UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VICENTE ALBERTO MARTINS MERLO

Chronos: uma ferramenta de profiling integrada à suíte de testes para Elixir

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Alvaro Freitas Moreira

Porto Alegre
2021

AGRADECIMENTOS

Agradeço imensamente ao meu orientador pela paciência e apreço durante o desenvolvimento deste trabalho, e a todos os professores do Instituto de Informática pelos ensinamentos durante a graduação.

RESUMO

Erros em programas de computador causam grandes prejuízos econômicos e sociais. Uma das principais estratégias para a prevenção de erros são os testes unitários, que testam programas em nível de implementação e asseguram que um comportamento se manteve de acordo com o especificado pelos programadores. Para a categoria dos erros causados pelo uso indesejado de recursos computacionais pelo programa, é difícil a escrita de testes unitários, pois normalmente são necessárias integrações com ferramentas de *profiling*, muitas vezes externas à suíte de testes. Neste trabalho, implementamos uma biblioteca chamada Chronos que integra um *profiler* à suíte de testes da linguagem Elixir. Avaliamos, com base em três experimentos, a viabilidade da biblioteca implementada, demonstrando a relevância do Chronos, levando em consideração o tempo de execução e o *overhead* de tamanho dos programas analisados, como mais uma ferramenta no arsenal de programadores para aumentar a confiança no software.

Palavras-chave: Elixir. Profiling.

Chronos: a profiling tool integrated to the test suite for Elixir

ABSTRACT

Malfunctions in computer programs cause huge economical and social damages. One of the main strategies for the prevention of malfunctions are unit tests, which test program at implementation level and make sure that a behavior is kept according to what was specified by the programmer. The writing of unit tests is difficult with respect to the categories of malfunctions caused by the unwanted use of computer resources by the program, since usually the integration of a profiling tool is necessary — many times external to the test suite. In this work, we implemented a library called Chronos, which integrates a profiler to the test suite of the Elixir language. We evaluated, based upon three experiments, the viability of the implemented library, which demonstrates the relevance, with respect to execution time and bytecode size overhead, of Chronos as another tool in the programmer's toolbox to increase software reliability.

Keywords: Elixir. Profiling.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BEAM	Bogdan/Björn's Erlang Abstract Machine
EVM	Erlang Virtual Machine
AST	Árvore de Sintaxe Abstrata
PID	Process Identifier

SUMÁRIO

1 INTRODUÇÃO

Todos os anos, erros de software causam grandes prejuízos econômicos e sociais (TRICENTIS, 2017). Entre as estratégias mais utilizadas para elevar a qualidade de projetos de software e diminuir o número de falhas, estão os testes unitários (AMMANN; OFFUTT, 2008). Testes unitários são casos de testes que têm a finalidade de avaliar a implementação das unidades do programa (AMMANN; OFFUTT, 2008) — em conjunto com os demais testes, compõem a suíte de testes.

Uma categoria perigosa de erros é a causada pelo uso indesejado de recursos. Esta categoria é caracterizada pelo consumo excessivo de um ou mais recursos computacionais (AMMANN; OFFUTT, 2008), sejam eles a memória RAM, número de instruções, requisições HTTP, chamadas de funções ou qualquer outro cujo uso em demasia causa efeitos indesejáveis.

Entre as estratégias para a detecção do consumo excessivo de recursos estão as ferramentas de *profiling*, consideradas como um dos métodos mais úteis para a detecção de problemas de desempenho (LEACH, 2020). Também chamadas de *profilers*, as ferramentas de *profiling* são programas capazes de analisar a quantidade de recursos utilizados pela execução de um programa (GROTKER, 2012): o número de funções executadas, a quantidade de memória utilizada, e a quantidade de instruções de processador executadas são alguns dos recursos normalmente examinados por essas ferramentas.

A fim de detectar e prevenir os erros causados pelo uso indesejado de recursos, é interessante adicionar à suíte de testes do programa casos de teste com asserções sobre o uso de recursos do programa ou de suas unidades. Porém, fazer verificações sobre os recursos utilizados a partir da suíte de testes pode ser uma tarefa difícil: como *profilers* são programas externos ao programa testado, não são facilmente alcançáveis pela suíte de testes, necessitando de integrações complexas ou alterações de código para sua integração. Por exemplo, utilizar um *profiler* externo à máquina virtual de uma linguagem requer chamadas ao sistema operacional para a execução de um processo — que é uma integração complexa para programadores interessados em apenas testar seus programas.

Elixir é uma linguagem funcional, criada em 2013 (ELIXIR, 2021b) e compilável para a máquina virtual de Erlang (ELIXIR, 2021b). Ela vêm ganhando adeptos na indústria de software (ERLANG SOLUTIONS, 2020), sendo utilizada com sucesso em várias áreas: pipelines de dados, processamento multimídia, software embarcado, websites (ELIXIR, 2021b). Aproveitando-se da máquina virtual de Erlang para prover funcio-

nalidades de tolerância a falhas e programação distribuída (SASA, 2019), Elixir é focada em concorrência e escalabilidade, favorecendo a escrita de código sem efeitos colaterais e facilmente paralelizável graças às suas primitivas de comunicação (ERLANG, 2021b).

Embutidas na máquina virtual de Erlang, estão duas bibliotecas que são as ferramentas padrão para o *profiling* de programas escritos em Elixir: *lcnt* (ERLANG, 2021e) e *cprof* (ERLANG, 2021c). Ambas analisam o número de execuções de funções de um programa utilizando mecanismos internos da máquina virtual, sendo de difícil integração à suíte de testes por utilizarem *flags* de *debugging* que interferem com os testes.

Este trabalho tem como objetivo implementar uma alternativa de *profiling* de programas integrada à suíte de testes, possibilitando que testes unitários com verificações sobre o consumo de recursos sejam escritos de maneira transparente, sem fardos extras aos programadores. A implementação resultante é um *profiler* chamado Chronos, de código aberto e disponível no gerenciador de pacotes da linguagem, integrado à suíte de testes e que se assemelha às bibliotecas *lcnt* e *cprof* por também realizar o *profiling* do número de execuções de funções. O Chronos injeta contadores nos pontos de entrada das funções do programa durante o processo de compilação da linguagem, realizando o *profiling* por instrumentação.

Como método de experimentação da viabilidade do Chronos, foram criados e executados testes unitários de *profiling* em três bibliotecas populares da linguagem de diferentes tamanhos e de diferentes domínios. Os resultados obtidos mostram a relevância do Chronos como mais uma ferramenta no arsenal dos programadores para aumentar a confiança nos seus programas.
Este trabalho está organizado da seguinte maneira:

- No Capítulo 2, são introduzidos os conceitos necessários para a compreensão do trabalho.
- No Capítulo 3, apresenta-se a arquitetura do Chronos com o auxílio de um programa de exemplo.
- No Capítulo 4, são listados os experimentos realizados para avaliar o Chronos, assim como a análise dos resultados obtidos.
- No Capítulo 5, é apresentada a conclusão obtida a partir dos experimentos realizados.

2 REFERENCIAL TEÓRICO

Este capítulo começa com uma apresentação da linguagem Elixir, com destaque para as construções e características da linguagem mais relevantes para a compreensão deste trabalho. A seguir, são discutidas as categorias de *profilers* e as estratégias de *profiling* para programas escritos em Elixir. O capítulo termina com uma breve revisão sobre testes e sobre a escrita de testes unitários em Elixir.

2.1 Linguagem Elixir

Elixir é uma linguagem funcional de código aberto criada em 2012, compilável para a máquina virtual de Erlang, que provê à linguagem características de escalabilidade e tolerância a falhas (ELIXIR, 2021b). Surgiu como uma alternativa à linguagem de programação Erlang, tendo como foco o código mais limpo e uma experiência de aprendizagem mais fácil (SASA, 2019). Elixir oferece bibliotecas e ferramentas padronizadas, reduzindo a quantidade de código necessária para o desenvolvimento de software.

Apesar de ser uma linguagem à parte de Erlang, todas as construções e estruturas de dados de Erlang também estão presentes em Elixir — sendo possível executar em Elixir qualquer biblioteca escrita em Erlang (SASA, 2019).

Nos últimos anos, Elixir vem ganhando popularidade na indústria de software, sendo utilizada por empresas líderes de seus segmentos em produtos de alta demanda como Discord e Pinterest (ERLANG SOLUTIONS, 2020). No ano de 2021, Elixir foi incluída no ranking TIOBE (TIOBE, 2021), que lista as linguagens de programação mais populares no mundo.

O objetivo desta seção é introduzir a linguagem Elixir ao leitor, começando por exemplos de projetos populares, apresentando a sintaxe da linguagem, descrevendo a imutabilidade e o modelo de atores, e então mostrando a árvore de sintaxe abstrata da linguagem para a compreensão do processo de instrumentação feito pelo Chronos.

2.1.1 Projetos populares

A linguagem possui um gerenciador de bibliotecas para lidar com dependências de projetos chamado *hex.pm*. Entre as bibliotecas mais populares no *hex.pm* estão:

- *ExUnit* (HEXDOCS, 2021b), com mais de 10 milhões de downloads, é a ferramenta padrão para a execução de testes unitários de programas escritos na linguagem;

- *Phoenix* (HEXDOCS, 2021d), com mais de 7 milhões de downloads, é um *framework* para a web muito parecido com o *Rails* (RUBY ON RAILS, 2021), sendo utilizado em muitos projetos e na indústria como o *framework* web padrão para Elixir;

- *Ecto* (HEXDOCS, 2021a), com mais de 8 milhões de downloads, é uma biblioteca para interação com banco de dados utilizada pelo *Phoenix* e por outras bibliotecas populares;

- *ExDoc* (HEXPM, 2021b), com mais de 7 milhões de downloads, é uma biblioteca utilizada na documentação de projetos em Elixir.

2.1.2 *Mix*

O *Mix* é uma ferramenta embutida em Elixir que provê rotinas para criar, compilar, testar, e gerenciar projetos escritos na linguagem (ELIXIR, 2021c). O *Mix* utiliza o arquivo *mix.exs* como fonte das configurações dos projetos — nele, estão especificadas dependências, a versão do Elixir que o projeto espera, a versão do projeto, entre outras. Algumas rotinas úteis do *Mix* são:

- *mix deps.get*: faz o download das dependências do projeto, definidas no arquivo de configuração.

- *mix compile*: compila os arquivos da pasta *lib* do projeto.

- *mix test*: utilizada para rodar a suíte de testes do projeto. Além de executar a rotina *mix compile*, também compila todos os módulos da pasta *test* do projeto e executa todos os módulos de teste utilizando o *ExUnit*.

Todas as rotinas do *Mix* devem ser executadas pelo programador no terminal, no mesmo diretório em que o projeto se encontra.

2.1.3 Tipos primitivos

Os tipos primitivos da linguagem são inteiros, floats, booleanos, átomos, strings, listas, tuplas, mapas, e keywords lists (ELIXIR, 2021a).

2.1.3.1 Inteiros

Código 2.1: Exemplos de Inteiros

```
1 32200
2 32_200
3 0b0110
4 0o644
5 0x1F
```

Além da notação decimal, Elixir possui suporte a notação binária, octal, e hexadecimal, como mostram as linhas 3 até 6 do Código 2.1.

2.1.3.2 Floats

Código 2.2: Exemplos de Floats

```
1 3.141592
2 3.0
3 2.0e-10
```

Como mostra o Código 2.2, Elixir possui suporte à notação de expoentes (linha 3). Floats em Elixir possuem dupla precisão de 64 bits (ELIXIR SCHOOL, 2021a).

2.1.3.3 Booleanos

Os valores *true* e *false* são os booleanos de Elixir. Qualquer valor dentro de uma condicional que não seja *false* ou *nil* será avaliado como verdadeiro (ELIXIR SCHOOL, 2021b).

2.1.3.4 Átomos

Código 2.3: Exemplos de Átomos

```
1 :chronos
2 Chronos
```

De maneira parecida com os literais de Scheme e com símbolos de Ruby, Elixir possui um tipo especial para guardar constante: o átomo. No Código 2.3, estão duas maneiras de declarar átomos na linguagem: através do prefixo *:* (linha 1); através da letra maiúscula na letra inicial do átomo (linha 2).

2.1.3.5 Strings

Código 2.4: Exemplos de Strings

```
1 "Chronos"
2 "Chronos" <> " interpolado"
3 variavel = 2
4 "Chronos #{variavel} interpolada"
```

Elixir conta com suporte a strings codificadas em UTF-8 (ELIXIR SCHOOL, 2021c). Como mostra o Código 2.4, strings são envoltas por aspas duplas, e podem ser interpoladas com variáveis ou outras strings através dos operadores <> (linha 2) e *#{}* (linha 4).

A string resultante da interpolação na linha 2 do Código 2.4 é *"Chronos interpolado"*, e a string resultante da interpolação da linha 4 é *"Chronos 2 interpolada"*.

2.1.3.6 Listas

Código 2.5: Exemplos de Listas

```
1 [1, 2, "two", 2, 2, 3, 4]
2 ["Chronos", "profiler", "ferramenta"]
3 ["primeiro_elemento" | [2, 3]]
```

Listas em Elixir podem possuir elementos de diferentes tipos, como mostra o Código 2.5. Listas também podem ser definidas na notação *head* e *tail*, como mostra a linha 3 do Código 2.5.

Os módulos que provêm funções para interação com listas são o *List* e o *Enum*. Mais detalhes sobre eles estão disponíveis na documentação da linguagem no HexDocs, em <https://hexdocs.pm/elixir/1.12/List.html> e <https://hexdocs.pm/elixir/1.12/Enum.html>.

2.1.3.7 Mapas

Código 2.6: Exemplos de Mapas

```
1 %{"projeto" => "Chronos", "linguagem" => :elixir}
```

Assim como em outras linguagens, mapas são estruturas de dados que ligam uma chave a um valor. Em Elixir, a chave de um mapa é necessariamente única, não podendo um mesmo mapa ter uma chave repetida. A biblioteca *Map* provê funções para a interação com mapas.

Como mostra o Código 2.6, um mapa é envolto por *%{}*. A ligação entre uma chave e um valor se dá através do símbolo =>.

2.1.3.8 Tuplas

Código 2.7: Exemplos de Tuplas

```
1 {:ok, 1}
2 {"Chronos", "profiler", "ferramenta"}
```

Tuplas são estruturas similares a listas, normalmente utilizadas como retornos de funções. O Código 2.7 mostra a definição de duas tuplas, que são envoltas por *{}*.

2.1.3.9 Keyword lists

Código 2.8: Exemplos de *Keyword Lists*

```
1 [projeto: "Chronos", linguagem: "Elixir"]
2 # Equivalente a lista:
3 # [{:projeto, "Chronos"}, {:linguagem, "Elixir"}]
```

Keyword lists são tipos especiais de listas na forma *[atomo_1: valor, atomo_2: valor, ...]*, cujos elementos são tuplas que ligam átomos a valores. As *keywords lists* são especialmente interessantes pois seus elementos podem ser acessados da mesma maneira

que um mapa, porém a inserção de um novo elemento em se dá da mesma forma que em uma lista (ELIXIR SCHOOL, 2021d) — de maneira mais eficiente do um mapa.

No Código 2.8, na linha 3 está uma *keyword list* equivalente à *keyword list* da linha 1 — porém, sem a aplicação do açúcar sintático que transforma listas da forma *[{:atomo_1, valor}, {:atomo_2, valor}, ...]* para a forma *[[atomo_1: valor, atomo_2: valor, ...]*.

2.1.4 Funções

A definição de uma função é composta por um nome, pelos parâmetros de entrada e um único ponto de saída — sem a presença de operadores de pulos como *return*, *break*, ou *goto*.

Código 2.9: Definição de uma função

```
1 def saudar(nome, idade) do
2   nova_idade = idade + 1
3   IO.puts("Olá, #{nome}! Você terá #{nova_idade} anos.")
4   true
5 end
```

O Código 2.9 mostra a definição da função *saudar*, que aceita dois argumentos: *nome* e *idade*. O resultado da execução da função será sempre *true*, que é a última expressão do corpo da função, na linha 4.

Por convenção, funções são referidas como *Modulo.seu_nome/aridade*. Ao referenciar uma função, é importante especificar sua aridade, pois podem existir múltiplas funções com o mesmo nome em um módulo — desde que tenham aridades diferentes.

Código 2.10: Módulo definindo múltiplas funções de mesmo nome

```
1 defmodule MultiplasDefinicoes do
2   def saudar(nome, cidade) do
3     IO.puts("Olá, #{nome}! Como está #{cidade}?")
4   end
5
6   def saudar(nome, cidade, estado) do
7     IO.puts("Olá, #{nome}! Como está #{cidade} em #{estado}?")
8   end
9 end
```

Como mostra o Código 2.10, referir-se a *MultiplasDefinicoes.saudar* sem uma aridade levantaria a dúvida sobre qual função está sendo referida — porém, referir-se a *MultiplasDefinicioes.saudar/3* deixa explícito que a função referida é a definida na linha 6.

Código 2.11: Função definindo duas funções anônimas

```
1 def executar do
2   saudar = fn nome ->  IO.puts("Olá, #{nome}!") end
3   executar = fn nome, funcao -> funcao.(nome) end
4   executar("Chronos", saudar)
5 end
```

Elixir tem suporte a funções anônimas, tratadas como cidadãs de primeira classe (ELIXIR SCHOOL, 2021f). O Código 2.11 mostra a definição de uma função que define duas funções anônimas (linhas 2 e 3), e as executa (linha 4). Ressalta-se que as funções anônimas podem receber um número arbitrário de argumentos: a da linha 1 recebe apenas um argumento, *nome*; a da linha 2 recebe dois argumentos, *nome* e *funcao*.

Chamadas de funções podem ser compostas por meio do *pipe operator*, um operador especial que obtém o resultado de uma função e o envia como parâmetro para outra.

Código 2.12: Função com múltiplas chamadas utilizando o *pipe operator*

```
1 def saudar_com_idade(nome, ano_nascimento) do
2   ano_nascimento
3   |> calcular_idade()
4   |> saudar(nome)
5 end
```

O Código 2.12 envia o argumento *ano_nascimento* para a função *calcular_idade/1*, cujo retorno é enviado para *saudar/2*. O Código 2.13 tem o mesmo efeito que o Código 2.12.

Código 2.13: Função com múltiplas chamadas sem o *pipe operator*

```
1 def saudar_com_idade(nome, ano_nascimento) do
2   saudar(calcular_idade(ano_nascimento), nome)
3 end
```

Funções são as unidades de execução de Elixir (SASA, 2019): quando chamadas, funções dão origem a processos, que recebem um identificador (chamado de *Process Identifier* ou *PID*) e podem se comunicar com outros processos em execução.

2.1.5 Módulos

Módulos são as unidades compiláveis da linguagem. Apesar de funções serem as unidades de execução, elas precisam estar dentro de módulos para serem compiladas (SASA, 2019).

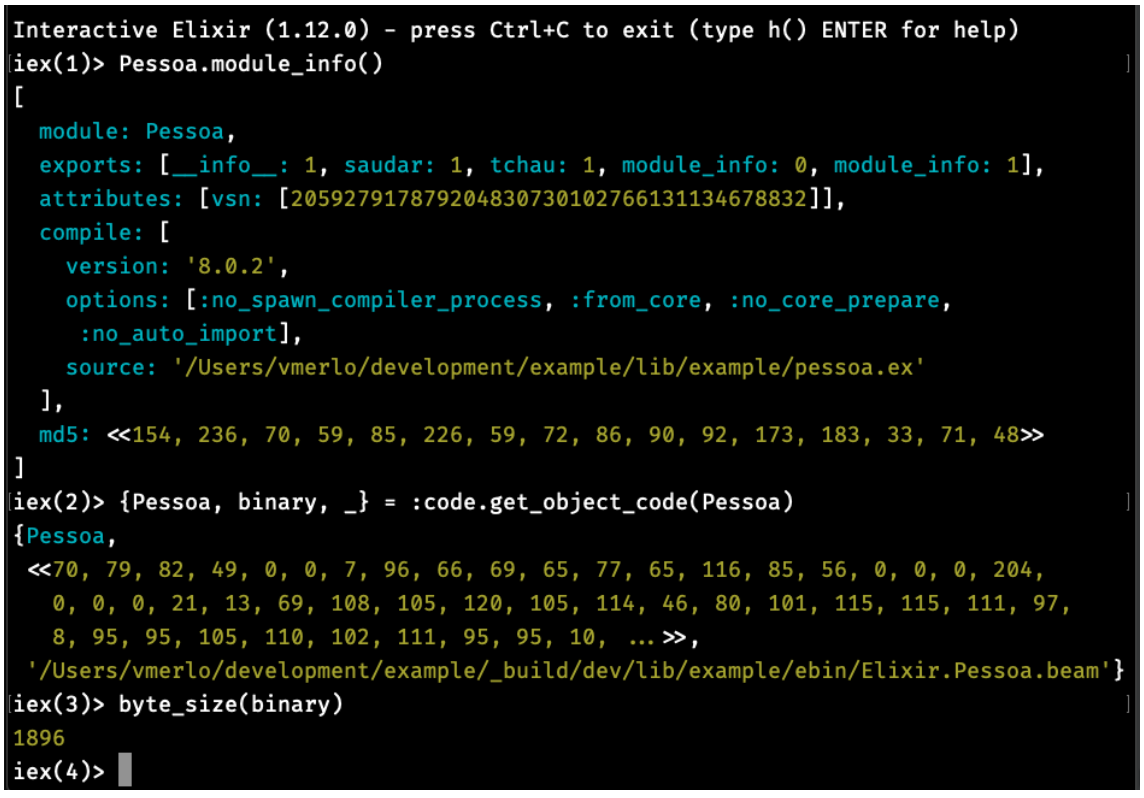
Diferente de classes, das linguagens orientadas a objetos, os módulos não possuem estado ou comportamento. Eles servem apenas para organizar as funções e torná-las compiláveis.

Código 2.14: Módulo com duas funções definidas

```elixir
defmodule Pessoa do
  def saudar(nome) do
    IO.puts("Olá #{nome}!")
  end

  def tchau(nome) do
    emoticon = Emoticon.sad()
    IO.puts("Até mais, #{nome}! #{emoticon}")
  end
end
```

Figura 2.1: Metadados do módulo *Pessoa* e o tamanho do binário gerado



O módulo *Pessoa* do Código 2.14 define as funções *saudar/1* e *tchau/1*, além das funções de metadados. O tamanho do módulo compilado é de 1896 bytes.

Ao ser compilado, o módulo do Código 2.14 proverá as duas funções definidas em seu corpo, como mostra o resultado da função *Pessoa.module_info/0* na Figura 2.1. Ape-

sar de não estarem definidas no código do módulo, as funções *__info__/1*, *module_info/0*, e *module_info/1* são inseridas pelo processo de compilação, a fim de prover metadados sobre módulos (ERLANG, 2021a).

Uma função de um módulo pode fazer uma chamada ou referenciar uma função de outro módulo utilizando a sintaxe *Módulo.função(argumento_1, argumento_2, ..., argumento_n)*, como é feito na linha 7 do Código 2.14.

2.1.6 *Pattern matching*

O *pattern matching* em Elixir tem o objetivo de verificar se um valor é válido para um determinado padrão, que pode ser composto por tipos primitivos, estruturas de dados e até funções (ELIXIR SCHOOL, 2021g).

A linguagem fornece duas maneiras de usar o *pattern matching*: através do operador de *matching* e através de cláusulas de funções — mostrados nos Códigos 2.15 e 2.16.

Código 2.15: *Pattern matching* com o operador de *matching* =

```
1 {:ok, resultado} = {:ok, 10}
```

O operador de *matching* é o símbolo =. Na esquerda do operador fica o *pattern*, e na direita fica o valor testado.

Após a execução do Código 2.15, o conteúdo de *resultado* será *10*. Uma exceção seria levantada caso o valor não estivesse de acordo com o *pattern* — como seria o caso do valor *{:outro_valor, 10}*, por exemplo.

Código 2.16: *Pattern matching* diretamente na cláusula de uma função

```
1 def tenta_match({:ok, resultado}) do
2   resultado
3 end
4
5 def tenta_match(_) do
6   nil
7 end
```

No *pattern matching* através das cláusulas de funções, o *pattern* fica nos argumentos da função. Diferentemente do que acontece com o operador de *matching*, caso o valor

não seja válido para o *pattern*, a próxima função de mesmo nome e mesma aridade será testada.

O Código 2.16 mostra o *pattern* definido em 2.15, mas recebido como argumento da função *tenta_match/2*. Quando a função *tenta_match/2* é executada com um valor, o primeiro *pattern* testado será o definido na linha 1; caso o valor não seja válido para ele, o *pattern* da linha 5 será testado — e, como ele é um *catch all* (definido por _), qualquer valor será válido.

2.1.7 Estruturas de controle

Elixir conta com 3 estruturas de controle: *if*, *case*, e *cond* (ELIXIR SCHOOL, 2021e).

Código 2.17: Controle do fluxo do programa utilizando blocos *if* e *else*

```
1 if notificacao_enviada?(pessoa) do
2   IO.puts("Notificação enviada")
3 else
4   IO.puts("Notificação não enviada")
5 end
```

O *if*, exemplificado no Código 2.17, recebe um valor e um bloco (ou dois, no caso do *else*) — e executa o bloco caso o valor seja verdadeiro. Em Elixir, os únicos valores falsos são *nil* e *false* (ELIXIR SCHOOL, 2021e).

Código 2.18: Controle do fluxo do programa utilizando *case*

```
1 case enviar_notificacao(pessoa) do
2   {:ok, _} -> IO.puts("Enviada")
3   {:error, _} -> IO.puts("Ocorreu um erro!")
4 end
```

O *case*, mostrado no Código 2.18, verifica se o valor passado se encaixa em algumas das cláusulas de *pattern matching*. Caso o *match* não ocorra, um erro de *CaseClauseError* é disparado.

Código 2.19: Controle do fluxo do programa utilizando *cond*

```
1 cond enviar_notificacao(pessoa) do
2   true -> IO.puts("Enviada")
3   false -> IO.puts("Ocorreu um erro ao enviar a notificação")
4 end
```

O *cond*, presente no Código 2.19, funciona de maneira parecida ao *case*, mas sem utilizar *pattern matching* — verificando apenas se o valor passado (linha 1) é exatamente igual a alguma de suas cláusulas (linha 2 e 3).

Além das estruturas básicas, é possível usar o *pattern matching* como uma forma de controlar o programa, conforme mostra o Código 2.20.

Código 2.20: Controle do fluxo do programa utilizando o *pattern matching* pelas cláusulas das funções

```
1  def enviar_mensagem(pessoa) do
2    pessoa
3    |> calcular_idade()
4    |> enviar_propaganda()
5  end
6
7  def enviar_propaganda(idade) when idade >= 21 do
8    enviar_propaganda_de_cerveja()
9  end
10
11 def enviar_propaganda(idade) when idade > 0 do
12   enviar_propaganda_de_refrigerante()
13 end
14
15 def enviar_propaganda(0) do
16   nil
17 end
```

As demais construções da linguagem podem ser vistas na documentação oficial da linguagem, acessível em <https://elixir-lang.org/getting-started/introduction.html>.

2.1.8 Imutabilidade da linguagem

Assim como Clojure (CLOJURE, 2021) e outras linguagens funcionais, todas as estruturas de dados de Elixir são imutáveis (SASA, 2019). Uma estrutura de dados imutável pode apenas ser transformada em uma nova estrutura, mas nunca alterada, como é possível em linguagens de programação com mutabilidade.

A imutabilidade das estruturas de dados é uma característica desejada em aplicações com paralelismo. Ela não permite que dois observadores vejam um mesmo dado de maneiras diferentes, evitando as condições de corrida (SASA, 2019). Entre outras aplicações, também é uma característica desejada para garantir a previsibilidade de programas, pois efeitos colaterais são mais difíceis de serem produzidos em ambientes imutáveis (SASA, 2019).

Exemplificando a imutabilidade de estruturas de dados, um programa que semanticamente modifica uma estrutura — mas não a modifica de verdade, apenas produz uma nova estrutura baseada na antiga.

Código 2.21: Exemplo da imutabilidade da linguagem

```
1 projeto = %{"nome" => "Chronos", "linguagem" => "Elixir"}
2 Map.put(projeto, "linguagem", "Clojure") |> IO.inspect()
3 IO.inspect(projeto)
```

No Código 2.21, um mapa é criado e ligado ao nome *projeto* (linha 1). Na linha 2, o mapa é modificado, dando origem a um novo mapa — ao invés de substituir o mapa anterior, como aconteceria em uma linguagem com mutabilidade. Ao verificar o conteúdo de *projeto* na linha 3, confirma-se que o mapa da linha 1 não foi alterado.

2.1.9 Modelo de atores e processos

O modelo de atores é um método de computação concorrente que utiliza atores como a primitiva universal de computação (HEWITT; BISHOP; STEIGER, 1973). Os atores do modelo de atores possuem um estado e trocam mensagens com outros atores, sem nunca alterar diretamente o estado dos outros atores — como aconteceria com instâncias de objetos, no paradigma de orientação a objetos.

A máquina virtual de Erlang, onde os programas escritos em Elixir são executados, implementa algo parecido com o modelo de atores (SASA, 2019): funções são executadas

como processos, que possuem uma caixa de entrada. Estes processos trocam mensagens entre si e possuem um estado individual, modificado apenas por si mesmos.

A comunicação entre atores se dá através das funções nativas da máquina virtual *send/2* e *receive/1*. A função *send/2* envia uma mensagem contendo qualquer tipo de dado da linguagem para um processo, sem a necessidade de serialização (SASA, 2019), executando de maneira não-bloqueante, seguindo com sua execução sem esperar retorno do processo destinatário.

Código 2.22: Envio de uma mensagem através da função *send/2*

```
1 pid = Process.whereis(Contador)
2 send(pid, {:hello, "world"})
```

No Código 2.22, a chamada para *Process.whereis/1* (linha 1) encontra o *PID* do processo com o nome *Contador*. Em seguida, está a chamada para a função *send/2*, que envia uma mensagem para o processo com o conteúdo *{:hello, "world"}*.

A função *receive/1* age de maneira bloqueante, lendo a caixa de mensagens do processo e executando *pattern matching* da mesma maneira que um *case* (visto na Seção 2.1.7) com a mensagem recebida:

Código 2.23: Recebimento de uma mensagem através da função *receive/1*

```
1 def ler_inbox do
2   receive do
3     {:hello, message} -> IO.puts("Recebi um oi")
4     message -> IO.puts("Recebi alguma outra mensagem.")
5   end
6   IO.puts("Mensagem lida")
7 end
```

No Código 2.23, a chamada da função *receive/1* (linha 2) bloqueia a execução da função *ler_inbox/0* até que uma mensagem chegue ao seu processo. Quando alguma mensagem chegar, a execução de *ler_inbox/0* é retomada.

A leitura da caixa de mensagens funciona de maneira sequencial, onde uma mensagem é lida de cada vez (SASA, 2019). — o *receive/1* (linha 2 do Código 2.23) lê apenas uma mensagem.

2.1.10 Árvore de Sintaxe Abstrata de Elixir

As árvores de sintaxe abstratas da linguagem, representadas usando tipos de dados da própria linguagem Elixir, são utilizadas pelo Chronos para realizar a instrumentação de código. ASTs da linguagem possuem duas possíveis formas:

- Na forma de uma 3-upla *{função_ou_definição, metadados, lista_de_argumentos}*, contendo o nome de uma função ou uma definição como primeiro elemento, os metadados de compilação como segundo, e uma lista de argumentos como terceiro. ASTs nesta forma podem expressar uma chamada de função, uma definição de funções, ou uma definição de módulo – no caso de definições, o primeiro elemento será *def* ou *defmodule*;
- Tipo primitivo da linguagem.

Como exemplos de ASTs da primeira forma, estão os Códigos 2.24 e 2.26, expressando as ASTs utilizando estruturas da própria linguagem. Note que no Código 2.24, acontece uma definição de função.

Código 2.24: AST da definição da função *saudar/0*

```
1 {:def, [],
2  [
3    {:saudar, [], Elixir},
4    [
5      do: {{:., [], [{:__aliases__, [], [:IO]}, :puts]}, [],
6       ["Olá! Como você está?"]}
7    ]
8  ]}
```

A AST presente no Código 2.24, representa a função definida no Código 2.25.

Código 2.25: Definição da função *saudar/0*

```
1 def saudar do
2   IO.puts("Olá! Como você está?")
3 end
```

No Código 2.26, a função *saudar/0* do Código 2.25 é chamada. Ambas ASTs possuem a mesma forma *{função_ou_definição, metadados, argumentos}*.

Código 2.26: AST de chamada de função

```
1 {:saudar, [], []}
```

As ASTs dos tipos primitivos da linguagem, listados na Seção 2.1.3, são os próprios tipos primitivos. O Código 2.24 contém algumas ASTs de tipos primitivos, como o átomo *:saudar* na linha 3 e a string *"Olá! Como você está?"* na linha 6.

2.2 *Profilers*

Profilers, também chamados de ferramentas de *profiling*, são utilizados para examinar o uso de recursos pela execução de um programa, extraindo um perfil da execução de um programa sob determinadas circunstâncias a partir da análise de sua execução (GROTKER, 2012). *Profilers* podem examinar diferentes recursos, como:

- O número de execuções das funções do programa;
- O tempo de execução das funções do programa;
- O número de instruções de baixo nível executadas pelo programa;
- O consumo de memória por cada função.

Os *profilers* fornecem análises para execuções específicas do programa, com entradas pré-determinadas, não sendo capazes de fornecer perfis genéricos para qualquer execução do programa. Mesmo assim, ferramentas de *profiling* são consideradas uma das abordagens mais úteis para a melhoria da desempenho de programas (LEACH, 2020). Outros autores, como Humble e Farley (HUMBLE; FARLEY, 2010), também classificam os *profilers* como importantes na solução de problemas de desempenho, especialmente através da análise de excessos no uso recursos para a identificação de gargalos.

Ferramentas de *profiling* podem usar diferentes técnicas para obter perfis de execução. Grotker (GROTKER, 2012) classifica as ferramentas de *profiling* em duas categorias: a dos *profilers* por amostragem e a dos *profilers* por instrumentação.

2.2.1 *Profilers* por amostragem

Profilers nesta categoria examinam o uso de recursos do computador em intervalos de amostragem, obtendo o perfil da execução a partir da amostragem do uso de recursos

do ambiente. Esta técnica não modifica o programa analisado e normalmente não adiciona *overhead*, porém a qualidade do perfil obtido depende do intervalo de amostragem utilizado (GROTKER, 2012).

Por extraírem o perfil do uso de recursos a partir do ambiente computacional, os *profilers* que utilizam esta técnica normalmente são de uso geral, podendo ser utilizados para analisar programas escritos em qualquer linguagem, desde que o programa seja executável no ambiente em que o *profiler* está executando.

Um exemplo de ferramenta de *profiling* que está na categoria dos *profilers* por amostragem é o *gprof* (GNU, 2021) — um *profiler* do projeto GNU que analisa programas compilados para sistemas operacionais compatíveis com Unix.

2.2.2 *Profilers* por instrumentação

Profilers nesta categoria inserem chamadas extras no programa examinado, a fim de extrair informações sobre a execução do programa examinado enquanto ela acontece. A instrumentação pode ocorrer em diferentes lugares: durante a compilação, através da modificação do código fonte do programa original; após a compilação, através da modificação das instruções do programa compilado; durante a execução, através de modificações no ambiente para a inserção de instruções adicionais.

Segundo (GROTKER, 2012), esta técnica oferece vantagens em relação à amostragem: o perfil resultante não está sujeito às variâncias estatísticas e ela permite a análise de um leque maior de recursos — pois o analisador pode inserir código arbitrário no programa fonte.

Uma grande desvantagem é que os *profilers* que utilizam esta técnica estão restritos às linguagens (sejam de baixo ou de alto nível), necessitando que o *profiler* saiba como injetar instruções de medição na linguagem em que o programa a ser analisado foi escrito ou compilado.

2.3 *Profiling* em Elixir

2.3.1 Ferramentas de *profiling*

Existem diferentes *profilers* para a máquina virtual de Erlang — e, por consequência, para programas escritos em Elixir. Alguns dos *profilers* destacados pelo trabalho (ŚLASKI; TUREK, 2019) são:

- ***lcnt***: Integrado à máquina virtual do Erlang, o *profiler lcnt* obtém o perfil de execução de um programa com relação ao seu uso de *threads* e quantidade de *locks* utilizando mecanismos internos da máquina virtual. O perfil de execução obtido é especialmente útil para analisar a paralelização do programa (ERLANG, 2021e).

- ***cprof***: É um *profiler* que exclusivamente obtém o número de execuções de funções durante a execução de um programa. O *cprof* está presente nativamente na máquina virtual, e utiliza instrumentação das instruções do programa compilado com instruções especiais da máquina virtual para obter o número de execuções (ERLANG, 2021c).

- ***eprof***: Parecido com o *cprof*, é utilizado para obter o tempo de execução das funções executadas por um programa. Também presente nativamente na máquina virtual, o *eprof* utiliza a instrumentação com instruções especiais da máquina virtual para obter o tempo de execução de cada função (ERLANG, 2021d).

2.3.2 *Profiling* manual

No *profiling* manual, o programador insere contadores na aplicação, fazendo manualmente o que uma ferramenta de *profiling* por instrumentação faria automaticamente. Os contadores são inseridos em pontos de interesse para o programador, como nas saídas de funções, por exemplo.

Em Elixir, uma maneira de fazer o *profiling* manual é utilizar módulos *mock* — que, aliados à uma biblioteca de *mocking*, permitem criar expectativas sobre chamadas de funções nos casos de teste do programa. Estas expectativas permitem aos programadores verificar quantas vezes uma função foi chamada, quais foram seus argumentos, entre outras introspeções sobre o código (HEXDOCS, 2021c).

Código 2.27: Exemplos de casos de teste utilizando *mocking*

```
1  # Caso de teste será executado com sucesso.
2  test "chama a função soma duas vezes" do
3    expect(Mock, 2, :soma, fn a, b -> a + b end)
4    Mock.soma()
5    Mock.soma()
6  end
7
8  # Caso de teste retornará um erro quando executado.
9  test "chama a função soma duas vezes" do
10   expect(Mock, 2, :soma, fn a, b -> a + b end)
11   Mock.soma()
12 end
```

O caso de teste definido no bloco que começa na linha 2 do Código 2.27 define a expectativa de que a função *Mock.soma/2* será chamada duas vezes — o que acontece nas linhas 4 e 5. Então, este caso será executado com sucesso pela suíte de testes.

Já o segundo caso de teste do Código 2.27, definido no bloco que começa na linha 9, retornará um erro, pois a expectativa definida na linha 10 não será atendida, pois a função *Mock.soma/2* será chamada apenas uma vez na linha 11.

O *profiling* manual é simples de ser realizado, não necessitando de ferramentas externas, apenas a escrita de código de medição pelos programadores. Porém, o método de *profiling* manual é desencorajado por ser suscetível a erros humanos, além de representar uma etapa a mais no desenvolvimento já que requer a adaptação dos módulos do programa para a compatibilidade com o *mocking* (GROTKER, 2012).

2.4 Teste de software

Teste de software é um método investigativo no processo de engenharia de software que tem como objetivo elevar a qualidade do software através da detecção de falhas. Testes são a principal maneira utilizada na indústria para medir a qualidade de um programa durante seu ciclo de desenvolvimento (AMMANN; OFFUTT, 2008).

O método consiste na criação e execução de casos de testes — cenários aos quais o software ou seus componentes são submetidos e avaliados. Existem diferentes estratégias

para a criação de casos de testes, sendo estas estratégias uma ampla área de estudo.

(JORGENSEN, 2014) define que um caso de teste é composto pelo identificador do teste, por um conjunto de entradas, e por um conjunto de resultados esperados. O grupo de casos de testes de um programa é chamado de suíte de testes.

Existem diferentes categorias de testes, conforme definido por (AMMANN; OFFUTT, 2008):

1. Testes de aceitação: avaliam o software de acordo com seus requisitos.

2. Testes de sistema: avaliam o software de acordo com sua arquitetura.

3. Testes de integração: avaliam o software de acordo com o design do sistema.

4. Testes de módulos: avaliam o software de acordo com o seu design detalhado.

5. Testes unitários: avaliam o software de acordo com sua implementação.

2.4.1 Testes unitários

A ferramenta de *profiling* implementada neste trabalho, o Chronos, tem por objetivo oferecer suporte a execução de testes unitários das funções de programas Elixir, possibilitando a análise do perfil do programa diretamente nos seus testes de implementação.

Testes unitários têm o objetivo de avaliar a implementação das unidades do programa. São os testes mais próximos do código, normalmente escritos pelos próprios programadores e embutidos junto do programa (AMMANN; OFFUTT, 2008).

(FOWLER, 2019) considera os testes unitários uma parte essencial do processo de desenvolvimento, sendo a escrita de testes determinada como obrigatória antes mesmo do início da implementação do programa.

Por sua proximidade com o código do programa, a execução de testes unitários está vinculada à linguagem ou plataforma na qual o programa foi escrito. Existem diferentes bibliotecas que auxiliam na escrita e execução de testes unitários, como a *jUnit* para programas escritos em Java (JUNIT, 2021); a *rspec* para programas escritos em Ruby (RSPEC, 2021); a *ExUnit* para programas escritos em Elixir (HEXDOCS, 2021b).

2.4.2 Testes unitários em Elixir

A biblioteca padrão para escrita e execução de testes unitários em Elixir é a *ExUnit*, que vêm embutida por padrão em novos projetos criados na linguagem (ELIXIR SCHOOL, 2021h).

Código 2.28: Exemplo de teste unitário utilizando a *ExUnit*

```
1  defmodule CalculadoraTest do
2    use ExUnit.Case
3
4    test "decrementa corretamente" do
5      assert Calculadora.decrementa(4) == 3
6      assert Calculadora.decrementa(3) == 2
7    end
8
9    test "não decrementa quando o valor é zero ou negativo" do
10         assert Calculadora.decrementa(0) == :error
11         assert Calculadora.decrementa(-1) == :error
12   end
13 end
```

O Código 2.28 contém o módulo *CalculadoraTest*, que testa o módulo *Calculadora*, que define a função *decrementa/1*. Na linha 2 do Código 2.28, através do *use*, é declarado que o módulo é um caso de testes da biblioteca *ExUnit* — o que importa uma série de macros para o módulo, como o *test*, *assert*, *refute*.

Nas linhas 4 e 9 do Código 2.28 estão definidos dois cenários de teste: o primeiro, define que a função *Calculadora.decrementa/1* deve subtrair *1* do número passado como argumento (testando os números *4* e *3* como entradas); o segundo teste define que a função *Calculadora.decrementa/1* deve retornar *:error* quando os números *0* e *-1* são passados.

Todos os módulos de testes devem ser salvos na pasta *test* do projeto, sendo executados pela rotina *mix test* (detalhada na Seção 2.1.2). Um projeto com o módulo do Código 2.28, quando executado pela *mix test*, tem a saída mostrada no Código 2.29.

Código 2.29: Execução da tarefa *mix test* em um projeto com o teste *CalculadoraTest*

```
1 $ mix test
2 Compiling 2 files (.ex)
3 ....
4 Finished in 2.2 seconds
5 2 tests, 0 failures
6
7 Randomized with seed 231834
```

3 PROPOSTA

Este trabalho propõe o Chronos, uma ferramenta de *profiling* integrada à suíte de testes para programas escritos em Elixir. O Chronos foi implementado na forma de uma biblioteca, também em Elixir, sendo compatível com qualquer outro projeto na linguagem. Ele está disponível no gerenciador de pacotes da linguagem e também pode ser obtido em <https://github.com/vamm/chronos>.

Através da instrumentação das funções do programa com contadores, o Chronos possibilita aos programadores fazerem asserções sobre a quantidade de execuções de funções diretamente nos casos de teste do programa, sem a necessidade de etapas de anotações ou a utilização de ferramentas externas.

Para utilizar o Chronos em um programa, basta adicioná-lo como uma dependência no arquivo de configuração *mix.exs*. Após isto, é possível chamá-lo em qualquer parte da suíte de testes do projeto utilizando a função *Chronos.profile/1*, que espera como argumento uma função anônima que contém as chamadas a serem analisadas. Para executar as análises, basta executar a suíte de testes normalmente utilizando a rotina *mix test*.

Código 3.1: Exemplo de caso de teste que utiliza o Chronos

```
1 test "executa o profiler" do
2   funcoes_executadas = Chronos.profile(fn ->
3     Calculadora.soma(4, 6)
4   end)
5
6   assert %{{Calculadora, :soma, 2} => 1} = funcoes_executadas
7   assert Map.keys(funcoes_executadas) == [{Calculadora, ↩
      :soma, 2}]
8 end
```

No momento em que o caso de teste do Código 3.1 é executado através do comando *mix test* junto dos demais testes da suíte de testes, a função *soma/2* do módulo *Calculadora* estará instrumentada com uma instrução de envio de mensagem para um contador em seu ponto de entrada – assim como todas as outras funções do programa, devido à etapa de instrumentação do Chronos (descrita na Seção 3.4).

A função *Chronos.profile/1* do Código 3.1 então executará a função anônima passada (linha 2), executando a chamada para *Calculadora.soma/2* (linha 3), retornando na

variável *funcoes_executadas* (linha 2) o estado do contador de execuções de funções do Chronos, sobre a qual poderão ser feitas asserções. Nas linhas 6 e 7 estão definidas duas asserções: uma verifica que a função *Calculadora.soma/2* foi executada exatamente 1 vez, e a outra verifica que nenhuma outra função foi executada além de *Calculadora.soma/2*.

Para facilitar a compreensão deste capítulo, o programa *Exemplo*, cujos módulos são definidos pelos Códigos 3.2 e 3.3, será usado nas próximas seções deste capítulo.

Código 3.2: Módulo *Exemplo.Downloader*

```
1 defmodule Exemplo.Downloader do
2   def download(urls, limite) do
3     urls
4     |> Enum.take(limite)
5     |> Enum.map(fn url ->
6       Exemplo.HTTP.get(url)
7     end)
8   end
9 end
```

No Código 3.2, o módulo *Exemplo.Downloader* define a função *download/2*, que espera como argumentos uma lista de *urls* e um inteiro *limite* como teto para o tamanho da lista.

Na linha 4 do Código 3.2, a chamada para a função *Enum.take/2* é responsável por limitar o tamanho da lista, restringindo o tamanho da lista *urls* a *limite* elementos. A lista limitada é então passada para a função *Enum.map/2* (linha 5), que faz uma chamada para *Exemplo.HTTP.get/1* com cada uma das *urls* como argumento (linha 6) e retorna uma lista contendo os resultados das chamadas.

Código 3.3: Módulo *Exemplo.HTTP*

```
1 defmodule Exemplo.HTTP do
2   def get(url) do
3     :http.get(url, 80)
4   end
5 end
```

No Código 3.3, o módulo *Exemplo.HTTP* define uma única função, *get/1*, que é uma encapsuladora para a biblioteca nativa *:http* — que faz uma requisição HTTP e

retorna sua resposta como binário.

O caso de teste descrito no Código 3.4 faz parte da suíte de testes do programa *Exemplo* e também é compartilhado pelas próximas seções deste capítulo.

Código 3.4: Cenário de teste para *Exemplo.Downloader*

```
1  test "limita corretamente as chamadas para HTTP.get/1" do
2    urls = [
3           "example.com/file_1.png",
4           "example.com/file_2.png",
5           "example.com/file_3.png",
6           "example.com/file_4.png",
7           "example.com/file_5.png",
8           "example.com/file_6.png"
9         ]
10
11   assert %{{Exemplo.HTTP, :get, 1} => 4} =
12             Chronos.profile(fn ->
13               Exemplo.Downloader.download(urls, 4)
14             end)
15 end
```

Como mostrado no Código 3.4, é passada para a função *Chronos.profile/1* uma função anônima contendo uma chamada para *Exemplo.Downloader.download/2* (linha 13), com uma lista de 6 elementos como primeiro argumento e o número *4* como segundo — estes argumentos representam, respectivamente, a lista de urls e um limite de downloads.

O resultado da execução de *Chronos.profile/1* no Código 3.4, contendo a contagem de execuções de funções pelo programa, é então verificado com o *pattern %{{Exemplo.HTTP, :get, 1} => 4}* (linha 11) — que será verdadeiro somente se o contador possuir a chave *{Exemplo.HTTP, :get, 1}* mapeada para *4*, ou seja, quando acontecerem 4 chamadas para *Exemplo.HTTP.get/1*.

Este capítulo fornece uma visão geral do Chronos na Seção 3.1. Após isto, apresenta detalhadamente as etapas de execução do Chronos nas Seções 3.2, 3.3, e 3.4. Então, termina mostrando a execução do caso de teste descrito no Código 3.4.

3.1 Visão Geral

Na perspectiva do programador, os seguintes passos são necessários para utilizar o Chronos como ferramenta de *profiling* na suíte de testes de um projeto em Elixir:

1. Adicionar o Chronos como uma dependência do projeto no arquivo de configuração *mix.exs*.

2. Criar o caso de teste desejado, passando uma função anônima contendo as funções a serem analisadas como argumento de *Chronos.profile/1* — como acontece no caso de teste do programa *Exemplo* (Código 3.4, entre as linha 12 e 14), e fazendo asserções sobre o resultado de *Chronos.profile/1* — que conterá um mapa com a quantidade de execuções de cada função executada durante a análise.

3. Executar a suíte de testes normalmente, através da rotina *mix test*.

Ressalta-se que a integração do *profiler* com a suíte de testes acontece sem etapas adicionais ao programador, sem a necessidade de anotações ou código extra além dos cenários de testes que utilizam o *profiling*.

Do ponto de vista de implementação, a arquitetura do Chronos é composta pelos seguintes componentes:

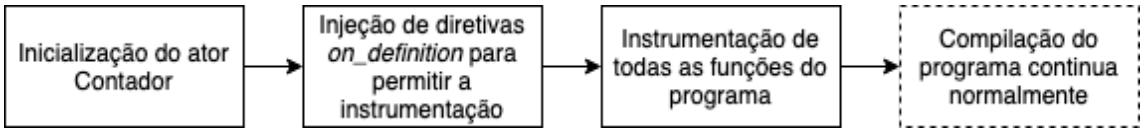
- **Contador:** é o ator utilizado como contador de execuções de funções pelo Chronos, sendo definido no módulo *Chronos.Collector*.

- **Injetor:** responsável pela injeção de diretivas *on_definition* (vistas na Seção 3.3), é definido pelo módulo *Chronos.Injector*.

- **Instrumentador:** neste componente, as ASTs das funções do programa analisado são instrumentadas com instruções de envio de mensagens para o Contador. Este componente é definido pelo módulo *Chronos.Instrumentor*.

- **Ajudante de testes:** definido pelo módulo *Chronos.TestHelper*, provê a integração entre a suíte de testes do programa testado e o Chronos através da função *Chronos.profile/1*.

Figura 3.1: Fluxo das etapas executadas pelo Chronos. Em tracejado, etapa já existente na compilação.



Estes componentes são utilizados pelas etapas no fluxo mostrado na Figura 3.1, que se inicia com a execução da suíte de testes pelo programador e termina com o prosseguimento normal da compilação do programa e execução da suíte de testes.

3.2 Inicialização do ator Contador

O ator Contador, definido pelo módulo *Chronos.Collector*, é iniciado quando a suíte de testes é executada. O Contador é responsável por contar as chamadas de funções que acontecem durante a execução do programa.

Como consequência da etapa de instrumentação, toda vez que uma função do programa é executada, uma mensagem é enviada para o ator Contador. Ao final da execução total do programa, o ator Contador terá em seu estado o número total de execuções de cada função. Assim, para a realização de asserções nos cenários de testes, basta executar as funções que deseja-se analisar e, após o término da execução, verificar se a asserção é válida para o estado final do ator Contador.

O estado do ator Contador é definido por um mapa contendo *{Módulo, função, aridade}* como chaves ligadas a valores inteiros que correspondem ao número de vezes que cada função é executada.

Código 3.5: Tratamento de mensagens do ator Contador no Chronos

```
1  defmodule Chronos.Collector do
2    def handle_message({:incrementar, funcao}, estado) do
3      {:noreply, incrementar_contador(estado, funcao)}
4    end
5
6    def handle_message(:expor_estado, estado) do
7      {:reply, estado, estado}
8    end
9
10   def handle_message(:limpar, estado) do
11     {:noreply, %{}}
12   end
13 end
```

O ator Contador pode receber 3 mensagens diferentes: *{:incrementar, função}*, *expor_estado*, e *limpar*, como mostra o Código 3.5, que define a implementação do tratamento de mensagens do ator Contador no Chronos. As mensagens *incrementar* (linha 2) e *limpar* (linha 6) não possuem retorno para a função, apenas a mensagem *expor_estado* (linha 10) tem um retorno: o estado interno do contador.

No Código 3.5, a mensagem *{:incrementar, função}* (linha 2) é enviada em todas as entradas das funções executadas pelo programa analisado, após ele ser instrumentado pelo Chronos. Já as mensagens *expor_estado* (linha 6) e *limpar* (linha 10) do Código 3.5 são enviadas pela suíte de testes ao Chronos, com o objetivo de limpar o contador e obter o estado interno do contador, respectivamente.

3.3 Injeção de diretivas *on_definition*

Para instrumentar as funções do programa, é necessário fazer com que o compilador de Elixir repasse as definições de funções do programa para o módulo de instrumentação do Chronos, o *Chronos.Instrumentor* — que é o responsável por adicionar instruções para enviar mensagens ao ator Contador nos pontos de entrada das funções.

A diretiva *on_definition* é usada para instruir o compilador da linguagem a enviar tuplas contendo as ASTs das funções definidas pelo programa ao Chronos, sendo a

diretiva injetada dinamicamente em todos os módulos do programa pelo módulo *Chronos.Injector*.

Código 3.6: Módulo Exemplo.Downloader com diretiva *on_definition* injetada

```
1 defmodule Exemplo.Downloader do
2   @on_definition Chronos.Instrumentor
3
4   def download(urls, limite) do
5     # ...
6   end
7 end
```

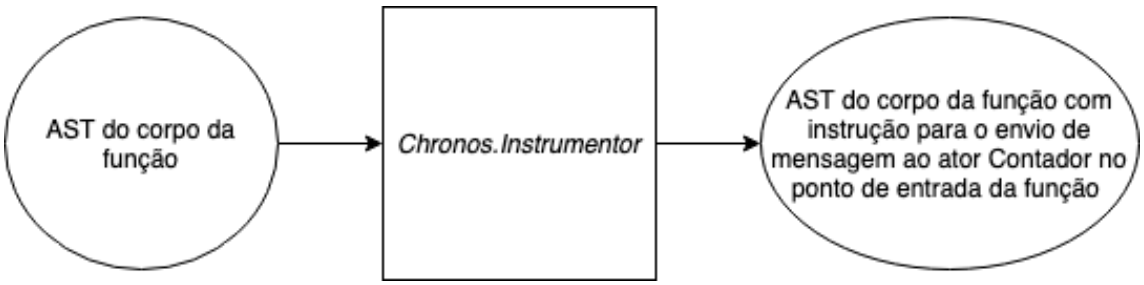
Quando o Chronos é executado no programa *Exemplo*, definido no início deste capítulo, o módulo *Exemplo.Downloader* tem a diretiva *on_definition* injetada pelo Chronos — como mostra a linha 2 do Código 3.6. O mesmo acontece com o módulo *Exemplo.HTTP*.

Durante a compilação dos módulos do programa, o compilador do Elixir enviará para o *Chronos.Instrumentor* as ASTs das funções sendo definidas pelos módulos. Assim, o módulo *Chronos.Instrumentor* consegue transformar as ASTs enquanto a compilação acontece. No caso do programa *Exemplo*, durante a compilação dos módulos *Exemplo.Downloader* e *Exemplo.HTTP*, o compilador envia ao *Chronos.Instrumentor* as ASTs de *Exemplo.Downloader.download/2* e *Exemplo.HTTP.get/1*, que são transformadas.

3.4 Instrumentação das funções do programa

O módulo responsável pela instrumentação das funções é o *Chronos.Instrumentor*, cujo objetivo é fazer a transformação da AST do corpo de uma função em uma AST do corpo da função com uma instrução de envio de mensagem ao ator Contador em seu ponto de entrada, como mostra a Figura 3.2.

Figura 3.2: Entrada e saída da etapa de instrumentação



Duas regras de transformação de ASTs são utilizadas nesta etapa. Suas implementações são mostradas no Código 3.7 e no Código 3.8.

Código 3.7: Regra de instrumentação para funções que realizam mais de uma chamada de função

```
1 def instrumentar({:__block__, [], chamadas}) do
2   {:__block__, [], [ast_incrementar_contador | chamadas]}
3 end
```

O Código 3.7 coloca a AST *ast_incrementar_contador* como o primeiro elemento da lista de chamadas da AST sendo transformada (linha 2). Na prática, o Código 3.7 faz com que a função transformada execute o código definido pela AST *ast_incrementar_contador* em seu ponto de entrada.

A regra do Código 3.7 cobre os casos de funções que fazem mais de uma chamada, como a função *Exemplo.Downloader.download/2* definida no início deste capítulo no Código 3.2, que faz duas chamadas: uma para *Enum.take/2* (linha 4 do Código 3.2) e outra para *Enum.map/2* (linha 5 do Código 3.2).

Código 3.8: Regra de instrumentação para funções que realizam apenas uma chamada

```
1 def instrumentar({_, _, _} = chamada_única) do
2   {:__block__, [], [ast_incrementar_contador, chamada_única]}
3 end
```

O Código 3.8, de maneira parecida com o Código 3.7, adiciona a AST *ast_incrementar_contador* como a primeira chamada de função da AST passada, também fazendo com que ela execute o código definido pela *ast_incrementar_contador* em seu ponto de entrada.

A regra do Código 3.8 cobre os casos de funções que realizam uma única chamada. Este é o caso da função *Exemplo.HTTP.get/1* do Código 3.3, que apenas realiza uma

chamada de função, para *:http.get/2* (linha 3).

São necessárias duas regras de transformação pois as ASTs das funções em Elixir podem possuir duas formas diferentes. Não há necessidade de outras regras de transformação pois apenas as ASTs de corpos de funções, que possuem somente duas formas, são passadas para a etapa de instrumentação.

Código 3.9: AST inserida pela instrumentação com uma chamada para o ator Contador

```
1 {
2   {
3     :.,
4     [],
5     [Chronos.Collector, :incrementa_contador]
6   },
7   [],
8   [assinatura_da_função_instrumentada]
9 }
```

O Código 3.9 mostra a AST *ast_incrementar_contador*, que é injetada pelas regras de transformação nas funções instrumentadas, mostradas nos Códigos 3.7 e 3.8. A AST injetada contém apenas uma instrução de envio de mensagem ao ator Contador.

3.4.1 Exemplos de instrumentação

Abaixo estão exemplos da instrumentação das funções definidas pelos módulos do programa *Exemplo*, apresentado no início deste capítulo.

3.4.1.1 Instrumentação do módulo Exemplo.Downloader

O módulo *Exemplo.Downloader* (Código 3.2 define apenas a função *download/2*, que tem chama mais do que uma função em seu corpo (linhas 1 e 3 do Código 3.10) — se encaixando, então, na regra de instrumentação para funções que fazem múltiplas chamadas (Código 3.7). O Código 3.10 define parcialmente a AST original da função *Exemplo.Downloader.download/2*, substituindo chamadas internas por reticências para facilitar a visualização.

Código 3.10: AST parcial da função *Exemplo.Downloader.download/2*

```
1 {:__block__, [],
2   [
3     {{:., [line: 7], [Enum, :map]}, [line: 7],
4      [
5        {{:., [line: 6], [Enum, :take]}, [line: 6], [...]},
6        {:fn, [line: 7], [...]}
7      ]}
8   ]}
```

O resultado da instrumentação da AST de *Exemplo.Downloader.download/2* está parcialmente definido no Código 3.11. Note a presença da AST *ast_envio_chamada* (Código 3.9) na linha 3.

Código 3.11: AST parcial da função *Exemplo.Downloader.download/2* instrumentada

```
1 {:__block__, [],
2   [
3     {{:., [], [Chronos.Collector, :incrementa_contador]}, ↩
          [], [Exemplo.Downloader, :download, 2]},
4     {{:., [line: 7], [Enum, :map]}, [line: 7],
5      [
6        {{:., [line: 6], [Enum, :take]}, [line: 6], [...]},
7        {:fn, [line: 7], [...]}
8      ]}
9   ]}
```

3.4.1.2 Instrumentação do módulo Exemplo.HTTP

O módulo *Exemplo.HTTP* (Código 3.3) também possui uma única função, *Exemplo.HTTP.get/1*, cuja AST é mostrada no Código 3.12.

Código 3.12: AST original da função *Exemplo.HTTP.get/1*

```
1  {{:., [], [:http, :get]}, [],
2   [{:url, [], nil}, 80]}
```

A função *Exemplo.HTTP.get/1* possui apenas uma única chamada de função em

seu corpo (linha 2 do Código 3.12), encaixando-se assim regra de instrumentação para funções com apenas uma chamada (Código 3.8). O resultado da instrumentação está no Código 3.13, com a AST *ast_envio_chamada*, do Código 3.9, presente na linha 2.

Código 3.13: AST instrumentada da função *get/2*

```
1 {:__block__, [], [
2     {{:., [], [Chronos.Collector, :incrementa_contador]}, ↩
          [], [Exemplo.HTTP, :get, 1]}
3     | {{:., [], [:http, :get]}, [], [{:url, [], nil}, 80]}
4   ]}
```

3.5 Execução dos testes

A execução dos testes não é uma etapa realizada pelo Chronos — ela é executada pela biblioteca de testes, a ExUnit, sem a interação direta com o Chronos, que teve suas etapas executadas durante a compilação. Porém, o Chronos provê para a execução dos testes a função *Chronos.profile/1*, que é utilizada pelos testes de *profiling* para analisar a execução de funções analisadas.

A função *Chronos.profile/1* é fornecida pelo módulo *Chronos.TestHelper* para a suíte de testes. O objetivo da função é auxiliar os testes, tornando a obtenção do estado do ator Contador uma tarefa fácil. O passo-a-passo em alto nível da função *Chronos.profile/1* é:

1. Receber uma função anônima;
2. Limpar o estado do ator Contador;
3. Executar a função anônima recebida;
4. Obter o estado do ator Contador e retorná-lo.

A execução do caso do teste definido no início deste capítulo pelo Código 3.4 acontece da seguinte maneira:

1. Execução da função *Chronos.profile/1* começa;
2. Estado do ator contador é limpo;

3. Função *Exemplo.Downloader.download(urls, 4)* é executada;

4. Estado do ator Contador é obtido e retornado pela função *Chronos.profile/1*, que termina sua execução;

5. *Pattern match* entre o retorno da função *Chronos.profile/1* e o mapa *%{{Exemplo.HTTP, :get, 1} => 4}* é executado — e caso seja inválido, retorna um erro da suíte de testes.

4 EXPERIMENTOS

Como medida de avaliação do impacto do Chronos na compilação e execução dos programas analisados, foram criados casos de testes que utilizam o Chronos para *profiling* em três projetos: *Phoenix*, *Ecto*, e *ExDoc*. As suítes de testes dos projetos foram executadas por duas sequências de 100 vezes, com e sem a instrumentação do Chronos, a fim de comparar o impacto do Chronos no tempo de execução da suíte e no tamanho dos módulos compilados.

Para a execução dos experimentos, foi utilizado um computador com processador I7 de 2.6 GHz em 6 cores e 16 GB de RAM, executando a máquina virtual de Erlang na versão 24.0.0 e o compilador de Elixir na versão 1.12.0.

Todos os resultados foram obtidos a partir de dois conjuntos de execução sequencial da suíte de testes: um utilizando o Chronos com um caso de testes que utiliza *profiling* e outro sem utilizar o Chronos. Para realizar os experimentos, o seguinte *bash script* foi utilizado:

```
for i in {1..100}; do mix test; done
```

Para obter o tamanho total dos módulos compilados dos projetos, o *script* em Elixir descrito no Código 4.1 foi utilizado. O *script* utiliza a função *:code.get_object_code/1* para obter os binários dos módulos dos projetos (linha 10), então extrai seus tamanhos em bytes (linha 11) e os soma (linha 12).

Código 4.1: Script em Elixir para obter o tamanho dos módulos de um projeto

```
1  project = "{nome_do_projeto}"
2  :code.all_loaded() |> Enum.filter(fn
3    {_, :preloaded} ->
4      false
5    {_, path} ->
6      path
7      |> List.to_string()
8      |> String.contains?(project)
9  end) |> Enum.map(fn {module, _} ->
10   {_, binary, _} = :code.get_object_code(module)
11   byte_size(binary)
12 end) |> Enum.sum()
```

4.1 *Phoenix*

O *Phoenix* é um *framework* web em Elixir inspirado no *Rails* da linguagem Ruby (RUBY ON RAILS, 2021). O projeto é de código aberto, mantido pela organização Phoenix Framework, tendo em sua versão 1.6.2 cerca de 118 módulos na sua base de código, mais de 17.743 linhas de código, e uma suíte de testes com 801 casos de teste. O código fonte do *Phoenix* está disponível no GitHub do projeto, <https://github.com/phoenixframework/phoenix>.

Código 4.2: Novo teste incluído na suíte de testes do Phoenix

```
1 test "calls Phoenix.Logger.install/0 only once" do
2   assert %{{Phoenix.Logger, :install, 0} => 1}} =
3     Chronos.profile(fn ->
4       stack_conn()
5       |> MyController.call(:show)
6     end)
7 end
```

À suíte de testes do *Phoenix*, em sua versão 1.6.2, foi incluído um novo cenário de teste que utiliza o Chronos, mostrado no Código 4.2. O objetivo do novo teste é verificar que apenas uma execução de *Phoenix.Logger.install/0* acontece quando a função *MyController.call/2* é executada. Esta verificação é realizada através da asserção da linha 2.

Figura 4.1: Resultado da execução do teste com *profiling* pelo Chronos

```
.%{
  {Mix.Phoenix.Context, {:__struct__, 0}} ⇒ 4,
  {Mix.Phoenix.Schema, {:__struct__, 0}} ⇒ 10,
  {Phoenix.Channel, {:__using__, 1}} ⇒ 5,
  {Phoenix.CodeReloader.Server, {:child_spec, 1}} ⇒ 1,
  {Phoenix.Config, {:child_spec, 1}} ⇒ 1,
  {Phoenix.Controller, {:__using__, 1}} ⇒ 13,
  {Phoenix.Controller.Pipeline, {:__using__, 1}} ⇒ 13,
  {Phoenix.Endpoint, {:__using__, 1}} ⇒ 14,
  {Phoenix.Endpoint.Cowboy2Adapter, {:child_specs, 2}} ⇒ 1,
  {Phoenix.Endpoint.Cowboy2Handler, {:init, 2}} ⇒ 8,
  {Phoenix.Endpoint.RenderErrors, {:__using__, 1}} ⇒ 1,
  {Phoenix.Logger, {:install, 0}} ⇒ 1,
  {Phoenix.Naming, {:resource_name, 2}} ⇒ 261,
  {Phoenix.Presence, {:__using__, 1}} ⇒ 2,
  {Phoenix.Router, {:__using__, 1}} ⇒ 16,
  {Phoenix.Socket, {:__struct__, 0}} ⇒ 69,
  {Phoenix.Socket.Broadcast, {:__struct__, 0}} ⇒ 122,
  {Phoenix.Socket.Message, {:__struct__, 0}} ⇒ 376,
  {Phoenix.Socket.PoolSupervisor, {:child_spec, 1}} ⇒ 2,
  {Phoenix.Socket.Reply, {:__struct__, 0}} ⇒ 50,
  {Phoenix.Socket.Transport, {:load_config, 2}} ⇒ 8,
  {Phoenix.Socket.V1.JSONSerializer, {:fastlane!, 1}} ⇒ 1,
  {Phoenix.Token, {:sign, 4}} ⇒ 2,
  {Phoenix.Transports.LongPoll.Server, {:child_spec, 1}} ⇒ 2
}
.

Finished in 8.6 seconds (3.3s async, 5.2s sync)
11 doctests, 803 tests, 0 failures, 811 excluded

Randomized with seed 800613
```

Quando executada a suíte de testes, o novo teste também é executado com sucesso. O tempo médio da sequência de execução da suíte de testes do projeto sem o Chronos foi de 59,0 segundos, e com o Chronos foi de 59,1 segundos, um acréscimo de 0,16%.

O tamanho total dos módulos compilados do *Phoenix* sem a utilização do Chronos é de 1.239.464 bytes. O tamanho total dos módulos compilados com a instrumentação do Chronos é de 1.269.336 bytes. Ou seja, a utilização do Chronos gerou um aumento de 2,41% no tamanho do *bytecode* do projeto, devido às instruções de envio de mensagens para contadores adicionadas.

4.2 *Ecto*

O *Ecto* é uma biblioteca para interação com o banco de dados utilizada pelo *Phoenix* e por outros projetos, sendo muito popular na linguagem (HEXPM, 2021a). Em sua versão mais recente, 3.7.1, possui 143 módulos e cerca de 25.794 linhas na sua base de

código, com 1.164 casos de teste em sua suíte de testes. O código fonte do *Ecto* está disponível no GitHub do projeto, <https://github.com/elixir-ecto/ecto>.

À suíte de testes do *Ecto*, em sua versão 3.7.1, foi adicionado um novo teste, mostrado no Código 4.3, com o objetivo de verificar que a função *Ecto.UUID.cast/1* não faz nenhuma chamada para o módulo *Ecto.Repo* — a fim de verificar que a função não chama nenhuma função do módulo de interação com o banco de dados.

Código 4.3: Novo teste incluído na suíte de testes do *Ecto*

```
1 test "cast does not call Repo" do
2   resultado = Chronos.profile(fn ->
3     Ecto.UUID.cast(@uuid)
4   end)
5
6   Enum.each(resultado, fn {modulo, _}, _ ->
7     assert modulo != Ecto.Repo
8   end)
9 end
```

Na linha 3 do Código 4.3, é atribuída à variável *resultado* o resultado da execução da função de *profiling* do Chronos, chamada com uma função anônima que faz uma chamada para *Ecto.UUID.cast/1* passando *@valid_uuid* como argumento (linha 4). Após isto, na linha 6, o resultado é iterado para garantir que nenhuma das funções executadas pertence ao módulo *Ecto.Repo* (linha 7).

As execuções da suíte de testes com o novo teste aconteceram com sucesso. O tempo médio de execução das rodadas de execução da suíte de testes do projeto sem o Chronos foi de 10,1 segundos, e com o Chronos foi de 10,2 segundos, um acréscimo de 0,09%.

O tamanho total dos módulos compilados do Ecto sem a utilização do Chronos é de 1.190.420 bytes. Quando o Chronos é utilizado, este tamanho aumenta para 1.217.400 bytes, um aumento de 2,26%.

4.3 *ExDoc*

A *ExDoc* é a biblioteca padrão para a escrita de documentação de projetos em Elixir (HEXPM, 2021b). Ela permite que programadores escrevam a documentação di-

retamente no código, na forma de anotações, como mostra o Código 4.4, nas linhas 1 até 5.

Código 4.4: Exemplo de anotação com o *ExDoc*

```
1 @doc """
2  A função ‘saudar/2‘‘ espera uma string e um inteiro como
3   argumentos. Ela possui o efeito colateral de imprimir
4   uma mensagem na tela.
5 """
6 def saudar(nome, idade) do
7   IO.puts("Olá, #{nome}! Você tem #{idade} anos.")
8 end
```

A *ExDoc*, em sua versão 0.25.5, possui 55 módulos e 5.301 linhas em sua base de código, com 244 casos de teste em sua suíte de testes. Seu código fonte está disponível no GitHub do projeto, em <https://github.com/elixir-lang/ex_doc>.

Código 4.5: Novo teste incluído na suíte de testes do *ExDoc*

```
1 test "empty input checks Earmark availability only once" do
2   assert %{
3         {ExDoc.Markdown.Earmark, :available?, 0} => 1
4       } =
5         Chronos.profile(fn ->
6           Markdown.to_ast("", [])
7         end)
8 end
```

À suíte de testes da *ExDoc*, foi adicionado um novo caso de teste que utiliza o Chronos, mostrado no Código 4.5. O caso de teste executa a função *Chronos.profile/1* (linha 5) passando uma função anônima que faz uma chamada para *Markdown.to_ast/2* (linha 6). Depois, é feito o *pattern matching* entre o resultado do *profiling* e um mapa contendo a chave *{ExDoc.Markdown.Earkmark, :available?, 0}* ligada ao inteiro *1*.

O objetivo do teste adicionado é verificar que a função *Markdown.to_ast/2*, quando chamada com os argumentos e *[]*, faz apenas uma execução da função *ExDoc.Markdown.Earkmark.available?/0*.

As execuções da suíte de testes com o Chronos aconteceram com sucesso. A

ExDoc foi a biblioteca com maior acréscimo na média de tempo de execução da suíte de testes, levando em média 24,5 segundos para a execução da suíte de testes sem o Chronos e 24,7 segundos para a execução com o Chronos — o que representa um acréscimo de 0,81%.

O tamanho total dos binários dos módulos do *ExDoc* sem a utilização do Chronos é de 558.440 bytes, enquanto com a utilização do Chronos, este tamanho aumenta para 598.972 bytes — um aumento de 7,25%, maior que nos outros experimentos.

4.4 Resultados

A Tabela 4.1 compara o tamanho total dos módulos compilados dos projetos, enquanto a Tabela 4.2 compara a média dos tempos de execução das suítes de teste dos projetos. Os resultados completos dos experimentos estão nos Anexos A, B, e C.

Nos experimentos, não houve um aumento significativo nos tempos médios das execuções das suítes de teste. Também não houve um aumento significativo nos tamanhos dos binários compilados dos módulos dos projetos testados.

Tabela 4.1: Tamanho total dos módulos compilados dos projetos

Projeto	*Tamanho sem o Chronos*	*Tamanho com o Chronos*	*Diferença*
Phoenix	1.239.464 bytes	1.269.336 bytes	2,41%
Ecto	1.190.420 bytes	1.217.400 bytes	2,26%
ExDoc	558.440 bytes	598.972 bytes	7,25%

Fonte: O Autor

Tabela 4.2: Média dos tempos de execução das suítes de teste dos projetos

Projeto	*Tempo sem o Chronos*	*Tempo com o Chronos*	*Diferença*
Phoenix	59,0 segundos	59,1 segundos	0,16%
Ecto	10,1 segundos	10,2 segundos	0,09%
ExDoc	24,5 segundos	24,7 segundos	0,81%

Fonte: O Autor

5 CONCLUSÃO

Nos experimentos, o Chronos não adicionou um *overhead* significativo ao tempo de execução das suítes de teste dos experimentos, e o *overhead* no tamanho dos binários dos módulos não se mostrou grande o suficiente para ser um impedimento ao *profiling*. Também não foram identificados cenários em que o Chronos não conseguiu instrumentar módulos.

Assim, o Chronos se mostrou uma alternativa viável para execução de *profiling* diretamente nos testes unitários de programas escritos em Elixir, porém algumas limitações foram encontradas e estão descritas abaixo.

Uma limitação importante encontrada é a impossibilidade do Chronos de fazer o *profiling* de bibliotecas externas ao projeto em que foi incluído. Isto é, não é possível analisar a quantidades de chamadas para uma dependência do programa analisado.

Esta limitação existe por dois motivos: a instrumentação ser feita nos pontos de entrada das funções e somente o projeto que inclui o Chronos como dependência ser compilado utilizando a instrumentação de módulos do Chronos. Diferentes soluções são possíveis para contornar a limitação:

- Passar a instrumentar as chamadas de funções ao invés de instrumentar as entradas de funções;
- Instrumentar as chamadas externas, modificando a ordem de execução do compilador para que o Chronos seja o primeiro a ser compilado e executado, antes de demais dependências;
- Incluir dinamicamente o Chronos como dependência das dependências do projeto.

Uma outra limitação encontrada foi a de que o Chronos analisa somente o número de chamadas de funções — seria interessante para o *profiling* que outras métricas também fossem analisadas, como o tempo de execução das funções, o consumo de memória, entre outras. Esta limitação é possível de ser contornada com modificações na instrumentação descrita na Seção 3.4.

Espera-se a utilização do Chronos pela comunidade, seja como um guia de referência para a injeção dinâmica de código no processo de compilação de Elixir, seja como uma alternativa aos *profilers* existentes. Com a adesão da comunidade, espera-se que a infraestrutura oferecida pelo Chronos seja estendida e aperfeiçoada para contemplar mais métricas e diferentes necessidades de *profiling*.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. New York: Cambridge University Press, 2008. OCLC: ocn180851905. ISBN 9780521880381.

CLOJURE. **Clojure - Transient Data Structures**. 2021. <https://clojure.org/reference/transients>, acesso em: 27/10/2021.

ELIXIR. **Basic Types - The Elixir programming language**. 2021. <https://elixir-lang.org/getting-started/basic-types.html>, acesso em: 02/11/2021.

ELIXIR. **Development**. 2021. <https://elixir-lang.org/development.html>, acesso em: 27/10/2021.

ELIXIR. **Introduction to Mix - The Elixir language**. 2021. <https://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html>, acesso em: 01/11/2021.

ELIXIR SCHOOL. **Basics · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/basics>, acesso em: 27/10/2021.

ELIXIR SCHOOL. **Basics · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/basics>, acesso em: 27/10/2021.

ELIXIR SCHOOL. **Basics · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/basics>, acesso em: 27/10/2021.

ELIXIR SCHOOL. **Collections · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/collections>, acesso em: 27/10/2021.

ELIXIR SCHOOL. **Control Structures · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/control_structures>, acesso em: 27/10/2021.

ELIXIR SCHOOL. **Functions · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/functions/>, acesso em: 01/11/2021.

ELIXIR SCHOOL. **Pattern Matching · Elixir School**. 2021. <https://elixirschool.com/en/lessons/basics/pattern_matching>, acesso em: 27/10/2021.

ELIXIR SCHOOL. **Testing · Elixir School**. 2021. <https://elixirschool.com/en/lessons/testing/basics#exunit-0>, acesso em: 27/10/2021.

ERLANG. **Erlang – Modules**. 2021. <https://www.erlang.org/doc/reference_manual/modules.html#module_info-0-and-module_info-1-functions>, acesso em: 01/11/2021.

ERLANG. **Erlang - Concurrent Programming**. 2021. <https://erlang.org/doc/getting_started/conc_prog.html>, acesso em: 31/10/2021.

ERLANG. **Erlang - cprof**. 2021. <http://erlang.org/doc/man/cprof.html>, acesso em: 27/10/2021.

ERLANG. **Erlang - eprof**. 2021. <http://erlang.org/doc/man/eprof.html>, acesso em: 27/10/2021.

ERLANG. **Erlang - lcnt**. 2021. <http://erlang.org/doc/man/lcnt.html>, acesso em: 27/10/2021.

ERLANG SOLUTIONS. **Which companies are using Elixir, and why?** 2020. <https://www.erlang-solutions.com/blog/which-companies-are-using-elixir-and-why-mytopdogstatus.html>, acesso em: 09/08/2020.

FOWLER, M. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley, 2019.

GNU. **GNU gprof - Introduction**. 2021. <https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_1.html>, acesso em: 27/10/2021.

GROTKER, T. **The developer's guide to debugging**. North Charleston, S.C.?: CreateSpace Independent Publishing Platform, 2012. OCLC: 816047009. ISBN 9781470185527.

HEWITT, C.; BISHOP, P.; STEIGER, R. A universal modular actor formalism for artificial intelligence. In: **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973. (IJCAI'73), p. 235–245.

HEXDOCS. **Ecto — Ecto v3.7.1**. 2021. <https://hexdocs.pm/ecto/Ecto.html>, acesso em: 27/10/2021.

HEXDOCS. **ExUnit — ExUnit v1.12.3**. 2021. <https://hexdocs.pm/ex_unit/1.12/ExUnit.html>, acesso em: 27/10/2021.

HEXDOCS. **Mox — Mox v1.0.1**. 2021. <https://hexdocs.pm/mox/Mox.html>, acesso em: 27/10/2021.

HEXDOCS. **Overview — Phoenix v1.6.2**. 2021. <https://hexdocs.pm/phoenix/overview.html>, acesso em: 27/10/2021.

HEXPM. **ecto**. 2021. <https://hex.pm/packages/ecto>, acesso em: 31/10/2021. Available from Internet: <https://hex.pm/packages/ecto>.

HEXPM. **ex_doc**. 2021. <https://hex.pm/packages/ex_doc>, acesso em: 31/10/2021. Available from Internet: <https://hex.pm/packages/ex_doc>.

HUMBLE, J.; FARLEY, D. **Continuous delivery: reliable software releases through build, test, and deployment automation**. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 9780321601919.

JORGENSEN, P. **Software testing: a craftsman's approach**. Fourth edition. Boca Raton, [Florida]: CRC Press, Taylor & Francis Group, 2014. ISBN 9781466560680.

JUNIT. **JUnit 5**. 2021. <https://junit.org/junit5/>, acesso em: 27/10/2021.

LEACH, R. J. **Introduction to software engineering**. [S.l.: s.n.], 2020. OCLC: 1220879052. ISBN 9780367575038.

RSPEC. **RSpec: Behaviour Driven Development for Ruby**. 2021. <https://rspec.info/>, acesso em: 27/10/2021.

RUBY ON RAILS. **Ruby on Rails**. 2021. <https://rubyonrails.org/>, acesso em: 27/10/2021.

SASA, J. **Elixir in action**. Second edition. Shelter Island, NY: Manning Publications Co, 2019. OCLC: on1027193404. ISBN 9781617295027.

TIOBE. **index | TIOBE - The Software Quality Company**. 2021. <https://www.tiobe.com/tiobe-index/>, acesso em: 27/10/2021.

TRICENTIS. **1.1 USD Trillion Impacted by Software Defects: A Testing Fail?** 2017. <https://www.tricentis.com/blog/1-1-trillion-in-assets-impacted-by-software-defects-a-software-testing-fail/>, acesso em: 27/10/2021.

ŚLASKI, M.; TUREK, W. Towards online profiling of Erlang systems. In: **Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang - Erlang 2019**. Berlin, Germany: ACM Press, 2019. p. 13–17. ISBN 9781450368100. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3331542.3342568>.

APÊNDICE A — EXPERIMENTO *PHOENIX*

Tabela A.1: Execuções da suíte de testes do *Phoenix* com o Chronos

N	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*
1	59.2	231317	*26*	59.1	119805	*51*	59.2	826980	*76*	59.1	419021
2	59.1	922634	*27*	59.0	673409	*52*	59.1	991062	*77*	59.1	362750
3	59.0	483238	*28*	59.1	316030	*53*	59.0	444164	*78*	59.1	823846
4	59.0	390595	*29*	59.1	158121	*54*	59.0	367843	*79*	59.0	607510
5	59.2	957387	*30*	59.0	551073	*55*	59.1	217630	*80*	59.2	582434
6	59.2	411142	*31*	59.0	494514	*56*	59.2	830673	*81*	59.1	374963
7	59.1	322727	*32*	59.2	818819	*57*	59.0	601353	*82*	59.1	490925
8	59.0	595915	*33*	59.0	235618	*58*	59.0	132731	*83*	59.2	486257
9	59.1	893714	*34*	59.1	465108	*59*	59.2	496997	*84*	59.1	760149
10	59.1	735448	*35*	59.1	489953	*60*	59.1	600216	*85*	59.0	773603
11	59.1	138062	*36*	59.1	425672	*61*	59.2	416457	*86*	59.1	472019
12	59.0	739314	*37*	59.2	311270	*62*	59.2	197949	*87*	59.1	347247
13	59.1	914324	*38*	59.2	372313	*63*	59.1	162033	*88*	59.1	371942
14	59.1	390775	*39*	59.1	182749	*64*	59.0	119199	*89*	59.2	833336
15	59.1	906172	*40*	59.0	135823	*65*	59.2	189275	*90*	59.0	424991
16	59.0	601734	*41*	59.0	245397	*66*	59.1	555169	*91*	59.0	901231
17	59.1	631842	*42*	59.1	845159	*67*	59.0	752617	*92*	59.2	527922
18	59.2	616078	*43*	59.0	336410	*68*	59.0	907922	*93*	59.1	316149
19	59.0	539915	*44*	59.0	210425	*69*	59.0	180041	*94*	59.0	140071
20	59.0	521628	*45*	59.0	324553	*70*	59.1	354529	*95*	59.1	863615
21	59.1	331422	*46*	59.0	698100	*71*	59.1	603925	*96*	59.2	241852
22	59.2	480599	*47*	59.2	915273	*72*	59.1	157548	*97*	59.1	779888
23	59.1	803488	*48*	59.1	779980	*73*	59.1	448188	*98*	59.1	561459
24	59.1	498383	*49*	59.1	652873	*74*	59.0	290057	*99*	59.1	900648
25	59.2	867999	*50*	59.2	577316	*75*	59.0	475552	*100*	59.2	259099

(Tempo em segundos)

Fonte: O Autor

Tabela A.2: Execuções da suíte de testes do *Phoenix* sem o Chronos

N	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*
1	58.9	216378	*26*	59.1	230343	*51*	59.0	229377	*76*	59.0	322740
2	59.0	944154	*27*	59.0	834589	*52*	59.0	673182	*77*	58.9	178900
3	58.9	825107	*28*	59.1	118306	*53*	59.0	168180	*78*	59.1	801986
4	59.0	169741	*29*	59.0	539064	*54*	58.9	300733	*79*	59.0	375025
5	59.0	965674	*30*	59.1	462576	*55*	58.9	910072	*80*	58.9	328704
6	59.0	459466	*31*	59.0	856357	*56*	59.1	114950	*81*	59.1	723163
7	59.1	942402	*32*	58.9	932977	*57*	58.9	367547	*82*	58.9	528884
8	59.0	980894	*33*	59.1	201208	*58*	59.0	217710	*83*	58.9	743062
9	59.1	878482	*34*	59.0	300698	*59*	59.0	610917	*84*	58.9	323813
10	58.9	748220	*35*	59.0	679196	*60*	59.1	406105	*85*	59.1	802044
11	59.1	184879	*36*	59.0	719205	*61*	59.0	822324	*86*	59.1	112953
12	58.9	307521	*37*	59.1	913756	*62*	58.9	909484	*87*	59.0	316709
13	58.9	816863	*38*	59.1	946817	*63*	59.0	140804	*88*	59.0	304992
14	59.0	544912	*39*	59.1	613605	*64*	59.0	322089	*89*	59.0	382965
15	59.0	746724	*40*	59.0	760158	*65*	59.0	474401	*90*	59.0	874640
16	59.1	603117	*41*	59.1	119606	*66*	58.9	744026	*91*	58.9	964590
17	58.9	970830	*42*	59.0	661511	*67*	59.0	122858	*92*	58.9	615576
18	58.9	817735	*43*	59.1	804481	*68*	59.1	923108	*93*	59.1	903052
19	59.0	463995	*44*	59.0	383709	*69*	59.1	536484	*94*	59.0	520639
20	59.0	836037	*45*	59.1	377392	*70*	59.0	924262	*95*	58.9	860683
21	59.0	192568	*46*	59.1	401118	*71*	59.1	922043	*96*	58.9	608160
22	58.9	112682	*47*	59.1	678045	*72*	59.0	743976	*97*	59.0	349898
23	58.9	235580	*48*	59.0	740194	*73*	59.0	872935	*98*	59.0	170426
24	59.1	875036	*49*	59.0	431622	*74*	58.9	515448	*99*	59.0	978313
25	59.0	214933	*50*	58.9	174142	*75*	59.1	386359	*100*	59.0	352142

(Tempo em segundos)

Fonte: O Autor

APÊNDICE B — EXPERIMENTO *ECTO*

Tabela B.1: Execuções da suíte de testes do *Ecto* com o Chronos

N	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*
1	10.2	648978	*26*	10.2	665209	*51*	10.1	648838	*76*	10.3	509250
2	10.2	249871	*27*	10.2	132566	*52*	10.3	283957	*77*	10.1	663368
3	10.1	395502	*28*	10.2	971637	*53*	10.1	136246	*78*	10.3	743232
4	10.3	693492	*29*	10.1	366268	*54*	10.1	799758	*79*	10.2	494981
5	10.1	381399	*30*	10.2	473453	*55*	10.3	701008	*80*	10.1	259418
6	10.3	705823	*31*	10.1	517768	*56*	10.1	166416	*81*	10.3	640211
7	10.1	490885	*32*	10.3	115800	*57*	10.3	791407	*82*	10.2	810170
8	10.2	914806	*33*	10.1	475790	*58*	10.3	802465	*83*	10.2	951178
9	10.1	782433	*34*	10.3	203746	*59*	10.3	852869	*84*	10.2	859357
10	10.2	192797	*35*	10.2	744003	*60*	10.1	409735	*85*	10.3	263213
11	10.3	880858	*36*	10.3	841146	*61*	10.2	225444	*86*	10.3	441806
12	10.1	112473	*37*	10.3	242545	*62*	10.3	410645	*87*	10.1	401759
13	10.3	549374	*38*	10.3	714121	*63*	10.3	858890	*88*	10.1	671615
14	10.2	641691	*39*	10.2	150183	*64*	10.3	723728	*89*	10.1	239752
15	10.2	769293	*40*	10.2	446274	*65*	10.2	946799	*90*	10.2	402418
16	10.2	464843	*41*	10.2	350337	*66*	10.2	319947	*91*	10.3	993294
17	10.2	328986	*42*	10.1	657347	*67*	10.3	937521	*92*	10.1	336639
18	10.3	802964	*43*	10.3	926196	*68*	10.2	759944	*93*	10.3	903657
19	10.1	600653	*44*	10.3	809553	*69*	10.1	528343	*94*	10.2	280072
20	10.3	547618	*45*	10.3	706687	*70*	10.2	222225	*95*	10.2	497526
21	10.2	586921	*46*	10.2	208041	*71*	10.2	718981	*96*	10.1	868358
22	10.3	799684	*47*	10.2	915740	*72*	10.3	877020	*97*	10.1	499140
23	10.2	334742	*48*	10.1	625490	*73*	10.3	837547	*98*	10.1	192528
24	10.2	889646	*49*	10.2	852136	*74*	10.2	540775	*99*	10.2	711210
25	10.2	168882	*50*	10.2	123242	*75*	10.1	388625	*100*	10.3	985728

(Tempo em segundos)

Fonte: O Autor

Tabela B.2: Execuções da suíte de testes do *Ecto* sem o Chronos

N	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*
1	10.0	563664	*26*	10.1	954463	*51*	10.1	335206	*76*	10.1	503803
2	10.0	185693	*27*	10.1	581586	*52*	10.1	326730	*77*	10.2	831958
3	10.1	480550	*28*	10.1	820513	*53*	10.0	156709	*78*	10.0	926805
4	10.1	392029	*29*	10.1	261875	*54*	10.2	700887	*79*	10.1	583298
5	10.0	953711	*30*	10.0	187967	*55*	10.0	390403	*80*	10.1	601029
6	10.1	756221	*31*	10.0	879710	*56*	10.2	738867	*81*	10.1	969592
7	10.0	715459	*32*	10.1	719435	*57*	10.1	961714	*82*	10.1	331732
8	10.1	735869	*33*	10.1	490978	*58*	10.1	608064	*83*	10.2	576877
9	10.0	391197	*34*	10.1	528083	*59*	10.1	395944	*84*	10.2	831128
10	10.2	599394	*35*	10.0	825135	*60*	10.2	299951	*85*	10.2	391567
11	10.1	760273	*36*	10.1	380766	*61*	10.1	762980	*86*	10.1	360517
12	10.1	469573	*37*	10.2	856631	*62*	10.2	989552	*87*	10.0	911981
13	10.1	794388	*38*	10.2	913974	*63*	10.1	595993	*88*	10.2	344588
14	10.2	599170	*39*	10.0	893559	*64*	10.0	789891	*89*	10.2	407784
15	10.0	367555	*40*	10.1	533353	*65*	10.1	409136	*90*	10.1	232656
16	10.1	887137	*41*	10.2	499755	*66*	10.1	906304	*91*	10.1	508685
17	10.2	273067	*42*	10.0	687785	*67*	10.1	605577	*92*	10.1	889029
18	10.1	408401	*43*	10.1	146924	*68*	10.2	446101	*93*	10.2	314367
19	10.0	421105	*44*	10.1	179674	*69*	10.0	175556	*94*	10.1	189674
20	10.1	918594	*45*	10.1	787971	*70*	10.2	277945	*95*	10.2	872800
21	10.1	269900	*46*	10.2	181299	*71*	10.0	568844	*96*	10.1	527778
22	10.1	452575	*47*	10.0	299504	*72*	10.0	290027	*97*	10.1	516946
23	10.0	326632	*48*	10.2	734715	*73*	10.1	122245	*98*	10.1	220786
24	10.0	981968	*49*	10.1	852129	*74*	10.0	939352	*99*	10.2	843867
25	10.2	296192	*50*	10.1	336358	*75*	10.1	589674	*100*	10.1	964912

(Tempo em segundos)

Fonte: O Autor

APÊNDICE C — EXPERIMENTO *EXDOC*

Tabela C.1: Execuções da suíte de testes do *ExDoc* com o Chronos

N	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*
1	24.6	575397	*26*	24.7	234832	*51*	24.6	318809	*76*	24.8	561771
2	24.7	624783	*27*	24.6	186436	*52*	24.7	858433	*77*	24.8	916897
3	24.7	715235	*28*	24.8	127449	*53*	24.8	421329	*78*	24.6	976614
4	24.7	229073	*29*	24.8	492380	*54*	24.6	374094	*79*	24.8	987308
5	24.7	701851	*30*	24.7	952200	*55*	24.7	692769	*80*	24.6	489004
6	24.7	723497	*31*	24.7	221243	*56*	24.6	596470	*81*	24.8	461183
7	24.6	664522	*32*	24.8	556823	*57*	24.8	759375	*82*	24.6	453454
8	24.8	577410	*33*	24.8	755894	*58*	24.7	661886	*83*	24.6	252372
9	24.8	595993	*34*	24.7	834157	*59*	24.6	185294	*84*	24.7	858962
10	24.6	962263	*35*	24.8	244652	*60*	24.6	479930	*85*	24.7	627942
11	24.7	582682	*36*	24.7	856152	*61*	24.8	443707	*86*	24.7	308412
12	24.7	343746	*37*	24.7	933531	*62*	24.8	427707	*87*	24.7	867807
13	24.6	374674	*38*	24.7	537777	*63*	24.7	874224	*88*	24.6	314529
14	24.7	381977	*39*	24.7	508774	*64*	24.7	678324	*89*	24.7	515501
15	24.8	824446	*40*	24.6	353793	*65*	24.8	978410	*90*	24.7	220303
16	24.8	892882	*41*	24.7	639701	*66*	24.7	223529	*91*	24.8	736673
17	24.7	173742	*42*	24.6	846233	*67*	24.8	451753	*92*	24.7	387069
18	24.7	970968	*43*	24.7	836893	*68*	24.6	153633	*93*	24.7	997282
19	24.7	519358	*44*	24.7	440913	*69*	24.8	226886	*94*	24.7	762570
20	24.7	589716	*45*	24.6	743040	*70*	24.6	379321	*95*	24.8	469060
21	24.7	808218	*46*	24.7	509665	*71*	24.7	913274	*96*	24.7	705066
22	24.7	281817	*47*	24.8	663243	*72*	24.7	596277	*97*	24.7	849805
23	24.7	495807	*48*	24.8	666370	*73*	24.7	361326	*98*	24.7	677767
24	24.7	628386	*49*	24.7	960273	*74*	24.7	504059	*99*	24.7	849388
25	24.8	600810	*50*	24.6	841183	*75*	24.7	602134	*100*	24.6	854008

(Tempo em segundos)

Fonte: O Autor

Tabela C.2: Execuções da suíte de testes do *ExDoc* sem o Chronos

N	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*	*N*	*Tempo*	*Seed*
1	24.5	972254	*26*	24.5	796603	*51*	24.4	271931	*76*	24.6	356029
2	24.6	228366	*27*	24.6	606028	*52*	24.5	171951	*77*	24.6	482227
3	24.5	386213	*28*	24.4	431404	*53*	24.5	997956	*78*	24.4	869905
4	24.5	472489	*29*	24.4	192094	*54*	24.5	778002	*79*	24.6	789564
5	24.5	788090	*30*	24.4	943156	*55*	24.5	340347	*80*	24.5	157887
6	24.5	977264	*31*	24.5	548264	*56*	24.5	339234	*81*	24.5	521828
7	24.5	467376	*32*	24.6	868191	*57*	24.5	800289	*82*	24.4	862034
8	24.4	910035	*33*	24.4	183346	*58*	24.5	943240	*83*	24.4	579770
9	24.6	770135	*34*	24.6	402617	*59*	24.6	748413	*84*	24.4	284131
10	24.4	399738	*35*	24.6	997214	*60*	24.5	639715	*85*	24.5	772337
11	24.4	418615	*36*	24.6	186114	*61*	24.4	165428	*86*	24.5	145579
12	24.5	299648	*37*	24.6	813606	*62*	24.4	270515	*87*	24.5	473509
13	24.4	992150	*38*	24.5	349694	*63*	24.6	298655	*88*	24.5	885097
14	24.4	640805	*39*	24.5	431871	*64*	24.5	952968	*89*	24.4	548379
15	24.4	152330	*40*	24.4	357321	*65*	24.5	945256	*90*	24.5	193657
16	24.6	636043	*41*	24.6	202691	*66*	24.4	192886	*91*	24.5	164427
17	24.5	695881	*42*	24.6	119795	*67*	24.6	708560	*92*	24.4	848492
18	24.5	912437	*43*	24.4	878632	*68*	24.4	683221	*93*	24.5	513445
19	24.5	727167	*44*	24.5	449938	*69*	24.5	829695	*94*	24.5	704623
20	24.6	390854	*45*	24.5	592055	*70*	24.6	191261	*95*	24.5	483162
21	24.6	891492	*46*	24.5	482924	*71*	24.5	373940	*96*	24.4	251354
22	24.4	272802	*47*	24.5	187219	*72*	24.5	537000	*97*	24.5	259186
23	24.5	209474	*48*	24.6	952759	*73*	24.5	534223	*98*	24.5	501122
24	24.4	419204	*49*	24.6	140529	*74*	24.5	504620	*99*	24.5	127456
25	24.4	151222	*50*	24.4	946947	*75*	24.6	758199	*100*	24.5	743201

(Tempo em segundos)

Fonte: O Autor