Titulo

Autor1 Autor2 Autor3

Data

Introducao

O paradigma funcional tem ganho notorieadade 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++`, através de algumas bibliotecas existentes para esse efeito e, aproveitaremos para efectuar uma análise comparativa de performance, sintaxe, etc, através de programas que resolvem o mesmo problema em `C++` e `Haskell`.

O uso de `template`s em `C++` traz algumas vantagens à programação em estilo funcional, nomeadamente a possibilidade de fazer programação genérica, isto é, criar programas polimórficos. Também é possível obter computação em tempo de compilação com `template`s, mas esta não é essencial a Programação Funcional, e portanto não vamos desenvolver sobre este assunto.[1]

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

- Imutabilidade
- Lazy Evaluation
- Composição
- 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.

[1] Para mais informação sobre este assunto, ler *Let Over Lambda*.

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

Explicação genérica sobre o paradigma funcional

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 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 pratica".

Um exemplo extremo e pouco realista seria:

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

Deste pequeno excerto, podemos concluir uma de duas hipoteses:

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 util, mas para tal acede e altera alguma estrutura de dados partilhada; esta relacao *input-output* não é explicita.

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

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

`Haskell` como linguagem funcionalmente pura

`Haskell` adopta o paradigma Funcionalmente Puro, o que quer dizer que um programa é uma funcao no sentido matematico, ou seja, dado o mesmo *input* é sempre devolvido o mesmo *output*.

Para se implementar efeitos secundários, em `Haskell`, em vez de se aceder ao mundo e (zaz) se alterar implicitamente, como na maioria das linguagens, este é recebido como um argumento, e as mudanças são feitas sobre esse argumento.

Para dar melhor a entender, vejamos um exemplo: `puts`. O seu protótipo em `C` é `int puts (const char *s)`. A string parâmetro `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 efeito secundário de imprimir a string de *input* no `stdout` está descrito no próprio tipo da função: `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.[2]

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, tambem pode ser boa ideia passar a estrutura por referencia `const`, que se traduz em menos movimentação de memoria.
- Para um estilo de programacao mais genérico, 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);
     return 0;
  }
  ```

  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`)

[2] Ver *Tackling the Awkward Squad.*

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

Neste capítulo, faremos uma comparação mais específica sobre programas escritos em âmbas 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++`, que vão de encontro aos objectivos do nosso projeto. Vamos começar por algumas funções sobre listas do *prelude* do `Haskell`, usando a biblioteca *"CPP Prelude"*[3], para uma comparação mais directa, e terminaremos com um programa mais robusto que foi utilizado na ronda de qualificação do *Google Hash Code 2020*, do qual tinhamos a versão em `Haskell` e fizemos a conversão para `C++` utilizando a biblioteca *"Functional Plus"*[4], para uma comparação mais realista.

Prelude

(**TODO**: que tipo de comparacao?) De forma a efectuar a comparação de pequenos programas geramos um ficheiro de *input* com uma lista de 10000000 de inteiros. Note-se que deixamos de fora da análise o processo de leitura do ficheiro. Focaremos a comparação na aplicação de funções específicas em `Haskell` e `C++`. Vamos apresentar primeiramente uma definição possível para cada função nas duas linguagens, e no fim apresentamos os tempos de execução.

`map`

Comecemos pelo `map`. Esta função *mapeia* todos os elementos de uma dada lista com uma dada função. Por exemplo, em `Haskell`, se tivermos uma lista de inteiros `l :: [Int]` e quisermos duplicar todos os elements da lista, basta chamar `map (*2) l`.

TODO: Uma definição possível do `map` em `Haskell` é:

```haskell
map :: (a -> b) -> [a] -> [b]

-- Recursividade explícita
map f [] = []
map f (h:t) = f h : map f t

-- Funções de ordem superior
map f = foldr (\a b -> f a : b) []
```

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>>
```

[3] Ver *CPP Prelude*.
[4] Ver *Functional Plus*.

```
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;
}
```

TODO: ~~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 "floreado".~~

Em `C++` já existe uma função parecida: `std::transform`. Esta função recebe os iteradores de início e fim da colecção de *input*, a forma como se deve inserir na colecção de resultado, e como transformar cada elemento da colecção de *input*; e devolve o iterador para o início da colecção de resultado.

Como tal, podemos aproveitar o `std::transform` para definir o `map` em `C++`. Como o `map` devolve uma colecção, temos de criar uma a colecção de resultado (`res`) – em `Haskell` isto é feito de forma automática.

`filter`

A segunda função que comparamos foi o `filter`, que recebe uma lista e um predicado, e calcula a lista que tem todos os elementos que satisfazem esse predicado. Por exemplo, se tivermos uma lista de inteiros `l :: [Int]`, e quisermos obter a lista dos inteiros pares, podemos usar o `filter` com o predicado `even`: `filter even l`.

TODO: Em `Haskell` uma definição possível do `filter` é:

```
filter :: (a -> Bool) -> [a] -> [a]

-- Recursividade explícita
filter p [] = []
filter p (h:t)
    | p h       = h : filter p t
    | otherwise =     filter p t

-- Funções de ordem superior
filter p = foldr (\a b -> if p a then a:b else b) []
```

Já do lado de C++ temos:

```
template <Predicate PR, Container CN, Type A,
          typename AllocA = std::allocator<A>>
auto filter(const PR& p, const CN<A, AllocA>& c) -> CN<A, AllocA> {
  auto res = CN<A, AllocA>{};
  res.reserve(c.size());
  std::copy_if(std::begin(c), std::end(c), std::back_inserter(res), p);
```

```
  res.shrink_to_fit();
  return res;
}
```

Tal como no caso do `map`, já existe uma função parecida: `std::copy_if`. Apesar de não sabermos à partida quantos elementos terá a colecção de resultado, por razões de performance, podemos na mesma reservar espaço. No fim, a colecção pode conter menos elementos que os reservados, e para remover a memória inutilizada, usa-se `shrink_to_fit`.

reverse

A nossa terceira função escolhida foi o `reverse` que, dada uma lista, inverte a ordem dos seus elementos. Por exemplo, se tivermos a lista `l = [1, 2, 3, 4, 5]`, e lhe aplicarmos o `reverse` obtemos `[5, 4, 3, 2, 1]`.

TODO: Em `Haskell` uma definição possível é:

```
reverse :: [a] -> [a]

-- Recursividade explícita
reverse = reverse' []
    where
        reverse' ret []    = ret
        reverse' ret (h:t) = reverse' (h:ret) t

-- Funções de ordem superior
reverse = foldl (flip (:)) []
```

Já do lado de C++ temos:

```
template <Container CN, Type A, typename AllocA = std::allocator<A>>
auto reverse(const CN<A, AllocA>& c) -> CN<A, AllocA> {
  auto res = CN<A, AllocA>{c};
  std::reverse(std::begin(res), std::end(res));
  return res;
}
```

Mais uma vez, já existe uma função parecida: `std::reverse`. No entanto, o `std::reverse` altera a colecção, em vez de devolver uma nova.

zip

Para concluir o primeiro conjunto de funções escolhemos a função `zip`. Esta recebe duas listas, e emparelha os seus elementos – o primeiro com o primeiro, o segundo com o segundo, etc. Caso as listas tenham tamanhos diferentes a menor lista dita o tamanho final.

TODO: Em `Haskell` uma definição possível é:

```haskell
zip :: [a] -> [b] -> [(a,b)]

-- Recursividade explícita
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys

-- Funções de ordem superior
zip = zipWith (,)
```

Já do lado de C++ temos:

```cpp
template <Container CA, Type A, typename AllocA = std::allocator<A>,
          Container CB, Type B, typename AllocB = std::allocator<B>,
          Container CRES = CA, typename RES = std::tuple<A, B>,
          typename AllocRES = std::allocator<RES>>
auto zip(const CA<A, AllocA>& left, const CB<B, AllocB>& right)
    -> CRES<RES, AllocRES> {
  auto res = CRES<RES, AllocRES>{};
  res.reserve((left.size() < right.size()) ? left.size() : right.size());
  auto l = std::begin(left);
  auto r = std::begin(right);
  while (l != std::end(left) && r != std::end(right)) {
    res.emplace_back(*l, *r);
    ++l;
    ++r;
  }
  return res;
}
```

Neste caso, não existe nenhuma função parecida na STL, e portanto, é definida à mão como um ciclo.

Resultados

Para comparar performance entre as duas linguagens, medimos o tempo de CPU de cada função, com os meios disponíveis em cada uma. Simultaneamente medimos o tempo de execução real do processo com o programa `/usr/bin/time`.

Em C++ usamos `std::clock()` do *header* `<ctime>`, com o seguite macro:

```cpp
#define benchmark(str, func) \
  do { \
    auto start = std::clock();                \
    (void) func;                              \
    auto stop = std::clock();                 \
    auto duration = 1000000000                \
                  * (stop - start)            \
```

```
                / CLOCKS_PER_SEC;            \
    std::cout << str << ": " << duration       \
              << " nanoseconds" <<  std::endl; \
  } while (0)
```

Em `Haskell` usamos `getCPUTime` de `System.CPUTime`, com a seguinte função:

```
timeSomething :: NFData a => String -> a -> IO ()
timeSomething str something = do
  start <- liftIO getCPUTime
  let !result = deepforce $! something
  end <- liftIO getCPUTime
  let diff = round . (/1000) . fromIntegral $ end - start
  putStrLn $ str ++ ": " ++ show diff ++ " nanoseconds"
```

	C++	Haskell
`map (*2)`	14 ms	149 ms
`filter even`	48 ms	139 ms
`reverse`	11 ms	806 ms
`uncurry zip . split id id`	36 ms	126 ms
Tempo real do processo	02.35 s	30.18 s

Google Hash Code 2020

Falemos agora sobre o problema do *Google Hash Code 2020*. O programa original, escrito em `Haskell`, foi desenvolvido durante a competição, que durou quatro horas, e está estruturado simplesmente como a composição de 3 passos – ler o *input*, resolver o problema, e escrever o *output* – como se pode verificar no código:

```
main = interact (outputToString . solve . readLibraries)
```

A conversão em `C++` segue a mesma estrutura, como se pode também verificar no código:

```
output_to_string(solve(read_libraries()));
```

Tem apenas duas pequenas excepções: enquanto que em `Haskell` temos as seguintes funções:

```
readLibraries :: String -> Libraries
outputToString :: Output -> String
```

Em `C++` temos dois procedimentos:

```
struct libraries read_libraries (void);
void output_to_string (output_t output);
```

Isto porque seria mais difícil implementar de uma forma mais funcional, e o resultado seria muito menos idiomático – estranho, até.

A conversão "imediata" para `C++`, com a biblioteca *"Functional Plus"*, demorou duas tardes a completar, um total de cerca de oito horas. Para alguns dos ficheiros de *input*, o programa em `C++` dá um resultado ligeiramente diferente do original. Acreditamos que isto se deva a diferenças entre as implementações do algoritmo de ordenação nas duas linguagens.

Quanto a performance, o programa original demora cerca de 7 segundos para processar todos os ficheiros de *input*, e o programa em `C++` demora cerca de 30 minutos. Esta diferença tão grande achamos que se deve às estruturas usadas em `C++` não serem adequadas para o uso que lhes estamos a dar – há muita cópia de memória.

Aspectos Importantes de Programação Funcional

Imutabilidade

Lazy Evaluation

Composição

ADTs