UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PHILLIPE CÉSAR GOMES DE QUEIROZ

CONHECENDO A PROGRAMAÇÃO FUNCIONAL

FORTALEZA

2024

PHILLIPE CÉSAR GOMES DE QUEIROZ

CONHECENDO A PROGRAMAÇÃO FUNCIONAL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Pablo Mayckon Silva Farias.

FORTALEZA

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

Q46c Queiroz, Phillipe César Gomes de.
Conhecendo a programação funcional / Phillipe César Gomes de Queiroz. – 2024.
88 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Ciências, Curso de Computação, Fortaleza, 2024.
Orientação: Prof. Dr. Pablo Mayckon Silva Farias.

1. Programação funcional. 2. Linguagens de programação. 3. Algoritmos. I. Título.
CDD 005

PHILLIPE CÉSAR GOMES DE QUEIROZ

CONHECENDO A PROGRAMAÇÃO FUNCIONAL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: 20/09/2024.

BANCA EXAMINADORA

Prof. Dr. Pablo Mayckon Silva Farias (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Francisco Heron de Carvalho Junior
Universidade Federal do Ceará (UFC)

Prof. Dr. Yuri Lenon Barbosa Nogueira
Universidade Federal do Ceará (UFC)

A minha querida mãe, Neuma Gomes, que sempre fez o possível e impossível para me oferecer a melhor educação. Sem a sua inspiração, exemplo de luta e orientação, nada disso seria possível.

AGRADECIMENTOS

Ao Prof. Pablo Mayckon Silva Farias, pela sabedoria, paciência e excelente orientação. Nos momentos difíceis, ele me motivou. Nos momentos de falhas, ele me cobrou. Sem seu profissionalismo, didática e inspiração, eu não teria continuado no curso de Ciência da Computação.

Aos professores participantes da banca examinadora Francisco Heron de Carvalho Junior e Yuri Lenon Barbosa Nogueira pelo tempo, pelas valiosas colaborações e sugestões.

Ao Instituto Federal do Ceará, que me apresentou o universo da programação funcional, e à Universidade Federal do Ceará, que me permitiu aprofundar conhecimentos nas áreas do paradigma funcional e Ciência da Computação. Sem esse espaço de aprendizado e cada profissional que o constrói, nada disso seria possível.

A minha tia Irineuma Gomes, meu tio Elenildo Linhares e minha prima Talita Gomes que me deram o sentimento de pertencimento familiar e suporte durante toda minha graduação. Estarão sempre em meu coração.

Aos meus avós, Maria Gomes, Josué Barbosa, Iracema Vasconcelos e Adriano Queiroz. Maria e Josué, de origem humilde e camponesa, nunca tiveram a oportunidade de estudar, mas batalharam muito para conseguir fornecer o necessário aos seus filhos. Minha graduação é também por eles e seus sonhos. Iracema e Adriano sempre foram pessoas de coração imenso que também buscaram dar o melhor aos filhos mostrando sempre gentileza e sabedoria. Além disso, sem meu avô Adriano, não existiria meu gosto pela leitura e curiosidade de aprender.

Aos trabalhadores e trabalhadoras da área de ensino e tantas outras áreas que estão todos os dias lutando para construir um Brasil para sua classe.

Aos amigos que me apoiaram nos momentos mais difíceis e me deram forças para eu continuar.

RESUMO

Esse texto tem como finalidade apresentar o paradigma de programação funcional. Inicialmente serão visualizados alguns conceitos que o moldam. Logo em diante, serão discutidas implementações, que fazem uso do paradigma funcional, de algoritmos apresentados em cursos de Ciência da Computação. Dessa forma, analisaremos na prática a utilização de alguns conceitos, examinando características do código e extraindo os pontos positivos e negativos das implementações.

Palavras-chave: programação funcional; linguagens de programação; algoritmos.

ABSTRACT

This text aims to present the functional programming paradigm. At first, some concepts that shape it will be visualized. Then implementations, within functional paradigm, of algorithms taught in Science Computer courses will be discussed. In this way, we will analyze in practice the use of some concepts, examining code characteristics and extracting the positive and negative points of the implementations.

Keywords: functional programming; programming languages, algorithms

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

O paradigma de programação funcional vem conseguindo espaço nas linguagens de programação. Linguagens como Kotlin[1], Java[2], Dart[3] e muitas outras vêm implementando conceitos desse paradigma. Isso acontece devido ao entendimento, tido pelos projetistas e as comunidades de usuários de linguagens de programação, de que a utilização de ideias do paradigma funcional, de forma isolada ou em conjunto com recursos de outros paradigmas, permite códigos menores, mais fáceis de serem mantidos e modularizados.

Por outro lado, os conceitos e as técnicas da Programação Funcional ainda são bem menos difundidos que os da Programação Imperativa, incluidos aqueles da Orientação a Objetos. Nesse contexto, este trabalho busca então introduzir o paradigma funcional àqueles programadores que já apresentam alguma experiência, mas não possuem ainda conhecimento do universo funcional.

1.2 Objetivos gerais

Apresentar uma introdução do paradigma funcional e a importância de alguns conceitos nele presentes, pois muito do que será visualizado no texto pode ser utilizado na linguagem que o programador faz uso com frequência, sem deixar para trás o que ele já sabe sobre programação. Nesse sentido, não temos o objetivo de nos aprofundarmos no âmbito específico da programação funcional pura. Por essa razão, alguns conceitos importantes dentro dessa visão mais pura do paradigma não serão abordados no texto, como é o caso de mônadas, que são amplamente utilizadas na linguagem Haskell[4]. Nós também abordamos o assunto das estruturas de dados imutáveis de forma breve; ao leitor interessado no assunto, nós recomendamos o livro de Okasaki (1998).[5]

---

1 https://kotlinlang.org/
2 https://www.java.com/pt-BR/
3 https://dart.dev/
4 https://www.haskell.org/
5 A tese de doutorado que deu origem a esse livro é livremente acessível online: https://www.cs.cmu.edu/~rwh/students/okasaki.pdf.

1.3 Objetivos específicos

1. Explicar alguns conceitos presentes no paradigma funcional.
2. Demonstrar a heterogeneidade do paradigma funcional através de implementações principalmente em TypeScript[6], mas também OCaml[7], Elixir[8] e ReScript[9].
3. Implementar os tipos abstratos de dados: pilhas, filas e listas encadeadas, com conceitos utilizados no paradigma funcional.
4. Implementar os algoritmos de interpolação de Newton e quicksort com conceitos utilizados no paradigma funcional.

1.4 Organização do trabalho

No capítulo 2, vamos explorar as definições presentes no paradigma, partindo de funções, tipos de funções e formas como elas avaliam, até *closures*, imutabilidade, efeitos colaterais, casamento de padrões e tipos, apresentando principalmente códigos em TypeScript, mas também explorando rapidamente algumas outras linguagens funcionais, buscando apresentar ao leitor a heterogeneidade do paradigma funcional. Em seguida, no capítulo 3, utilizando TypeScript, iremos mostrar na prática a utilização desses conceitos em algoritmos ensinados em cursos de Ciência da Computação, apresentando uma análise crítica sobre os pontos positivos e negativos da utilização das ferramentas do nosso paradigma. Por fim, no capítulo 4, faremos as considerações finais, a partir de uma análise do paradigma e dos algoritmos implementados com conceitos funcionais.

1.5 Outras considerações

A nossa expectativa é que esse texto sirva ao leitor como porta de entrada para o estudo do paradigma funcional e para a utilização dos seus recursos na prática.

Para desenvolvimento do texto, realizamos a leitura dos artigos de Hudak (1989), Hughes (1989) e Turner (2012). Além disso, utilizamos os livros de Atencio (2016), Clarkson (2024), Lonsdorf e Benkort (2017) e Resig e Bear (2013). De forma não menos importante, utilizamos Cormen *et al.* (2012) para auxiliar na implementação de alguns algoritmos.

---

[6] https://www.typescriptlang.org/
[7] https://ocaml.org/
[8] https://elixir-lang.org/
[9] https://rescript-lang.org/

2 PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

Essencialmente, o paradigma funcional é a programação a partir de composição de funções. Logo, um programa escrito no paradigma é representado por uma função que utiliza outras funções e assim por diante. Além de estar sendo fortemente adotado pelas linguagens, como falaremos posteriormente, apresenta grandes contribuições na industria. Por exemplo, através de Erlang[1], o Whatsapp[2] alcançou uma escalabilidade de envio de mensagem em tempo real com mais de um bilhão de usuários ativos diariamente.[3] Próximo desse exemplo, através da linguagem de programação Elixir, o Discord[4] alcançou uma escalabilidade de envio de mensagem e de conversa utilizando (VoIP) com onze milhões de usuários ao mesmo tempo.[5] Assim, cada vez mais é necessário o entendimento do paradigma funcional.

Logo, iremos explicar alguns dos principais conceitos do paradigma funcional, partindo da unidade elementar, que é a função, explorando as categorias que elas podem assumir em diferentes linguagens. Em seguida, apresentaremos imutabilidade, efeitos colaterais, casamento de padrões e uma classificação sobre diferentes formas de tipagem que uma linguagem funcional pode ter.

2.1 Funções como alicerce

Quando estuda um novo idioma, o estudante em um determinado momento deve entender como as frases podem ser estruturadas. Por exemplo, no português, uma frase afirmativa comum segue a sequência de sujeito, predicado e objeto. Assim como na língua portuguesa, linguagens de programação têm regras que formalizam como os programas podem ser preenchidos de significados. Explorando um pouco mais a analogia, quando se entende como essas regras funcionam, é possível escrever diferentes tipos de texto. Caso alguém queira escrever um artigo, as formas de tal texto devem ser seguidas, juntando frases e mais frases. É de se observar que a ideia de juntar palavras e frases está presente em todo o processo da criação de um texto. O mesmo vale para as linguagens de programação e seus paradigmas. Um código seguindo o paradigma de programação funcional utiliza estruturas específicas, chamadas de funções, para integrar suas partes e gerar um significado, assim como o paradigma de orientação a objeto

[1] https://www.erlang.org/
[2] https://pt.wikipedia.org/wiki/WhatsApp
[3] https://favtutor.com/articles/whatsapp-discord-and-the-secret-to-handling-millions-of-concurrent-users/
[4] https://pt.wikipedia.org/wiki/Discord
[5] https://favtutor.com/articles/whatsapp-discord-and-the-secret-to-handling-millions-of-concurrent-users/

utiliza objetos para integrar suas partes e realizar uma computação.

A função é a principal parte do idioma e agora "juntar partes para formar uma frase", ou, melhor dizendo, criar um programa, consiste em realizar composição de funções. Recapitulando a história da computação, observamos que, durante a década de 1930, o matemático estadunidense Alonzo Church apresenta ao mundo o *lambda calculus* (Church 1932, 1936, 1940). Seu objetivo era criar um cálculo que capturasse a intuição sobre o comportamento das funções. Esse modelo consistia em um esquema de função único e suas regras de transformações.

Para o *lambda calculus*, uma função é definida como $\lambda x.A$. Em sua anatomia, a ocorrência de x após a letra grega λ significa que a função tem um parâmetro identificado como x. A letra A é o corpo da função e pode ser um identificador representado por uma letra minúscula, uma aplicação ou uma outra função. A função $\lambda x.x$ representa a função identidade e quando a aplicarmos a um identificador qualquer y, será retornado esse mesmo identificador. Denotamos esse comportamento através da notação $[x \to y](x)$ que significa substituir x por y no corpo da função. Nesse caso, aplicamos um passo de computação ou reduzimos a expressão $(\lambda x.x)y$ a y. Vale ressaltar que no *lambda calculus* proposto por Alonzo Church as funções não são nomeadas.

Criemos a função $\lambda x.xy$. Nessa expressão, a segunda ocorrência de x é uma variável ligada, pois está no escopo do parâmetro indicado pela primeira ocorrência de x, enquanto y é uma variável livre. Por outro lado, se analisamos x e y no contexto do termo xy ao invés do termo $\lambda x.xy$, temos que x e y são variáveis livres. Dito isso, "reduzir a expressão" $(\lambda x.A)y$, isto é, realizar nela um passo de computação, consiste, essencialmente, em substituir em A todas as ocorrências livres de x por y. Seguindo a terminologia original de Church, chamamos de redução-β a regra de aplicar uma função a uma expressão. Vejamos a redução da expressão $(\lambda x.xz)y$.

$(\lambda x.xz)y$

$[x \to y](xz)$

yz

Quando uma substituição gerar uma "captura", isto é, quando a variável substituta se tornar ligada por ocorrer dentro do escopo de um parâmetro, devemos renomear o parâmetro na expressão. Essa regra é conhecida como redução-α e nos diz, por exemplo, que $\lambda x.x = \lambda y.y = \lambda z.z$. Vejamos a redução da expressão $((\lambda x.(\lambda y.xy))y)z$

$((\lambda x.(\lambda y.xy))y)z$	
$((\lambda x.(\lambda w.xw))y)z$	Renomeando o parâmetro y para w
$[x \to y](\lambda w.xw)$	Aplicando $\lambda x.(\lambda w.xw)$ a y [redução-β]
$(\lambda w.yw)z$	
$[w \to z](yw)$	Aplicando $\lambda w.yw$ a z [redução-β]
yz	

Através da captura do comportamento de uma função presente no *lambda calculus*, da possibilidade de nomear funções e de ter zero ou mais parâmetros, materializamos as funções no contexto do paradigma funcional. Nesse caminho, o paradigma funcional implementa programas através da composição de funções. Devido a heterogeneidade das linguagens que implementam o paradigma, declarar funções tem diferentes sintaxes. Utilizaremos para o nosso texto, como principal linguagem de programação, o TypeScript. De acordo com a *Stack Overflow Developer Survey 2024*[6], a linguagem é a quinta linguagem mais popular. Além disso, apresenta a mesma sintaxe de JavaScript[7], que é a primeira linguagem mais popular, com a adição de tipos. Nesse sentido, é uma linguagem mais fácil de ser apresentada com os conceitos do paradigma funcional, pois é possível que o leitor já tenha utilizado-a. Logo, vejamos no Código-fonte 1 como declarar uma função nessa linguagem. Na linha 1, definimos a função de nome `identidade`, com um único parâmetro `x` de tipo `T` qualquer (mais adiante explicaremos tipos e polimorfismo paramétrico e como ele nos permite construir funções reutilizáveis).

Código-fonte 1 – TypeScript: Função identidade

```
1 const identidade = <T>(x: T) => x;
2
3 identidade(1) // retorna 1
```

[6] https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language
[7] https://ecma-international.org/publications-and-standards/standards/ecma-262/

A redução do *lambda calculus* corresponde ao que é conhecido como *avaliação de função*. Na linha 3 do Código-fonte 1, a função `identidade` é aplicada ao valor 1 e será reduzida ao próprio valor passado. Note que um valor é uma expressão que não pode mais ser reduzida (mais adiante, veremos que nas linguagens de programação, as funções podem ser avaliadas de forma ansiosa ou preguiçosa).

Então, partindo do *lambda calculus*, definimos as funções como recurso primário de nosso paradigma, aprendemos como declarar funções em nossa linguagem principal do texto. Vamos agora entender os tipos de funções e como através delas podemos criar nossos programas.

2.1.1 Funções Puras

Assuma que um programa A tenha entradas a_1, a_2 e saída s. Esse programa A, no paradigma funcional, é representado por uma função com parâmetros a_1, a_2 e retorno s. Nessa função, podem ser chamadas outras funções que então chamam outras funções, até que não exista mais nada para ser avaliado.

Exploremos mais o exemplo do programa A acima e adicionemos a informação que a_1, a_2 e s são do tipo inteiro. Suponhamos ainda que, para $a_1 = 2$ e $a_2 = 3$, a função sempre retornará 5. De forma ampla, assuma que para os mesmos valores de a_1 e a_2, o resultado de saída seja sempre o mesmo. Temos então que A é uma função pura.

Definição 1 (Versão preliminar) *Dizemos que uma função é pura se, recebendo os mesmos argumentos, a função sempre retorna a mesma saída.*

Escrever funções puras permite que um programa seja determinístico, pois sempre que informado os mesmos argumentos, será obtido o mesmo resultado.

2.1.2 Funções de alta ordem

Uma das formas que o paradigma funcional utiliza para unir as partes é permitir que funções sejam retornadas e passadas como argumento por outra função. O Código-fonte 2 exemplifica esse comportamento. A função anônima `(elemento) => elemento + 2` é um argumento da função `map`.

Código-fonte 2 – TypeScript: Soma cada elemento de uma lista por 2, gerando uma nova lista como resultado

```
1 const lista = [1, 2, 3, 4].map((elemento) => elemento + 2)
2 // [3, 4, 5, 6]
```

Assuma que o código acima precise de uma atualização em seu comportamento. Agora ele deve somar a mesma lista por qualquer valor em vez de somar sempre por dois. Como funções podem retornar outras funções, a solução para o novo problema pode ser feita da seguinte forma:

Código-fonte 3 – TypeScript: Soma cada elemento de uma lista por um inteiro qualquer

```
1  const somaListaPor = (lista: number[]) => {
2      return (valor: number) => {
3          return lista.map((elemento) => elemento + valor)
4      }
5  };
6
7  const lista = [1, 2, 3, 4];
8
9  const somaPor = somaListaPor(lista);
10
11 const somaPorDois = somarPor(2); // [3, 4, 5, 6]
12 const somaPorSete = somarPor(7); // [8, 9, 10, 11]
```

É notável que, caso seja necessário em algum trecho de código após a linha 12, somar a lista `lista` por qualquer outro valor, chamar a função *somarPor* é suficiente, não necessitando adicionar mais linhas de código e mais lógica para resolver o problema. De maneira mais ampla, elegantemente o paradigma permite compor programas. Sejam A e B dois programas escritos no paradigma funcional. Contanto que o resultado de B seja um tipo válido de entrada para A, B ou a avaliação de B pode ser argumento de A.

Então, uma função é dita de primeira ordem se todos o seus argumentos e o seu retorno não são funções. No entanto, se a função possui pelo menos um argumento ou retorno que é uma função de primeira ordem, ela é dita de segunda ordem. Seguindo a linha de raciocínio, uma função é de ordem $n+1$ se ela tem pelo menos um argumento ou retorno de ordem n e

nenhum de ordem mais alta. Assim, programação de alta ordem é aquela que as funções podem ser de qualquer ordem (PETER; HARIDI, 2004, p. 177).

Entretanto, a definição mais simplificada abaixo é normalmente adotada pela comunidade. Logo, durante o texto utilizaremos essa definição ao mencionar funções de alta ordem, com o intuito de facilitar o entendimento do leitor.

Definição 2 *Uma função é de alta ordem se algum dos seus parâmetros é uma função, ou se o seu retorno é uma função. Em caso contrário, a função é de primeira ordem.*

De forma transparente, ao criarmos funções com mais de um parâmetro, um conjunto de linguagens automaticamente transforma essa função em uma função de alta ordem que recebe o primeiro argumento e retorna uma função que recebe o próximo argumento e assim sucessivamente até que se chegue ao último argumento e o retorno da função possa finalmente ser expresso. Dizemos portanto que temos uma função *currificada*. O termo vem da palavra "Currying" em homenagem ao matemático Haskell Curry, que usou o conceito[8] amplamente. Apesar disso, a ideia foi desenvolvida pelo matemático Moses Schönfinkel. Em OCaml, no Código-fonte 4, podemos criar a função soma que recebe dois argumentos e será por padrão, uma função *currificada*.

Código-fonte 4 – OCaml: Função soma currificada

```
1  let soma a b = a + b;;
2
3  soma 7 3;; (* 10 *)
4
5  (* ou *)
6  let somaPor7 = soma 7;;
7  somaPor7 3;; (* 10 *);;
8
9  (* ou *)
10 (soma7)(3);; (* 10 *)
```

Outro ponto a se destacar, é que nas linguagens que buscam implementar o paradigma de programação funcional, as funções são chamadas de *funções de primeira classe* ou

[8] https://en.wikipedia.org/wiki/Currying

cidadãos de primeira classe. Isso significa dizer que uma função pode ser utilizada como qualquer outro tipo de dado. Ou seja, podem ser atribuídas a uma variável, armazenadas em uma lista, passadas como argumento e retornadas por outra função.

Apesar de linguagens funcionais possuírem funções de alta ordem, LISP[9], a primeira linguagem de programação funcional era de primeira ordem, ou seja, possuía apenas funções de primeira ordem (TURNER, 2012).

2.1.3 Avaliação Preguiçosa e Ansiosa

Além de classificar as funções como puras ou impuras e de alta ordem ou primeira ordem, podemos classificá-las com base em como são avaliadas. Em linguagens de programação, uma expressão segue um conjunto de regras para ser avaliada. Por exemplo, na expressão $f(g(),h())$, os argumentos podem ser avaliados completamente antes da função f ser aplicada. No entanto, a função pode ser aplicada antes de as funções g e h serem avaliados. Para o primeiro caso, temos uma avaliação estrita[10] ou avaliação aplicativa, pois os argumentos são avaliados completamente antes de aplicar a função. Para o segundo caso, temos uma avaliação não estrita ou avaliação normal e, portanto, os argumentos serão avaliados quando forem necessários na função. Assim, as funções g e h podem ser avaliadas apenas quando necessárias em f.

Quando os argumentos são avaliados o mais breve possível, temos uma avaliação ansiosa. Mas quando os argumentos são avaliados toda vez que forem necessários na função, temos uma estratégia conhecida como *call by name*. O resultado da avaliação de um argumento pode ser guardado para os pontos subsequentes que utilizam-no, sem necessidade de avaliá-lo toda vez que ele for necessário. Essa estratégia é conhecida como *call by need* ou avaliação preguiçosa.

Suponha a função $f(x) = 1$. Vamos aplicá-la a função identidade mencionada no Código-fonte 1. Vamos também aplicar $identidade$ a 2. Dessa forma, teremos a expressão $f(identidade(2))$. Se a expressão avaliar ansiosamente, $identidade(2)$ será avaliada e, portanto, teremos que agora avaliar a expressão $f(2)$, pois $identidade(2) = 2$. Mas f não utiliza x e assim, estaremos fazendo um passo de computação a mais para que a avaliação da expressão inicial retorne 1. Em contrapartida, se a expressão $f(identidade(2))$ avaliar preguiçosamente,

[9] https://pt.wikipedia.org/w/index.php?title=Lisp&oldid=67012202
[10] https://en.wikipedia.org/w/index.php?title=Evaluation_strategy&oldid=1247275836

a expressão irá avaliar para 1 sem uma avaliação desnecessária de $identidade(2)$.

Na prática, as linguagens podem utilizar a implementação das duas avaliações. Em JavaScript, a avaliação de funções é ansiosa, entretanto, a avaliação preguiçosa é simulada através de *geradores*. Através do símbolo `*`, após a palavra-chave `function`, criamos uma função geradora. Essa função gera um objeto gerador, que possui a propriedade `next`. As funções geradoras pausam a execução, em um ponto definido no corpo da função, com a palavra-chave `yield`. Assim, quando aplicamos `next`, a função irá executar até encontrar o próximo `yield`. Exploremos o Código-fonte 5 para entender melhor o funcionamento de funções geradoras.

Código-fonte 5 – JavaScript: Avaliação Preguiçosa

```
1  function* gerarListaInfinita(inicio = 0) {
2    let i = inicio;
3    const lista = [];
4    while (true) {
5      lista.push(i);
6      yield lista;
7      i += 1;
8    }
9  }
10
11 const coletar = (listaInfinita, total) => {
12   let i = 0;
13   while (i < total - 1) {
14     listaInfinita.next();
15     i += 1;
16   }
17
18   const { value } = listaInfinita.next();
19   return value;
20 };
21
22 const listaInfinita = gerarListaInfinita(0);
```

```
23 coletar(listaInfinita, 4) // [0, 1, 2, 3]
```

Na linha 1, criamos a função geradora `gerarListaInfinita`. Olhando o corpo da função rapidamente, podemos pensar que o laço da linha 4 nunca irá parar, mas observando atentamente, podemos notar o uso de `yield` na linha 6. Logo, ao chamarmos `next`, `i` será inserido em `lista` e a execução será parada. Dessa forma, podemos fazer sucessivas chamadas a `next` para ir inserido inteiros na lista. Note que é exatamente o que a função `coletar` faz. Ela recebe um objeto gerador `listaInfinita` e a quantidade de passos `total`. Então criamos uma lista com `total - 1` itens através de sucessivas chamadas de `listaInfinita.next()`, como pode ser visto na linha 14. Para criar nossa lista com `total` elementos, fazemos uma última chamada de `listaInfinita.next()` na linha 18.

Assim, criamos através de funções geradoras, uma lista "infinita". Uma vantagem presente na avaliação preguiçosa é a possibilidade de trabalhar com estruturas infinitas. Porém, para realizar esse tipo de computação, é necessário uma implementação de avaliação mais complexa em nossas linguagens, quando comparada com a implementação de avaliação ansiosa.

Em linguagens como OCaml, ReScript, Elixir e JavaScript, por padrão, as funções avaliam ansiosamente, enquanto em Haskell, as funções avaliam preguiçosamente. Portanto, podemos classificar nossas funções, baseando-se na forma em que elas são avaliadas. Vamos agora falar sobre o conceito de *Closures*.

2.1.4 Closures

No corpo de uma função, podemos atribuir valores a variáveis. Assim, as operações sobre esses valores podem ser feitas de qualquer forma que a linguagem permite que sejam feitas. Como funções são de primeira classe, é possível também atribuir funções a variáveis. Portanto, é possível declarar funções no corpo de uma outra função. Todas essas variáveis em conjunto com os argumentos formam o contexto daquela função. Observa-se que cada função, seja ela declarada no corpo de outra função ou não, possui o seu próprio contexto. Esse contexto não pode ser acessado externamente à função. Em teoria, o mesmo contexto não existiria após a avaliação da função.

Agora digamos que f seja o retorno da função g. Quando g é chamada, f é retornada, podendo acessar qualquer nome e função declarada no corpo da função g, mesmo que ela já tenha sido avaliada. Em teoria, f não poderia mais acessar algo do corpo de uma função que

já finalizou sua execução. Isso não acontece em virtude de uma implementação nas linguagens de programação chamada de *Closure*.

De acordo com Atencio (2016, p. 45), *Closure* é uma estrutura de dados que liga uma função ao seu ambiente no momento que ela é declarada.

Em um retorno breve ao Código-fonte 3, a linha 9 passa `lista` como argumento para a função `somaListaPor`. Essa função, então, é avaliada, retornando uma nova função. Mesmo que `somaListaPor` já tenha sido avaliada nas linhas 11 e 12, `somaPorDois` e `somaPorSete` conseguem utilizar o argumento `lista` de `somaListaPor` e avaliar corretamente.

Se nossa linguagem permite mutabilidade, o uso de Closures pode tornar uma função impura. No Código-fonte 6, a função `somar` utiliza a variável global `algumValor`. Nesse caso, para os mesmos valores de entrada, a função pode retornar valores diferentes como é mostrado nas linhas 5 a 7.

Código-fonte 6 – TypeScript: Mal uso de closure em TypeScript

```
1 let algumValor = 3;
2
3 const somar = (x: number, y: number) => x + y + algumValor;
4
5 somar(1, 1) // retorna 5
6 algumValor = 1
7 somar(1, 1) // retorna 3
```

Assim, é interessante que o usuário faça uma análise do uso de closures com a utilização de variáveis do contexto global que sejam mutáveis, pois em determinadas situações e linguagens, isso pode causar inconsistência no resultado esperado da avaliação da função. Note que nossa função também tem um maior acoplamento, pois agora ela depende de uma variável externa ao corpo da função. Vale ressaltar também que o contexto global falado anteriormente apenas se encerra quando o programa termina enquanto contexto da função pode encerrar antes do fim de um programa. Além disso, toda função declarada pode usar as variáveis globais daquele programa.

2.2 Imutabilidade e efeitos colaterais

Em programação distribuída, que vários computadores trabalharão juntos para resolver um determinado problema, a imutabilidade se torna de grande importância para que diferentes máquinas que se comunicam entre si não lidem com estados inconsistentes. Como exemplo, cita-se a linguagem Elixir de paradigma funcional para o desenvolvimento de sistemas distribuídos, a qual utiliza a imutabilidade como um dos seus pilares.

Quando programamos em linguagens que dão apoio ao paradigma imperativo, é comum que existam variáveis que mudem de valor no ciclo de vida de um programa. Uma simples instrução `for` é uma ferramenta que altera o valor de uma variável por *n* passos. Veja o exemplo do Código-fonte 7. Nele, a variável `i` tem seu valor somado mais 1 a cada passo até chegar ao valor dez. Dizemos, portanto, que `i` é uma variável mutável. Caso `i` não pudesse ter seu valor alterado ao longo da execução do programa, teríamos que `i` é imutável.

Código-fonte 7 – TypeScript: Instrução for

```
1 for (let i = 0; i < 10; i += 1) {
2     console.log(i);
3 }
```

Definição 3 *Dizemos que uma **variável** é imutável, se após um valor ser vinculado a essa variável, a linguagem não permitir que lhe seja associado outro valor.*

Algumas linguagens de programação buscam implementar a imutabilidade fazendo com que uma variável nunca possa mudar o valor ao qual ela se refere. Em TypeScript, através do uso da palavra-chave `const`, uma variável nunca poderá apontar para outro valor. Podemos visualizar isso no Código-fonte abaixo.

Código-fonte 8 – TypeScript: Criando variáveis imutáveis em TypeScript

```
1 const x = 1;
2
3 x = 2 // TypeError: Assignment to constant variable.
```

Porém, o uso de `const` nessa linguagem não proíbe que um valor possa ser mudado. Exploremos o Código-fonte 9. Na linha 1 dizemos que `obj` aponta para a estrutura par chave-valor `{ nome: 'A' }`. Na linha 2, em `obj`, alteramos o valor da propriedade `nome` de `'A'` para `'B'` resultando em `obj = { nome: 'B' }`. No entanto, se tentarmos atribuir uma nova estrutura par chave-valor para `obj`, como vemos na linha 4, o checador de tipos da linguagem TypeScript reportará um erro.

Código-fonte 9 – TypeScript: Alterando valores mesmo com utilização da palavra-chave const

```
1 const obj = { nome: 'A' };
2 obj.nome = 'B'; // obj = { nome: 'B' }
3
4 obj = { nome: 'C' } // TypeError: Assignment to constant
    variable.
```

Para outras linguagens, a imutabilidade é implementada no valor e não na associação de uma variável a um valor. Vejamos o caso de Elixir em que uma variável pode se referir a um outro valor após sua declaração.

Código-fonte 10 – Elixir: Imutabilidade em Elixir

```
1 # Permitido em elixir
2 x = [1, 2, 3]
3 x = ["a", "b", "c"]
```

Por mais que x assuma outras formas, as listas `[1, 2, 3]` e `["a", "b", "c"]` nunca serão alteradas durante todo o tempo de execução do código. Note que, ao atribuir `["a", "b", "c"]` a `x`, a lista `[1, 2, 3]`, por não ser mais utilizada em nenhum ponto do código, é elencada para ser removida da memória pelo *Garbage Collector*[11]. Veja que, para o mesmo conceito de imutabilidade, têm-se duas formas de implementá-lo que não necessariamente são opostas. Para ser justo com o TypeScript, ele possui a função `Object.freeze` que não permite alterações em valores do tipo lista, estruturas par chave-valor e outros. Portanto, são implementações que podem coexistir em uma mesma linguagem. O código abaixo tenta

[11] https://pt.wikipedia.org/w/index.php?title=Coletor_de_lixo_(informÃątica)&oldid=68252940

mudar o valor da propriedade `nome` do objeto "*freezed*" `obj` para `'B'`, mas o valor continua sendo `'A'`.

Código-fonte 11 – TypeScript: Utilizando Object.freeze

```
const obj = Object.freeze({ nome: 'A' })
obj.nome = 'B' // obj = { nome: 'A' }
```

Na linguagem de programação multiparadigma Dart, temos a coexistência dos dois tipos de imutabilidade.

Código-fonte 12 – Dart: Imutabilidade em Dart

```
var x = const [1, 2, 3] // imutabilidade sobre o valor
const y = ["a", "b", "c"] // imutabilidade na atribuicao
   variavel-valor
```

Com intuito de complementar a ideia de imutabilidade em uma linguagem de programação, temos a seguinte definição:

Definição 4 *Dizemos que um **valor** é imutável se após sua criação, a linguagem não permitir que esse valor seja alterado em memória.*

2.2.1 Efeitos colaterais

Uma Função pode realizar computações que alteram o estado do sistema. Quando isso acontece, ela não realiza apenas uma computação baseada nas entradas para gerar o resultado final, como também faz modificações em recursos externos à ela ou ao programa. Dizemos então que esta função causa efeitos colaterais. Operações desse tipo podem ser variadas, dependendo do ambiente no qual o programador está desenvolvendo seu trabalho. Algumas delas são (LONSDORF; BENKORT, 2017, p. 21-22):

1. Alterar o sistema de arquivos
2. Inserir um registro no banco de dados
3. Fazer uma chamada HTTP
4. Mutabilidade como foi explicado anteriormente

5. Fazer *logs*
6. Obter entradas do usuário
7. Alterar estados do sistema

Agora que definimos efeitos colaterais, o conceito de funções puras apresentado na Definição 1 pode ser complementado (LONSDORF; BENKORT, 2017, p. 20).

Definição 5 *Dizemos que uma função é pura se:*

1. *É livre de efeitos colaterais, e*
2. *Recebendo os mesmos argumentos, a função sempre retorna a mesma saída.*

No conjunto representado pelo paradigma de programação funcional, existe o subconjunto da programação funcional pura. Nele, todas as funções devem ser puras, e, portanto, livres de efeitos colaterais. No escopo da programação funcional pura, pode ser necessário criar programas que não são livres de efeitos colaterais devido à natureza das máquinas e problemas do mundo real. Uma das formas de lidar com essa situação é utilizando mônadas. Este TCC não tem como objetivo se restringir a esse subconjunto da programação funcional e em particular nós não explicaremos o que são mônadas. Para os leitores que quiserem entender melhor esse conceito, Atencio (2016), Lonsdorf e Benkort (2017) e Clarkson (2024) ajudam a entendê-las e como utilizá-las.

2.3 Casamento de padrões

Ao desenvolver um programa, é comum criarmos fluxos de execução. Por exemplo, "se $x = 0$, faça isso, caso contrário, faça aquilo". Logo, as linguagens nos fornecem a estrutura/comando `if <cond> then <expression> else <expression>`.

Código-fonte 13 – JavaScript: Estrutura/comando if else

```
1 const nome = 'A';
2
3 if (nome === 'A') {
4     console.log('nome = A')
5 } else {
6     console.log('nome != A')
7 }
```

No contexto de algumas linguagens que dão apoio ao paradigma funcional, `if else` se torna uma expressão. Portanto, pode ser reduzida a um valor.

Código-fonte 14 – OCaml: Expressão if else

```
let nome = "A"

let frase = if nome = "A" then "nome = A" else "nome != A"

print_endline frase (* "nome = A" *)
```

Quando queremos tratar mais de uma verificação, podemos aninhar os `ifs`.

Código-fonte 15 – JavaScript: Estrutura/comando if aninhada

```
const nome = "A";

if (nome === "A") {
  console.log("nome = A")
} else {
  if (nome === "B") {
    console.log("nome = B")
  } else {
    if (nome === "C") {
      console.log("nome = C")
    } else {
      console.log("nome desconhecido")
    }
  }
}
```

Quanto mais condições são necessárias verificarmos, mais ifs aninhado vão existir. Portanto, temos a estrutura/comando `else if` para evitar aninhamentos.

Código-fonte 16 – JavaScript: Estrutura/comando else if

```
const nome = "A";

if (nome === "A") {
  console.log("nome = A")
} else if (nome === "B") {
  console.log("nome = B")
} else if (nome === "C") {
  console.log("nome = C")
} else {
  console.log("nome desconhecido")
}
```

Além do `else if`, temos uma maneira mais fácil através da estrutura/comando `switch case`, que permite que o valor a ser avaliado seja mencionado apenas uma vez.

Código-fonte 17 – JavaScript: Estrutura/comando switch case

```
const nome = "A";

switch(nome) {
  case "A":
    console.log("nome = A")
    break;
  case "B":
    console.log("nome = B");
    break;
  case "C":
    console.log("nome = C");
    break;
  default:
    console.log("nome desconhecido")
}
```

Vejamos que todos os exemplos tratados anteriormente, com exceção do Código-fonte 14, são comandos e realizam um passo de computação baseado na avaliação de uma condição que verifica um determinado valor, como, por exemplo, se *nome* é igual a $'A'$. O casamento de padrões (*pattern matching*) vai além, avaliando também a estrutura do objeto.

Ao escrever `let x = (1, 2, 3)`, nosso paradigma funcional vincula `x` a `(1,2,3)` fazendo uma igualdade verdadeira. Logo, fazendo `let (a, 2, _) = (1, 2, 3)`, a linguagem verifica se o lado esquerdo da igualdade tem a mesma estrutura do lado direito. Veja que verificamos a igualdade tanto dos valores quanto do padrão da estrutura. O que se fez foi um casamento de padrão (*pattern matching*) em que `a` será vinculado a 1 apenas se o segundo elemento da tupla `(1, 2, 3)` for 2. Vale ressaltar que o símbolo "_" significa dizer que podemos ter qualquer valor nessa posição e que esse valor será irrelevante e, portanto, ignorado.

Em algumas linguagens inseridas no escopo do paradigma funcional, podemos utilizar casamento de padrões (*pattern matching*) em conjunto com expressões `switch`, `match` ou até na assinatura de nossas funções. No código abaixo, na linguagem de programação ReScript,

utilizamos casamento de padrões com a expressão `switch`.

Código-fonte 18 – ReScript: Casamento de padrão com a expressão switch

```
1  type universidade = UFC(string, int) | UECE(string) | IFCE
2
3  let minhaUniversidade = UFC("Phillipe", 2018)
4
5  let estuda = switch minhaUniversidade {
6  | UFC(nome, ano) => `UFC ${nome} - ${Belt.Int.toString(ano)
       }`
7  | UECE(nome) => `UECE ${nome}`
8  | IFCE => `IFCE -`
9  }
10
11 Console.log(estuda) // `UFC Phillipe - 2018`
```

Veja que o código checa se o nome `minhaUniversidade` tem uma das seguintes estruturas: `UFC(string, int)`, `UECE(string)` ou `IFCE`. Na linha 11, obtemos o resultado `"UFC Phillipe - 2018"` impresso na tela. Vamos agora visualizar o casamento de padrões em funções na linguagem Elixir.

Código-fonte 19 – Elixir: Casamento de padrões em funções

```
1  defmodule Exemplo do
2    defp filter([], _, lista_filtrada), do: lista_filtrada
3    defp filter([cabeca | cauda], f, lista_filtrada) do
4      nova_lista_filtrada = case f.(cabeca) do
5        true -> [cabeca | lista_filtrada]
6        false -> lista_filtrada
7      end
8
9      filter(cauda, f, nova_lista_filtrada)
```

```
10    end
11
12    def filter(lista, f) do
13      lista |> Enum.reverse() |> filter(f, [])
14    end
15  end
```

Nesse exemplo, temos a implementação da função `filter`. Ela percorre uma lista e retorna uma nova lista cujos elementos são aqueles que retornaram verdadeiro para um predicado passado como argumento. Em Elixir, uma função se diferencia por quantos argumentos ela possui. Por exemplo, a função `filter` da linha 3 é diferente da função da linha 12, pois a primeira recebe três argumentos e a segunda recebe apenas dois. Dizemos então que a função `filter` da linha 3 tem aridade 3, enquanto a função da linha 12 tem aridade 2.

A função da linha 2 é a mesma da linha 3, pois possuem o mesmo nome e a mesma aridade. Portanto, utilizamos um casamento de padrões em funções para saber qual corpo de função será executado. O corpo na linha 2 será executado apenas se o primeiro argumento for uma lista vazia. Nesse caso, retornamos uma lista com os elementos da lista de entrada para os quais a aplicação de `f` retorna verdadeiro. Caso a lista de entrada não seja vazia, executamos o corpo das linha 3 a 10, em que verificamos se `cabeca` estará na nova lista. Isso acontecerá caso a avaliação de `f.(cabeca)` retorne verdadeiro. Caso retorne falso, o elemento não fará parte da nova lista; logo, `nova_lista_filtrada` será igual à lista acumulada `lista_filtrada` do passo anterior. Por fim, chamamos `filter` recursivamente para a lista `cauda` e assim passamos a iteração para o próximo elemento da lista.

As funções declaradas com a palavra-chave `defp` são privadas. Dessa maneira, nossa função das linhas 2 e 3 não poderá ser utilizada externamente ao módulo. Entretanto, na linha 12 criamos uma função que estará visível para o usuário do módulo `Exemplo`. Note que no corpo dela utilizamos o operador `|>`. Esse operador passa o valor ou a avaliação de uma função à esquerda como primeiro argumento da função à direita. Por exemplo, na linha 13 passamos `lista` como argumento da função `Enum.reverse()` (que inverte os elementos de uma lista). Em seguida, a avaliação de `Enum.reverse(lista)` é passada como argumento de `filter(f, [])`. Ou seja, o programa irá avaliar `filter(lista, f, [])`.

No Código-fonte 20 temos um exemplo em que filtramos os elementos ímpares da

lista `[1, 2, 3]` através da chamada da função `rem(x, 2)`, que retorna o resto da divisão de x por 2, e, portanto, podemos verificar se o valor resultante é diferente de 0. Caso verdadeiro, o elemento será impar e, portanto, estará na lista filtrada.

Código-fonte 20 – Elixir: Filtrar elementos ímpares de uma lista

```
1 Exemplo.filter([1, 2, 3], fn x -> rem(x, 2) != 0 end) # 1 3
```

Observe que se removermos `Enum.reverse` na linha 13 do Código-fonte 19, ao chamar `filter(lista, f, [])`, nossa lista estará numa ordem incorreta. Suponha que `lista` seja `[1, 2, 3]`. Ao aplicar `Exemplo.filter` a `lista` e `fn x -> rem(x, 2) != 0 end`, nosso resultado será a lista `[3, 1]` que não respeita a propriedade do `filter` de manter os elementos filtrados na mesma ordem da lista original (o número 1 vem antes do número 3 em `[1, 2, 3]`).

Através do casamento de padrões em funções, conseguimos definir recursão de uma forma diferente. "Declaramos" duas funções (nas linhas 2 e 3 do Código-fonte 19) em que a primeira é a condição de parada, enquanto a segunda é o passo recursivo. Temos assim, uma sintaxe bem próxima de como representamos funções na matemática. A título de exemplificação, vejamos a função fatorial.

$$\text{fat}(n) = \begin{cases} 1, \text{ se } n = 0 \\ n \cdot \text{fat}(n-1), \text{ se } n \geq 1. \end{cases}$$

Implementando essa função em Elixir, temos o Código-fonte 21.

Código-fonte 21 – Elixir: Função fatorial

```
1 defmodule Fatorial do
2   def fat(0), do: 1
3   def fat(n), do: n * fat(n-1)
4 end
```

2.4 Tipos

As linguagens de programação que dão apoio ao paradigma funcional são classificadas em duas formas quando se fala em tipos: tipagem estática ou tipagem dinâmica. Uma linguagem tem tipagem estática quando os tipos dos dados são checados com base apenas no código-fonte do programa; tipicamente, essa checagem ocorre em tempo de compilação, e, portanto, na compilação é verificado se as operações feitas entre os tipos são válidas. Uma linguagem tem tipagem dinâmica quando os tipos dos dados são checados em tempo de execução, e a verificação se uma operação entre dois tipos de dados é válida acontece ao executar aquela linha de código. Linguagens como TypeScript, ReScript, OCaml e Haskell têm tipagem estática, enquanto JavaScript, Python[12], Elixir têm tipagem dinâmica.

Analisemos agora o escopo mais restrito das linguagens que possuem tipagem estática: suponha que queremos criar uma função que insere um elemento no final de uma lista. Ao invés de alterar a lista de entrada, queremos retornar uma lista nova. Se nossa lista é uma sequência de números, podemos implementar como mostra o código abaixo.

Código-fonte 22 – TypeScript: Função inserir um elemento em uma lista de números

```
1  const copiar = (lista: number[]) => {
2    const novaLista: number[] = [];
3    for (let i = 0; i < lista.length; i+=1) {
4      novaLista[i] = lista[i];
5    }
6
7    return novaLista;
8  }
9
10 const inserir = (lista: number[], valor: number) => {
11   const novaLista = copiar(lista);
12   novaLista[novaLista.length] = valor;
13
14   return novaLista;
15 }
```

[12] https://www.python.org/

Na função `copiar` declarada na linha 1, a partir de uma lista de entrada `lista`, retornamos uma `novaLista` com os mesmos elementos de `lista`. Já na linha 10, utilizamos a função `copiar` para não alterar a lista original e criar a lista-cópia `novaLista`; ao final dela, inserimos o novo elemento `valor`. Essas são funções bastante simples, mas limitadas a uma lista de números. Se quisermos inserir elementos em uma lista de *strings*, as funções acima não serão mais úteis. Precisaríamos criar uma função `inserir` para cada tipo que desejássemos. Isso pode se tornar inviável, a depender do contexto, além de aumentar a chance de problemas em futuras manutenções do código, uma vez que, ao se modificar uma versão da função, é preciso verificar também se a mesma modificação deve ser feita nas outras versões. Para resolver essa questão, as linguagens de tipagem estática utilizam o polimorfismo paramétrico. Nele mencionamos um tipo parametrizado, que será definido de acordo com o contexto de utilização. Vejamos o código abaixo:

Código-fonte 23 – TypeScript: Função que adiciona um elemento em uma lista de um tipo T qualquer

```
1  const copiar = <T>(lista: T[]) => {
2    const novaLista: T[] = [];
3    for (let i = 0; i < lista.length; i+=1) {
4      novaLista[i] = lista[i];
5    }
6
7    return novaLista;
8  }
9
10 const inserir = <T>(lista: T[], valor: T) => {
11   const novaLista = copiar(lista);
12   novaLista[novaLista.length] = valor;
13
14   return novaLista;
15 }
```

Veja que utilizamos uma nova notação <T>. Ela indica que receberemos um tipo T. O parâmetro lista será então uma lista com elementos de um tipo T qualquer. Assim, podemos passar uma lista de números, de *strings* ou de qualquer tipo válido na linguagem para as funções copiar e inserir, como exemplificado a seguir:

Código-fonte 24 – TypeScript: Exemplo de utilização da função inserir

```
1 const listaDeNumeros = [1, 2, 3];
2 const listadeStrings = ['Maria', 'Pablo', 'Phillipe'];
3
4 inserir(listaDeNumeros, 4); // [1, 2, 3, 4]
5 inserir(listadeStrings, ['Vladimir']); // ['Maria', 'Pablo
    ', 'Phillipe', 'Vladimir']
6 inserir(listaDeNumeros, 'Rice'); // Error de Tipo!
```

Observe como as nossas funções ganharam em generalidade. Claramente, portanto, o polimorfismo paramétrico é uma ferramenta extremamente poderosa no cinto de utilidades das nossas linguagens do paradigma funcional, que, em conjunto com funções de alta ordem, imutabilidade e casamento de padrões, nos permitirá construir algoritmos analisando as vantagens e desvantagens do paradigma, assim como comparações com versões que mesclam ou utilizam exclusivamente técnicas de outros paradigmas.

3 APLICANDO NO MUNDO REAL

Definidos os conceitos presentes em linguagens funcionais, neste momento, iremos explorar a utilização dos mesmos em algoritmos ensinados em um curso de Ciência da Computação. Começaremos explicando, através do tipo pilha, um padrão de escrita encontrado nas linguagens funcionais, assim como também analisaremos o tempo de execução dos algoritmos. Em seguida iremos falar sobre filas e a relação entre complexidade e utilização de estruturas imutáveis. Logo após, iremos mostrar a implementação das funções `reduce` e `map` em listas encadeadas, assim como obter resultados através da mesclagem de conceitos do paradigma funcional com o paradigma orientado a objetos. Por fim, vamos descrever os algoritmos de interpolação de Newton e quicksort sob a ótica do paradigma funcional, contrastando um caso em que obtemos um bom resultado com outro em que obtemos um resultado lento comparado à sua versão imperativa.

3.1 Pilhas

Pilhas são sequências nas quais a operação de remoção é definida de acordo com a ordem de inserção dos elementos (CORMEN *et al.*, 2012, p. 168). A seguir, usaremos os conceitos aprendidos no capítulo 2 para implementar essas estruturas e analisar suas versões.

O tipo abstrato de dados (TAD) pilha tem uma estratégia de remoção LIFO (*last-in*, *first-out*) em que a operação `DELETE` remove o último elemento inserido. A pilha apresenta as funções *push* (empilhar) que insere um elemento no fim da sequência e *pop* (desempilhar) que remove o último elemento inserido na sequência.

Nossas sequências de itens a serem empilhados serão representadas por listas ligadas e listas indexadas. Uma lista indexada pode ser declarada em TypeScript através de valores entre colchetes como podemos ver no código abaixo.

Código-fonte 25 – TypeScript: Declarando listas indexadas

```
1 const numeros = [1, 2, 3, 4];
2
3 const strs = ["Arnaldo", "Mariana", "Pablo"];
4
5 const booleans = [true, false, false, true];
```

```
6
7  const objetos = [
8    { nome: "Arnaldo", idade: 20 },
9    { nome: "Mariana", idade: 24 },
10   { nome: "Pablo", idade: 28 },
11 ];
```

Essa lista tem os métodos *push* e *pop*. Assim, já podemos utilizá-la como uma pilha sem escrever nenhuma implementação.

Código-fonte 26 – TypeScript: Funções push e pop nativas da linguagem

```
1  const pilha: number[] = [] // Uma lista vazia
2
3  pilha.push(1);
4  pilha.push(2);
5  pilha.push(3);
6  pilha.pop(); // retorna 3
7  pilha.pop(); // retorna 2
8  pilha.pop(); // retorna 1
9  pilha.pop(); // retorna undefined
```

A implementação acima utiliza programação orientada a objetos. Nela, temos um objeto do tipo lista com os métodos `push` e `pop`. Assim como linguagens que utilizam o paradigma funcional irão, por padrão, nos fornecer funções que se comportam sobre uma lista como o `push` e o `pop` de uma pilha, linguagens que utilizam o paradigma orientado a objetos possuem objetos nativos para representar o mesmo tipo abstrato de dados. Apesar dessas implementações nativas das funções facilitarem o trabalho dos programadores, pois não é preciso seguir um fluxo de implementação, otimização e criação de testes para as funções, não conseguimos explorar as diferentes abordagens entre o paradigma orientado a objetos e o paradigma funcional apenas utilizando funcionalidades que já estão prontas na linguagem. Portanto iremos implementar o tipo abstrato de dados pilha e analisar as diferenças entre as implementações. Vale ressaltar que o texto não tem o objetivo de dizer qual paradigma é melhor. O que buscamos é analisar com

o leitor a abordagem de escrita de código funcional em paralelo com a implementação mais comum aos programadores com o paradigma orientado a objeto.

Iniciemos com a implementação utilizando o paradigma funcional. Para isso, usaremos uma lista encadeada. Cada elemento da lista terá o tipo `Elemento<T>`:

Código-fonte 27 – TypeScript: Tipo do nó de uma lista encadeada

```
type Elemento<T> = {
  valor: T;
  proximo: Elemento<T> | null;
}
```

Nosso tipo utiliza polimorfismo paramétrico através da notação `<T>`. Isso permite que um elemento da lista possa ter um tipo `T` qualquer e assim não precisamos criar um ElementoString, ElementoNumber ou qualquer outro como podemos ver no Código-fonte 28. Vale ressaltar que quando definido o tipo `T` para uma lista específica, todos os elementos da lista terão aquele mesmo tipo.

Código-fonte 28 – TypeScript: Tipos sem polimorfismo paramétrico

```
type ElementoString = {
  valor: string;
  proximo: ElementoString | null
};

type ElementoNumber = {
  valor: number;
  proximo: ElementoNumber | null;
}

// ...
```

Ao invés de fazer como o código acima, podemos apenas utilizar `Elemento<string>` ou `Elemento<number>`. Seguindo isso, vamos implementar nossa pilha.

Código-fonte 29 – TypeScript: Implementação de Pilha no Paradigma Funcional

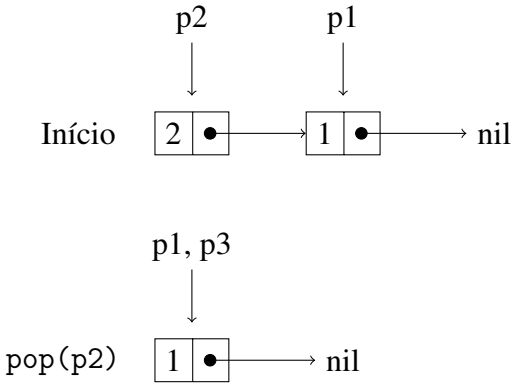
```
type Pilha<T> = {
  topo: Elemento<T> | null;
}

const criar = <T>(): Pilha<T> => {
  return { topo: null }
}

const estaVazia = <T>(p: Pilha<T>) => p.topo === null;

const empilhar = <T>(p: Pilha<T>, v: T) => {
  return {
    topo: {
      valor: v,
      proximo: p.topo
    }
  }
}


const desempilhar = <T>(p: Pilha<T>) => {
  if (estaVazia(p)) {
    return { valor: undefined, pilha: p };
  }

  const topo = p.topo as Elemento<T>;
  return {
    valor: topo.valor,
    pilha: {
      topo: topo.proximo
    }
```

```
32    };
33  }
34
35  export const Pilha = {
36    criar,
37    estaVazia,
38    empilhar,
39    desempilhar
40  }
```

Na linha 1 temos o tipo `Pilha<T>` que também tem um tipo paramétrico. Nossa pilha portanto é uma estrutura que tem uma propriedade `topo` com o tipo `Elemento<T> | null`. Essa pilha será suficiente para qualquer tipo válido `T` que seja passado. Mais uma vez, através do polimorfismo paramétrico, não teremos que implementar um tipo de pilha para cada tipagem ao utilizar essa implementação.

A partir da linha 5, implementamos todas as funcionalidades que nossa pilha terá. Começamos com a função `criar` que cria uma pilha vazia. Logo em seguida implementamos a função `estaVazia` que é responsável por verificar se a pilha está vazia ou não através da comparação `p.topo === null`. Depois implementamos as funções `empilhar` e `desempilhar` e por fim, na linha 29, exportamos as funções que poderão ser utilizadas externamente ao módulo.

Figura 1 – Compartilhamento de memória com lista encadeada



Vale ressaltar que o `empilhar` cria um novo elemento com um valor e uma referência para o elemento do topo antigo. Dessa forma, se fizermos `p1 = Pilha.push(p0, 1)` e logo em seguida fizermos `p2 = Pilha.push(p1, 2)`, as duas pilhas compartilharão um

mesmo elemento cujo o `valor` é 1, conforme ilustrado na Figura 1. Assim, de forma inteligente, consumimos menos recursos computacionais. Note também que isso é possível apenas porque nossa implementação de pilha é imutável e portanto, temos a certeza que nenhum nó da lista encadeada mudará seu valor durante a execução do programa. Dito isso, vamos visualizar abaixo como usar as funções apresentadas.

Código-fonte 30 – TypeScript: Utilização de Pilha implementado no paradigma funcional

```
1 const p0 = Pilha.criar<number>();
2
3 const p1 = Pilha
4   .empilhar(Pilha.empilhar(Pilha.empilhar(p0, 1), 2), 3);
5
6 Pilha.desempilhar(p1).valor // Retorna 3
```

Veja que `Pilha.empilhar(p, v)` retorna uma pilha com topo v. Então, após criar uma pilha vazia na linha 1 do Código-fonte 30, inserimos os elementos 1, 2 e por último 3 através de composição de funções `Pilha.empilhar`. Logo, ao realizar um `Pilha.desempilhar`, teremos o retorno do último valor inserido 3.

Mas note que realizar composições de funções pode gerar um código complexo de entender. Na verdade, linguagens com um maior propósito de utilizar o paradigma de programação funcional utilizam o operador pipe (em geral com notação `|>`). Assim, é possível criar um código mais legível, como podemos ver no Código-fonte 31.

Código-fonte 31 – Elixir: Usando o pipe operator

```
1 Pilha.criar()
2 |>Pilha.empilhar(1)
3 |>Pilha.empilhar(2)
4 |>Pilha.empilhar(3)
5 |>Pilha.desempilhar() // Retorna 3
```

O mesmo resultado pode ser obtido no JavaScript se implementarmos um *curried*

(Verifique a subseção 2.1.2) `Pilha.empilhar` e utilizarmos a biblioteca Ramda[1].

Código-fonte 32 – TypeScript: Utilizando a função pipe

```
import * as R from "ramda";

const empilhar = <T>(v: T) =>
  (p: Pilha<T>) => Pilha.empilhar(p, v);

const getP1 = R.pipe(
  Pilha.make<number>,
  empilhar(1),
  empilhar(2),
  empilhar(3),
)

const p1 = getP1();

p1.desempilhar().valor // Retorna 3
```

Voltando ao Código-fonte 29, podemos destacar que cada função utiliza polimorfismo paramétrico para que elas funcionem em qualquer tipo de pilha, seja ela uma `Pilha<number>`, `Pilha<string>`, ou qualquer outra. Outro ponto a se destacar são os argumentos que as funções implementadas podem receber. Veja, todas possuem um parâmetro `p` do tipo `Pilha<T>`. Temos de um lado de nossa implementação a informação com um determinado tipo e de outro as operações — funções — sobre essa informação. Se a função é o alicerce de nossa linguagem, a filosofia de escrita de código é separar as estruturas das operações que poderão ser aplicadas sobre elas, deixando com que as funções tenham uma única responsabilidade — Realizar passos de computação sobre uma determinada estrutura.

Diferente da filosofia do paradigma funcional, a filosofia do paradigma orientado a objetos não separa informação das operações. Na verdade, temos um objeto que contém propriedades(informações) e métodos(operações). Para visualizar o que foi definido, vamos analisar o código de uma implementação de pilha no paradigma orientado a objetos.

[1] https://ramdajs.com/docs/

Código-fonte 33 – TypeScript: Implementação de Pilha no Paradigma Orientado a Objetos

```
class Pilha<T> {
  private _topo: Elemento<T> | null;

  constructor() {
    this._topo = null;
  }

  estaVazio = () => {
    return this._topo === null;
  };

  empilhar = (valor: T) => {
    const antigoTopo = this._topo;

    const novoTopo: Elemento<T> = {
      valor: valor,
      proximo: antigoTopo
    };

    this._topo = novoTopo;
  };

  desempilhar = () => {
    if (this.estaVazio()) {
      return undefined;
    }

    const topoRemovido = this._topo as Elemento<T>;
    this._topo = topoRemovido.proximo;

    return topoRemovido.valor;
```

```
32    };
33  }
```

No código acima, definimos uma classe `Pilha`. Dentro dela, na linha 2, definimos a propriedade `_topo`. Ela guardará o topo da pilha caso exista um. Caso contrário, ela será `null`. Logo em seguida definimos uma função especial conhecida como construtor que se responsabiliza por fazer a configuração inicial ao criar o objeto. Basicamente, ele tem a mesma função que `criar` da implementação no paradigma funcional. Em seguida, da linha 8 a 32, temos as implementações das funções `estaVazia`, `empilhar` e `desempilhar` semelhantes ao que foi apresentado no Código-fonte 29.

No Código-fonte 33, não separamos informação de operações. Na verdade criamos uma planta do nosso objeto — uma classe — com a implementação de suas propriedades e métodos. Nesse paradigma, a divisão de responsabilidades ocorre entre classes. Cada classe tem seu objetivo dentro da modelagem de um sistema.

Código-fonte 34 – TypeScript: Utilização de Pilha implementada no paradigma orientado a objetos

```
1  const p1 = new Pilha<number>();
2  p1.empilhar(1);
3  p1.empilhar(2);
4  p1.desempilhar(); // Retorna 2
5
6  const p2 = new Pilha<string>();
7  p2.empilhar("Lia");
8  p2.empilhar("Pablo");
9  p2.desempilhar(); // Retorna "Pablo"
```

No código acima, criamos um objeto a partir da classe `Pilha` e ligamos ao nome `p1`. Logo em seguida realizamos `push(1)` e `push(2)`. Então, na linha 7, fazemos um `pop` que remove e retorna o valor do topo da pilha. Realizamos as mesmas operações a partir da linha 6 em diante, todavia, `p2` é uma pilha de *string* e não uma pilha de números como é o caso de `p1`.

3.2 Filas

Assim como Pilhas (verificar seção 3.1), Filas são sequências nas quais a operação de remoção são definidas de acordo com a ordem de inserção dos elementos (CORMEN *et al.*, 2012, p. 168). A seguir, usaremos os conceitos aprendidos no capítulo 2 para implementar essa estrutura e analisar suas versões.

O tipo abstrato de dados (TAD) Fila tem uma estratégia de remoção FIFO (*first-in*, *first-out*) em que a operação `DELETE` remove o primeiro elemento inserido. A fila apresenta as funções *enqueue* (enfilar) que insere um elemento no fim da sequência e *dequeue* (desenfilar) que remove o primeiro elemento inserido no conjunto dentre aqueles elementos que ainda não foram removidos.

Assim como podemos criar uma pilha através de listas indexadas e duas funções nativas da linguagem TypeScript, com a fila podemos fazer o mesmo utilizando a função *push* para inserção e *shift* para a remoção do primeiro elemento inserido.

Código-fonte 35 – TypeScript: Criando uma pilha através de uma lista indexada

```
1 const queue = [];
2
3 queue.push(1);
4 queue.push(2);
5 queue.push(3);
6
7 queue.shift(); // Retorna 1
```

Mas veja que no código acima, a lista `queue` é alterada durante a execução das linhas 3 a 7. No capítulo 2 destacamos a ideia de imutabilidade e, portanto, vamos explorá-la melhor agora com o tipo fila nos colocando o desafio de implementá-lo sem perda no tempo amortizado de inserção e remoção em O(1).

Nossa fila será representada por um tipo com a propriedade `primeiro` que indica o primeiro elemento da fila e a propriedade `ultimo` que indica o último elemento da fila. Caso `primeiro` e `ultimo` seja `null`, a fila estará vazia. Além disso, para que nossa fila seja reutilizada em qualquer situação necessária, usaremos polimorfismo paramétrico. O resultado será o tipo `Fila` apresentado abaixo.

Código-fonte 36 – TypeScript: Tipo Fila

```
type Fila<T> = {
  primeiro: Elemento<T> | null;
  ultimo: Elemento<T> | null;
}
```

Em seguida, vamos implementar quatro funções:

1. `criar`: cria uma lista vazia
2. `estaVazia`: verifica se a fila está vazia
3. `enfilar`: Adiciona um elemento no final da fila
4. `desenfilar`: Remove o primeiro elemento da fila

O resultado será o código abaixo.

Código-fonte 37 – TypeScript: Implementação de fila mutável

```
const criar = <T>(): Fila<T> => {
  return { primeiro: null, ultimo: null }
};

const estaVazia = <T>(f: Fila<T>) => {
  return f.primeiro === null && f.ultimo === null;
};

const enfilar = <T>(f: Fila<T>, v: T) => {
  const elemento = {
    valor: v,
    proximo: null
  };

  if (estaVazia(f)) {
    f.primeiro = elemento;
  } else {
    const antigoUltimo = f.ultimo as Elemento<T>;
```

```
19      antigoUltimo.proximo = elemento;
20    }
21
22    f.ultimo = elemento;
23  };
24
25  const desenfilar = <T>(f: Fila<T>) => {
26    if (estaVazia(f)) return null;
27
28    const primeiro = f.primeiro as Elemento<T>;
29    f.primeiro = primeiro.proximo;
30
31    if (primeiro.proximo === null) {
32      f.ultimo = null;
33    }
34
35    return primeiro.valor;
36  };
37
38  export const Fila = {
39    criar,
40    estaVazia,
41    enfilar,
42    desenfilar
43  }
```

A função `enfilar` recebe uma fila e um valor `v`. Se a fila estiver vazia, ligamos `f.primeiro` e `f.ultimo` ao novo `elemento`. Caso contrário, trocamos o valor da propriedade `antigoUltimo.proximo` para ser o novo `elemento`. Por fim, atualizamos o último elemento da lista com `f.ultimo = elemento`. Já a função `desenfilar` recebe apenas uma fila. Caso ela esteja vazia, retornamos `null`. Todavia, se a lista não estiver vazia, ligamos o nome `primeiro` a `f.primeiro`, atualizamos `f.primeiro` para `primeiro.proximo`. Corrigimos o valor de

`f.ultimo` caso estejamos desenfilando de uma fila com um único elemento e por fim retornamos `primeiro.valor`. Até então, a implementação acima continua sendo uma fila mutável com `enfilar` e `desenfilar` em tempo amortizado O(1).

Um pensamento rápido no primeiro instante para tornar nossa fila imutável é na função `desenfilar`, em vez de alterar a propriedade `proximo` de um elemento da fila, seja criado uma cópia da fila e então inserido um novo elemento nessa cópia. Além disso, vamos atualizar a função `desenfilar` para que ela retorne uma estrutura com o valor removido e a nova fila após essa remoção. Dessa forma, a fila passada para essas funções permanece não alterada e atingimos nosso objetivo de implementar uma fila imutável. Abaixo temos a implementação de nossa fila imutável.

Código-fonte 38 – TypeScript: Implementação de fila imutável obtida por cópia

```
const criar = <T>(): Fila<T> => {
  return { primeiro: null, ultimo: null }
};

const estaVazia = <T>(f: Fila<T>) => {
  return f.primeiro === null && f.ultimo === null;
};

const copiarElemento = <T>(
  elemento: Elemento<T> | null,
  anterior: Elemento<T>
): Elemento<T> => {
  if (elemento === null) {
    return anterior;
  }

  const novoElemento: Elemento<T> = {
    valor: elemento.value,
    proximo: null,
  }
```

```
21
22    anterior.proximo = novoElemento;
23
24    return copiarElemento(elemento.proximo, novoElemento);
25  }
26
27  const copiar = <T>(f: Fila<T>): Fila<T> => {
28    if (estaVazia(f)) {
29      return criar<T>();
30    }
31
32    const primeiro = f.primeiro as Elemento<T>;
33    const novoPrimeiro = {
34      valor: primeiro.valor,
35      proximo: null
36    }
37
38    const novoUltimo = copiarElemento(primeiro.proximo,
        novoPrimeiro);
39    return {
40      primeiro: novoPrimeiro,
41      ultimo: novoUltimo
42    };
43  }
44
45  const enfilar = <T>(f: Fila<T>, v: T) => {
46    if (estaVazia(f)) return f;
47    const novaFila = copiar(f);
48
49    const elemento = { valor: v, proximo: null }
50    const ultimo = novaFila.ultimo as Elemento<T>;
51    ultimo.proximo = elemento
```

```
52    novaFila.ultimo = elemento;
53
54    return novaFila;
55  };
56
57  const desenfilar = <T>(f: Fila<T>) => {
58    if (estaVazia(f)) {
59      return { valor: null, fila: f };
60    }
61
62    const primeiro = f.primeiro as Elemento<T>;
63    const novaFila = {
64      primeiro: primeiro.proximo,
65      ultimo: primeiro.proximo === null ? null : f.ultimo
66    }
67
68    return { valor: primeiro.valor, fila: novaFila };
69  };
```

Na linha 27, implementamos a função copiar. Se a fila `f` estiver vazia, nossa função retorna uma nova fila vazia através de `criar<T>()` na linha 29. Caso contrário, vamos criar uma cópia do primeiro elemento e atribuir a variável `novoPrimeiro` como podemos ver nas linhas 32 e 33. Logo em seguida, criamos uma cópia do último elemento ao chamar a função `copiarElemento`. Por fim, na linha 39 retornamos uma nova fila. Vale ressaltar que a função `copiarElemento` é uma função recursiva que vai iterar sobre cada elemento da fila e lhe copiar. Quando `elemento` for `null`, quer dizer que iteramos toda a lista. Nesse caso, `anterior` será o último elemento copiado da fila e portanto iremos retorná-lo.

Na linha 47 da função `enfilar` criamos uma cópia da fila `f` chamada de `novaFila`. Essa função `copiar` transforma nosso enfilar em um algoritmo que executa em tempo O(n), sendo *n* o tamanho da fila recebida, pois toda vez que for inserido um elemento na fila, será criado uma cópia de `f`. Apesar de obtermos uma fila imutável no algoritmo acima, nossa operação de enfilar se torna pior, pois ao contrário do `enfilar` anterior que executava em tempo

amortizado O(1), agora temos um tempo O(n).

Será que existe uma implementação de fila imutável que mantém as funções `enfilar` e `desenfilar` executando em tempo amortizado O(1)? A resposta é sim. Para isso, podemos raciocinar que uma fila pode ser construída através de duas pilhas. Quando enfilarmos um valor, ele será inserido na primeira pilha. Quando desenfilarmos, iremos realizar um desempilhar na segunda pilha caso ela não seja vazia. Caso contrário, iremos desempilhar a primeira pilha até que esteja vazia. A cada passo desse desempilhamento, iremos inserir o elemento removido na segunda pilha. Por fim, iremos realizar um desempilhar na segunda pilha.

Frequentemente, nosso algoritmo de desenfilar executará em tempo O(1). Apenas quando nossa segunda pilha for vazia, teremos um tempo O(n). Então, apesar de respondermos sim para a pergunta do parágrafo anterior, podemos ver que o algoritmo `desenfilar` nem sempre executará em tempo O(1) no pior caso, embora fique mantido o tempo amortizado constante. Vale ressaltar que o algoritmo de pilha apresentado em Código-fonte 29 é uma implementação de pilha imutável. Vamos reutilizá-lo para fazer a implementação de uma fila através de duas pilhas.

Código-fonte 39 – TypeScript: Implementação de fila com duas pilhas

```
type Fila<T> = {
  entrada: Pilha<T>;
  saida: Pilha<T>
}

const criar = <T>(): Fila<T> => {
  return {
    entrada: Pilha.criar<T>(),
    saida: Pilha.criar<T>(),
  }
};

const estaVazia = <T>(f: Fila<T>) => {
  return Pilha.estaVazia(f.entrada) && Pilha.estaVazia(f.
     saida);
```

```
15 }
16
17 const enfilar = <T>(f: Fila<T>, v: T): Fila<T> => {
18   const novaEntrada = Pilha.empilhar(f.entrada, v);
19
20   return {
21     entrada: novaEntrada,
22     saida: f.saida
23   };
24 };
25
26 const transferirDeEntradaParaSaida = <T>(
27   entrada: Pilha<T>,
28   saida: Pilha<T>,
29 ): Fila<T> => {
30   if (Pilha.estaVazia<T>(entrada)) {
31     return { entrada, saida };
32   }
33
34   const resumo = Pilha.desempilhar(entrada);
35
36   return transferirDeEntradaParaSaida(
37     resumo.pilha,
38     Pilha.empilhar(saida, resumo.valor)
39   );
40 };
41
42 const desenfilar = <T>(f: Fila<T>) => {
43   if (!Pilha.estaVazia(f.saida)) {
44     const resumo = Pilha.desempilhar(f.saida);
45
46     return {
```

```
47        valor: resumo.valor,
48        fila: {
49          entrada: f.entrada,
50          saida: resumo.pilha
51        }
52      };
53    }
54
55    const novaFila = transferirDeEntradaParaSaida(f.entrada,
        f.saida);
56    const resumo = Pilha.desempilhar(novaFila.saida);
57
58    return {
59      valor: resumo.valor,
60      fila: {
61        entrada: novaFila.entrada,
62        saida: resumo.pilha
63      }
64    };
65  };
```

Recapitulando, primeiro implementamos uma fila através de uma lista indexada e as funções nativas da linguagem. Logo em seguida, fizemos a nossa própria implementação de fila que apresentava um tempo O(1) nas operações de enfilar e desenfilar, mas ainda era uma fila mutável. Depois fizemos uma alteração no nosso algoritmo de fila criando uma estrutura de dados imutável, mas chegamos ao tempo O(n) na operação `enfilar` pois usamos uma estratégia de copiar os dados de uma fila antes de enfilar o valor propriamente dito. Por fim, apresentamos uma implementação de fila através de duas pilhas que é imutável e tem uma operação de enfilar que executa em tempo amortizado O(1).

Apesar de chegarmos em uma fila imutável com as operações de enfilar e desenfilar em tempo O(1), veja que ganhamos uma complexidade a mais na implementação, pois agora utilizamos duas pilhas. Mesmo que a imutabilidade proteja o nosso código de erros causados

por mudanças de estado, podemos nos colocar em soluções mais complexas, como foi o caso de nossa fila. Além disso, temos toda uma cultura de desenvolvimento de algoritmos na qual, por vezes, as melhores implementações conhecidas utilizam estruturas de dados mutáveis. Portanto, entre um algoritmo mutável eficiente e um algoritmo imutável ineficiente, aderir ao primeiro é a melhor escolha dependendo do contexto.

Em um primeiro contato com o mundo do paradigma de programação funcional, podemos ficar aflitos ao buscar soluções imutáveis. Mas o universo desse paradigma não nos obriga a programar estritamente dessa forma: OCaml, por exemplo, é uma linguagem de programação funcional e apresenta mutabilidade. Na verdade, a ideia de aderirmos estritamente à imutabilidade está presente apenas em um subconjunto do universo da programação funcional.

Porém, isso não quer dizer que devamos também utilizar irrestritamente mutabilidade no dia a dia. Veja que um bom uso de mutabilidade é encapsular valores mutáveis em funções. Por exemplo, suponha que queremos calcular a média aritmética dos elementos de uma lista. Vamos criar a função `media`:

Código-fonte 40 – TypeScript: Função que calcula a média aritmética

```
1 const media = (lista: number[]) => {
2   const n = lista.length;
3   let soma = 0;
4   for (let elemento of lista) {
5     soma = soma + elemento
6   }
7
8   return soma / n;
9 }
```

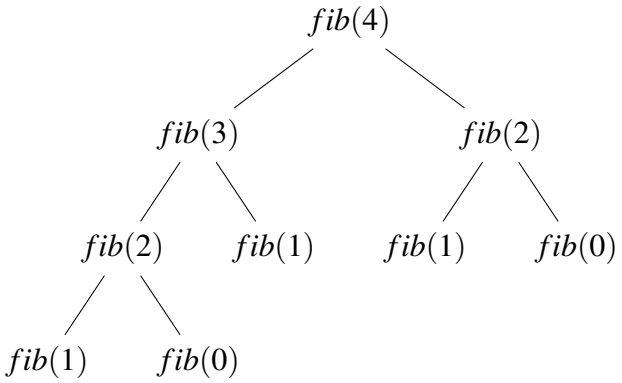
Na linha 3 atribuímos o valor 0 a variável `soma` e na linha 5 atualizamos esse valor fazendo `soma = soma + elemento` a cada iteração do `for` da linha 4 a 6, mas `soma` está encapsulada no contexto da função. Logo, essa mutabilidade será isolada para apenas esse contexto. Uma outra boa utilização é o uso de pré computações para tornar o código mais eficiente. Vamos supor que queremos calcular o n-ésimo número da série de Fibonacci. Vamos criar a função `fib`:

Código-fonte 41 – TypeScript: Função Fibonacci

```
1 const fib = (n: number) => n < 2 ? 1 : fib(n-1) + fib(n-2);
```

Veja que `fib` é uma função recursiva em que `fib(n)` é `fib(n-1) + fib(n-2)` caso `n >= 2`. Podemos visualizar a seguinte árvore de recursão a partir da execução de `fib(4)`:

Figura 2 – Árvore de recursão para fib(4)



Note que para calcular `fib(4)`, precisamos calcular `fib(3)` e `fib(2)`, mas para calcular `fib(3)` precisamos de `fib(2)` novamente. É um desperdício calcular `fib(2)` duas vezes. Portanto, vamos otimizar o cálculo de nossa função de Fibonacci:

Código-fonte 42 – TypeScript: Função Fibonacci

```
1  const fib = (m: number) => {
2    const cache: number[] = [];
3
4    const mem = (n: number) => {
5      if (cache[n]) {
6        return cache[n];
7      }
8
9      const resultado = n < 2 ? 1 : mem(n-1) + mem(n-2);
10     cache[n] = resultado;
11     return resultado;
12   };
13
```

```
14    return mem(m);
15 }
```

Na linha 2 do código acima, criamos uma lista `cache` vazia que será responsável por guardar os valores já calculados. Logo em seguida criamos a função `mem`. Nela, verificamos na linha 5 se n já foi calculado. Se verdadeiro, retornamos o valor já calculado e armazenado em `cache`. Caso contrário, calculamos na linha 9 o número de Fibonacci na n-ésima posição e na linha 10 guardamos o resultado em `cache[n]`. Por fim, na linha 11 retornamos o valor calculado. Logo em seguida, já na função `fib`, aplicamos `mem` a `m` e retornamos o resultado. O tempo de execução para `fib(48)` é $0,00073$ segundos para a versão memoizada, enquanto para a versão sem memoização, levamos $19,93$ segundos. As funções foram executadas no *runtime* Bun.js[2] em um computador com sistema operacional Ubuntu 24.04 LTS, processador *12th Gen Intel® Core™ i5-1235U* × *12*, 32 GB de memória RAM, 1 TB de SSD NVMe com leitura de 7000 MB/s e gravação de 6000 MB/s.

Figura 3 – Tempo de execução das duas versões da função de Fibonacci

```
❯ bun run fib.ts
7778742049
[0.73ms] Memoized Fibonacci
7778742049
[19.93s] Fibonacci
```

Na última implementação de `fib`, estamos alterando a lista `cache`, mas com a motivação de não recomputar `fib(j)` ao chamar `fib(n)` tal que $0 \leq j \leq n$. Utilizamos portanto um valor pré computado que esta guardado em `cache`. Na verdade, esse tipo de pré computação é conhecido como memoização, que é um caso específico de programação dinâmica (na seção 3.4, iremos explicar o conceito). Essa técnica utiliza um cache que é populado quando chamado uma função. Se a função é aplicada a um `n` e para esse argumento, temos o resultado em cache, então temos um *hit*. Caso contrário, temos um *miss*.

[2] https://bun.sh/

3.3 Listas encadeadas com reduce e map

Na seção 3.1 e 3.2 sobre pilhas e filas, falamos sobre a possibilidade de utilizar lista encadeada ou lista indexada para fazer a implementação de nosso tipos abstratos de dados pilha e fila. Então, no Código-fonte 29, através do tipo `Elemento` apresentado no Código-fonte 27, fizemos uma implementação de pilha com lista encadeada. Nessa seção vamos explorar no tipo lista encadeada algumas funções muito comuns em códigos que utilizam o paradigma funcional.

3.3.1 Implementando uma lista encadeada

Cada elemento de nossa lista será do tipo `Elemento` como apresentado no Código-fonte 27. Além disso, teremos o tipo `ListaEncadeada` com a propriedade `primeiro` que indica o primeiro elemento da lista e a propriedade `ultimo` que indica o último elemento da lista. Caso os valores das duas propriedades sejam `null`, a lista está vazia. Abaixo temos a representação desse tipo.

Código-fonte 43 – TypeScript: Tipo ListaEncadeada

```
type ListaEncadeada<T> = {
  primeiro: Elemento<T> | null;
  ultimo: Elemento<T> | null;
}
```

Então podemos implementar as funções `criar` que cria uma lista vazia, `estaVazia` que retorna se a lista esta vazia ou não e `adicionar` que cria uma nova lista com um novo elemento inserido no fim dela de forma que todos os outros valores são da lista passada nos argumentos.

Código-fonte 44 – TypeScript: funções criar, estaVazia e adicionar

```
const criar = <T>(): ListaEncadeada<T> => {
  return {
    primeiro: null,
    ultimo: null
  };
}
```

```
7
8  const estaVazia = <T>(lista: ListaEncadeada<T>) => {
9    return lista.primeiro === null && lista.ultimo === null;
10 }
11
12 const adicionar = <T>(lista: ListaEncadeada<T>, valor: T):
       ListaEncadeada<T> => {
13   const elemento: Elemento<T> = {
14     valor: valor,
15     proximo: null,
16   }
17
18   if (estaVazia(lista)) {
19     return {
20       primeiro: elemento,
21       ultimo: elemento
22     }
23   }
24
25   const ultimo = lista.ultimo as Elemento<T>;
26   ultimo.next = elemento;
27
28   return {
29     primeiro: lista.primeiro,
30     ultimo: elemento
31   }
32 }
```

Agora podemos implementar as funções `reduce` e `map`.

3.3.2 Função reduce

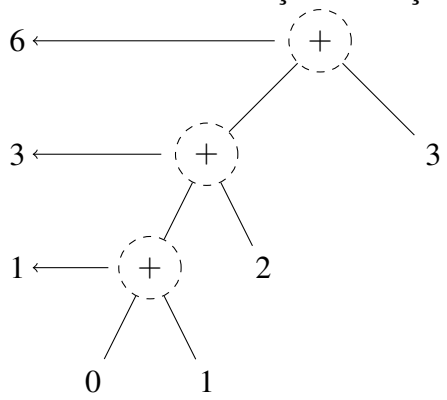
Suponha que temos a lista encadeada $1 \rightarrow 2 \rightarrow 3 \rightarrow nil$ e queremos somar seus elementos do início para o fim. Vamos começar por um valor inicial 0. Ao percorrer a lista encadeada, encontramos o primeiro valor 1 e somamos $0+1=1$. Esse valor será utilizado ao iterarmos sobre o próximo elemento da lista pois agora iremos somar $1+2=3$. Analogamente, ao iterarmos sobre o último valor que é 3, iremos somar $3+3=6$. Quando chegarmos em `nil`, temos nossa condição de parada e portanto retornamos o valor 6. Portanto, nossa função ficará como apresentado no Código-fonte 45. A figura 4 representa o passo a passo de nossa função.

Código-fonte 45 – TypeScript: Soma dos elementos de uma lista encadeada

```
1  const _soma = <T>(
2    elemento: Elemento<T> | null,
3    acumulador: number
4  ) => {
5    if (elemento === null) {
6      return acumulador;
7    }
8
9    return _soma(elemento.proximo, elemento.valor +
       acumulador);
10 }
11
12 const soma = <T>(lista: ListaEncadeada<T>) => {
13   return _soma(lista.primeiro, 0);
14 }
15
16 const lista = // lista encadeada 1, 2, 3
17 soma(lista); // 6
```

Observe que se quisermos agora multiplicar os elementos da mesma lista do início para o fim, precisamos alterar a linha 9 trocando o operador "+" por "$*$" e a linha 13 trocando o valor "0" por "1". O resultado obtido é o Código-fonte 46. A Figura 5 representa o passo a

Figura 4 – Árvore de execução da função soma



passo de nossa função.

Código-fonte 46 – TypeScript: Multiplicação dos elementos de uma lista encadeada

```
const _multiplicacao = <T>(
  elemento: Elemento<T> | null,
  acumulador: number
) => {
  if (elemento === null) {
    return acumulador;
  }

  return _multiplicacao(elemento.proximo, elemento.valor *
     acumulador);
}

const multiplicacao = <T>(lista: ListaEncadeada<T>) => {
  return _multiplicacao(lista.primeiro, 1);
}

const lista = // lista encadeada 1, 2, 3
multiplicacao(lista); // 6
```

Pode ser que desejemos transformar uma lista de números em um outro tipo. Por

Figura 5 – Árvore de execução da função multiplicação

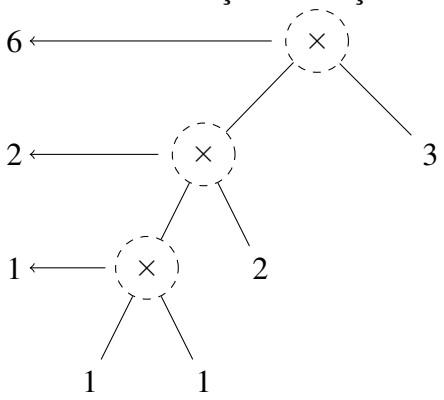
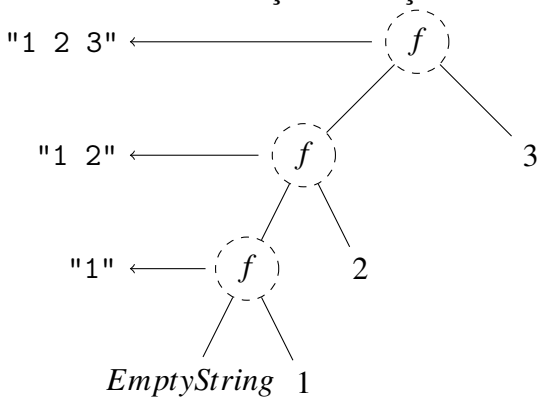


Figura 6 – Árvore de execução da função concatenação



exemplo, podemos levar uma lista de números para uma string como mostra a Figura 6. Ao analisar nossas operações de soma ou multiplicação como funções (de fato, algumas linguagens já fazem isso) e o valor inicial ser exposto através de um parâmetro, podemos criar uma função genérica que resolverá tanto a soma e multiplicação como outras situações que não operam necessariamente sobre uma lista de números ou retornam o mesmo tipo da lista de entrada.

Assim, inserimos a função `reduce` em nosso idioma. Ela receberá uma operação de redução e um valor representando o estado inicial de acumulação. Então, ela itera sobre a lista aplicando a função de redução ao acumulador atual e o valor do item atual. O resultado disso será um novo valor para o acumulador. Quando finalizada a iteração pela lista, o valor acumulado até então será retornado. Vale ressaltar que algumas linguagens nomeiam essa função como `fold`.

A operação de redução terá o tipo apresentado no Código-fonte 47. Veja que temos um tipo função — no *lambda calculus* simplesmente tipado, chamamos de tipo seta — em que o parâmetro `acumulador` tem o tipo parametrizado `A`, `itemAtual` tem o tipo parametrizado `C` e o retorno da função será o mesmo tipo `A` de `acumulador`, que em funções recursivas no TypeScript deve ser explicitado o tipo do retorno, como acontece na linha 5.

Código-fonte 47 – TypeScript: Tipo FuncaoReduce

```
type FuncaoReduce<A, C> = (acumulador: A, itemAtual: C) =>
  A
```

A função `reduce` pode ser implementada como podemos ver abaixo.

Código-fonte 48 – TypeScript: Função reduce auxiliar

```
const _reduce = <C, A>(
  elemento: Elemento<C> | null,
  funcao: FuncaoReduce<A, C>,
  acumulador: A
): A => {
  return elemento === null
    ? acumulador
    : _reduce(elemento.proximo, funcao, funcao(acumulador,
      elemento.valor));
}
```

Quando iteramos sobre uma lista encadeada e estamos no último elemento, temos a certeza que `elemento.proximo` será `null`. Dito isso, na linha 6, tratamos essa situação como a nossa condição de parada. Portanto, quando isso acontecer, `elemento` será `null` e a função retorna `acumulador`. Caso contrário, será realizada uma chamada recursiva para `_reduce`, passando o próximo elemento `elemento.proximo`, a operação de redução e um novo valor para `acumulador` que será o resultado da aplicação de `funcao` aos valores de `acumulador` e `elemento.valor`. Note que `funcao` é fornecido pelo usuário de `_reduce`, assim como seu retorno deve sempre ser o mesmo tipo de `acumulador` devido ao contrato estabelecido com o tipo `FuncaoReduce`. Isso permite que nossa função seja utilizada para resolver diferentes problemas que envolvam transformar uma lista em um outro valor qualquer válido da linguagem.

Fornecidas as ferramentas auxiliares, vamos implementar a função `reduce` para nossa lista encadeada:

Código-fonte 49 – TypeScript: Função reduce para listas encadeadas

```
1  const reduce = <C, A>(
2    lista: ListaEncadeada<C>,
3    fn: FuncaoReduce<A, C>,
4    acumulador: A
5  ) => {
6    return _reduce(lista.primeiro, fn, acumulador);
7  }
```

Implementamos a função `reduce` que recebe uma lista encadeada, uma operação de redução e um valor inicial para acumulador. Na linha 6, retornamos o resultado da chamada de `_reduce` que será o acumulador após iterar por toda a lista encadeada. Vale dizer que os valores acumulados serão orientados pela função `fn` que é passada pelo usuário de `reduce`.

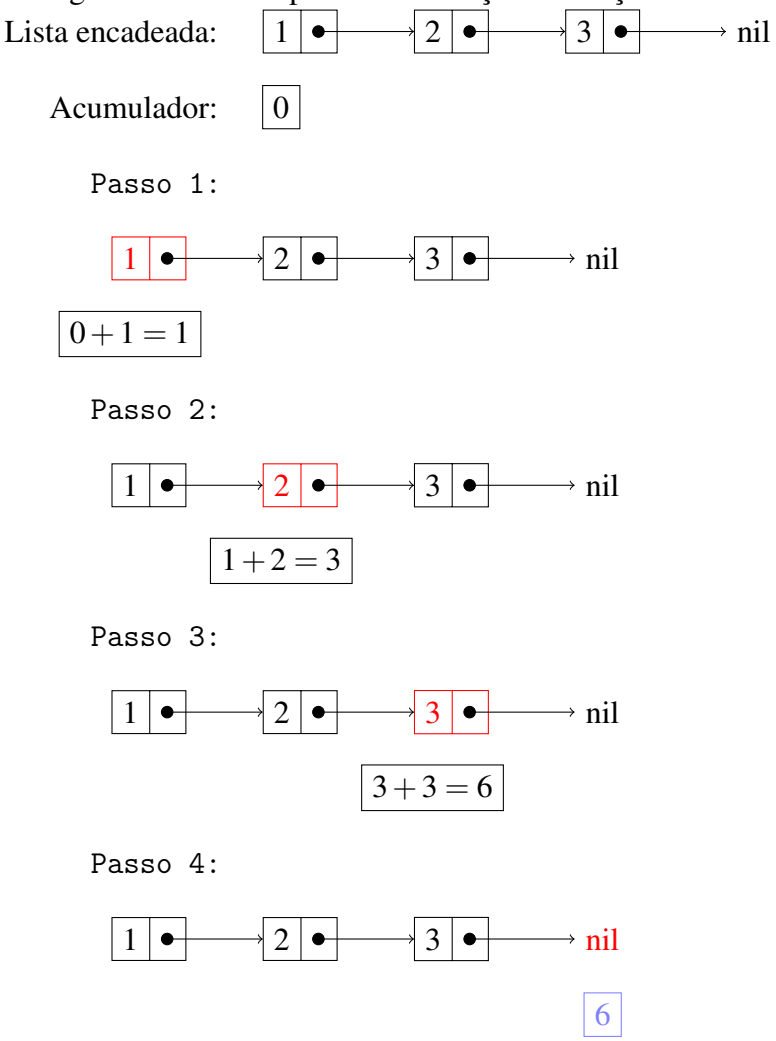
O Código-fonte 50 apresenta as funções `soma`, `multiplicação` e `concatenacao` utilizando a função `reduce` para computar seus respectivos objetivos. Na Figura 7, podemos ver o passo a passo da função `soma`.

Código-fonte 50 – TypeScript: Função de soma dos elementos de uma lista encadeada

```
1  const soma = (lista: ListaEncadeada<number>) =>
2    reduce(lista, (ac, item) => ac + item, 0)
3
4  const multiplicacao = (lista: ListaEncadeada<number>) =>
5    reduce(lista, (ac, item) => ac * item, 1);
6
7  const concatenacao = (lista: ListaEncadeada<string>) =>
8    reduce(lista, (ac, item) => {
9      return ac === "" ? `${item}` : `${ac} ${item}`;
10   }, "")
```

Vamos agora utilizar `reduce` para transformar uma lista em outra lista.

Figura 7 – Passo a passo da execução da função soma



3.3.3 Função map

Suponha que queremos ter uma lista encadeada resultante da multiplicação de cada elemento por 2. Por exemplo, se temos a sequência "1, 2, 3" queremos a sequência resultante "2, 4, 6". Podemos então implementar as duas funções abaixo.

Código-fonte 51 – TypeScript: Multiplicar cada elemento de uma lista por dois

```
1 const _multiplicarPorDois = (
2   elemento: Elemento<number> | null,
3   lista: ListaEncadeada<number>) => {
4   if (elemento === null) {
5     return lista;
6   }
```

```
7
8     const novoValor = elemento.valor * 2;
9     const novaLista = ListaEncadeada.adicionar(lista,
          novoValor);
10
11    return _multiplicarPorDois(elemento.proximo, novaLista);
12  }
13
14  const multiplicarPorDois = (lista: ListaEncadeada<number>)
        => {
15    const acumulador = ListaEncadeada.criar();
16    return _multiplicarPorDois(lista.primeiro, acumulador);
17  }
```

Mas ora, acabamos de implementar a função reduce. Vamos utilizá-la para simplificar essa implementação.

Código-fonte 52 – TypeScript: Multiplicar cada elemento de uma lista por dois utilizando reduce

```
1  const multiplicarPorDois = (lista: ListaEncadeada<number>)
       => {
2    const novaLista = ListaEncadeada.criar<number>();
3    return reduce(lista, (ac, item) => {
4      const novoValor = item * 2;
5      return ListaEncadeada.adicionar(ac, novoValor);
6    }, novaLista)
7  }
```

E se quisermos agora uma função que multiplica por três ao invés de dois? Ou se quisermos transformar o primeiro caractere de cada string de uma lista para maiúsculo? Se seguirmos os passos das implementações acima, teremos que criar uma função para cada problema definido. Ao invés disso, vamos implementar uma função que poderá ser reutilizada

em qualquer problema de transformar um elemento de uma lista em outro. Essa função será a `map`. Podemos ainda utilizar a função `reduce` para implementá-la. Além disso, iremos criar o tipo `FuncaoMap` e em nosso `map` receberemos uma função `funcao` do tipo `FuncaoMap`.

Código-fonte 53 – TypeScript: Função map

```
type FuncaoMap<T, U> = (current: T) => U;

const map = <T, U>(
  lista: ListaEncadeada<T>,
  funcao: FuncaoMap<T, U>
) => {
  return reduce(lista, (ac, item) => {
    const novoValor = funcao(item);
    return ListaEncadeada.adicionar(ac, novoValor);
  }, ListaEncadeada.criar<U>());
}
```

Assim, podemos utilizar o map para multiplicar todos os elementos de uma lista por dois, por três ou deixar todo primeiro caractere de cada string de uma lista em maiúsculo. Vejamos como fazer isso:

Código-fonte 54 – TypeScript: Exemplo de uso da função map

```
const lista = // sequencia 1, 2, 3

const listaMultiplicadaPor2 = map(lista, (v) => v * 2); //
   sequencia 2, 4, 6
const listaMultiplicadaPor3 = map(lista, (v) => v * 3); //
   sequencia 3, 6, 9

const nomes = // sequencia 'maria', 'pablo', 'phillipe'

```

```
const nomesMaiusculos = map(lista, (v) => {
  const primeiraLetra = v.at(0).toUpperCase();
  const restoDoNome = v.substring(1);

  return `${primeiraLetra}${restoDoNome}`
}); // sequencia 'Maria', 'Pablo', 'Phillipe'
```

3.3.4 Amizade entre paradigmas

As funções `map` e `reduce` são funções de alta ordem (verificar Definição 2). E se fosse possível utilizar elementos da programação funcional em conjunto com elementos da programação orientada a objetos? Vamos escrever nossa lista encadeada utilizando classe e as funções de alta ordem `map` e `reduce`.

Código-fonte 55 – TypeScript: Lista encadeada utilizando o paradigma orientado a objetos e o paradigma funcional

```
export class ListaEncadeada<T> {
  private _primeiro: Elemento<T> | null;
  private _ultimo: Elemento<T> | null;

  private constructor() {
    this._primeiro = null;
    this._ultimo = null;
  }

  estaVazia = () => {
    return this._primeiro === null && this._ultimo === null
        ;
  }

  adicionar = (valor: T) => {
    const elemento = {
```

```
16        valor,
17        proximo: null
18      }
19
20      if (this.estaVazia()) {
21        this._primeiro = elemento;
22        this._ultimo = elemento;
23      } else {
24        const ultimo = this._ultimo as Elemento<T>;
25        ultimo.proximo = elemento;
26        this._ultimo = elemento;
27      }
28      return this;
29    }
30
31    private _reduce = <C, A>(
32      elemento: Elemento<C> | null,
33      fn: FuncaoReduce<A, C>,
34      acumulador: A
35    ): A => {
36      return elemento === null
37        ? acumulador
38        : this._reduce(elemento.next, fn, fn(acumulador,
            elemento.value));
39    }
40
41    reduce = <A>(fn: FuncaoReduce<A, T>, acumulador: A): A =>
        {
42      return this._reduce(this._primeiro, fn, acumulador);
43    }
44
45    map = <U>(fn: FuncaoMap<T, U>) => {
```

```
46      let lista = new ListaEncadeada<U>();
47
48      return this.reduce((ac, item) => {
49        return ac.adicionar(fn(item));
50      }, lista);
51    }
52  }
```

Veja, temos classes, métodos, propriedades, mutação de propriedades ao mesmo tempo que alguns de nossos métodos utilizam recursão e são funções de alta ordem. Extraímos conceitos de diferentes paradigmas de forma a encontrar uma implementação elegante para nossas listas encadeadas. De fato, algumas linguagens vêm adotando uma abordagem multiparadigma: veja por exemplo, C++[3] (desde a versão 2011), Scala[4], Kotlin, JavaScript, TypeScript, Python e Dart.

Por fim, vamos visualizar como utilizamos nossa classe `ListaEncadeada`:

Código-fonte 56 – TypeScript: Utilizando a implementação de Lista Encadeada através de código multiparadigma

```
1  const lista = new ListaEncadeada<number>();
2
3  lista
4    .adicionar(1)
5    .adicionar(2)
6    .adicionar(3)
7
8  const total = lista.reduce((ac, v) => acc + v, 0) // 6
9  const listaMultiplicadaPor2 = lista.map((v) => v * 2); //
     2, 4, 6
10 const listaMultiplicadaPor3 = lista.map((v) => v * 3); //
     3, 6, 9
11
```

[3] https://learn.microsoft.com/pt-br/cpp/cpp/?view=msvc-170
[4] https://www.scala-lang.org/

```
12 const nomes = new ListaEncadeada<string>();
13
14 nomes
15   .adicionar('maria')
16   .adicionar('pablo')
17   .adicionar('phillipe');
18
19 const nomesMaiusculos = nomes.map((nome) => {
20   const primeiraLetra = nome.at(0).toUpperCase();
21   const restoDoNome = nome.substring(1);
22
23   return `${primeiraLetra}${restoDoNome}`;
24 }) // 'Maria', 'Pablo', 'Phillipe'
```

Em resumo, algumas linguagens conseguem extrair as melhores ferramentas do paradigma funcional e orientado a objeto para possibilitar a escrita de códigos modulares. Além disso, através das funções `map` e `reduce` que são funções de alta ordem, é possível criar linhas de códigos que poderão ser utilizada para resolver diferentes problemas computacionais de forma modular. Vamos em seguida explorar mais do paradigma funcional ao escrever o algoritmo de Interpolação de Newton.

3.4 Interpolação de Newton

Seja $f(x)$ a função que encontra os resultados da tabela 1. Queremos encontrar o valor de $f(0.4)$, mas não sabemos qual a função f. Vamos descobrir um polinômio que se aproxima da função f utilizando interpolação. Seja $x \in \mathbb{R}$ e P o polinômio interpolador da função f. O resultado de $P(x)$ será suficientemente próximo de $f(x)$.

Tabela 1 – Função f(x)

i	x	f(x)
0	0.0	1.008
1	0.2	1.064
2	0.3	1.125
3	0.5	1.343
4	0.6	1.512

Figura 8 – Cálculo de diferenças divididas

		1ª ordem	2ª ordem	3ª ordem
x_0	y_0			
		$[y_0,y_1]$		
x_1	y_1		$[y_0,y_1,y_2]$	
		$[y_1,y_2]$		$[y_0,y_1,y_2,y_3]$
x_2	y_2		$[y_1,y_2,y_3]$	
		$[y_2,y_3]$		
x_3	y_3			

Figura 9 – Cálculo de diferenças divididas

		1ª ordem	2ª ordem	3ª ordem
x_0	y_0			
		$\frac{y_1-y_0}{x_1-y_0}$		
x_1	y_1		$\frac{[y_1,y_2]-[y_0,y_1]}{x_2-x_0}$	
		$\frac{y_2-y_1}{x_2-x_1}$		$\frac{[y_1,y_2,y_3]-[y_0,y_1,y_2]}{x_3-x_0}$
x_2	y_2		$\frac{[y_2,y_3]-[y_1,y_2]}{x_3-x_1}$	
		$\frac{y_3-y_2}{x_3-x_2}$		
x_3	y_3			

Figura 10 – Cálculo de diferenças divididas

ordem	notação	fórmula
0	b_0	y_i
1	b_1	$\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$
2	b_2	$\frac{[y_i,y_{i+1}]-[y_{i-1},y_i]}{x_{i+1}-x_0}$
$\vdots$	$\vdots$	$\vdots$
n	b_n	$\frac{[y_0,\ldots,y_n]-[y_0,\ldots,y_{n-1}]}{x_n-x_0}$

Existem algumas formas de encontrar esse polinômio P. Para essa discussão, utilizaremos o polinômio interpolador de Newton que pode ser calculado pela fórmula 3.1.

$$P_n(x) = y_0 + (x-x_0)b_0 + (x-x_0)(x-x_1)b_1 + \ldots + (x-x_0)(x-x_1)\ldots(x-x_{n-1})b_n \tag{3.1}$$

Para descobrir a tabela de diferenças divididas (denotado na fórmula por b_i), vamos antes visualizar uma tabela para 4 pontos de (x_0,y_0) a (x_3,y_3) na Figura 8. Expandindo as notações com suas definições, chegamos ao resultado apresentado na Figura 9. Generalizando para n pontos, chegamos a Figura 10.

Veja que, na fórmula 3.1, temos um cálculo recursivo em que $(x-x_0)(x-x_1)$ usa o $(x-x_0)$ do termo anterior $(x-x_0)b_0$ e assim sucessivamente. No cálculo das diferenças divididas, temos também um cálculo recursivo como podemos ver na Figura 9 e Figura 10 que para calcular b_i precisamos dos valores do passo anterior. Dito isso, vamos desenvolver a

primeira versão de nosso algoritmo que irá receber um x e uma lista de pares $(x_i, f(x_i))$ para descobrirmos nosso polinômio. A mesma retornará o resultado para $f(x)$. Abaixo podemos ver nossa versão inicial.

Código-fonte 57 – TypeScript: Primeira versão do algoritmo de Interpolação de Newton

```
type Ponto = {
  x: number;
  y: number;
};

const polinomio = (x: number, pontos: Ponto[]) => {
  const tabela: number[][] = [];
  const n = pontos.length;

  tabela[0] = [];

  for (let j = 0; j < n; j+= 1) {
    tabela[0][j] = pontos[j].y;
  }

  for (let j = 1; j < n; j+=1) {
    tabela[j] = [];

    for (let i = 0; i < (n - j); i += 1) {
      tabela[j][i] = (tabela[j-1][i+1] - tabela[j - 1][i])
        / (pontos[j + i].x - pontos[i].x)
    }
  }

  let s = pontos[0].y;
  let m = 1;

```

```
27    for (let i = 1; i < n; i += 1) {
28      m = m * (x - pontos[i - 1].x);
29
30      s = s + (m * tabela[i][0]);
31    }
32
33    return s;
34  }
```

O algoritmo acima calcula a tabela de diferenças divididas nas linha 7-22. Logo em seguida, calcula $f(x)$ utilizando a fórmula 3.1 nas linhas 24-33. Por último, retorna o resultado. Calcular a tabela de diferenças divididas tem tempo de execução $O(n^2)$ e apesar do cálculo de $f(x)$ ter tempo de execução $O(n)$, nosso algoritmo ainda assim terá um tempo de execução quadrático.

Podemos extrair para uma função o nosso cálculo da tabela de diferenças divididas. Além disso, podemos transformar a função `polinomio` em uma função de alta ordem fazendo com que ela retorne uma função que recebe um `x` e retorna `f(x)`.

Código-fonte 58 – TypeScript: Segunda versão do algoritmo de Interpolação de Newton

```
1  type Ponto = {
2    x: number;
3    y: number;
4  };
5
6  const calcTabelaDiferencaDividida = (pontos: Ponto[]) => {
7    const tabela: number[][] = [];
8    const n = pontos.length;
9
10   tabela[0] = [];
11
12   for (let j = 0; j < n; j+= 1) {
13     tabela[0][j] = pontos[j].y;
```

```
14    }
15
16    for (let j = 1; j < n; j+=1) {
17      tabela[j] = [];
18
19      for (let i = 0; i < (n - j); i += 1) {
20        tabela[j][i] = (tabela[j-1][i+1] - tabela[j - 1][i])
              / (pontos[j + i].x - pontos[i].x)
21      }
22    }
23
24    return tabela;
25  }
26
27  const polinomio = (pontos: Ponto[]) => {
28    const tabela = calcTabelaDiferencaDividida(pontos);
29    const n = pontos.length;
30
31    return (x: number) => {
32      let s = pontos[0].y;
33      let m = 1;
34
35      for (let i = 1; i < n; i += 1) {
36        m = m * (x - pontos[i - 1].x);
37
38        s = s + (m * tabela[i][0]);
39      }
40
41      return s;
42    }
43  }
```

Agora quando chamamos a função `polinomio`, ao invés de retornar o valor de $f(x)$, retornamos uma função que recebe x e retorna o valor de $f(x)$. Assim, podemos usar essa função da seguinte forma:

Código-fonte 59 – TypeScript: Usando a função polinomio

```
const f = polinomio(pontos);

f(0.4); // Aproximadamente 1.216
```

Note que chamar `polinomio` custa $O(n^2)$. Enquanto qualquer chamada de `f` — função retornada por `polinomio` — custa $O(n)$. Através de funções, separamos as responsabilidades ao criar uma função para realizar todos os cálculos envolvidos para construção da tabela de diferenças divididas (`calcTabelaDiferencaDividida`) e outra função de alta ordem para chamar `calcTabelaDiferencaDividida` e retornar a função f que calcula o valor de $f(x)$. Por fim, descobrimos que $f(0.4)$ é 1.216.

Em uma abordagem mais funcional, vamos agora utilizar recursão para calcular nossa tabela de diferenças divididas.

Código-fonte 60 – TypeScript: Terceira versão do algoritmo de Interpolação de Newton

```
type Ponto = {
  x: number;
  y: number;
};

const calcTabelaDiferencaDividida = (
  pontos: Ponto[],
  tabelaAcumulada: number[][],
  i: number = 1
): number[][] => {
  if (i === pontos.length) {
    return tabelaAcumulada;
  }

```

```
15    const ordemAntigo = tabelaAcumulada[i - 1];
16    tabelaAcumulada[i] = R
17      .range(0, pontos.length - i)
18      .map((j) => {
19        const a = (ordemAntigo[j+1] - ordemAntigo[j]);
20        const b = (pontos[j+i].x - pontos[j].x)
21
22        return a / b;
23      })
24
25    return calcTabelaDiferencaDividida(pontos,
        tabelaAcumulada, i+1);
26  }
27
28  const polinomio = (pontos: Ponto[]) => {
29    const y0 = pontos.map((p) => p.y);
30    const tabela = calcTabelaDiferencaDividida(pontos, [y0]);
31
32    return (x: number) => {
33      const n = pontos.length;
34      let s = pontos[0].y;
35      let m = 1;
36
37      for (let i = 1; i < n; i += 1) {
38        m = m * (x - pontos[i - 1].x);
39        s = s + (m * tabela[i][0]);
40      }
41
42      return s;
43    }
44  }
```

Na função `calcTabelaDiferencaDividida` temos agora dois argumentos acumuladores `tabelaAcumulada` e `i`. A cada passo, guardamos os valores da tabela acumulada para ordem i. Na linha 25, incrementamos i em 1. Nossa condição de parada será quando i for igual ao tamanho de `pontos.length`. Veja que, se temos n pares (x_i, y_i), ou seja, `pontos.length` pontos, precisamos calcular a tabela de diferenças divididas até a ordem $n-1$. Então, quando i for igual a n, precisamos apenas retornar `tabelaAcumulada`.

Vale ressaltar que, através da programação dinâmica, pela abordagem do método de baixo para cima, criamos a lógica de nossa função `calcTabelaDiferencaDividida`. Nas linhas 16 a 23 do Código-fonte 60, nós usamos os valores calculados no passo anterior para calcular os valores da ordem i. Fazemos isso criando uma lista de tamanho `pontos.length - i`, pois, se estamos em uma ordem i, temos $n-i$ valores daquela ordem a serem calculados. Logo em seguida, criamos uma nova lista com os valores da ordem i através dos valores da ordem $i-1$ como podemos ver das linhas 18 a 23. Observe que após chamar `polinomio` e ela terminar seus passos de computação retornando a função da linha 32, a função retornada ainda terá acesso aos valores de `tabela`. Isso é possível devido a closures (verifique a subseção 2.1.4).

Note ainda que quando vamos calcular os elementos da tabela de ordem i, usamos os valores calculados anteriormente na ordem $i-1$. A programação dinâmica, de acordo com Cormen *et al.* (2012, p. 262), se aplica quando os subproblemas compartilham subsubproblemas. Um algoritmo de programação dinâmica resolve cada subsubproblema só uma vez e depois grava sua resposta em uma tabela, evitando assim, o trabalho de recalcular a resposta toda vez que resolver cada subsubproblema.

Portanto, através do uso de programação dinâmica, funções de alta ordem e closures, conseguimos implementar de maneira elegante um algoritmo de interpolação de Newton que, dado os pontos $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$, computa em tempo $O(n^2)$ o polinômio interpolador f, fornecido na forma de uma função que, recebendo qualquer x no intervalo $[x_0, x_n]$, calcula e retorna $f(x)$ em tempo $O(n)$.

3.5 Quicksort

O quicksort é um algoritmo de ordenação com tempo de execução de pior caso $O(n^2)$ e tempo de execução esperado de $O(n \lg n)$. O algoritmo aplica o paradigma de divisão e conquista. As três etapas podem ser definidas da seguinte forma (CORMEN *et al.*, 2012, p. 123):

1. Divisão: Particionar o arranjo $A[p...r]$ em dois subarranjos $A[p...q-1]$ e $A[q+1...r]$ em que o primeiro subarranjo tem elementos menores que $A[q]$, o segundo tem elementos maiores $A[q]$ (vale ressaltar que $A[q]$ ficará entre os dois arranjos delimitando à esquerda os menores elementos que ele e à direita os maiores que ele). Nesse passo, também calculamos o índice q.
2. Conquista: ordenar os dois subarranjos $A[p...q-1]$ e $A[q+1...r]$ por chamadas recursivas a `quicksort`.
3. Combinação: como os subarranjos já estão ordenados, não é necessário nenhum trabalho para combiná-los: o arranjo A[p...r] encontra-se ordenado.

Escolhemos esse algoritmo para apresentar que apesar de tornarmos a escrita mais legível com as técnicas da programação funcional, podemos produzir um algoritmo não otimizado ou falsamente otimizado, caso não tenhamos atenção ao custo correspondente a cada trecho do código.

Inicializemos nossa discussão utilizando uma função bastante comum em códigos que utilizam o paradigma funcional: `filter`. Essa função irá receber uma lista l e um predicado p. Então irá percorrer l e executar o predicado p para cada elemento da lista. Assim, criaremos uma nova lista com os elementos para os quais o predicado p retornou "`true`".

Código-fonte 61 – TypeScript: Exemplo de uso da função filter

```
1 const lista = [1, 2, 3, 4];
2
3 const listaFiltrada = lista
4   .filter((elemento) => elemento % 2 === 0); // [2, 4]
```

Agora vamos visualizar a implementação de nosso algoritmo quicksort utilizando a função filter para separar nosso arranjo em dois subarranjos como falamos no passo de divisão do algoritmo. Além disso, escolheremos um pivô de forma randomizada.

Código-fonte 62 – TypeScript: Quicksort utilizando concat e filter

```
1 const ordenar = (lista: number[]): number[] => {
2   if (lista.length < 2) {
3     return lista;
4   }
```

```
5
6     const pivot = lista[Random.random(0, lista.length - 1)];
7
8     return Array<number>()
9       .concat(ordenar(lista.filter((v) => v < pivot)))
10      .concat(lista.filter((v) => v === pivot))
11      .concat(ordenar(lista.filter((v) => v > pivot)))
12  }
```

Observe que a função de ordenação acima é curta e fácil de ler. Infelizmente, porém, ela também é particularmente lenta. O motivo é que cada operação de filtragem e de concatenação, além de realizar um percurso pela lista recebida, gera uma nova cópia dos elementos como resultado, fazendo com que o algoritmo realize muitas cópias e alocações de memória desnecessárias.

Visualizemos agora o algoritmo com melhor performance apresentado no livro de Cormen *et al.* (2012, p. 123-130) utilizando a linguagem de programação TypeScript.

Código-fonte 63 – TypeScript: Quicksort

```
1  const trocar = (lista: number[], i: number, j: number) => {
2    if (i === j) return;
3    const aux = lista[i];
4    lista[i] = lista[j];
5    lista[j] = aux;
6  }
7
8  const particao = (lista: number[], p: number, r: number) =>
       {
9    const x = lista[r];
10
11   let i = p - 1;
12   for (let j = p; j <= r-1; j += 1) {
13     if (lista[j] <= x) {
```

```
14        i = i + 1;
15
16        trocar(lista, i, j);
17      }
18    }
19
20    trocar(lista, i + 1, r);
21    return i + 1;
22  }
23
24  const ordenarRecursivo = (lista: number[], p: number, r:
      number) => {
25    if (p < r) {
26      const i = Random.random(p, r);
27      trocar(lista, r, i);
28
29      const q = particao(lista, p, r);
30      ordenarRecursivo(lista, p, q - 1);
31      ordenarRecursivo(lista, q + 1, r);
32    }
33  }
34
35  const ordenar = (lista: number[]) => {
36    ordenarRecursivo(lista, 0, lista.length - 1);
37
38    return lista;
39  }
40
41  export const Quicksort = {
42    ordenar
43  }
```

A implementação acima, apesar de ser mais rápida, altera a lista a ser ordenada e também é mais complexa de entender. Temos a função auxiliar `trocar`, que troca o elemento de uma posição `i` com o elemento da posição `j` na lista; a função `particao`, que a partir do pivô na posição `r`, coloca os elementos menores que `lista[r]` à esquerda e os maiores que `lista[r]` à direita; a função `ordenarRecursivo` que escolhe um pivô aleatoriamente e realiza chamadas recursivas para ordenar a lista. Por fim, temos a função `ordenar` que basicamente abstrai a escolha inicial do arranjo a ser ordenado, precisando receber apenas uma lista (começamos sempre pelo arranjo A[0...r] em que r é o tamanho da lista menos 1).

Note que podemos alterar a função `ordenar` para realizar uma cópia de `lista` antes de começar a ordená-la, como mostra o Código-fonte 64. Como essa mudança mantem a complexidade assintótica do tempo de execução, essa alteração não impacta significativamente a performance do algoritmo na prática.

Código-fonte 64 – TypeScript: Função ordenar com cópia

```
1  const ordenar = (lista: number[]) => {
2    const copiaDeLista = [];
3
4    for (const elemento of lista) {
5      copiaDeLista.push(elemento)
6    }
7
8    ordenarRecursivo(copiaDeLista, 0, copiaDeLista.length -
       1);
9    return copiaDeLista;
10 }
```

Os resultados para ordenação de uma lista com um milhão de elementos foram a execução do algoritmo não otimizado em 778,13 milissegundos, 166,08 milissegundos para o algoritmo otimizado e 180,80 para o algoritmo otimizado com cópia da lista de entrada. As funções foram executadas no *runtime* Bun.js[5] em um computador com sistema operacional Ubuntu 24.04 LTS, processador *12th Gen Intel® Core™ i5-1235U* de 10 núcleos, 32 GB de memória RAM, 1 TB de SSD NVMe com leitura de 7000 MB/s e gravação de 6000 MB/s.

[5] https://bun.sh/

Figura 11 – Tempo de execução do Quicksort

```
❯ bun run index.ts
[778.13ms] Slow Quicksort
[166.08ms] Fast Mutable Quicksort
[180.80ms] Fast Imutable Quicksort
```

Vale ressaltar que os resultados da Figura 11 ao serem replicados, podem não gerar resultados iguais devido aos recursos computacionais usados da máquina no momento de execução. De qualquer forma, replicar os resultados irá gerar um resultado bem menor para o quicksort com a implementação do Código-fonte 63.

4 CONSIDERAÇÕES FINAIS

No texto, foram definidos conceitos que edificam o paradigma funcional. Começando com a função como unidade elementar do paradigma, explicamos que elas podem ser funções puras ou impuras, de alta ordem ou primeira ordem, e avaliar de forma ansiosa ou preguiçosa. Por exemplo, OCaml é uma linguagem com avaliação ansiosa, enquanto em Haskell as funções avaliam preguiçosamente. Veja que LISP era uma linguagem com funções de primeira ordem, enquanto a maioria das linguagens modernas têm funções de alta ordem. Voltando a Haskell, trata-se de uma linguagem que faz parte do subconjunto de linguagens funcionais puras, e, portanto, nela, funções devem ser puras. Note que, algumas linguagens podem ter tipagem dinâmica, como é o caso de Elixir, enquanto outras podem ter uma tipagem estática, como é o caso de OCaml. Historicamente, a possibilidade de definir funções genéricas sobre listas, árvores e grafos de qualquer tipo de dado foi alcançado inicialmente pelas linguagens funcionais através do retardamento da checagem de tipos para o tempo de execução (tipagem dinâmica). Foi apenas com o sistema de tipos polimórficos de Milner, em 1978, que nossas linguagens funcionais começaram a adotar uma tipagem estática (TURNER, 2012).

Nós não defendemos uma visão exclusivista do paradigma funcional; mais especificamente, o texto não segue estritamente o subconjunto do paradigma funcional conhecido como programação funcional pura. Não contentes com uma ótica apenas dentro do paradigma funcional, mostramos vantagens em mesclá-lo com o paradigma orientado a objetos. Veja o caso de JavaScript, linguagem à qual, como se preocupa Resig e Bear (2013, p. 58), os programadores podem chegar com uma bagagem de linguagens não funcionais e, portanto, se frustrarem com a linguagem por usá-la apenas sob uma perspectiva orientada a objetos. Segundo eles, “A principal diferença entre escrever um código JavaScript mediano e escrevê-lo como um ninja JavaScript é o entendimento desta como uma linguagem funcional” (p. 57). Mais adiante, eles afirmam também: “Uma das razões pelas quais funções e conceitos funcionais são tão importantes em JavaScript é que a função é a unidade modular primária de execução” (p. 60). Diante disso, não é bom, na escrita de código JavaScript, utilizar apenas conceitos do paradigma orientado a objetos. Dessa forma, apresentamos uma situação em que faz sentido não apenas utilizar um paradigma ou outro, mas sim a junção dos dois.

Implementar uma linguagem multiparadigma não é um caso isolado de JavaScript. Kotlin, uma linguagem com sua primeira versão estável lançada em 2016[1], utiliza o mesmo

[1] https://pt.wikipedia.org/wiki/Kotlin

princípio de utilização de conceitos tanto do paradigma orientado a objetos como do paradigma funcional. O mesmo acontece para Java, que, a partir da versão 8, adiciona aspectos do paradigma funcional. Não paramos por aí e podemos ver que C++ (a partir da versão 2011), Scala e Dart (Primeira versão estável lançada em 2013[2]) fazem o mesmo. É fato que os recursos da programação funcional vêm sendo progressivamente reconhecidos e incorporados em linguagens tradicionalmente imperativas (procedurais ou orientadas a objeto) e em linguagens criadas recentemente.

Se existe um padrão de escrita de código para o paradigma orientado a objetos, o mesmo acontece para o paradigma funcional. Se no primeiro, uma classe define as propriedades e métodos de um objeto[3], o segundo definirá módulos com funções que realizam operações sobre um tipo de dados. Especificamente, nossos dados não guardarão métodos/funções. Apesar desse padrão baseado em módulos ser muito utilizado em linguagens funcionais, como é o caso de Elixir e OCaml, JavaScript é um exemplo de linguagem funcional em que é possível, mas não recomendável, utilizar esse padrão. Veja no Código-fonte 30 que para chamar funções consecutivamente, que precisam de outras funções e assim em diante, precisamos escrever composições grandes e difíceis de entender. Em contrapartida, em linguagens como OCaml e Elixir, temos o operador *pipe*, que torna conveniente realizar chamadas consecutivas de funções passando o retorno de uma como argumento para a próxima (veja o Código-fonte 31).

No capítulo 3, ao implementarmos algoritmos ensinados em um curso de Ciência da Computação, extraímos na implementação de pilhas um padrão de escrita de código presente no paradigma funcional como mencionamos no parágrafo acima. Já com o algoritmo de fila, concluímos que, a depender do caso, pode ser necessário optar por uma lógica mais complexa para atingir tempos de execução similares ao de utilizar estruturas mutáveis. Na implementação do tipo abstrato de dados (TAD) lista encadeada, ao utilizar funções de alta ordem, conseguimos alcançar uma estrutura de dados mais modular, como Hughes (1989) propôs como consequência da utilização desse conceito da programação funcional. Como continuação dessa implementação, concluímos também que podemos utilizar, ao nosso favor, a combinação do melhor de cada paradigma, como ilustramos na seção 3.3.4. Entretanto, os recursos da programação funcional devem ser utilizados de maneira consciente: se, por um lado, através de *closures*, conseguimos chegar em uma solução interessante para a implementação do algoritmo de Interpolação de

[2] https://pt.wikipedia.org/wiki/Dart_(linguagem_de_programaÃğÃčo)

[3] Subentendemos aqui a abordagem da orientação a objetos baseada no uso de classes, cientes de que existe também a abordagem baseada em protótipos, que inclusive é aquela adotada nas linguagens JavaScript e Lua.

Newton, por outro lado, para o algoritmo de ordenação quicksort, ao utilizar alguns conceitos do paradigma funcional (Código-fonte 62), obtivemos um tempo ruim quando comparado ao tempo da execução de uma implementação imperativa do algoritmo (Código-fonte 63). Isso ocorre devido à realização de muitas cópias e alocações de memória desnecessárias (veja os tempos de execução na Figura 11). Portanto, fica a lição de que, assim como no paradigma imperativo, é importante atentar ao custo das operações utilizadas no código. Logo, o texto introduz o leitor ao paradigma funcional, a algumas de suas definições principais e à sua aplicação na implementação de algoritmos e estruturas de dados. Dessa forma, o leitor talvez descubra que sua linguagem de programação imperativa favorita já possui muitas das funcionalidades aqui apresentadas, viabilizando utilização imediata.

REFERÊNCIAS

ATENCIO, L. **Functional programming in JavaScript**. Shelter Island, NY: Manning Publications Co., 2016. ISBN 9781617292828.

CHURCH, A. A set of postulates for the foundation of logic. **Annals of Mathematics**, [Annals of Mathematics, Trustees of Princeton University on Behalf of the Annals of Mathematics, Mathematics Department, Princeton University], v. 33, n. 2, p. 346–366, 1932. ISSN 0003486X, 19398980. Disponível em: http://www.jstor.org/stable/1968337.

CHURCH, A. An unsolvable problem of elementary number theory. **American Journal of Mathematics**, The Johns Hopkins University Press, v. 58, n. 2, p. 345–363, 1936. ISSN 00029327, 10806377. Disponível em: http://www.jstor.org/stable/2371045.

CHURCH, A. A formulation of the simple theory of types. **The Journal of Symbolic Logic**, Association for Symbolic Logic, v. 5, n. 2, p. 56–68, 1940. ISSN 00224812. Disponível em: http://www.jstor.org/stable/2266170.

CLARKSON, M. R. **OCaml Programmin: Correct + Efficient + Beautiful**. 2024.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST RONALD LAND STEIN, C. **Algoritmos - Teoria e Prática**. 3. ed. Rio de Janeiro, RJ: Elsevier Editora Ltda., 2012. ISBN 9788535236996.

HUDAK, P. Conception, evolution, and application of functional programming languages. **ACM Comput. Surv.**, Association for Computing Machinery, v. 21, n. 3, p. 359–411, 1989. ISSN 0360-0300.

HUGHES, J. Why functional programming matters. **The computer journal**, Oxford University Press, v. 32, n. 2, p. 98–107, 1989. ISSN 00104620.

LONSDORF, B.; BENKORT, M. **Professor Frisby's mostly adequate guide to functional programming**. 2017.

OKASAKI, C. **Purely Functional Data Structures**. Cambridge, UK: Cambridge University Press, 1998. ISBN 0521663504.

PETER, V. R.; HARIDI, S. **Concepts, techniques, and models of computer programming"**. London, England: MIT Press, 2004. (The MIT Press). ISBN 9780262220699.

RESIG, J.; BEAR, B. **Segredos do ninja JavaScript**. São Paulo, SP: Novatec Editora Ltda., 2013. ISBN 9788575223284.

TURNER, D. A. Some history of functional programming languages: (invited talk). In: LOIDL, H.-W.; PEÑA, R. (Ed.). **International Symposium on Trends in Functional Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 1–20. ISBN 9783642404474.