -> b) ->
              (b -> b -> b)               ->
              (b -> b -> b)               ->
              (b -> b)                    ->
              Constr a                    ->
              b
foldConstr = undefined
```

que consiste do operador `fold` para o tipo de dados `Constr a`. O tipo da função `foldConstr` espera 5 argumentos, sendo que os 4 primeiros são funções que devem ser utilizadas para processar o caso base (construtor `Atom`) e as demais para combinar os resultados de chamadas recursivas. O segundo parâmetro (tipo `b -> b -> b`) é usado para combinar os resultados de chamadas recursivas para o construtor `:&:`, o terceiro para o construtor `:|:` e o quarto para o construtor `Not`.

Questão 2. Desenvolva a função

```
validate :: Constr a -> a -> Bool
validate = undefined
```

que a partir de uma restrição de validação (valor de tipo `Constr a`) e de um valor de tipo `a` retorne verdadeiro caso o segundo parâmetro satisfaça a restrição codificada pelo primeiro parâmetro. Sua implementação deverá ser feita utilizando a função `foldConstr` definida por você na Questão 1.

Questão 3. Outro tipo importante a ser utilizado nesta biblioteca de validação é os que "explica" o sucesso ou a falha de uma validação. Esses valores podem ser utilizados para emitir melhores mensagens de erro ou mesmo para depurar o motivo de um valor ser aceito. Representamos esses "certificados" utilizando o tipo `Cert`.

```
type Cert a = Either (Failure a) (Success a)
```

O tipo `Success` representa certificados de que uma restrição foi executada com sucesso sobre um valor de tipo `a`. Por sua vez o tipo `Failure` representa que uma restrição não é satisfeita.

```haskell
data Success a
     = SAtom Label a
     | SPair (Success a) (Success a)
     | SLeft (Success a)
     | SRight (Success a)
     | SFail (Failure a)
     deriving (Eq, Ord, Show)
```

O construtor `SAtom` representa que uma restrição construída pelo construtor `Atom` foi atendida por uma valor de tipo `a` recebida como segundo argumento. O construtor `SPair` indica que uma restrição de conjunção foi executada com sucesso. Por sua vez, os construtor `SLeft` e `SRight` indicam que uma disjunção foi executada de maneira apropriada. Finalmente, o construtor `SFail` representa que a restrição construída usando o construtor `Not` teve sucesso.

```haskell
data Failure a
     = FAtom Label a
     | FPair (Failure a) (Failure a)
     | FLeft (Failure a)
     | FRight (Failure a)
     | FFail (Success a)
     deriving (Eq, Ord, Show)
```

Os construtores para o tipo `Failure a` possuem significado dual aos de `Success a`. Isto é, o construtor `FPair` é usado para justificar uma falha da disjunção; os construtores `FLeft` e `FRight` são usados para construir certificados para a conjunção e `FFail` para a negação.

Diante do exposto, desenvolva a função

```haskell
certify :: Constr a -> a -> Cert a
certify p v
   = undefined
```

que constrói evidências de validação de uma restrição para um valor de tipo `a` fornecido como segundo argumento. Sua definição de `certify` deverá ser feita utilizando a função `foldConstr`.

Parte II: Classes de tipos e functores aplicativos

Na segunda parte desta avaliação vamos considerar o problema de construir abstrações para a representação de restrições.

Questão 4. Toda linguagem de domínio específico possui uma interface amigável para expressar suas construções. O objetivo desta questão é o desenvolvimento de funções para representar esta interface.

a) Implemente a função

```
(.:) :: Label -> (a -> Bool) -> Constr a
(.:) = undefined
```

que a partir de um rótulo (tipo `Label`) e uma função de tipo `a -> Bool` constrói uma restrição.

b) Implemente a função

```
(.&.) :: Constr a -> Constr a -> Constr a
(.&.) = undefined
```

que constrói a conjunção de duas restrições fornecidas como argumentos.

c) Implemente a função

```
(.|.) :: Constr a -> Constr a -> Constr a
(.|.) = undefined
```

que constrói a disjunção de duas restrições fornecidas como argumentos.

d) Implemente a função

```
no :: Constr a -> Constr a
no = undefined
```

que constrói a negação de duas restrições fornecidas como argumentos.

e) Implemente as funções

```
true :: Constr a
true = undefined

false :: Constr a
false = undefined
```

que denotam restrições para as constantes lógicas verdadeiro (`true`) e (`false`).

Questão 5. A partir da interface simples construída por você na questão anterior, podemos modelar restrições que podem ser usadas para validar permissões em uma rede social. Para isso, considere os seguintes tipos de dados.

```
type Name = String

data User
      = User {
          name :: Name
        , isRegistered :: Bool
        } deriving (Eq, Ord, Show)

data Post
      = Post {
          user :: User
        , text :: String
        , public :: Bool
```

```
        , anonymous :: Bool
        } deriving (Eq, Ord, Show)
```

O tipo `User` representa usuários da rede social e armazena o nome e se este está ou não registrado. Um post da rede é representado por um valor do tipo `Post` e este é composto pelo usuário que criou o post, o texto que forma o post e dois valores booleanos que representam se o post é público e se este pode receber comentários anônimos.

```
data Network = Network User Post deriving (Eq, Ord, Show)
```

Faça o que se pede:

a) Implemente a seguinte restrição utilizando as funções da Questão 4: "Comentários podem ser feitos apenas por usuários registrados ou se o post permitir comentários anônimos ou se este for público."

```
registeredCanComment :: Constr Network
registeredCanComment
    = undefined
```

b) Implemente a seguinte restrição utilizando as funções da Questão 4: "Um usuário pode editar um post se ele é o autor do post e se o autor é registrado na rede".

```
canEdit :: Constr Network
canEdit
    = undefined
```

Parte III: Provas de propriedades de programas

Considere a seguinte função que cria a negação de uma restrição

```
negg :: Constr a -> Constr a
negg (Atom n f) = Atom n (not . f)
negg (c :&: c') = (negg c) :|: (negg c')
negg (c :|: c') = (negg c) :&: (negg c')
negg (Not c) = negg c
```

Prove as seguintes propriedades por indução:

Questão 6. Para toda restrição `c` e valores `v`,

```
not (validate c v) = validate (negg c) v
```

Questão 7. Para toda restrição `c` e valores `v`,

```
validate (negg (negg c)) v = validate c v
```