Programação Funcional em C++

Com o aumento em complexidade de problemas a resolver, e importante manter codigo facil de ler e escrever. O paradigma de programação Funcional e conhecido por ser conciso, normalmente a custa de conceitos dificeis de compreender, muitas vezes matematicos. Esta complexidade aumenta vertiginosamente quando os problemas requerem um programa multi-threaded.

Neste trabalho pretendemos abordar C++ do ponto de vista funcional.

Intro

- Importancia/vantagens/pontos positivos de Programação Funcional
- Como programar funcionalmente em C++?

TODO: De que topicos vamos falar

Lazy Evaluation

TODO: Overview

Ranges V3

TODO: Overview

Imutabilidade

TODO: Overview

ADTs & Pattern Matching

TODO: Overview

Concorrencia

TODO: Overview

Os topicos a fundo

Lazy Evaluation ou avaliação preguiçosa

A avaliação preguiçosa é uma técnica de programação que atrasa a execução de um código, até que o seu valor seja necessário e evita chamadas repetidas. Como normalmente não partilhamos o resultado das nossas operações entre threads/funcões, estamos a sobrecarregar o sistema ao executar as mesmas operações varias vezes, resultando numa perda de performance, no pior dos casos tendo um tempo exponencial.

Como por exemplo, no case de um algoritmo que recorra ao calculo do produto entre 2 matrizes A e B com alguma ou muita frequência, a avaliação preguiçosa

propõe calcular uma única vez o produto das matrizes e reutilizar o resultado sempre que o produto seja utilizado. Deste modo evita-se o custo computacional associado a repetição da mesma operação, o que contribui para o aumento da performance.

No entanto, pode dar-se o caso do resultado não ser necessário, o que consome desnecessariamente os recursos computacionais. Assim sendo o cálculo é atrasado até que seja objectivamente necessário, para tal, criamos uma função para calcular o produto entre duas matrizes:

```
auto P = [A, B] {
    return A * B;
};
```

No contexto de programação paralela, onde o resultado de operações pode ser reutilizado por múltiplas threads, pretende-se partilhar esse valor para que outros fios de execução tenham acesso a esse resultado, evitando cálculos desnecessários.

Laziness in C++

Em C++ temos de ser nós a programar dessa forma garantindo uma avaliação preguiçosa. Para isso criamos um **template** com uma função que é a operação de se pretende executar, uma flag que indica se a operação ja foi calculada e por fim o resultado da função. O processo associado a utilização deste template denomina-se por *memorização*, visto que os resultados são memorizados.

- Função.
- Uma flag que indica se já calculamos o resultado da função.
- O resultado calculado.

```
template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable bool m_cache_initialized;
    mutable decltype(m_computation()) m_cache;
    mutable std::mutex m_cache_mutex;

public:
    ...
};
```

É necessária a utilização de um mutex para evitar a execução paralela da função. As variáveis são declaradas como *mutable* e não *conts* pois eventualmente vamos ter de alterar esse valores. Como a dedução automática para tipos de tamplates só é suportada a partir do C++17, temos de ter uma abordagem diferente. Na maioria das vezes, não podemos especificar explicitamente o tipo da função pois pode ser definindo por "lambdas", e não é posível determinar o seu tipo. É

necessário criar uma função auxiliar para que a dedução dos argumentos da função em tamplates seja automática, assim podemos usar o modelo principal com compiladores que não suportam C++17.

```
template <typename F>
class lazy_val {
private:
    ...

public:
    lazy_val(F computation)
        : m_computation(computation)
        , m_cache_initialized(false)
{
}
};

template <typename F>
inline lazy_val<F> make_lazy_val(F&& computation)
{
    return lazy_val<F>(std::forward<F>(computation));
}
```

O construtor precisa de armazenar a função e definir como *false* a flag que indica se já calculamos o resultado.

NOTA Como não sabemos o tipo de retorno da função, temos de garantir que o membro `m_cache` tenha um construtor por defeito.

Para finalizar falta criar a função responsável pelo calculo e/ou retorno do valor da operação. A representação em C++ traduz-se na aplicação do mutex no inicio da função, para evitar acesso simultâneo a cache, seguido da verificação da condição da inicialização e da execução da operação caso a cache não tenha sido inicializada. A função retorna o valor calculado.

```
template <typename F>
class lazy_val {
private:
    ...

public:
    ...

    operator const decltype(m_computation())& () const
    {
        std::unique_lock<std::mutex>
            lock{m_cache_mutex};
```

```
        if (!m_cache_initialized) {
            m_cache = m_computation();
            m_cache_initialized = true;
        }

        return m_cache;
    }
};
```

{}

Como se pode observar este algoritmo não é óptimo, pois apesar de o mutex ser necessário apenas na primeira execução este é utilizado em todas as execuções.

Uma solução alternativa ao mutex é a utilização de um operador de `casting` que permite limitar a execução de uma porção do código uma única vez por instancia. A função correspondente a operação casting na biblioteca padrão é `std :: call_once`:

```
template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable decltype(m_computation ()) m_cache;
    \textbf{mutable std::once_flag m_value_flag;}

public:
    ...
    operator const decltype(m_computation())& () const
    {
        std::call_once(m_value_flag, [this] {
            m_cache = m_computation();
        });
    return m_cache;
    }
};
```

Assim substituímos o mutex pela função `std::call_once` que recebe como argumentos a flag e a operação que deve executar uma única vez. A flag será actualizada automaticamente pela operação de casting.

Laziness as an optimization technique

$\xrightarrow[\text{world}]{\text{hello}}$

...

Ranges V3

An important part of Funtional Programming is the composition of functions. By writing small and generic functions, and reutilising them with composition, it's possible to quickly write complete programs with fewer bugs than if everything was written from scratch. In Function Programming Languages composition is used everywhere. In a language like C++, composition everywhere is not very convenient, mainly because of syntax and the semantics of passing vriables by value or reference. There's a place, however, where composition doesn't have to be pointwise: working with collections. When there's a set of operations to do on a collection, be it the whole collections or a subset of it, expressing these operations as some sort of pipeline is very intuitive, human-readable, and cheap in the number of characters typed. This idea is not new, and has been adopted even by large mainstream languages like Java (in the form of Streams). In Functional Programming Languages this concept is usually implemented in the form of lists in lazy-by-default languages, like Haskell, or lazy-lists or streams in eager-by-default languages, like Scheme.

There are many operations over collections that can be mapped to a pipeline, and many of these operations are very common. Coding them each time as a loop is tedious and most likely harder to read than simply using the abstractions. Some of these common operations include summing, multiplying, filtering, mapping, and the swiss-army knife, with which many of the other operations can be coded, the fold (also commonly known as reduce, but with slightly different semantics).

The C++ STL already has some of these operations implemented. For the most common and simplest of use cases they could be enough. Certainly beats writing a for-loop by hand. These could, however, be improved, and the Ranges library does so in two very important aspects: usage and performance.

As an example, given a vector (or a list in Haskell), filtering the elements given some predicate, applying a mapping function to each of them, and then multiplying the results could be done like so in Haskell:

```
product . map mapper . filter pred $ xs
```

A handwritten for-loop in C++ could look something like this:

```
for (auto x : xs) {
    if (pred(x)) {
        ret *= mapper(x);
    }
}
```

But one doesn't have to handwrite for-loops most of the time! It would be nice if it was possible to do the following:

```
xs | filter(pred) | transform(mapper) | product
```

And with the Ranges library it is! Learning about its inner workings a bit could

help understand why it's performant in comparison to the counterparts in the STL.

In the STL, the functions have as parameters two iterators, the beginning and end of the input collection, an iterator to the beginning of the output collection, and an inserter, i.e., how to insert elements into the output collection. Right away it's easy to see some points for improvement:

- Why do we have to pass the beginning and end iterators to the input collection? (I know why, but it doesn't have to be that way) It's possible to pass iterators of different collections by mistake. It's tedious to pass two arguments instead of one.
- Passing an iterator and inserter to the output collection is also tedious work; but worse, it means a new collection is always created as a result!

Ranges improves both of these aspects simply by abstracting collections and the result of operations as "ranges", which could be very simply be thought of as a pair iterators: the beginning and end.

Now it's possible to give an operation the collection itself, which will be converted into a range automatically, or the result of an operation. This last bit is very important, because it means it's possible to compose operations pointfree.

Usability is explaing. Now for performance. The pair of iterators mentioned above in a range structure are just pointers. For operations that don't mutate the original collection, there's no need to copy data around. For example, to implement `filter`, all is needed is to implement the `++` ("next") operator of the beginning iterator as searching for the next element in the input range (note *range*) that passes the predicate. If there's no element passing the predicate, the end of the resulting range has been reached.

Where change is needed there are two options: copying the input range, or mutating the original collection in-place.

All this happens *"on-demand"* (i.e., lazily), like in Haskell. If the result is not collected, nothing is done.

Imutabilidade vs mutabilidade

Uma das características mais importantes do paradigma funcional, nomeadamente na linguagem Haskell (existem linguagens funcionais impuras) é a noção de imutabilidade das expressões. Isto faz com que, por exemplo, não seja possível alterar o valor de variáveis já existentes mas sim, criar novas variáveis com os novos valores.

Analisando o excerto de código a seguir podemos verificar que quando *action* é invocado, o valor mostrado é o primeiro valor de *a=123* pois é o valor avaliado para *a* até então.

```
> a = 123
```

```
> action = print a
> a = 456
> action
123
```

A linguagem *c++* tenta também lidar com esta noção de imutabilidade. A noção de funções puras é dada pela avaliação de *referential transparency*. Uma função é referencialmente transparente se o programa não se comportar de maneira diferente ao substituír a expressão inteira apenas pelo seu valor de retorno, por exemplo:

```
int g = 0;

int ref_trans(int x) {
    return x + 1;
} // referencialmente transparente

int not_ref_trans(int x) {
    g++;
    return x + g;
} // não referencialmente transparente(cada vez que a função é invocada, tem um valor de
```

Se uma expressão é referencialmente transparente, não tem efeitos colaterais observáveis e, portanto, todas as funções usadas nessa expressão são consideradas puras. A ideia de imutabilidade é particularmente útil em ambientes em que se gera concorrência, pois, existem variáveis partilhadas que podem gerar comportamentos inesperados nos programas se não for devidamente protegida a sua alteração. Em *c++* está disponível a keyword *const* que permite controlar a imutabilidade de uma variável. Ao declarar uma variável *const x* estamos a dizer ao compilador que esta variável é imutável e, qualquer tentativa de alteração à variável irá originar um erro. De seguida analisamos a declaração de uma variável *const* e os possíveis erros que podem ser cometidos ao tentar manipular essa variável.

```
const std::string name{"John Smith"};

1 - std::string name_copy = name;
2 - std::string& name_ref = name; // erro
3 - const std::string& name_constref = name;
4 - std::string* name_ptr = &name; // erro
5 - const std::string* name_constptr = &name;
```

Em 1 não há ocorrências de erros pois apenas se está a associar o valor de name a uma nova variável. Em 2 teremos um erro de compilação pois estamos a passar *name* por referência a uma variável não *const*, situação que é resolvida em 3. Em 4 voltamos a ter um erro de compilação pois estamos a criar um pointer não *const* para name. Este erro é resolvido em 5. O facto de em 2 e 5 ocorrer um erro de compilação deve-se ao facto de \textit{name_ref} e \textit{name_ptr}

não estarem qualificados com *const* e poderem ser alterados. No entanto, como apontam para uma variável *const*, gera-se uma contradição.

No entanto, por vezes existe necessidade de ter variáveis mutáveis mas que, em determinados casos, a sua alteração esteja protegida. Esse controlo pode ser feito recorrendo a mutexes que permitem o controlo de concorrência em determinadas zonas crítica de código. Esta abordagem será aprofundada mais à frente na secção de concorrência.

ADTs & Pattern Matching

TODO: Aprofundar

Concorrencia

TODO: Aprofundar

Programa Exemplo

Conclusoes