Introducao

O paradigma funcional tem ganho notorieadade junto de grandes empresas e programadores em detrimento de outros paradigmas pois este paradigma permite que em poucas linhas de código (quando comparado com outros paradigmas) se consiga criar soluções robustas e eficientes.

Neste documento abordaremos as vantagens do paradigma funcional, como pode ser feita a sua implementação em `C++` e compararemos `C++` em estilo funcional com uma linguagem funcional, `Haskell`.

Iremos detalhar alguns aspectos de Programação Funcional tais como:

- Imutabilidade
- Lazy Evaluation
- Composicao
- ADTs

Destes, composicao é, provavelmente, o mais importante e talvez o único inerente a Programação Funcional.

A ideia central de Programação Funcional é que construindo pecas pequenas, faceis de entender e de provar como correctas, é também fácil construir um sistema, mesmo que complexo, correctamente.

De seguida, imutabilidade, a ideia em que objectos não são alterados, mas copiados, para implementar mudanças. Esta propriedade ajuda a evitar erros comuns em Programação Imperativa, causados pela partilha de memória e a não especificação da relação entre estados.

Lazy Evaluation, não sendo adoptada como estratégia de avaliação, pode ser usada como estratégia de optimização, especialmente quando combinada com imutabilidade e partilha de memória.

Finalmente, ADTs (*Algebraic Data Types*) são um forma de definir formalmente novos tipos de dados a partir de tipos já existentes. Apesar de não serem essenciais para a Programação Funcional, é desejavel criar abstrações no sistema de tipos que ajudem a descrever o problema com que nos deparamos, dando significado a valores e tentando limitar o conjunto de valores possíveis aos estritamente válidos.

Para cada um destes, mostraremos e analisaremos exemplos de como se faz em `Haskell` e como se pode fazer em `C++`. A unica excepcão será Lazy Evaluation, visto que em `Haskell` é adoptada como estrategia de avaliação e, como tal, não há necessidade de mostrar como se faz em `Haskell`.

Ao longo do documento serão usados pequenos programas ou excertos de código `Haskell` e `C++` de forma a auxiliar a análise e a comparação das linguagens.

— A introdução é só até aqui. Daqui para baixo ainda é um esboço e fará parte de um novo capitulo

Abordagem ao Paradigma Funcional em Haskell e C++

Explicação genérica sobre o paradigma funcional

Como dado a entender na introducao, a ideia fulcral do paradigma Funcional e a composicao. Enquanto que no paradigma Imperativo/Procedimental e comum um procedimento ser uma longa lista de instrucoes a executar, sendo tanto o "input" como "output" implicitos. Com isto quer-se dizer que os procedimentos podem ter acesso a estado global e/ou partilhado entre varios procedimentos. Esta partilha nao esta especificada de forma nenhuma, e o programador que tem o trabalho de mentalmente verificar que o seu uso esta dentro do expectavel.

O paradigma Funcional evita este problema parcial ou completamente, ao desencorajar ou impedir esta practica e ao mesmo tempo encorajar e facilitar "boa practicas".

Um exemplo extremo e pouco realista seria:

```
void accoes (void)
{
    accao1();
    accao2();
    accao3();
}
```

Deste pequeno excerto, podemos concluir que:

1. Como nenhum dos procedimentos `accao1`, `accao2` ou `accao3` recebe argumentos, e o resultado nao e utilizado, entao estes procedimentos nao fazem nada de util, e portanto, `accoes` tambem nao faz nada de util;
2. Ou, cada um dos procedimentos faz algo de util, mas para tal acede e altera alguma estrutura de dados partilhada; esta relacao input-output nao e explicita.

Em contraste, numa linguagem funcional escreveriamos (em notacao `Haskell`) `accoes = accao3 . accao2 . accao1` para representar a mesma sequencia de accoes, mas sem partilha de memoria nem estruturas de dados a serem mutadas: cada uma das accoes e uma funcao que devolve uma estrutura de dados, dada outra estrutura de dados.

Este problema de alteracao implicita de estado agrava-se ainda mais num contexto concorrente, num modelo "tradicional", com threads e partilha de memoria.

Haskell como linguagem funcionalmente pura

`Haskell` adopta o paradigma Funcionalmente Puro, o que quer dizer que um programa e uma funcao no sentido matematico, ou seja, dado o mesmo input, e

sempre devolvido o mesmo output.

Para se implementar "*side-effects*", em `Haskell`, em vez de se aceder ao mundo e se alterar implicitamente, como na maioria das linguagens, e recebido como um argumento, e as mudancas sao feitas sobre esse argumento.

Para dar melhor a entender, vejamos um exemplo: `puts`. O seu prototipo em `C` e `int puts (const char *s)`. A string parametro `s` vai ser impressa no `stdout`, mas nada no tipo da funcao nos diz que assim e.

Em `Haskell`, a funcao equivalente e `putStrLn`, com tipo `String -> IO ()`, e o "*side-effect*" de imprimir a string de input no `stdout` esta descrito no proprio tipo da funcao, com o tipo `IO ()`.

Pode-se pensar neste `IO a` como sendo `World -> (a, World)`, ou seja, dado um mundo, e devolvido o resultado da computacao, e o novo mundo.[1]

Breve descrição sobre como pensar funcionalmente em C++

`C++`, devido a sua heranca, promove um estilo fragil de programacao, e o programador que tem de se esforcar para programar num melhor. Por exemplo:

- Evitar dados mutáveis. Numa funcao que altera uma estrutura, em vez de receber a estrutura por referencia e a alterar, sera melhor receber a estrutura por valor e devolver uma nova. Por razoes de performance, tambem pode ser boa ideia passar a estrutura por referencia `const`, que incur(??) menos movimentacao(??) de memoria.
- Para um estilo de programacao mais generico, mas ao mesmo tempo mais seguro, preferir `templates` a `void *`, o que permite uma abstraccao de tipos, indo de encontro ao que acontece em `Haskell`. Vejamos o exemplo de uma função que soma dois valores passados como argumento.

  ```
  template <typename T> T add(T a, T b) {
     return a + b;
  };

  int main(){
     auto i = add(2,3);
     auto f = add(2.2,4.1);
     ...
  }
  ```

  Esta função pode ser invocada com diferentes tipos, tornando desnecessária a implementação da mesma função para tipos diferentes e ganhando de forma gratuita a inferência de tipos por parte do compilador, através da keyword *auto*.

[1] Ver *Tackling the Awkward Squad*.

- Definir o que é o resultado esperado ao invés de especificar as instruções que deverão ser aplicadas
- Recorrer ao uso de *lambdas* para criar abstrações.
- Definir o que é o resultado esperado ao invés de especificar as instruções que deverão ser aplicadas