Eduardo Ribeiro de Oliveira 11201810478

Desenvolvimento guiado por tipos em Idris

Introdução

Desenvolvimento guiado por tipos é uma metodologia de desenvolvimento de software que se baseia na criação de tipos de dados significativos antes de implementar a lógica de negócios. Em Idris, essa abordagem é especialmente poderosa devido ao sistema de tipos dependente da linguagem, que permite expressar propriedades sofisticadas sobre nossos programas diretamente na assinatura de tipos.
Neste tutorial, vamos usar um exemplo simples de um sistema de tipos para veículos que podem ser movidos por diferentes fontes de energia: combustão ou elétrica. Nosso objetivo é explorar como podemos usar o desenvolvimento guiado por tipos para criar um sistema robusto e seguro.

Problema

Primeiro vamos definir e entender o problema, vamos supor que precisamos desenvolver um sistema simples para uma locadora de veículos. Essa locadora trabalha com dois tipos de veículos: a combustão e elétricos. Além disso, também há a distinção do porte do veículo, podendo ser carros ou ônibus. Ou seja, precisamos trabalhar com diferentes tipos de veículos, sendo que existem diferenças entre eles e cada um tem suas peculiaridades, por exemplo, não podemos abastecer um veículo elétrico, não faz sentido, precisamos que caso haja uma função para abastecer um veículo, ela seja capaz de verificar se o tipo do veículo parametrizado é capaz de ser abastecido, caso não seja, a função não poderá ser chamada. Podemos também estender esse pensamento para a recarga de um veículo elétrico, criando uma função que aceita apenas veículos do tipo elétrico.
Ademais, como tipos diferentes de veículos produzem diferentes resultados, podemos criar uma função que calcula a autonomia de um veículo a partir de seu combustível/bateria dependendo do tipo do parâmetro que foi passado a ela. Por fim, podemos criar uma função que simula uma viagem de qualquer tipo de veículo, verificando se tal veículo possui a energia necessária para completar uma viagem dada uma distância e retornando o estado do veículo após tal viagem.

Desenvolvimento

Vamos começar definindo nossos tipos de dados em Idris. Primeiro, precisamos definir os tipos de fonte de energia e os tipos de veículos associados a essas fontes de energia:

```
data FonteEnergia = Combustao | Eletrica

-- Definição do tipo Veiculo parametrizado pelo tipo de fonte de energia
data Veiculo : FonteEnergia -> Type where
 CarroC : (combustivel : Double) -> Veiculo Combustao
 CarroE : (bateria : Double) -> Veiculo Eletrica
 Onibus : (combustivel : Double) -> Veiculo Combustao
```

Aqui, *FonteEnergia* é um tipo de dados enumerado que representa as diferentes fontes de energia. O tipo *Veiculo* é dependente do tipo *FonteEnergia* e define os diferentes tipos de veículos associados a cada fonte de energia tal que cada veículo possui um quantificador representativo da sua energia armazenada, para os veículos a combustão temos o combustível e para os elétricos temos a bateria.
Para ter certeza se está tudo correto, podemos utilizar o REPL, criar um CarroC e verificar seu tipo:

```
Main> :let x = CarroC 50
Main> :t x
Main.x : Veiculo Combustao
```

Agora que os tipos já foram definidos podemos começar a implementar as funções. Vamos começar com as funções abastecer e recarregar, porém como declarado anteriormente, essas funções possuem restrições, não podemos deixar o tipo veículo elétrico conseguir chamar essa função, apenas veículos à combustão. Aqui que realmente começa as imposições sobre uma ou mais restrições. Como na assinatura da função podemos definir os tipos dos parâmetros, podemos simplesmente definir que uma função *x* recebe apenas o tipo *y*. No caso da função *abastecer*, podemos definir que só pode receber apenas o tipo *Veiculo Combustao*. Analogamente, devemos fazer o mesmo para função *recarregar*, para que ela tenha restrição de receber apenas o tipo *Veiculo Eletrico*. Então, ficamos com a seguinte implementação:

```
-- Função para abastecer veículos movidos a combustão
abastecer : Veiculo Combustao -> Veiculo Combustao
abastecer (CarroC _) = CarroC 100.0
abastecer (Onibus _) = Onibus 200.0

-- Função para recarregar veículos elétricos
recarregar : Veiculo Eletrica -> Veiculo Eletrica
recarregar (CarroE _) = CarroE 100
```

Agora, temos que testar para ver se a restrição está realmente funcionando. Podemos usar o CarroC declarado anteriormente e tentar usá-lo como parâmetro na função *recarregar*:

```
Main> recarregar x
Error: When unifying:
      Veiculo Combustao
and:
      Veiculo Eletrica
Mismatch between: Combustao and Eletrica.

(Interactive):1:12--1:13
 1 | recarregar x
                ^
```

Não pode! O compilador deixa explícito o problema: a função espera o tipo *Veiculo Eletrica*, porém o tipo recebido como parâmetro é *Veiculo Combustao*, como os tipos não se casam, ocasiona-se o erro. Tudo como esperado, queríamos restringir as funções de acordo com os tipos e estamos conseguindo, podemos seguir em frente.
Seguindo o roteiro, podemos usar o mesmo raciocínio e implementar funções que retornam a autonomia de cada tipo de veículo:

```
-- Função para calcular a autonomia de veículos movidos a combustão
autonomiaC : Veiculo Combustao -> Double
autonomiaC (CarroC x) = x * 15.0
autonomiaC (Onibus x) = x * 8.0

-- Função para calcular a autonomia de veículos elétricos
autonomiaE : Veiculo Eletrica -> Double
autonomiaE (CarroE x) = x * 9.5
```

A restrição dessas funções é igual as funções anteriores, ou seja, ela cumpre o papel de segurança a nível de tipos, porém, será se é possível criar uma função que recebe um tipo a (nesse caso, *Combustao* ou *Eletrico*) e seja capaz de a partir desse tipo a, conseguir chamar uma das funções implementadas acima? Em Idris isso é possível e bem simples, basta definir na assinatura da função que receberemos um tipo *Veiculo a* como parâmetro, desde que a seja do tipo *FonteEnergia*. Com isso, podemos comparar *a* com cada tipo de *FonteEnergia* e então chamar a respectiva função de autonomia, ou seja, caso *a* for igual a *Combustao*, chamamos *autonomiaC*. Para ficar mais claro, segue a implementação:

```
-- Função para calcular a autonomia de qualquer tipo de veículo
autonomia : {a : FonteEnergia} -> Veiculo a -> Double
autonomia {a = Combustao} x = autonomiaC x
autonomia {a = Eletrica} x = autonomiaE x
```

Nesse caso, a função *autonomia* consegue funcionar com diferentes tipos desde que o parâmetro seja do tipo *Veiculo*. Alguns testes:

```
Main> :let x = CarroC 50
Main> :let y = CarroE 80
Main> autonomiaC x
750.0
Main> autonomiaE y
760.0
Main> autonomia x
750.0
Main> autonomia y
760.0
Main> autonomiaC y
Error: When unifying:
      Veiculo Eletrica
and:
      Veiculo Combustao
Mismatch between: Eletrica and Combustao.

(Interactive):1:12--1:13
 1 | autonomiaC y
            ^
```

Tudo correto! Seja a distribuição feita pela função *autonomia,* seja a segurança a nível de tipos de cada função específica.
Por fim, para finalizar nosso plano de desenvolvimento, resta a função que simula uma viagem de um veículo. Para implementar essa função podemos utilizar a estratégia adotada na função *autonomia* para trabalhar com diferentes tipos de veículos em uma mesma função, manter a segurança para garantir que estamos lidando apenas com um tipo de *Veiculo FonteEnergia*, e então tratar cada caso como quisermos. Assim ficou a implementação:

```
-- Função que simula uma viagem com um determinado veículo, levando em
consideração
-- a distância do destino e imprime o estado final do veículo após a
viagem ou uma
-- mensagem de erro caso a viagem nao seja possivel.
simularViagem : {a : FonteEnergia} -> Veiculo a -> Double -> IO ()
simularViagem {a = Combustao} v dist =
 case v of
   (CarroC x) =>
```

```
     if autonomia v >= dist
       then print $ "O veiculo chegara ao destino com combustivel = " ++
show (x - (dist / 15.0)) ++ "L"
        else print "A distancia da viagem e maior que a autonomia do
veiculo!"

   (Onibus x) =>
     if autonomia v >= dist
       then print $ "O veiculo chegara ao destino com combustivel = " ++
show (x - (dist / 8.0)) ++ "L"
         else print "A distancia da viagem e maior que a autonomia do
veiculo!"

simularViagem {a = Eletrica} v dist =
 case v of
   (CarroE x) =>
     if autonomia v >= dist
        then print $ "O veiculo chegara ao destino com a bateria = " ++
show (x - (dist / 9.5)) ++ "%"
         else print "A distancia da viagem e maior que a autonomia do
veiculo!"
```

Alguns testes:

```
Main> autonomia x
750.0
Main> simularViagem x 500
MkIO (prim__putStr "\"O veiculo chegara ao destino com combustivel =
16.666666666666664L\"")
Main> simularViagem x 800
MkIO (prim__putStr "\"A distancia da viagem e maior que a autonomia do
veiculo!\"")
Main> autonomia y
760.0
Main> simularViagem y 456
MkIO (prim__putStr "\"O veiculo chegara ao destino com a bateria =
32.0%\"")
```

```
Main> simularViagem y 1000
MkIO (prim__putStr "\"A distancia da viagem e maior que a autonomia do
veiculo!\"")
```

Perfeito, tudo funcionando corretamente! A função está tratando cada tipo de Veiculo e verificando se o veículo possui autonomia suficiente para a viagem, em caso positivo, imprime o estado final do veículo após a simulação, caso contrário, imprime uma mensagem de erro.

Conclusão

Neste tutorial conseguimos implementar um exemplo simples de como seria uma locadora de carros com diferentes tipos de veículos e usando a abordagem de desenvolvimento guiado por tipos, conseguimos restringir os parâmetros das funções e consequentemente aumentar a segurança do sistema. Vale ressaltar que o desenvolvimento guiado por tipos também promove uma compreensão mais profunda do sistema e facilita a manutenção do código, e a linguagem Idris foi criada justamente para promover essa abordagem, oferecendo um sistema de tipos dependente e flexível.

Referências

https://idris2.readthedocs.io/en/latest/tutorial/index.html - A Crash Course in Idris 2 (Acessado em 08/05/2024)

Código usado: https://github.com/eduribeiro8/DGPT