

Paradigma Funcional em Haskell e C++

André Sá (A76361) João Rodrigues (A84505)
Pedro Oliveira (A86328)

2020/07/22

Licenciatura em Ciências da Computação



Índice

1	Introdução	3
2	Abordagem ao Paradigma Funcional em Haskell e C++	3
2.1	O paradigma funcional	3
2.2	Haskell como linguagem funcionalmente pura	4
2.3	C++ "funcional"	5
3	Comparação e Análise de Programas Equivalentes em Haskell e C++	5
3.1	<i>Prelude</i>	6
3.1.1	<code>map</code>	6
3.1.2	<code>filter</code>	7
3.1.3	<code>reverse</code>	8
3.1.4	<code>zip</code>	8
3.1.5	Resultados	9
3.2	<i>Google Hash Code 2020</i>	10
3.2.1	Notas Sobre Funções Relevantes Utilizadas	11
3.2.2	Tipos e Estruturas de Dados	12
3.2.3	Ler o <i>Input</i>	13
3.2.4	Resolver o Problema	15
3.2.5	Escrever o <i>Output</i>	17
3.2.6	Resultados	18
4	Aspectos Importantes de Programação Funcional	19
4.1	Imutabilidade	20
4.2	<i>Lazy Evaluation</i>	22
4.3	Composição	23
4.4	ADTs	25
4.4.1	<code>struct/class</code>	26
4.4.2	<code>enum</code>	26
4.4.3	<code>union</code>	27
4.4.4	ADTs em C++	27
5	Conclusão	29

1 Introdução

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++, 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 `templates` 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 `templates`, mas esta não é essencial a Programação Funcional, e portanto não vamos desenvolver sobre este assunto.¹

Aproveitaremos também para aprofundar alguns aspectos/características importantes da 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.

2 Abordagem ao Paradigma Funcional em Haskell e C++

2.1 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 ao 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”.

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

Um exemplo extremo e pouco realista seria:

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

Deste pequeno excerto, podemos concluir uma de duas hipóteses:

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.

2.2 Haskell como linguagem funcionalmente pura

`Haskell` adota 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 efeitos secundários, em `Haskell`, em vez de se aceder ao mundo e alterá-lo 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

²Ver *Tackling the Awkward Squad*.

2.3 C++ "funcional"

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 abstração 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).
- Utilizar bibliotecas funcionais existentes, como “*Functional Plus*”³, “*CPP Prelude*”⁴, ou “*Ranges*”⁵.

3 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

³Ver *Functional Plus*.

⁴Ver *CPP Prelude*.

⁵Ver *Ranges*.

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*”, 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 tínhamos a versão em `Haskell` e fizemos a conversão para C++ utilizando a biblioteca “*Functional Plus*”, para uma comparação mais realista.

3.1 *Prelude*

De forma a comparar a performance de pequenos programas em âmbas as linguagens, 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++. Para cada função, vamos apresentar uma definição com recursividade explícita e uma definição recorrendo a funções de ordem superior em `Haskell`, seguidas de uma implementação em C++ e no final apresentamos os tempos de execução.

3.1.1 `map`

Começamos 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`.

Em `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 -> (f a :)) []
```

Em C++:

```
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;
}
```

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 devolve uma colecção, temos de criar uma a colecção de resultado (*res*) – em Haskell isto é feito de forma automática.

3.1.2 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`.

Em Haskell:

```
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 -> if p a then (a:) else id) []
```

Em C++:

```
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`.

3.1.3 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]`.

Em Haskell:

```
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 (:)) []
```

Em C++:

```
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.

3.1.4 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.

Em 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 (,)
```


Em C++:

```
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 manualmente como um ciclo.

3.1.5 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 seguinte macro:

```
#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"
```

Como Haskell é *lazy-by-default*, para obter-mos uma comparação justa é necessário forçar a avaliação das expressões que pretendemos testar. Para isso usamos o `deepforce`, que está definido como `deepforce x = deepseq x x`, sendo `deepseq a b` a função que força a avaliação de `a` e devolve `b`.

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

3.2 Google Hash Code 2020

Falemos agora sobre o problema do *Google Hash Code 2020*. O problema é de optimização, e consiste em planear que livros serão examinados e de que biblioteca, de forma a maximizar a pontuação. Não iremos detalhar aqui o problema em si, os ficheiros de input e output, – visto que estão disponíveis na página da competição⁶ – nem a estratégia usada para o resolver, visto não estar directamente relacionado com o tema deste trabalho.

O programa original, escrito em `Haskell`, foi desenvolvido durante a competição, que durou quatro horas, e está estruturado simplesmente como a composição de três 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:

⁶O enunciado do problema e ficheiros de input podem ser descarregados daqui: <https://codingcompetitions.withgoogle.com/hashcode/archive>

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

Em C++ temos estes 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.

3.2.1 Notas Sobre Funções Relevantes Utilizadas

3.2.1.1 `fplus::fwd::apply`

Esta função é usada para aplicar funções sobre colecções a uma colecção. Por exemplo, quando em Haskell escreveríamos `func3 . func2 . func1 $ col`, em C++, com a biblioteca “*Functional Plus*”, escrevemos `fplus::fwd::apply(col, func1, func2, func3)`.

Funções uteis para usar com `fplus::fwd::apply` podem ser encontradas no *namespace* `fplus::fwd`.

3.2.1.2 `fplus::transform`

Esta função serve um propósito similar ao da função `map` em Haskell. Enquanto que em Haskell escreveríamos `map func col`, em C++, com a biblioteca “*Functional Plus*”, escrevemos `fplus::transform(func, col)`.

3.2.1.3 `fplus::keep_if`

Esta função serve um propósito similar ao da função `filter` em Haskell. Enquanto que em Haskell escreveríamos `filter pred col`, em C++, com a biblioteca “*Functional Plus*”, escrevemos `fplus::keep_if(pred, col)`.

3.2.1.4 `fplus::fwd`

Como em C++ não há *auto-carrying*, de forma a ser possível compor funções da biblioteca, existem no *namespace* `fplus::fwd` versões para este propósito. Por exemplo, podemos escrever

```
fplus::fwd::apply(col,
                  fplus::fwd::keep_if(pred),
                  fplus::fwd::transform(func));
```

em vez de

```
fplus::transform(func, fplus::keep_if(pred, col));
```

3.2.1.5 std::tie

Para destruturar tuplos em C++ – mas não pares – podemos usar `std::tie`. Isto é equivalente a fazer *pattern match* sobre um tuplo em Haskell. Vejamos um exemplo:

Em Haskell:

```
t :: (Int, Float, Char, String, Double)
t = (0, 1, 'a', "ola", 2)
(x, _, _, _, d) = t
```

Em C++:

```
std::tuple<int, float, char, std::string, double> t
    = std::make_tuple(0, 1, 'a', "ola", 2);
int x;
double d;
std::tie(x, std::ignore, std::ignore, std::ignore, d) = t;
```

3.2.2 Tipos e Estruturas de Dados

Definimos apenas três tipos para uso no programa, – um para representar o *input*, outro para representar o *output*, e um para representar uma biblioteca – e as únicas estruturas de dados usadas foram listas e vectores – só vectores do lado de C++. Apresentamos as suas definições a seguir.

Em Haskell:

```
--                               ID      nBooks  Signup books/day books
type LibraryDesc = (Int, (Int,      Int,      Int,      V.Vector Int))

data Libraries = Libraries {
    nBooks :: Int,
    nLibraries :: Int,
    nDays :: Int,
    bookScore :: V.Vector Int,
    libraries :: [LibraryDesc]
} deriving Show

--                               lib  nb  books
newtype Output = Output [(Int, Int, V.Vector Int)]
```

Em C++:

```
typedef std::pair<int, std::tuple<int, int, int, std::vector<int>>>
    library_desc_t;

struct libraries {
    int n_books;
```

```

    int n_libraries;
    int n_days;
    std::vector<int> book_scores;
    std::vector<library_desc_t> libraries;
};

typedef std::vector<std::tuple<int, int, std::vector<int>>> output_t;

```

3.2.3 Ler o *Input*

Para a leitura da *input* tiramos proveito da *lazyness* do Haskell, e tomamos o *input* como uma *String*. Do lado do C++ usamos também `std::string`, simplesmente porque pretendíamos uma conversão mais directa – esta não é a melhor escolha para performance, mas para os ficheiros de *input* não é expectável qualquer penalização, visto que o maior destes tem apenas 3.4MB.

De seguida apresentamos o ficheiro de exemplo mais pequeno:

```

6 2 7      -- 6 livros, 2 bibliotecas, 7 dias
1 2 3 6 5 4 -- pontuação de cada livro
5 2 2      -- A biblioteca 0 tem 5 livros, 2 dias de signup, 2 livros/dia
0 1 2 3 4   -- Os livros que a biblioteca 0 tem
4 3 1      -- A biblioteca 1 tem 4 livros, 3 dias de signup, 1 livro/dia
0 2 3 5     -- Os livros que a biblioteca 1 tem

```

Em Haskell:

```

onTSBD (_, (_, ts, bd, _)) = (ts, bd)

readLibraries :: String -> Libraries
readLibraries = proc . map (map read . words) . lines
  where
    proc ([nb, nl, nd]:scores:library_desc) = Libraries nb nl nd scs libs
      where
        scs = V.fromList scores
        libs = (proc3 . zip [0..] $ proc2 library_desc)

    proc3 = sortOn onTSBD

    proc2 ([nb, su, bd]:books:t) = (nb, su, bd, V.fromList books):(proc2 t)
    proc2 _ = []

```

Em C++:

```

struct libraries read_libraries (void)
{
    std::string input(std::istreambuf_iterator<char>{std::cin}, {});
    std::vector<std::vector<int>> values = fplus::fwd::apply(
        input,

```

```

fplus::fwd::split_lines(false),
fplus::fwd::transform([](std::string line) -> std::vector<int> {
    return fplus::fwd::apply(
        line,
        fplus::fwd::split_words(false),
        fplus::fwd::transform(fplus::read_value_unsafe<int>));
}));

struct libraries ret;

{ /* Primeira linha */
    ret.n_books = values[0][0];
    ret.n_libraries = values[0][1];
    ret.n_days = values[0][2];
}

{ /* Segunda linha */
    ret.book_scores = values[1];
}

{ /* Resto */
    ret.libraries.reserve(ret.n_libraries);
    for (int i = 0; i < ret.n_libraries; i++) {
        std::vector<int> props = values[2 * i + 2 + 0];
        std::vector<int> books = values[2 * i + 2 + 1];

        int n_books = props[0];
        int sign_up = props[1];
        int books_per_day = props[2];

        ret.libraries.push_back(
            std::make_pair(
                i,
                std::make_tuple(n_books,
                                sign_up,
                                books_per_day,
                                books)));
    }
}

ret.libraries = fplus::sort_on(&on_ts_bd, ret.libraries);

return ret;
}

```

Como se pode verificar, o passo de separar a *string* de *input* em linhas e palavras,

e de ler essas palavras para inteiros, está muito parecido ao original – a definição da variável `values`. O passo `zip [0..] . proc2` já foi fundido num só *loop for*.

3.2.4 Resolver o Problema

Em Haskell:

```
ordBookScore :: Libraries -> Int -> Int
ordBookScore l id = (V.! id) $ bookScore l

solve :: Libraries -> Output
solve l = cenas . sortOn onTSBD $ solve' (nDays l) (distinct $ libraries l)
  where
    distinct :: [LibraryDesc] -> [LibraryDesc]
    distinct = map mapper . filter pred . fst . foldl' folder def
      where
        mapper (id, (nb, ts, bd, bs)) = (id, (nb, ts, bd, sorted))
          where
            sorted = V.fromList . sortOn (ordBookScore l) $ V.toList bs

        pred (_, (_, _, _, bs)) = not $ V.null bs

        def = ([], S.empty)
        folder (ret, s) (id, (nb, ts, bd, bs)) = let
          bss = S.fromList $ V.toList bs
          bs' = S.difference bss s
          ss = S.union s bs'
          bs'' = V.fromList $ S.toList bs'
        in ((id, (S.size bs', ts, bd, bs'')):ret, ss)

    cenas = Output . map (\(id, (nb, ts, bd, bs)) -> (id, nb, bs))

    solve' 0 _ = []
    solve' _ [] = []
    solve' nd (e@(_, (_, ts, _, _)):t)
      | ts <= nd = e:(solve' (nd - ts) t)
      | otherwise = solve' nd t
```

Em C++:

```
/* Esta função representa a função `distinct` acima */
struct libraries distinct (struct libraries libs)
{
  libs.libraries = fplus::fwd::apply(
    libs.libraries,
    fplus::fwd::fold_left(
      [](std::pair<std::vector<library_desc_t>, std::set<int>> a,
```

```

        library_desc_t b) {
std::vector<library_desc_t> ret = a.first;
std::set<int> s = a.second;

int id = b.first;
int ts, bd;
std::vector<int> bs;
std::tie(std::ignore, ts, bd, bs) = b.second;

std::set<int> bss(bs.begin(), bs.end());
std::set<int> bs_ = fplus::set_difference(bss, s);
std::set<int> ss = fplus::set_merge(s, bs_);

return std::make_pair(
    fplus::prepend_elem(
        std::make_pair(
            id,
            std::make_tuple(
                bs_.size(),
                ts,
                bd,
                std::vector<int>(bs_.begin(), bs_.end()))),
        ret),
    ss);
},
std::make_pair(std::vector<library_desc_t>(), std::set<int>()),
fplus::fwd::fst(),
fplus::fwd::keep_if([](library_desc_t pt)
    { return !std::get<3>(pt.second).empty(); }),
fplus::fwd::transform(
    fplus::fwd::transform_snd(
        [libs](std::tuple<int, int, int, std::vector<int>> tup) {
            int nb, ts, bd;
            std::vector<int> bs;
            std::tie(nb, ts, bd, bs) = tup;
            bs = fplus::sort_on([libs](int bid) { return libs.book_scores[bid]; }, bs);
            return std::make_tuple(nb, ts, bd, bs);
        })
    );
return libs;
}

/* Esta função representa a função `solve` acima */
std::vector<library_desc_t>
solve_ (int n_days, std::vector<library_desc_t> libs)
{

```



```

std::vector<library_desc_t> ret;
int len = libs.size();
assert(len > 0);

for (int i = 0; i < len && n_days > 0; i++) {
    library_desc_t e = libs[i];
    /*
     * `e` é um `std::pair`
     * `e.second` acede à componente da direita.
     *
     * `e.second` é um `std::tuple`
     * `std::get<1>` acede à componente de índice 1
     */
    int ts = std::get<1>(e.second);

    if (ts <= n_days) {
        ret.push_back(e);
        n_days -= ts;
    }
}

return ret;
}

output_t solve (struct libraries libs)
{
    libs = distinct(libs);
    return fplus::fwd::apply(
        solve_(libs.n_days, libs.libraries),
        fplus::fwd::sort_on(&on_ts_bd),
        fplus::fwd::transform(
            [](library_desc_t pt) {
                int id = pt.first;
                int nb;
                std::vector<int> bs;
                std::tie(nb, std::ignore, std::ignore, bs) = pt.second;
                return std::make_tuple(id, nb, bs);
            }));
}

```

3.2.5 Escrever o *Output*

Aqui apresentamos o *output* gerado pelos dois programas, dado o ficheiro de exemplo mostrado acima:

```
2          -- 2 bibliotecas
```

```

0 5          -- 5 livros, da biblioteca 0
0 1 2 4 3    -- Livros 0, 1, 2, 4, 3, por esta ordem
1 1          -- 1 livro, da biblioteca 1
5            -- Livro 5

```

Em Haskell:

```

outputToString :: Output -> String
outputToString (Output libs) = unlines
    . ((show nLibs):)
    $ concatMap mapper libs

where
    mapper = map (unwords . map show)
              . (\(x, y, l) -> [[x, y], V.toList l])
    nLibs = length libs

```

Em C++:

```

void output_to_string (output_t output)
{
    /* Esta operação representa `(show nLibs):` acima */
    std::cout << output.size() << std::endl;

    /* O corpo deste ciclo representa a função `mapper` acima */
    for (const std::tuple<int, int, std::vector<int>> & lib : output) {
        int x, y;
        std::vector<int> l;
        std::tie(x, y, l) = lib;

        std::cout << x << " " << y << std::endl;

        int len = l.size();
        assert(len > 0);
        std::cout << l[0];

        for (int i = 1; i < len; i++)
            std::cout << " " << l[i];

        std::cout << std::endl;
    }
}

```

3.2.6 Resultados

A conversão “imediata” para C++, com a biblioteca “*Functional Plus*”, demorou duas tardes a completar, um total de cerca de oito horas. Relativamente a performance, apresentamos de seguida a tabela com os tempos de processamento por cada ficheiro.

Ficheiro	Haskell	C++	Output igual
a_example.txt	0s	0s	Sim
b_read_on.txt	0.7s	9.95s	Não
c_incunabula.txt	1.26s	4m 40.83s	Sim
d_tough_choices.txt	1.57s	23m 59.30s	Não
e_so_many_books.txt	3.01s	3m 5.39s	Não
f_libraries_of_the_world.txt	2.9s	3m 13.55s	Sim
Total	9.44s	35m 9.02s	

Como é possível verificar na tabela, o tempo total de execução para todos os ficheiros de *input* é muito superior em C++. Pensamos que esta diferença acentuada se deve ao facto de as estruturas usadas em C++ não serem adequadas para o uso que lhes estamos a dar – existe muita cópia de memória. Por outro lado, a biblioteca que estamos a utilizar poderá ter sido pensada para satisfazer os requisitos funcionais descurando performance.

Para alguns dos ficheiros de *input*, o programa em C++ dá um resultado ligeiramente diferente do original. Como os programas são deterministas, acreditamos que isto se deve a diferenças entre as implementações do algoritmo de ordenação nas duas linguagens.

4 Aspectos Importantes de Programação Funcional

Neste capítulo detalharemos as características da programação funcional, mencionadas na introdução.

Composição é, provavelmente, o mais importante e talvez o único aspecto inerente a Programação Funcional. A ideia central de Programação Funcional é que construindo peças pequenas, fáceis de entender e de provar como correctas, é também “simples” construir um sistema complexo, correctamente.

De seguida, imutabilidade, em que objectos não são alterados mas sim 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 desejável 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.

A seguir, para cada um destes pontos, mostraremos e analisaremos exemplos de como se faz em `Haskell` e como se pode fazer em `C++`.

4.1 Imutabilidade

Uma das características mais importantes do paradigma funcional, é a noção de imutabilidade das expressões. Isto faz com que não seja possível alterar o valor de variáveis já existentes mas sim, criar novas variáveis com os novos valores.

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*, é sempre devolvido o mesmo valor de retorno – ou seja, substituindo uma expressão pelo seu valor de retorno, o seu significado permanece inalterado – e não tem efeitos colaterais observáveis. Por exemplo:

```
int g = 0;

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

/*
 * Não referencialmente transparente -- cada vez que a função é
 * invocada, tem um valor de retorno diferente
 */
int not_ref_trans1 (int x) {
    g++;
    return x + g;
}

/*
 * Não referencialmente transparente -- embora o valor de retorno seja sempre o
 * mesmo, dado o mesmo input, há um efeito colateral, o incremento de `g`
 */
int not_ref_trans2 (int x) {
    g++;
    return x + 1;
}
```

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 compilação. 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 apontador 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 `name_ref` e `name_ptr` não estarem qualificados com `const` e poderem ser alterados. No entanto, como apontam para uma variável `const`, gera-se uma contradição.

4.2 *Lazy Evaluation*

Lazy Evaluation é uma técnica de programação que adia a avaliação de uma expressão até que, e se, o seu valor for realmente necessário. Além disso, é possível evitar a reavaliação de uma expressão.

Muitas vezes, o resultado da avaliação de uma expressão é comum a várias operações. Se todas essas operações avaliassem a expressão, o sistema seria sobrecarregado desnecessariamente, resultando numa perda de performance. Por exemplo, no caso de um algoritmo que recorra ao cálculo do produto entre duas matrizes com alguma frequência, *lazy evaluation* 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 à repetição da mesma operação, o que contribui para o aumento da performance.

C++ não é *lazy-by-default*, e como tal, deverá ser o programador a aplicar esta técnica.

Vejamos uma possível implementação de *lazy evaluation* em C++, sendo necessário ter em atenção os seguintes pontos:

- Função sobre a qual queremos adiar o cálculo
- Uma *flag* que indica se já calculamos o resultado da função
- O resultado calculado

```

template <typename F>
class lazy_funcall
{
    const F func;
    typedef decltype(func()) RetType;
    mutable std::optional<RetType> ret;
    mutable std::once_flag call_once_flag;

public:
    lazy_funcall (F f) : func(f) { }

    const RetType & operator() () const
    {
        std::call_once(call_once_flag, [this] { ret = func(); });
        return ret.value();
    }
};

```

4.3 Composição

Uma parte importante de Programação Funcional é a composição de funções. Ao escrever funções pequenas e genéricas, e ao reutilizá-las com composição, é possível escrever programas completos rapidamente e com menos bugs. Em linguagens funcionais, composição é usada frequentemente; numa linguagem como C++ não é muito conveniente usar composição em todo o lado, principalmente por causa da sintaxe e da semântica de passar variáveis por valor ou referência. Há um sitio, no entanto, onde composição não tem de ser *pointwise*: trabalhar com colecções. Quando há um conjunto de operações a aplicar a uma colecção, seja no seu todo ou parte dela, expressar estas operações com algum tipo de *pipeline* é bastante intuitivo, legível e barato em número de caracteres escritos. Esta ideia não é nova – em linguagens funcionais este conceito é normalmente implementado como listas em linguagens *lazy-by-default*, como Haskell, ou *lazy-lists/streams* em linguagens *eager-by-default*, como Scheme.

Existem muitas operações sobre colecções que podem ser mapeadas numa *pipeline*, sendo muitas delas bastante comuns. Programá-las de cada vez manualmente como um *loop* é tedioso e muito provavelmente menos legível do que simplesmente usar as abstrações. Algumas destas operações comuns incluem somar, multiplicar, filtrar, mapear e o canivete suíço, com o qual muitas das outras operações são implementadas, o *fold* – também comumente conhecido como *reduce*, mas com semântica ligeiramente diferente.

A STL de C++ já tem algumas destas operações. Para os casos mais simples e comuns estas podem ser suficientes. É definitivamente melhor do que escrever um *loop* manualmente. Estas podem, no entanto, ser melhoradas. Existem várias bibliotecas que implementam conceitos funcionais em C++; vamos usar apenas a “*Functional Plus*” no documento. No entanto, existe uma outra biblioteca

parecida, “*Ranges*”, com melhor performance, mas a documentação é escassa, o que torna difícil perceber como a usar.

Como exemplo, dado um vector (ou uma lista em `Haskell`), filtrar os elementos dado um predicado, aplicar uma função 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 manualmente podia ser como o que se segue – omitindo a declaração e inicialização da variável `ret`:

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

Mas não é preciso escrever `loops for` manualmente grande parte das vezes – podemos em vez disso escrever o seguinte⁷:

```
fplus::fwd::apply(  
    xs  
    , fplus::fwd::keep_if(pred)  
    , fplus::fwd::transform(mapper)  
    , fplus::fwd::product()  
)
```

O nosso estudo não se centrou apenas na “*Functional Plus*”, no entanto. A partir do livro *Functional Programming in C++*, do Ivan Čukić, pudemos obter uma amostra do que é possível neste tipo de biblioteca. Em particular, o livro explica por alto porque é que a biblioteca “*Ranges*” tem melhor performance que a STL, e que a “*Functional Plus*”.

Começando por usabilidade: na STL, as funções têm como parâmetros dois *iterators*, – o início e fim da colecção de *input*, ou de parte dela – um iterador para o início da colecção de *output*, e um *inserter*, que dita como os elementos serão inseridos na colecção de *output*. De imediato, alguns pontos a melhorar saltam à vista:

1. Porque é que é preciso passar os iteradores de início e fim da colecção de *input*? Na verdade sabemos para que serve, mas não precisa de ser assim. Dá mais trabalho passar dois argumentos em vez de um, mas mais importante: é possível passar iteradores para o início e fim de duas colecções diferentes por engano.
2. Passar iterador e *inserter* da colecção de *output* também é tedioso; mas pior, significa que é sempre criada uma colecção de *output*.

⁷Para mais exemplos de uso, ver o programa do *Google Hash Code 2020*.

A “*Functional Plus*” melhora o primeiro destes pontos, – basta passar a colecção em si, não iteradores para ela – mas o segundo ponto continua um problema presente – e sempre devolvida uma nova operação como resultado.

A “*Ranges*” melhora estes dois aspectos ao simplesmente abstrair colecções como *ranges*, e devolver *ranges* como resultado das operações. Pode-se pensar nesta abstracção de *range* como um par de iteradores: o início e fim.

Agora é possível passar a uma operação uma colecção, que é automaticamente transformada num *range*, ou o resultado de uma outra operação, que já é um *range*. Esta última parte é crucial – significa que podemos compor operações *pointfree*.

Usabilidade está explicada. Vamos agora a performance. O par de iteradores que forma um *range* é só um par de apontadores. Para operações que não alteram a colecção original não há necessidade de copiar memória. Para implementar, por exemplo, o `filter`, basta implementar o operador `++` (*next*) para o *range* de *output*, sobre o iterador de início, procurando pelo elemento seguinte no *range* de *input* que satisfaz o predicado. Se nenhum elemento satisfaz o predicado chegamos eventualmente ao fim da colecção, ou seja, temos um *range* vazio.

Quando há a necessidade de alterar o *range* de *input* temos duas opções: copiar o *range* de *input*, ou mutar a colecção original *in-place*.

Todas as operações são *lazy*, ou seja, acontecem *on-demand*, como em `Haskell` – se o resultado não for usado, nada é feito.

4.4 ADTs

ADTs (*Algebraic Data-Types*), ou Tipos de Dados Algebricos, são tipos de dados criados a partir de tipos já existentes, de duas maneiras diferentes. Vamos dar uma breve descrição, para completude, simplesmente porque nem todas as linguagens funcionais têm ADTs nativamente ou com este nome.

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

A segunda, presente em grande parte das linguagens, mesmo que indirectamente, é o co-produto. Dados dois tipos A e B , o co-produto deles, $A + B$, é o conjunto cujos elementos ou são do tipo A ou do tipo B , mas é possível distingui-los – união disjunta. Este conjunto pode ser representado indirectamente como $Bool \times (A \cup B)$: um elemento de A ou B , e uma flag a indicar se é de A ou de B . Note-se que esta flag indica na verdade se o elemento é da esquerda ou direita: $A + A$ é um tipo válido.

Com estas duas técnicas de composição é possível representar qualquer estrutura de dados. Será então útil saber como usar estas duas técnicas numa linguagem de programação. Em `Haskell`, com o seu sistema de tipos avançado, âmbas estão disponíveis nativamente. Em `C++`, tal como em `C`, só o produto está disponível,

sob a forma de `structs`. Na STL também existem `std::pair` e `std::tuple` que podem considerar-se alternativas em alguns casos.

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

4.4.1 `struct/class`

Juntamos estas duas *keywords* visto que servem o mesmo propósito; a única diferença está em que, caso nada seja dito em contrário, numa `struct` todos os membros são públicos, enquanto que numa `class` são privados.

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

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

```
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 teórico; a segunda é uma definição mais comum; a terceira, com *records*, dá “nomes” aos vários campos e é mais parecida com uma definição 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 é mais idiomática, e a segunda, que é mais parecida com o tipo teórico, como a primeira definição em Haskell.

4.4.2 `enum`

Um exemplo simples e conhecido a todos do uso de *enums* é na definição 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 o co-produto de `false` e `true`, isto é, $Bool \cong False + True$.

Poderíamos assim achar que `enum` em C++ serve para representar co-produtos em geral mas estaríamos errados. `enum` serve apenas para representar o co-produto de vários conjuntos singulares ou um único conjunto enumerável de valores não inteiros e sem ordem. Veremos mais a frente como representar tipos de soma.

4.4.3 union

Esta é a menos comum das três *keywords*, por ser de uso muito limitado, e não existe equivalente em Haskell. Esta “falha” do lado de Haskell na verdade não é grave – possivelmente nem sequer é uma falha. Ao contrário do que o nome sugere, `union` não serve para representar a união de tipos, e não vamos aqui listar os seus usos além do necessário para este texto.

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

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

4.4.4 ADTs em C++

Vamos agora, finalmente, descrever como implementar ADTs em C++. A maneira mais idiomática, possível também em C, é usar uma *tagged union*.

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

Em Haskell:

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

Em C++:

```
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 definição em C++ é muito maior do que a definição em `Haskell`, não só devido à verbosidade de C++, como à necessidade de usar o truque mencionado acima de transformar um co-produto 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 implementação,

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

Neste caso, a `union` está realmente a simular a união de conjuntos.

Uma alternativa à *tagged union*, é 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>;
```

5 Conclusão

Ao longo deste documento, é possível constatar visualmente as diferenças sintáticas entre as duas linguagens. Em `Haskell` o código é bastante mais conciso do que em `C++`, pelo que a sua leitura e compreensão se torna mais simpática. Relativamente a eficiência e usabilidade das linguagens, em `Haskell` torna-se mais simples escrever programas relativamente eficientes com “pouco cuidado” uma vez que não há necessidade de preocupação com certos detalhes de implementação – gestão de memória, ordem de execução, etc. Embora `C++` não tenha sido inicialmente pensado para o paradigma funcional, é de notar que têm sido incluídas nas suas revisões alguns conceitos directamente relacionados com este mesmo paradigma, sendo por isso natural que em próximas revisões, a afinidade com este paradigma seja reforçada.

Relativamente ao nosso projecto, este deu-nos uma amostra sobre uma linguagem que até então nos era desconhecida, `C++`, e proporcionou-nos uma diferente perspectiva sobre alguns conceitos do paradigma funcional. Notámos, no entanto, que existe muito mais conteúdo a ser explorado no âmbito deste tema, tal como *Concorrência*, *Monads*, *Error Handling*, cuja investigação poderá ser realizada em projectos futuros, ou mesmo por outros alunos, dando continuidade ao material já existente.