**Problema das mãos de poker da disciplina de paradigmas da programação.**

Ana Beatriz

Anderson Mateus

Marcos Eduardo

UFRPE – Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Campus Dois Irmãos (Sede) – Recife – PE - Brasil

**Abstract.** *This report involves the resolution of the "poker hand problem" and the various aspects related to its development.*

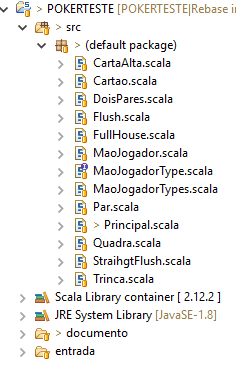
**Resumo.** *Esse relatório envolve a resolução do “problema das mãos de poker” e os vários aspectos relacionados ao seu desenvolvimento.*

1. **Informações preliminares**

Nesta etapa, iremos implementar o problema anterior utilizando o paradigma **Funcional**. Tendo como objetivo base, comparar o desempenho desta implementação, a anterior, discernindo os pontos em que ambas apresentam variação nos dados colhidos para o desempenho do código. A linguagem selecionada pelo grupo, foi **Scala**, devido à proximidade sintático-semântica a **Java**. O arquivo contendo o código fonte pode ser baixado através do link:

https://github.com/MattLack/PokerPrevisionSCALA

Para executar o código, basta importar o projeto para o Scala IDE (IDE baseada no Eclipse para desenvolver em Scala), e executar a classe **Principal**, que contém a função main(). Não esquecer de apontar na linha 7, o endereço do arquivo de testes no qual deseja-se utilizar.



A imagem acime descreve os diretórios do projeto. Dentro do diretório **src**, o pacote **(default package)**, contém o algoritmo implementado. O diretório **entrada** contém as entradas teste disponibilizadas pelo professor, sendo estas, de mil, um milhão de entradas e cem milhões de entradas (pokerK, pokerM, poker100M respectivamente). E o diretório **documento** contém alguns documentos referentes ao projeto.

1. **Como o problema foi resolvido**

Ao tratar o problema com Scala e Funcional, foi criada uma classe para cada tipo de jogada possível, onde as funções de reconhecimento das mãos foram aplicadas, exceto a lógica de jogo e a divisão de mãos, que por sua vez, são feitas na classe ***MaoJogador***. Deste modo, foram criadas as classes scala e as suas devidas funções, estas que são utilizadas dentro do algoritmo em ordem decrescente em nível de força, **Straight Flush, Quadra, Full House, Flush, Trinca, Dois pares, Par e Carta alta**, não tivemos tempo de implementar todas as jogadas, faltando as jogadas **Royal Flash e Sequencia** e também não foram implementadas funções de desempate.

Os métodos recebem o valor e o naipe. As listas “*h1 e h2*“ referem-se aos valores das cartas que a mão contém, estes valores ordenados dentro de **MaoJogador** para que possam ser transmitidos ás funções de validação, com a finalidade de facilitar na decisão da jogada.

Na classe ***Cartao*** é feita toda parte de divisão das cartas nas mãos dos jogadores, e implementado que as cartas **T, J, Q , K e A** são substituídas respectivamente para 10, 11, 12, 13 e 14 para conseguirmos estabelecer a força de uma sobre a outra, além de receber as listas como entrada nos métodos de teste.

Basicamente, é feita a leitura do arquivo na classe **Principal**, onde nela é dividida em duas listas, uma com a primeira e outra com a segunda mão, onde ambas são enviadas pela função MaoJogador, que retorna qual das duas listas ganhou o jogo, ao final, comparamos a resposta com as listas de entrada e obtemos a quantidade de vezes em que o jogador 1 ganha.

1. **Linguagem escolhida**

Visando uma proximidade as linguagens que dominamos, uma curva menor de aprendizagem e a aplicação possível ao paradigma Funcional, selecionamos a linguagem **Scala**, que é uma linguagem **multiparadigma**. Iremos a seguir apresentar algumas informações sobre a mesma.

1. **História da Linguagem**

Scala surgiu como objetivo para validar duas hipóteses, sendo a primeira: As linguagens de programação para componentes precisam ser escaláveis, onde os mesmos conceitos devem descrever partes pequenas e grandes, e a segunda hipótese: A escalabilidade pode ser conseguida unificando e generalizando conceitos de programação funcional e orientada a objetos.

Scala foi desenvolvida em 2001 por Martin Odersky e pelo grupo dele na École Polytechnique Fédérale de Lausanne (EPFL), Lausana na Suíça. A meta era combinar programação orientada a objetos e programação funcional, mas sem as restrições impostas pela linguagem Java. O primeiro passo foi o Funnel, uma linguagem minimalista baseada em redes funcionais.

Surgiu assim Scala, que trouxe algumas das ideias de Funnel, colocando dentro de uma linguagem mais pragmática com foco especial no funcionamento com plataformas padrões. Scala não é uma extensão de Java, mas é completamente inter operável com ele. O design do Scala começou em 2001. Um primeiro lançamento ao público foi em 2003. Em 2006, uma segunda versão remodelada foi lançada como Scala v 2.0. Desde então a linguagem tem ganhado popularidade. Atualmente a linguagem está na versão 2.12.3.

1. **Paradigmas Envolvidos**

A linguagem Scala é multiparadigma, podendo implementar os paradigmas orientação a objetos e funcional. Porém Scala é puramente orientada a objetos, no sentido em que todos os valores são objetos, e permite paradigma funcional no sentido de que cada função é um valor.

1. **Vantagens e Desvantagens**

Assim como toda linguagem possui suas vantagens e desvantagens, Scala não é diferente, as suas principais vantagens são: **Objetos imutáveis**, sendo um trunfo para reduzir a preocupação nas implementações thread-safe; **Velocidade na implementação**, pois Scala permite que o programador digite menos código para implementar o desejado; **Facilidade em resolver problemas de recorrência**, pois Scala possui uma biblioteca de “Ator”, específica para resolver problemas de concorrência mais rapidamente.

Já suas principais desvantagens são: **Ilegibilidade do código**, onde seu código em casos se trata apenas de poucas linhas que executam na verdade muitas coisas aninhadas; **Os programas podem ser menos eficiente**, por se tratar de um paradigma de abstração alta, a quantidade de coisas que rodam por trás do que foi implementado é imensa; **Difícil aprendizado, Se você já programa em Java, poderá ter dificuldades na adoção**, em função da abordagem diferenciada, tanto na sintaxe quanto no paradigma de programação que o Scala oferece.

1. **Possíveis Aplicações**

Scala é ideal para projetos funcionais, sistemas complexos de engenharia ou ambientes onde você precisa:

* Resolver problemas de concorrência;
* Executar tarefas DSL (Linguagem de Domínio-específico);
* Implementar soluções funcionais;
* Integrar com recursos compartilhados, arquivos, bancos de dados ou processos de back-end e serviços múltiplos;
* Aprender um novo paradigma de desenvolvimento;

1. **Relacionando o uso dos Aspectos Estudados**

**VARIÁVEIS E TIPOS**

Scala permite a tipagem de variáveis, mas podemos utilizar também declarações de variáveis genéricas que assumem o tipo do valor que os mesmos recebem. Assim como Java, no caso quando se declara o tipo, nem sempre precisamos inicializar (atribuir valores) estas variáveis, diferente da forma de declaração genérica. Os trechos de código abaixo exemplificam estas considerações.

Exemplo referente ao projeto anterior:

String hand1;

String hand2;

**int**[] valorHAND1 = **new** **int**[5];

Outro exemplo, referente a um dos métodos do projeto anterior:

**byte** cont = 0;

**int** value = 0;

No novo projeto temos os exemplos que foram forçados para apresentar exemplos de declaração de variável com tipos genéricos que podem ser vistos através do código abaixo, estes que podem ser por **val** ou **var**, explicaremos mais à frente a diferença de cada um.

**var** line = linha.next().split(" ").map(\_.trim).toList

**val** h1: Hand = Hand(line.slice(0,5).toList )

Como exemplo para declaração de tipos definidos, apresentamos um exemplo fictício da forma completa de declaração de variáveis.

private val name: String = "Diego"

[**escopo**] [**tipo da acesso**] [**nome da variável**]**:** [**tipo da variável**] = [**Valor padrão**]

Essa é a forma completa de declaração, porém não precisamos declarar sempre a forma completa, podemos omitir alguns parâmetros. Mas antes disso iremos entender cada parte da declaração completa.

* **Escopo**:

Certas variáveis tem um **escopo**, que nada mais é do que onde esta variável pode ser vista. A variável “name” tem escopo no bloco de execução da classe ou função em que a mesma é definida. Fora dessa classe ou função ela não existe. O escopo pode ser dos tipos **public**, **private**, **protected**. A variável “i” também possui um **tempo de vida**, ou seja, elas só existem enquanto a classe ou função em que ela faz parte estiver sendo executada.

* **Tipo de acesso:**

No **tipo de acesso** definimos se nossa variável é mutável ou imutável, ou seja, quando imutável, temos seu comportamento como uma constante.

**a) val**

Utilizado para deixar a variável imutável. Uma vez o valor definido, não poderá ser mais alterado.

**b) var**

Utilizado para deixar a variável mutável. Podemos definir e re-definir a qualquer momento o valor da variável.

* **Nome da variável:**

Analisando o trecho de código anterior, podemos observar detalhes interessantes. A primeira é que cada variável tem um **nome** e “name”, por exemplo, é o nome de uma variável.

* **Tipo da variável:**

Assim como em outras linguagens, o Scala possui todos os tipos objetos padrões: String, Int, Double, Boolean entre muitos outros, mas não implementa tipos primitivos. Quando o tipo da variável for omitido, o Scala irá atribuir um tipo baseando-se no valor que foi passado.

* **Valor padrão**

Por último, podemos informar algum valor padrão para nossa variável. Note que caso a variável seja declarada utilizando o **val**, esse será seu valor único, pois variáveis declaradas com val não podem ser alteradas.

**Declaração múltipla**

Podemos definir várias variáveis ao mesmo tempo e informando um valor padrão para elas também. Obs: Não utilizamos esta abordagem no projeto.

val variable1, variable2 = 100

**Lazy (variável preguiçosa)**

Um recurso bastante interessante é o lazy, quando informado na declaração de uma variável, faz com que o valor padrão somente seja executado no momento da primeira chamada a variável. Vamos ver como funciona:

Sem lazy

val numbers = for(i <- 1 to 10) yield i

for(e <- numbers) println(e)

Repare que no momento que declaramos a variável **numbers**, automaticamente foi criado um vetor com o retorno do nosso loop **for**. Agora iremos criar o mesmo exemplo utilizando o **lazy**.

Com lazy

lazy val numbers = for(i <- 1 to 10) yield i

for(e <- numbers) println(e)

Agora utilizando **lazy**, somente foi processado nosso loop **for** no momento em que chamamos a variável. Isso é bastante útil, pois pode evitar um processamento desnecessário. Obs: Não utilizamos esta abordagem no projeto.

**ESTRUTURAS DE CONTROLE E OPERADORES LÓGICOS**

Em Scala é possível utilizar estruturas de controle e operadores lógicos assim como em Java, porém com algumas peculiaridades. Não utilizamos no projeto algumas delas, devido à restrição do modelo de paradigma funcional a ser seguido. Porém, citaremos exemplos fictícios das mesmas.

scala> for (i <- 1 to 2; j <- 1 to 2) println(s"i = $i, j = $j")

Este loop permite a iteração de múltiplos contadores. Imagine que temos uma matriz, que no caso é representada por um array, poderíamos iterar todos os itens da matriz com uma única estrutura for.

while (x < 5) { println(x); x += 1}

A abordagem da estrutura while é semelhante a java, onde esta executa um bloco de código enquanto uma condição de teste é verdadeira.

Foram utilizadas algumas estruturas de controle no código. Exemplificaremos algumas:

**if**(linha.hasNext){

calcular(linha );

}

**else**{""}

Vemos a estrutura “if (testa condição)”. Ela executa o código que está contido nela apenas se a condição for verdadeira. A estrutura “else”, tem uso opcional, e irá tratar do que acontece caso o “if” não aconteça. Reparar que podem ser utilizados vários “if” dentro do código, de forma encadeada.

**val** q: *List*[Int] = matches **match** {

**case** **true** => m.pares.collect { **case** (a,b) => b.head.value }.toList

**case** **false** => 0 :: **Nil**

}

Também fazemos uso da estrutura match, que representa o case na linguagem scala, que vai funcionar como um interruptor, pois dependendo da entrada que você der a ele, ele vai acionar somente certo(s) comando(s) dentre os que você disponibilizou.

**TESTES COMPARATIVOS DE PERFORMARCE**

Agora testaremos o código implementado e o desempenho do mesmo em relação.

Para definir o arquivo de entrada, você deve seguir para a classe **POKERPREVISION** e dirigir-se ao método **MAIN**, neste por sua vez, você identificará uma chamada ao método ***PokerHands***, através dele, você deve dizer o diretório e o arquivo a ser lido.

Atualmente dentro do diretório entradas, que está dentro do projeto, temos os arquivos de mil valores e um milhão de valores, para usá-los em seus testes, basta apenas trocar o último caractere referente ao nome do arquivo, sendo K para o de mil e M para o de um milhão de entradas.

A máquina de testes é um notebook Asus, processador core i7 – 4720HQ 2.6GHz – com turbo bust para 3.6GHz, contendo 4 núcleos físicos e mais 4 virtuais, com 24Gb de memória RAM, rodando Windows 10 e Java 8.

As tabelas a seguir apresentam os resultados colhidos com mil, um milhão e cem milhões de entradas respectivamente, para a melhor versão do código java e a versão do código em scala. São colhidos três momentos (tempos) em escala de segundos, para cada versão e arquivo de número de entradas e, ao final, é apresentado uma média destes tempos, para servir melhor como medida de comparação.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada K | Tempo1 /s | Tempo2 /s | Tempo3 /s | Média Final /s |
| Scala | 0.534 | 0.531 | 0.524 | 0.529 |
| Java | 0.016 | 0.015 | 0.015 | 0.0153 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada 1M | Tempo1 /s | Tempo2 /s | Tempo3 /s | Média Final /s |
| Scala | 69.0 | 82.0 | 69.0 | 73.3 |
| Java | 0.484 | 0.484 | 0.516 | 0.495 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada 100M | Tempo1 /s | Tempo2 /s | Tempo3 /s | Média Final /s |
| Scala | +3h | x | x | x |
| Java | 40.057 | 40.263 | 40.902 | 40.407 |

Não foi possível rodar a entrada de cem milhões, devido ao grande tempo de computação para obter a resposta do algoritmo, o mesmo passou três horas computando sem apresentar retorno. Considerando as médias obtidas e relacionando com as médias da versão java do projeto, observa-se que não a utilização da linguagem scala acarreta em um menor desempenho para a solução do problema, isto não somente pelo uso da linguagem, mas pela aplicação do paradigma funcional para a solução do problema, demonstrando que, para tratar grandes massas de dados, como no caso do arquivo de cem milhões, usar Scala é inviável.

1. **Referências**
2. IME USP. Uma introdução a linguagem scala. Disponível em:  
   <http://ccsl.ime.usp.br/pt-br/uma-introducao-a-linguagem-scala>. Acesso em: 30 de agost. 2017.
3. SCALA. Scala Documentation. Disponível em:  
   < http://docs.scala-lang.org/?pageId=294934>. Acesso em: 29 de agost. 2017.
4. SCALA. Official Webpage. Disponível em:  
   < http://www.scala-lang.org/>. Acesso em: 30 de agost. 2017.
5. W3ii. Scala Tipos de Dados. Disponível em:  
   < http://www.w3ii.com/pt/scala/scala\_data\_types.html>Acesso em: 28 de agost. 2017.
6. LAMPx. The Scala Experiment. Disponível em:  
   < http://www.w3ii.com/pt/scala/scala\_data\_types.html>Acesso em: 25 de agost. 2017.
7. TRILHANDO SCALA. Scala, os prós e contras. Disponível em:  
   <https://trilhandoscala.wordpress.com/2012/10/24/scala-os-pros-e-os-contras/>Acesso em: 17 de agost. 2017.
8. IMERSÃO COM DIVERSÃO. Paradigma Funcional. Disponível em:  
   <http://informacaocomdiversao.blogspot.com.br/2009/02/paradigma-funcional.html>Acesso em: 14 de agost. 2017.
9. ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. **Programming in scala**. Artima Inc, 2008.
10. FELIPE.O.GUTIERREZ. Trabalhando com estruturas de controle com Scala. Disponível em:  
    < https://sites.google.com/site/felipeogutierrez/Home/diaadia/control-structure-scala>Acesso em: 1 de setemb. 2017.
11. SCALA. Scalacheat. Disponível em:  
    < https://docs.scala-lang.org/pt-br/cheatsheets/index.html >Acesso em: 25 de agost. 2017.