Programação Funcional em C++

Com o aumento da complexidade dos problemas a resolver, é importante manter codigo simples e cuja interpretação seja relativamente simples de conseguir. O paradigma de programação Funcional e conhecido por ser conciso, normalmente a custa de conceitos matematicos dificeis de compreender. A complexidade dos programas 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 e do Haskell

O paradigma funcional é um paradigma que descreve uma computação como uma expressão a ser avaliada. Os tipos de dados estáticos pelo que, podemos despistar muitos erros em tempo de compilação e assim reduzir os possíveis problemas que poderão aparecer durante a execução. O sistema de tipos é muito poderoso e os tipos podem ser facilmente inferidos, permitindo desta forma criar programas genéricos que poderão executar com diferentes tipos de dados. O paradigma funcional é também conhecido pelo facto de pouca escrita de código permitir criar operações muito poderosas. Isto permite que a leitura e interpretação de código seja facilitada, bem como o debug e correcção de erros. Haskell não executa funções nem calcula dados até que sejam necessários(lazy) Uma função nao tem efeitos colaterais Possibilidade de definir funções dentro de funções Actualmente, com processadores multi-core e multi-threaded, exige-se uma grande escabilidade dos programas. É sobretudo neste ponto que as linguagens funcionais têm ganho uma grande ênfase pois uma das suas caracteristicas principais é o facto de o paralelismo e threading serem bastante simples de implementar, muito à custa da imutabilidade dos dados, o que faz com que não sejam necessárias grandes preocupações ao nível do controlo de concorrência.

- Como programar funcionalmente em C++?

TODO: De que topicos vamos falar

Lazy Evaluation

TODO: Overview

Ranges V3

TODO: Overview

Imutabilidade

TODO: Overview

Recursividade

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.

NOTE: isto era uma `section` dantes

Laziness as an optimization technique

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

...

Ranges V3

Uma parte importante de Programacao Funcional e a composicao de funcoes. Ao escrever funcoes pequenas e genericas, e ao as reutilizar com composicao, e possivel escrever programas completos rapidamente, com menos bugs do que se escrevessemos o mesmo programa todo do zero. Em linguagens funcionais composicao e usada em todo o lado. Numa linguagem como `C++` nao e muito conveniente usar composicao em todo o lado, principalmente por causa de sintaxe e da semantica de passar variaveis por valor ou referencia. Ha um sitio, no entanto, onde composicao nao tem de ser pointwise: trabalhar com coleccoes. Quando ha um conjunto de operacoes a fazer numa coleccao, seja na coleccao completa ou parte dela, expressar estas operacoes coo algum tipo de pipeline e bastante intuitivo, legivel, e barato em numero de caracteres escritos. Esta

ideia nao e de agora. Em linguagens funcionais este conceito e normalmente implementado como listas em linguagens lazy-by-default, como Haskell, ou lazy-lists/streams em linguagens eager-by-default, como Scheme.

Existem muitas operacoes sobre coleccoes que podem ser mapeadas numa pipeline, e muitas destas operacoes sao muito comuns. Programa-las de cada vez a mao como um loop e tedioso, e muito provavelmente menos legivel do que simplesmente usar as abstraccoes. Algumas destas operacoes comuns incluem somar, multiplicar, filtrar, mapear, e o canivete suico, com o qual muitas das outras operacoes sao implementadas, o fold (tambem comummente conhecido como reduce, mas com semantica ligeiramente diferente).

A STL de `C++` ja tem algumas destas operacoes. Para os casos mais simples e comuns estas podem ser suficientes. Definitivamente e melhor do que escrever um loop for a mao. Estas podem, no entanto, ser melhoradas, e a biblioteca Ranges faz isso mesmo em dois aspectos muito importantes: usabilidade e performance.

Como exemplo, dado um vector (ou uma lista em Haskell), filtrar os elementos dado um predicado, aplicar uma funcao a cada um deles, e depois multiplicar os resultados pode ser feito assim em Haskell:

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

Um loop for em `C++` escrito a mao podia ser escrito como se segue:

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

Mas nao e preciso escrever loops for a mao grande parte das vezes! Era bom se fosse possivel escrever o seguinte:

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

E com a biblioteca Ranges e! Perceber como funciona internamente pode dar alguma luz sobre o porque de ter melhor performance em comparacao com a STL.

Na STL, as funcoes tem como parametros dois iterators, – o inicio e fim da coleccao de input, ou de parte dela – um iterator para o inicio da coleccao de output, e um inserter, que define como os elementos serao inseridos na coleccao de output. De imediato, alguns pontos a melhorar saltam a vista:

- Porque e que e preciso passar os iteradores de inicio e fim da coleccao de input? Na verdade sabemos para que serve, mas nao precisa de ser assim. Da mais trabalho passar dois argumentos em vez de um, mas mais importante, e possivel passar iteradores para o inicio e fim de duas coleccoes diferentes por engano.

- Passar iterador e inserter da coleccao de output tambem e tedioso; mas pior, significa que e sempre criada uma coleccao de output.

A Ranges melhora estes does aspectos ao simplesmente abstrair coleccoes como *ranges*, e devolver *ranges* como resultado das operacoes. Pode-se pensar nesta abstraccao de *range* como um par de iteradores: o inicio e fim.

Agora e possivel passar a uma operacao uma coleccao, que e automaticamente transformada num range, ou o resultado de uma outra operacao. Esta ultima parte e crucial, porque significa que podemos compor operacoes pointfree.

Usabilidade esta explicada. Vamos agora a porformance. O par de iteradores que forma um range e so um par de apontadores. Para operacoes que nao alteram a coleccao original nao e preciso copiar nada. Para implesmentar, por exemplo, o `filter`, basta implementar o operador `++` (*next*) para o range de output, sobre o iterador de inicio, procurando pelo elemento seguinte no range de input que satisfaz o predicado. Se nenhum elemento satisfaz o predicado chegamos eventualmente ao fim da coleccao, ou seja, temos um range vazio.

Quando ha a necessidade de alterar o range de input temos duas opcoes: copiar o range de input, ou mutar a coleccao original *in-place*.

Todas as operacoes sao lazy, ou acontecem *on-demand*, como em Haskell – se o resultado nao for usado, nao se faz nada.

Imutabilidade

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 para o mesmo input, a função devolve sempre o mesmo 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 retorno diferente)
```

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.

Recursividade

Uma vez que as linguagens funcionais puras preservam a caracteristica de imutabilidade dos dados, é usada recursividade em vez de loops. Analisemos semelhanças e diferenças sobre este recurso em Haskell e C++.

Tomemos como exemplo o cálculo do factorial de um número arbitrário 'n'. Em Haskell esta função torna-se bastante simples de implementar.

```haskell
fac:: Int -> Int
fac 0 = 1
fac n = n * fact (n-1)
```

No entanto, em C++, não é possível escrever um código tão conciso quanto em Haskell.

```
template <int N>
struct Fac{
  static int const value= N * Fac<N-1>::value;
};

template <>
struct Fac<0>{
  static int const value = 1;
};
```

A declaração do template para Fac<0> deve-se ao facto de pretender-mos que uma função se comporte de determinada forma para um certo argumento. Esta técnica tem o nome de *specialization* e podemos compará-la com o caso de paragem em Haskell.

Existe, no entanto, uma "ligeira" diferença entre a função factorial recursiva em Haskell e C++. Na verdade, a versão em C++ não é recursiva. Cada invocação do template com argumento N instancia uma novo template com argumento (N-1). Vejamos com detalhe o desenvolvimento da função para N = 5.

```
Fac<5>::value   = 5 * Fac<4>::value
                = 5 * 4 * Fac<3>::value
                = 5 * 4 * 3 * Fac<2>::value
                = 5 * 4 * 3 * 2 * Fac<1>::value
                = 5 * 4 * 3 * 2 * 1 * Fac<0>::value
                = 5 * 4 * 3 * 2 * 1 * 1
                = 120
```

ADTs & Pattern Matching

ADTs

ADTs (*Algebraic Data-Types*), ou Tipos de Dados Algebricos, em Portugues, sao tipos de dados criados a partir de tipos ja existentes, de duas maneiras diferentes. Vamos dar uma breve descricao, para completude, simplesmente porque nem todas as linguagens funcionais tem ADTs nativamente, tao omnipresentes, ou com diferentes nomes.

A primeira, e mais comum, e o producto. Dados dois tipos A e B, o produto deles, $A \times B$, e simplesmente o produto cartesiano entre A e B.

A segunda, presente em grande parte das linguagens, mesmo que indirectamente, e a soma. Dados dois tipos A e B, a soma deles, $A + B$, e o conjunto cujos elementos ou sao do tipo A ou do tipo B, mas e possivel distingui-los. Este conjunto e isomorfo a $Bool \times (A \cup B)$ (**TODO:** e mesmo? confirmar isto), ou seja, pode ser representado indirectamente como $Bool \times (A \cup B)$: um elemento de A ou B, e uma flag a indicar se e de A ou de B. Note-se que esta flag indica na verdade se o elemento e da esquerda ou direita; $A + A$ e um tipo valido.

Com estas duas tecnicas de composicao e possivel representar qualquer estrutura de dados. Sera entao util saber como usar estas duas tecnicas numa linguagem de programacao. Em Haskell, com o seu sistema de tipos avancado, ambas estao disponiveis nativamente. Em `C++`, tal como em C, so o produto esta disponivel, sobre a forma de structs. Na STL tambem ha `std::pair` e `std::tuple`, que vamos comparar a seguir. **vale a pena falar disto?**

De seguida vamos apresentar as tres formas de compor tipos em `C++`, com as *keywords* `struct`, `enum`, e `union`, qual o equivalente em `Haskell`, e como cada uma se relaciona com ADTs.

`struct`

Por exemplo, para representar um filme, com o seu titulo (`String`), ano de lancamento (`Int`), e uma pontuacao (`Float`), podemos definir o tipo `Filme` como o produto dos seus tres atributos, ou seja $Filme \cong String \times Int \times Float$.

Em `Haskell` existem varias maneiras de definir o tipo `Filme`.

```haskell
type Filme = (String, Int, Float)

data Filme = Filme String Int Float

data Filme = Filme {
    titulo :: String,
    ano :: Int,
    pontuacao :: Float
}
```

A primeira reflecte mais directamente o tipo teorico; a segunda e uma definicao mais comum; a terceira, com *records*, da "nomes" aos varios campos (isto nao e verdade, mas para ja, serve), e e mais parecida com uma definicao em `C++`.

Em C++, existem duas alternativas:

```
struct Filme {
    std::string titulo;
    unsigned ano;
    float pontuacao;
};

typedef std::tuple<std::string, unsigned, float> Filme;
```

A primeira, que e mais idiomatica, e a segunda, que e mais parecida com o tipo teorico, como a primeira definicao em Haskell.

enum

Um exemplo simples e conhecido a todos do uso de *enums* e na definicao do tipo dos booleanos: `enum bool { false, true };` em C++, e `data Bool = False | True` em Haskell.

Se pensarmos nos valores de falso e verdadeiro como pertencentes a um conjunto singular, e denotarmos esse conjunto por `false` e `true` respectivamente, podemos pensar no tipo booleano como a soma de `false` e `true`, i.e., $Bool \cong False + True$.

Poderiamos assim achar que `enum` em C++ serve para representar tipos de soma, mas estariamos errados. `enum` serve apenas para representar a soma de varios conjuntos conjuntos singulares, ou um unico conjunto enumeravel de valores nao inteiros e sem ordem. Veremos mais a frente como representar tipos de soma.

union

Esta e a menos comum das tres *keywords*, por ser de uso muito limitado, e nao existe equivalente em Haskell. Esta "falha" do lado de Haskell na verdade nao e grave – possivelmente nem sequer uma falha. Ao contrario do que o nome sugere, `union` nao serve para representar a uniao de tipos, e nao vamos aqui listar os seus usos alem do necessario para este texto.

`union` pode ser usada quando se pretende guardar qualquer um de varios valores, mas nao varios em simultaneo. Por exemplo se se pretender um tipo para guardar ou inteiros ou *floats*, pode-se usar a seguinte `union`:

```
union {
    int i;
    float f;
}
```

ADTs em C++

Vamos agora, finalmente, descrever como implementar ADTs em `C++`. A maneira mais idiomatica, possivel tambem em `C`, e usar uma *tagged union*.

Como exemplo, vamos definir um tipo de arvores binarias de nos, com valores nos nos e nas folhas: $BTree\ A \cong A + (A \times BTree\ A \times BTree\ A)$.

```
data BTree a = Leaf a
             | Node a (BTree a) (Btree a)
```

```
template <typename A>
struct BTree {
    enum {
        BTree_Leaf,
        BTree_Node,
    } variant;

    union {
        A leaf;
        struct {
            A x;
            BTree<A> left;
            BTree<A> right;
        } node;
    } tree;
};
```

Esta definicao em `C++` e muito maior que a definicao em `Haskell`, nao so devido a verbosidade (???) de `C++`, como a necessidade de usar o truque mencionado acima de transformar uma soma num produto, ou seja,

$$BTree\ A \cong A + (A \times BTree\ A \times BTree\ A) \cong Bool \times (A \cup (A \times BTree\ A \times BTree\ A))$$

Ou, para aproximar melhor a implementacao,

$$BTree\ A \cong \{\ Leaf,\ Node\ \} \times (A\ \cup\ (A \times BTree\ A \times BTree\ A))$$

Neste caso, a `union` esta realmente a simular a uniao de conjuntos.

Uma alternativa a tagged union, e usar `std::variant`, como a que se segue:

```
template <typename A>
struct Node {
    A x;
    BTree<A> left;
    BTree<A> right;
```

```
};

template <typename A>
using BTree = std::variant<A, struct Node>;
```

Pattern Matching

Em C++ nao vale a pena...

Concorrencia

TODO: Aprofundar

Programa Exemplo

Conclusoes