

## Introducao

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

Neste documento será dada ênfase às vantagens do paradigma funcional e de que forma podemos aproveitar essas vantagens em C++.

Iremos estudar e analisar as características funcionais em programas escritos em C++ **templates**, através de algumas bibliotecas existentes para esse efeito e, aproveitaremos para efectuar análise comparativa de performance, sintaxe, etc, através de programas que resolvem o mesmo problema em ambas as linguagens.

O uso de C++ **templates** traz algumas vantagens à programação em estilo funcional em C++ nomeadamente a possibilidade de fazer programação genérica, isto é, criar programas polimórficos e também a possibilidade de obter computação em tempo de compilação.

Aproveitaremos também para aprofundar alguns aspectos importantes de Programação Funcional tais como:

- Imutabilidade
- Lazy Evaluation
- Composicao
- ADTs

Quando necessário e para uma melhor elucidação sobre as questões que estão a ser analisadas, serão usados pequenos excertos de código em ambas as linguagens.

## Abordagem ao Paradigma Funcional em Haskell e C++

### Explicação genérica sobre o paradigma funcional

O paradigma funcional é um estilo de programação que modela a computação como avaliação de expressões. Na programação funcional, os programas são executados através da avaliação de expressões em contraste, por exemplo, com o paradigma imperativo onde os programas são compostos por declarações/instruções que vão alterando o estado global à medida que executam. Isto significa que os procedimentos podem ter acesso a estado global e/ou partilhado entre varios procedimentos. Esta partilha não está especificada de forma nenhuma e, portanto, tem de ser o programador a cuidar e evitar que problemas aconteçam. O paradigma Funcional evita este problema parcial ou completamente, ao desencorajar ou impedir esta prática 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 não é utilizado, então estes procedimentos não fazem nada de útil e, portanto, `accoes` também não faz nada de útil;
2. Cada um dos procedimentos faz algo de útil, mas para tal acede e altera alguma estrutura de dados partilhada; esta relação input-output não é explícita.

Por outro lado, numa linguagem funcional escreveríamos (em notação `Haskell`) `accoes = accao3 . accao2 . accao1` para representar a mesma sequência de ações mas sem partilha de memória nem estruturas de dados a serem mudadas: cada uma das ações é uma função que devolve uma estrutura de dados, dada outra estrutura de dados.

Este problema de alteração implícita de estado agrava-se ainda mais num contexto concorrente com threads e partilha de memória.

---

## Haskell como linguagem funcionalmente pura

`Haskell` adopta o paradigma Funcionalmente Puro, o que quer dizer que um programa é uma função no sentido matemático, ou seja, dado o mesmo input é 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, é recebido como um argumento, e as mudanças são feitas sobre esse argumento.

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

Em `Haskell`, a função equivalente é `putStrLn`, com tipo `String -> IO ()`, e o “*side-effect*” de imprimir a string de input no `stdout` está descrito no próprio tipo da função, com o tipo `IO ()`.

Pode-se pensar neste `IO a` como sendo `World -> (a, World)`, ou seja, dado um mundo, é devolvido o resultado da computação, e o novo mundo.<sup>1</sup>

---

<sup>1</sup>Ver *Tackling the Awkward Squad*.

## Breve descrição sobre como pensar funcionalmente em C++

Devido à sua herança, C++ promove um estilo frágil de programação, devendo ser o programador a ter alguma atenção e a tomar algumas decisões quando pretende usar o paradigma funcional em C++. Por exemplo:

- Evitar dados mutáveis. Numa função que altera uma estrutura, em vez de receber a estrutura por referência e a alterar, será melhor receber a estrutura por valor e devolver uma nova. Por razões de performance, também pode ser boa ideia passar a estrutura por referência `const`, que se traduz em menos movimentação de memória.
- Para um estilo de programação mais genérico, mas ao mesmo tempo mais seguro, preferir `templates` a `void *`, o que permite uma abstracção 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*.

- Recorrer ao uso de *lambdas* para criar abstrações (desde C++11)

## Comparação e análise de programas equivalentes em Haskell e C++

Neste capítulo, faremos uma comparação mais específica sobre programas escritos em ambas as linguagens e cujo propósito é o mesmo, ou seja, podem considerar-se equivalentes. Durante a pesquisa que efectuamos, encontramos duas bibliotecas que tentam transpôr o paradigma funcional para C++, de encontro ao que estamos também a fazer. Vamos tomar como exemplo alguns programas pequenos para facilitar a comparação, usando a biblioteca `Cpp Prelude`<sup>2</sup> e terminaremos com um programa mais robusto que foi utilizado na ronda de qualificação do *Google Hash Code 2020*, do qual tínhamos a versão em Haskell e fizemos a conversão para C++ utilizando a biblioteca `Functional Plus`<sup>3</sup>.

---

<sup>2</sup>Ver *CppPrelude*.

<sup>3</sup>Ver *FunctionalPlus*.

De forma a efectuar a comparação de pequenos programas geramos um ficheiro de input com ??? de entradas que serão lidas pelo programa e depois serão aplicadas algumas funções para se poder efectuar a comparação. Note-se que deixamos de fora da análise de performance o processo de leitura do ficheiro. Focaremos a comparação na aplicação de funções específicas em **Haskell** e **C++ templates**.

A biblioteca Cpp Prelude tem os seguintes *defines* para simplificar a leitura.

```
#define Container template <typename, typename> class
#define Function typename
#define Predicate typename
#define Type typename
#define Number typename
#define Ordinal typename
```

Vejamos então a aplicação de uma função que multiplica todos os elementos de uma lista por 2. Em Haskell uma definição possível é:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (h:t) = f h : map f t
```

Já do lado de C++ temos:

```
template <Function FN, Container CN, Type A,
          Type B = typename std::result_of<FN(A)>::type,
          typename AllocA = std::allocator<A>,
          typename AllocB = std::allocator<B>>
auto map(const FN& f, const CN<A, AllocA>& c) -> CN<B, AllocB> {
    auto res = CN<B, AllocB>{};
    res.reserve(c.size());
    std::transform(std::begin(c), std::end(c), std::back_inserter(res), f);
    return res;
}
```

Existem alguns aspectos notórios apenas pela análise visual e sintática. A sintaxe de Haskell é bastante mais simples, tornando o código mais conciso e com menos “floriado”.

Ao contrário do Haskell, em C++ é necessário criar um *container* (res) onde serão guardados os resultados depois de aplicar a função. Em Haskell isto é feito de forma escondida ao programador. No entanto, o programador pode tirar proveito da inferência de tipos do compilador de C++ e relaxar um pouco nos tipos utilizando *auto*.

Uma outra diferença é também na definição da função passada ao *map*. Em Haskell fazendo

```
map (*2) lista
```

obtemos imediatamente o resultado. Por outro lado, em C++ é necessário definir explicitamente, podendo ser feito da seguinte forma:

```
auto f = [] (int x) {return x * 2;};
```

e agora sim, poderá ser invocado *map* com a função *f*:

```
auto r = map(lista,f);
```

Relativamente à eficiência, executando ambos os programas com a mesma lista de input, Haskell sai claramente em vantagem. Em Haskell obtivemos 0.002 milissegundos e em C++ 48 milissegundos.