Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# LambdaTransformer: Uma Solução para o Tratamento de Expressões Lambda no JimpleFramework

Luisa Sinzker Fantin

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# LambdaTransformer: Uma Solução para o Tratamento de Expressões Lambda no JimpleFramework

Luisa Sinzker Fantin

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof.a Dr.a Edna Dias Canedo

Me. Walter Lucas

Prof. Dr. João José Costa Gondim
Coordenador do Curso de Engenharia da Computação

Brasília, 09 de junho de 2021

# Dedicatória

À minha avó, Jesi. Sinto sua falta.

# Agradecimentos

Agradeço meus pais, Cristina e Fidelis, por sempre fazer o melhor para me preparar emocionalmente, moralmente e intelectualmente para os desafios da vida.

Ao meu mentor Prof. Dr. Rodrigo Bonifácio de Almeida que me guiou durante minha jornada acadêmica e me apresentou desafios, me impulsionando profissionalmente.

À minha amiga, Juliana Mayumi Hosoume, que sempre me ajudou e me manteve focada.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

# Resumo

Análise de fluxo de dados é um tipo de análise estática que permite a coleta de informações sobre o comportamento dos dados de um programa em tempo de execução sem que esse código seja executado. Isso é feito com o uso de ferramentas como Grafos de Controle de Fluxo, CFG, uma representação de programa que facilita a visualização do comportamento do código e o desenvolvimento de análises. Códigos Java possuem *bytecode* baseado em pilha o que torna a criação de CFGs mais difícil. *Frameworks* como o Soot utilizam Representações Intermediárias, RIs, com estruturas mais amigáveis a criação do CFGs e escrita de análises para analisar códigos Java. O Jimple Framework implementa sua própria versão de Jimple, a principal RI de Soot, utilizando a linguagem de meta-programação Rascal com o intuito de tornar a escrita de análises menos verbosas em comparação ao Soot. A descompilação de *bytecode* Java da origem a código Jimple que pode passar por refinamentos com o intuito de tornar o código mais legível ou simplificar a realização de alguma análise. A partir de Java 8, expressões lambda foram introduzidas a linguagem, essas expressões são traduzidas em *bytecode*como instruções *invokedynamic*. Como todas as instruções presentes no *bytecode* Java, o *Jimple Framework* deve oferecer ferramentas que permitam a realização de análises estáticas, como análise de fluxo de dados, em códigos que possuam instruções esse tipo, porém esse tipo de instrução faz uso de ferramentas dentro da JVM que escondem o caminho dos dados, dessa forma impossibilitando a análise de fluxo e criação do CFG. O *Jimple Framework* deve refinar código Jimple para permitir análises de códigos com esse tipo de instruções. Este trabalho descreve o processo de desenvolvimento do LambdaTransformer um módulo do *Jimple Framework* capaz de transformar instruções *invokedynamic* em *invokestatic* com o uso de funções de travessia de árvore e casamento de padrões.

**Palavras-chave:** Expressões Lambda, Java, Rascal, JVM, Soot Framework, Jimple Framework, invokedynamic, Visiting, Casamento de Padrões, Travessia de Árvore, Análise Estática, Análise de Fluxo de Dados, Otimização de códigos

# Abstract

Dataflow analysis is a type of static analysis that allows gathering information about the behavior of a program's data at runtime without executing the code. This is done using tools such as Control Flow Graphs, CFG, a program representation that facilitates the visualization of code behavior and the development of analysis. Java code has stack-based *bytecode* which makes CFG creation more difficult, sometimes impossible. *Frameworks* like Soot use Intermediate Representations, IRs, with more user-friendly structures for creating CFGs and writing analyzes to analyze Java code. *Jimple Framework* implements its own version of Jimple, Soot's main IR, using the Rascal meta-programming language in order to make writing analysis less verbose compared to Soot. The decompilation of Java *bytecode* creates a Jimple code that can undergo refinements in order to make the code more readable or simplify performing some analysis. As of Java 8, lambda expressions were introduced to the language, these expressions are translated into *bytecode* as *invokedynamic* instructions. Like all instructions present in Java *bytecode*, *Jimple Framework* must offer tools that allow static analysis, such as data flow analysis, in codes that have instructions of this type, but this type of instruction makes use of tools within the JVM that hide the data flow, thus making flow analysis and CFG creation impossible. The *Jimple Framework*must refine the Jimple code to allow code parsing with this type of instructions. This work describes the process of developing LambdaTransformer, a *Jimple Framework* module capable of transforming *invokedynamic* instructions into *invokestatic* instructions using tree traversal and pattern matching functions.

**Keywords:** Lambda Expressions, Java, Rascal, JVM, Soot Framework, Jimple Framework, invokedynamic, Visiting, Pattern Matching, Tree Traversal, Static Analysis, Data Flow Analysis, Code Optimization

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Abreviaturas e Siglas

**ADT** Algebraic Data Type.

**AST** Abstract Syntax Tree.

**BSM** Bootstrap Method.

**CFG** Control Flow Graph.

**CID** ClassOrInterfaceDeclaration.

**CS** CallSite.

**GFC** Grafo de Fluxo de Controle.

**JIT** just-in-time.

**JVM** Máquina Virtual Java.

**MH** MethodHandle.

**RCP** Rich Client Platform.

**RI** Representação Intermediária.

# Capítulo 1

# Introdução

Java oferece recursos que facilitam o desenvolvimento de aplicações e sistemas, como independência de plataforma, segurança de execução e orientação a objetos. Esses recursos, porém, vem as custas de desempenho. Quando comparados com outras linguagens como C e C++ programas Java se mostram mais lentos, otimizações de código se tornaram necessárias para mitigar os custos de desempenho.

Um dos recursos mencionados anteriormente, independência de plataforma, está diretamente ligada ao fato de códigos Java serem executados ou interpretados, não em *hardware*, mas em *software*, pela Máquina Virtual Java (JVM). Isto torna essencial não somente as otimização aplicadas ao código fonte na máquina virtual, mas também a otimização da máquina em si. Esse passa a ser, ao menos nos primórdios da existência da linguagem [1], época em que ainda não haviam compiladores com otimizações sofisticadas, um dos principais motivos por trás de sua lentidão. Com a evolução da linguagem, seus compiladores e interpretadores ficaram mais eficientes, códigos Java rapidamente passaram a ser submetidos à otimizações robustas em tempo de execução com compiladores *just-in-time (JIT)*, compiladores adaptativos como *Hotspot*™, entre outros [2].

Soot [3] é *framework open source* que fornece um conjunto de Representações Intermediárias (RIs) e APIs que facilitam o desenvolvimento de otimização de *bytecode* Java [4], assim como ferramentas para a visualização e análise de códigos. O *framework* foi originalmente desenvolvido com o intuito de permitir que pesquisadores experimentassem com análises e otimizações de *bytecode* Java.

Há diversas razões para otimizar ao nível de *bytecode* [4]. Primeiramente, o *bytecode* otimizado pode ser executado usando qualquer implementação de JVM (interpretador, JIT, adaptativo), ou pode ser usado como *input* para um compilador de *bytecode* → C ou de *bytecode* → código nativo. Assim, o aumento de performance ao todo se dá devido tanto as otimizações estáticas de *bytecode*, quanto as otimizações realizadas em tempo de execução pela máquina virtual. Em segundo lugar, uma variedade de compiladores diferentes para uma variedade de

linguagens (Ada, Scheme, Fortran, Eiffel, etc.) agora produzem *bytecode* Java como seu código alvo. Dessa forma, as técnicas de otimização agora podem ser aplicada como um *backend* para todos esses compiladores.

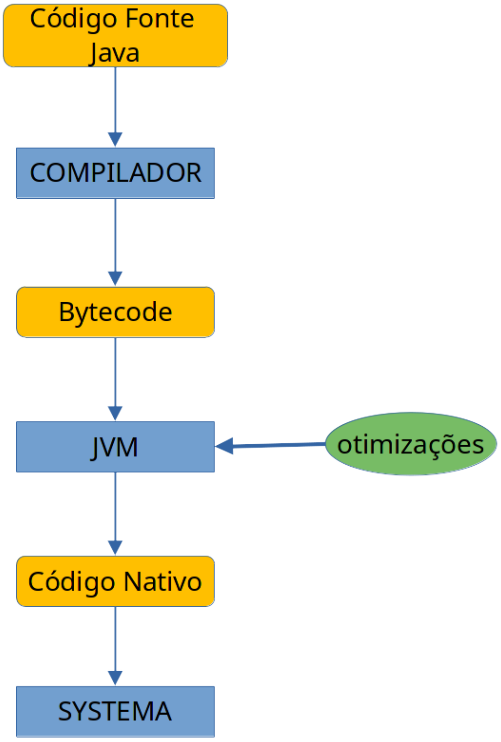


Figura 1.1: Fluxo Compilador Genérico

Técnicas de análise de programas oferecem ferramentas e diretrizes para otimização, verificação da corretude de algoritmos, identificação de falhas de segurança e áreas de vulnerabilidade [5]. Estas análises podem ser estáticas ou dinâmicas, sem e com execução do programa, respectivamente. Com o passar dos anos o Soot se tornou um dos, se não o *framework*mais utilizado para análises estáticas de código Java [6, 7].

Análise de fluxo de dados é uma técnica de coleta de dados sobre os valores de conjunto de variáveis de um segmento de código em diferentes pontos de um programa. É um tipo de análise estática que usa Grafo de Fluxo de Controle, em inglês *Control Flow Graph (CFG)*, para determinar as partes de um programa em que um determinado valor pode se propagar. Compiladores usam as informações coletadas pelas análises de fluxo de dados para otimizar o código a ser executado [8]. Algumas técnicas de análise de fluxo de dados, são: *Reaching definition, variable liveness analysis, available expression analysis, constant propagation analysis*, entre outros.

Soot facilita a criação do CFG transformando *bytecode* em código Jimple, uma RI tipada de três endereços com instruções simples. Essa transformação se faz necessária pois em códigos baseados em pilha, neste caso *bytecode* Java, suas expressões não são explicitas, dificultando a criação do CFG e tornando sua análise trabalhosa. Sua natureza não tipada também limita

e dificulta alguns tipos de análise que esperam informações explicitas sobre os tipos de suas variáveis, além de possuir instruções que tornar informações de contexto possivelmente inacessíveis. Estudos experimentais suficientes foram realizados para validar pode se transformar *bytecode* em código Jimple e código Jimple de volta para *bytecode* sem perder performance e que otimizações feitas em código Jimple resultam em otimizações no *bytecode* final [4].

Soot possui alguns entraves, sua API é complexa e sua implementação e análises são feitas em Java, sendo esta uma linguagem de proposito geral e não voltada para análise de códigos, o que torna suas análises verbosas e de difícil compreensão. Tendo em vista as dificuldades causadas pela implementação das análises em Java, foi proposta a criação do *Jimple Framework*, a implementação de um novo *framework* de análise de código Java que também usa Jimple como RI, mas usando a linguagem de meta-programação Rascal no lugar de Java para refinar o código Jimple, assim como implementar as análises de código.

Para a implementação da RI Jimple, uma das coisas que o *framework* deve fazer é identificar e tratar os diferentes tipos de chamadas a métodos a partir da descompilação do *bytecode*. Uma dessas chamadas é a `invokedynamic`, utilizada para, entre outras coisas, traduzir expressões lambda, que passaram a ser aceitas pela linguagem a partir de Java 8. Também conhecida como chamada `indy`, ela permite que métodos sejam chamados pelo nome sem que o programador determine o local de sua implementação [9].

## 1.1 Problema

O *Jimple Framework* é capaz de, a partir de um arquivo .class identificar e traduzir o *bytecode* referente a chamadas `indy` para a representação Jimple no formato de Árvore Sintática Abstrata, em inglês *Abstract Syntax Tree (AST)* porém, em versões anteriores, não possuía os refinamentos necessários para tratar a chamada de uma forma que permitisse sua integração nas análises do código. Essencialmente a instrução é traduzida para a representação Jimple mas acaba abstraindo e embaralhando informações sobre o fluxo de controle do código de tal forma que inviabiliza a análise do código. A solução paliativa para lidar com esse problema antes desse trabalho era de ignorar os blocos de código com esse tipo de instrução.

## 1.2 Objetivo

Esse trabalho tem como objetivo principal refinar a AST da representação Jimple do *Jimple Framework* aplicando transformações de programa para deixar explicito o fluxo dos dados de programas com expressões lambda. Para isso outros objetivos específicos precisam ser alcançados:

**OE1** Entender como o Soot trata expressões lambda

**OE2** Entender quais informações extrair da AST não refinada

**OE3** Aplicar transformações de programa na AST

## 1.3 Metodologia

Primeiramente foi estudado análise de programas, mais especificamente análise estática de programas focando em análise de fluxo de dados, ao mesmo tempo o Soot *Framework* foi estudado focando em sua Representação Intermediária Jimple. Foi estudado também as peculiaridades do tratamento de expressões lambda pela JVM e, consequentemente, o funcionamento das instruções *invokedynamic*. Por fim foi preciso estudar a linguagem de meta-programação Rascal, usada na implementação dos refinamentos no *Jimple Framework*, e conceitos transformação de programas e padrões de projeto como *visting design pattern* e *pattern matching*.

Após a revisão da teoria e funcionamento da linguagem de implementação, iniciou-se a etapa de investigação do tratamento das instruções *invokedynamic* pelo Soot *Framework* e comparação com o estado, até então, de seu tratamento pelo *Jimple Framework*. Para isso foram criadas amostras de código Java com a presença de expressões lambda, inicialmente foi analisado o exemplo mais simples e a medida que seu tratamento foi sendo compreendido exemplos mais complexos foram analisados para determinar peculiaridades em seu tratamento e evidenciar equívocos em compreensões anteriores da transformação. Uma vez que o entendimento da solução Soot foi considerada satisfatória, as informações necessárias para a replicação da solução foram mapeadas no código Jimple bruto, não refinado, do *Jimple Framework*e então extraídas e usadas na implementação do refinamento pretendido.

O trabalho foi desenvolvido com o sistema operacional Arch Linux, kernel versão 5.12.1-arch-1, com processador Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz quad-core, 8 GiB de memória RAM e placa de vídeo NVidia GeForce GTX 1050.

Em todas as etapas, foi utilizado Maven com java-8-openjdk, para isso foi preciso configurar o PATH da seguinte forma:

```
$ archlinux-java set java-8-openjdk
```

Para facilitar o uso do Soot versão 4.2.1 foi criado um *alias* rodando o seguinte comando:

```
$ alias soot='java -jar ~/dev/soot-4.2.1-jar-with-dependencies.jar'
```

O IDE, Ambiente de Desenvolvimento Integrado em português, usado foi o Eclipse Rich Client Platform (RCP), versão 2019-03-R, para que fosse possível usar alguns *plugins* necessários para o desenvolvimento, sendo eles rascal-0.18.0 e lombok-1.18.12. Uma vez importado para o Eclipse como projeto Maven, é possível rodar os comandos `mvn install` e `mvn test`, o último gera os arquivos .class dos códigos de *sample* que serão descompilados e usados para a realização das análises e testes.

## 1.4 Estrutura

Dividido em cinco capítulos, este trabalho descreve o processo de formulação do plano para o tratamento de `indy calls` no *Jimple Framework* por meio de refinamentos aplicados ao código Jimple oriundo da descompilação de *bytecode* Java, assim como sua implementação e verificação. Este, Capítulo 1, apresenta o contexto e motivação por trás da concepção do *Jimple Framework* e aponta o problema a ser resolvido. No Capítulo 2 os conceitos necessários para acompanhar o desenvolvimento da solução são discutidos, assim como as ferramentas utilizadas na implementação. O Capítulo 3 exemplifica o funcionamento do *Jimple Framework*, expondo detalhadamente as estruturas utilizadas na implementação do *LambdaTransformer* e como a AST é manipulada para obter as informações necessárias para fazer a transformação. A implementação é descrita no Capítulo 4, assim como sua verificação. Por fim, o Capítulo 5 conclui do trabalho.

# Capítulo 2

# Fundamentação Teórica

Esse capítulo procura apresentar as principais motivações, técnicas, ferramentas e contextos necessárias para compreender a implementação das transformações ao código Jimple do *Jimple Framework*. A Seção 2.1 explica o que é análise de fluxo de dados e *Reaching Definition Analysis*. Seção 2.2 fala sobre expressões lambda e como a JVM usa a instrução dynamicinvoke para lidar com elas. Seção 2.3 fala da principal RI do Soot Framework e os motivos de sua re-implementação no *Jimple Framework*. Seção 2.4 faz uma introdução a linguagem de meta-programação Rascal e lista algumas das principais estruturas usadas na implementação desse trabalho.

## 2.1 Análise de Fluxo de Dados

Análise de fluxo de dados é uma técnica de análise estática que captura informações sobre variáveis, seus valores e onde são usadas, rastreando o fluxo dos dados de um programa ou parte de um programa sem executá-lo, analisando o comportamento de expressões e atribuições de variáveis[10], de forma que as informações coletadas representem precisamente o comportamento do programa em tempo de execução. Esse comportamento é comumente expresso na forma de um grafo de controle, CFG, no qual os nós representam linhas do programa, ou blocos de linhas sem quebra de fluxo, e os vértices descrevem como e quais informações são passadas para o próximo nó.

Originalmente concebida com o objetivo de auxiliar na otimização de programas, análise de fluxo de dados possui diversas aplicações, Khedker, Sanyal e Karkare [10], descrevem as principais delas como sendo:

- Determinar a validade de um programa, em relação a uma propriedade específica, como verificação de tipo e uso de variáveis não inicializadas;

- Entender o comportamento de um programa para auxiliar na sua depuração, manutenção ou testagem de um programa;
- Transformação de programas [1], essa aplicação é comum para a criação de representações do mesmo programa para facilitar a extração de informações específicas ou sua visualização, por exemplo. O Framework Soot oferece uma variedade de representações intermediárias, RI, sendo a representação Jimple a mais importante para este trabalho sendo a base para o Jimple Framework Capítulo 3.

A implementação de uma análise de fluxo de dados pode mudar drasticamente, dependendo do escopo em que é aplicada, principalmente entre análises intra e inter-procedurais. Considerando programas como sendo um conjunto de estruturas e sub-estruturas, pode se dizer que análises procuram descobrir informações sobre fatos do programa, podendo, até, estender a estrutura analisada ao programa todo, dependendo do tamanho do programa.

O escopo de uma análise pode ser local, global ou inter-procedural, análises locais são limitadas a uma sequencia arbitrária de instruções dentro de um bloco básico[2]. Análises globais, chamadas comumente de análises intra-procedurais, analisam uma função ou processo por inteiro mas não se propagam para dentro de chamadas de funções que ocorrem dentro do bloco analisado. Já análises *inter-procedurais* se estendem entre as funções que são invocadas dentro do código analisado, aumentando sua complexidade, o que é facilmente observado quando comparando a estrutura de CFGs de análises intra e inter-procedurais, exigindo técnicas adicionais para a implementação dessas análises.

Ao projetar uma análise é preciso definir mais algumas propriedades que devem ser levadas em consideração para coletar as informações sobre o fluxo, essas propriedades de interesse são chamadas de sensibilidades. Análises sensíveis ao fluxo levam em consideração a ordem dos comandos, dessa forma, precisam de ferramentas de controle para armazenar as informações em cada ponto do programa, como, por exemplo, um grafo de controle de fluxo. Análises de fluxo de dados são por natureza sensíveis ao fluxo. Análises sensíveis ao caminho distinguem os caminhos feitos por um dado para chegar a determinado ponto do programa, dependendo das instruções de controle de fluxo presentes. Apenas análises inter-procedurais podem ser sensíveis ao contexto já que podem levar em consideração o contexto do programa no local da chamada da função quando analisando a função chamada. Análises intra-procedurais são sempre insensíveis ao contexto já que não saem da função/processo analisado, não mudando de contexto.

[1] Transformar um programa consiste em qualquer operação que gera um programa a partir de outro (e.g., código fonte Java em Jimple)

[2] Bloco básico é uma sequência de comandos sem estruturas de controle de fluxos, como *branches* e *jumps.*

### 2.1.1 Grafos de fluxo de controle

Control Flow Graphs (CFGs) são grafos direcionados com vértices representando comandos singulares ou blocos básicos e suas arestas representando o fluxo do código. Blocos básicos são agrupamentos de comandos sempre executados juntos de forma sequencial, sem instruções de fluxo de controle exceto, possivelmente, a última instrução. A análise coleta fatos sobre cada comando e relaciona esses fatos ao resto do programa, quando a estratégia de blocos básicos é usada, uma análise de fluxo local precisa ser feita uma vez para cada bloco, em seguida é possível realizar a análise global ou intra-procedural do código como um todo.

Comparar os efeitos entre os comandos requer propagar as informações de fluxo de dados coletadas de vértice em vértice, essa propagação pode ser feita de forma favorável a direção do fluxo de controle ou contrário a ele. Quando a propagação da informação coletada para a análise é favorável ao fluxo de controle o fluxo da análise é chamado *forward flow*, quando essa propagação é contrária ao fluxo de controle, *backward flow*.

Para compreender análises de fluxo de dados é importante entender as estruturas usadas para passar as informações entre os blocos e como elas se relacionam. Considerando um bloco $n$, o conjunto de blocos *predecessores* de $n$ são denotados *pred(n)* e o conjunto dos blocos sucessores, *succ(n)*. Essas relações são estabelecidas pelas arestas do CFG, a figura 2.1 ilustra um grafo simples representando o código 2.1, nesse grafo o bloco $n_2$ possui duas arestas conectadas a ele: $n_1 \rightarrow n_2$ e $n_2 \rightarrow n_3$, então $n_1$ é um predecessor de $n_2$ e $n_3$ um sucessor de $n_2$.

Os pontos de entrada e saída de cada bloco são denotadas *Entrada(n)* e *Saída(n)*, elas representam os possíveis estados do programa antes e depois da execução do bloco, respectivamente. As informações sobre o fluxo de dados associadas a esses pontos são os conjuntos denominados $In_n$ e $Out_n$, esses conjuntos armazenam *fatos* sobre o programa no ponto determinado, $In_n$ é o conjunto de *fatos* válidos imediatamente antes da execução do bloco e $Out_n$ o conjunto de *fatos* válidos imediatamente após a execução. Em análises com *forward flow*, o fluxo das informações ocorre atribuindo o conjunto $Out_{pred(n)}$ ao conjunto $In_n$, ou seja, o conjunto de *fatos* válidos que entram no bloco é o conjunto de *fatos* que saem de seus *predecessores* e o contrário é verdade para análises com *backward flow*, ou seja, o conjunto $In_n$ é atribuído ao conjunto $Out_{pred(n)}$. Na Figura 2.1, usando análise com *forward flow* $In_{n2}$ = a1, sendo a1 a declaração `x = 10` definida no bloco $n_1$ faz parte do conjunto $Out_{n1}$. Os *fatos* gerados no bloco analisado estão contidos no conjunto $Gen_n$ e os *fatos* que passam a ser inválidos ao final do bloco em $Kill_n$.

Código 2.1: Código simples, sem instruções de controle de fluxo

```
1  int foo() {
2      int x = 10;
3      int y = 2;
4
5      x = x*y;
6
7      return x;
8  }
```

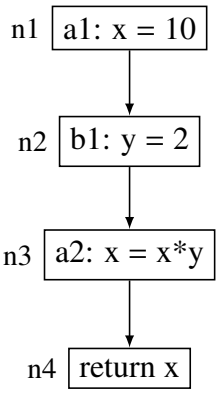


Figura 2.1: Grafo do Código 2.1

### 2.1.2 Reaching Definition Analysis

Análise de *reaching definition* nada mais é do que determinar a existência de um caminho pelo qual a atribuição de um valor a uma variável atinge um determinado ponto de um código sem sofrer uma nova atribuição. Uma mesma variável pode ter seu valor sobrescrito inúmeras vezes durante um programa, dessa forma, para a execução da análise, rótulos são designados para representar a definição das variáveis em cada ponto do código.

A definição $d_i$ de uma variável x, *alcança* o ponto $p$ do programa se existir algum caminho da definição $d_i$ até o ponto $p$, sem que haja outra definição de x dentro desse caminho. A propagação da análise segue a direção do fluxo de controle, sendo assim, a análise é de *forward flow* e as equações de fluxo que descrevem esse tipo de análise são:

$$IN_n = \bigcup_{p \in pred(n)} OUT_p$$
$$OUT_n = (IN_n - Kill_n) \cup Gen_n$$

Código 2.2: Código com estruturas de controle de fluxo

```
1    public static void main(String[] args) {
2        int x = foo();
3        int y = 4;
4
5        while(x<y) {
6            x++;
7        }
8        System.out.println(x);
9    }
```

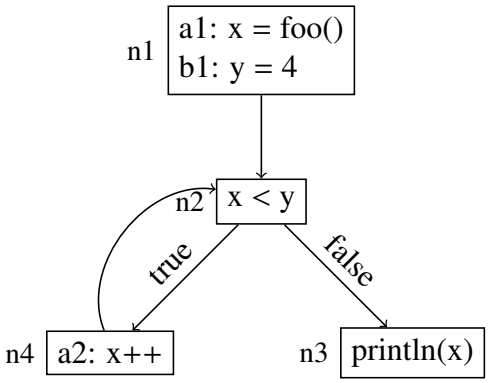


Figura 2.2: Grafo do Código 2.2

| Bloco | Informações Locais | | Informações Globais | | | |
|---|---|---|---|---|---|---|
| | | | Iteração 1 | | Iterações seguintes | |
| | Gen | Kill | In | Out | In | Out |
| n1 | {a1, b1} | {a0, b0} | {a0, b0} | {a1, b1} | | |
| n2 | {} | {} | {a1, b1} | {a1, b1} | {a2, b1} | {a2, b1} |
| n3 | {} | {} | {a1, b1} | {a1, b1} | {a1, a2, b1} | {a1, a2, b1} |
| n4 | {a2} | {a1} | {a1, b1} | {a2, b1} | {a2, b1} | {a2, b1} |

Tabela 2.1: Reaching Definitions

.

## 2.2 Expressões Lambda com Invokedynamic

### 2.2.1 Expressões Lambda

A partir de Java 8 expressões lambda, também chamadas de funções anônimas ou de primeira classe, foram incorporadas a linguagem. Alguns dos benefícios oferecidos por expressões lambda são a introdução de aspectos de paradigmas de programação funcional e formas alternativas mais enxutas de implementar interfaces, por exemplo. O Código 2.3 implementa o comparador usando *anonymous inner class*, o mesmo resultado pode ser obtido usando expressões lambda, Código 2.4, porém reduzindo o número de linhas necessárias para a implementação do comparador de 6 para 1 e dispensando a criação de outro *.class*.

Java não possui um tipo *function*, então a forma encontrada para simular esse tipo é implementar interfaces com apenas um método, `Comparator`, `Runnable` e `Consumer` são algumas delas, essas interfaces são objetos mas podem ser tratadas como funções, podendo ser chamadas de *interfaces funcionais* [11]. O tipo de uma expressão lambda será, então, o tipo da *interface funcional* que estiver implementando e dessa forma é possível atribuí-la a uma variável ou até ser passá-la como argumento de outra função.

Código 2.3: AnonyImplementation.java

```
1  package samples;
2  import java.util.Comparator;
3
4  public class AnonyImplementation {
5      public static void main(String args[]) {
6          Comparator<String> stringComparator = new Comparator<String>(){
7              @Override
8              public int compare(String s1, String s2) {
9                  return s1.compareTo(s2);
10             }
11         };
12         System.out.println(stringComparator.compare("hello", "world"));
13     }
14 }
```

Código 2.4: SimpleLambdaExpression.java

```java
package samples;
import java.util.Comparator;

public class SimpleLambdaExpression {
    public static void main(String args[]) {
        Comparator<String> c = (l, r) -> l.compareTo(r);
        System.out.println(c.compare("hello", "world"));
    }
}
```

### Closures

*Closures* são funções anônimas que encapsulam variáveis locais em seu escopo e utilizam-as em sua execução sem que estas tenham sido passadas como argumento, isso torna seu resultado dependente do estado das variáveis encapsuladas. Também chamadas de *Stateful Lambda Expressions*, em java, uma variável encapsulada deve ser *effectively final*, não podem ser alteradas após seu encapsulamento, os objetos encapsulados estão ligados a função e permanecem inalterados pelo tempo que essa função persistir. No Código 2.5, `incBy` é um *closure* que encapsula `i`.

Código 2.5: IncClosure.java

```java
package samples;

interface Incrementable {
    int inc(int a);
}
public class IncClosure {
    public static void main(String[] args) {
        int i = 1;
        Incrementable incBy = (a) -> (a+i);
        System.out.println(incBy.inc(1));
    }
}
```

### 2.2.2 Invokedynamic

A instrução `invokedynamic`, ou "Indy", foi a forma escolhida para tratar expressões lambda [12]. A instrução foi adicionada no Java 7, sua motivação original era de melhorar o suporte da JVM a linguagens dinâmicas e encontrar uma forma melhor de lidar com tipagem dinâmica do que *Reflection*. A instrução é usada para chamar métodos, sem determinar estaticamente onde a implementação do método chamado deve ser encontrada, isso torna a ligação entre a implementação de um método e sua chamada não evidente quando avaliando *bytecode*. Isso pode ser evidenciado quando comparando o *bytecode*dos códigos 2.4 e 2.3, Apêndice A e Apêndice B, descompilados usando `javap -c -p -v`.

Apesar de produzirem o mesmo resultado, os códigos 2.4 e 2.3 não são equivalentes para o compilador. O programa que utiliza *anonymous inner class*, tem apenas o método `main`, linha 62, e seu primeiro comando, linha 67, é `0: new #2`, que cria uma nova classe, essa classe terá seu próprio *bytecode* separado do arquivo principal, e ela que possui a implementação do método `compare` como definida pelo usuário. Na linha 75, quando faz `16: interfaceinvoke #7, 3`, chama o método `compare` implementado na nova classe. Já o programa que usa expressões lambda terá seu método `main`, linha 88, e um novo método `lambda$main$0`, linha 107, onde o método da interface funcional é implementado. A primeira instrução do método `main` é `0: invokedynamic #2, 0`, linha 93, que, analisando o *Constant Pool*, invoca o método `compare` da interface funcional `Comparator`. Mais para frente o código chama a instrução `14: invokeinterface #6, 3` que também chama o método `compare`, mas a implementação esperada pelo usuário não está em um método com o nome `compare` e sim `lambda$main$0`, então como a JVM encontra a implementação correta?

Para entender como `invokedynamic` faz a ligação entre as implementações dos métodos é preciso entender três estruturas fornecidas pela biblioteca *java.lang.invoke* API que auxiliam no funcionamento da instrução `indy`:

- *MethodHandle (MH)*: aponta para um método contendo uma implementação
- *CallSite (CS)*:
  - contém um MethodHandle
  - local na pilha onde ocorre a invocação do método
  - individual para cada `invokedynamic`, mas pode ter MethodHandles diferentes durante a execução
- *Bootstrap Method (BSM)*:
  - executado apena a primeira vez que um determinado `invokedynamic` é encontrado
  - retorna um CallSite ligando-o ao `invokedynamic` em questão

### Bootstrapping

A inicialização de `invokedynamic` é *lazy*, o *bootstrapping* só é feito a primeira vez que a instrução é executada, então dividimos seu funcionamento em duas situações, ligação e invocação:

- Ligação (*bootstrap*): invocação inicial
    1. instrução `invokedynamic` especifica é encontrada pela primeira vez
    2. JVM executa o BSM
    3. BSM retorna um CallSite *object* permanentemente associado aquela *indy call*
    4. execução do programa pula para o método apontado pelo MethodHandle do CallSite associado a instrução
- Invocações subsequentes:
    1. execução do programa pula para o método apontado pelo MethodHandle do CallSite associado a instrução

As números precedidos de # referenciam posições na *Constant pool*, seguindo suas referências a partir da posição #2 passada pela instrução `0: invokedynamic #2, 0` é possível coletar algumas informações relevantes para o *bootstrapping*.

Código 2.6: SimpleLambdaExpression: Constant Pool

```
#2 = InvokeDynamic    #0:#27
...
#27 = NameAndType     #43:#44       // compare:()Ljava/util/Comparator;
...
#43 = Utf8            compare
#44 = Utf8            ()Ljava/util/Comparator;
```

A pilha começa em #1, a referência a posição #0 é apenas um lembrete de que o verdadeiro BSM está localizado em outro *.class*. A constante em #27 leva às informações de nome e tipo, apontadas pelas posições #43 e #44, respectivamente, o *Name* se refere ao nome do método da interface funcional sendo chamada, *Type* é o tipo de retorno da interface e consequentemente o tipo de retorno instrução `invokedynamic`, neste exemplo o tipo tem sua lista de argumentos vazia, isso por ser uma expressão lambda que não captura contexto, isso só acontecerá no caso de *Closures*.

O trecho de código em 2.7 mostra a lista de BSMs do Código 2.4, que no caso de tratamento de expressões lambda serão sempre do tipo `LambdaMetafactory`, e embaixo a lista de argumentos estáticos adicionais para o tratamento de expressões lambda. O primeiro dele é a assinatura do método lambda, seguido do MethodHandle apontando para a implementação final da corpo do lambda e por último a assinatura apagada do método lambda.

Código 2.7: SimpleLambdaExpression: Bootstrap Methods

```
BootstrapMethods:
 0: #23 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(···)Ljava/lang/invoke/CallSite;
   Method arguments:
     #24 (Ljava/lang/Object;Ljava/lang/Object;)I
     #25 invokestatic SimpleLambdaExpression.lambda$main$0:(Ljava/lang/String;Ljava/lang/String;)I
     #26 (Ljava/lang/String;Ljava/lang/String;)I
```

## 2.3 Jimple-Soot

Neste trabalho a representação Jimple do Soot será referênciada como *Jimple-Soot*.

Soot usa a Jimple como sua principal Representação Intermediária (RI), sua estrutura de três endereços tipada é conveniente para aplicação de transformações de código e criação de CFG para análises. Outro aspecto de *Jimple-Soot* que facilita a realização de análises é sua quantidade reduzida de instruções, Jimple possui apenas 15 tipos de instruções (*statements*) comparadas com as mais de 200 instruções de *bytecode*Java [13].

- *Core statements*: `NopStmt`, `IdentityStmt` e `AssignStmt`
- *Intraprocedural control-flow statements*: `IfStmt`, `GotoStmt`, `TableSwitchStmt` e `LookupSwitchStmt`
- *Interprocedural control-flow statements*: `InvokeStmt`, `ReturnStmt` e `ReturnVoidStmt`
- *Monitor statements*: `EnterMonitorStmt`, `ExitMonitorStmt`
- Outros: `ThrowStmt` e `RetStmt`

Transformando Código 2.8 em *Jimple-Soot* por meio do comando `Soot Foo -f J` resulta no Código 2.9, nota-se as seguintes características estruturais do código *Jimple-Soot*: Suas instruções, declarações de variáveis e atribuições são semelhantes a de código fonte Java, já suas estruturas de fluxo de controle e invocação de métodos se aproximam mais com a do *bytecode* Java. O critério de nomeação de variáveis escolhido dita que variáveis locais que começam com $ representam posições na pilha e não variáveis locais do código original.

Valores de parâmetros e referências *this* são atribuídas usando `IdentityStmt`, ao usar essa instrução é garantido que todas as variáveis locais tem pelo menos um ponto de definição. Atribuição de expressões e valores imediatos à variáveis feitas com `AssignStmt` e instruções de retorno podem ser feitas usando `ReturnStmt` ou `ReturnVoidStmt`. Essas são as principais e mais comumente usadas instruções *Jimple-Soot* que não envolvem estruturas de controle de

fluxo. O Código 2.9 mostra um *pretty printer* do código *Jimple-Soot* derivado do Código 2.8, é possível ver alguns casos de uso dessas instruções.

Uma característica do *Jimple-Soot* importante para análise de dados que é a separação uma instrução por linha, de forma que comporte as especificações do código de três endereços. A instrução da linha 5 de `Foo` é quebrado em 2. Além disso aqui é possível ver uma otimização aplicada pelo Soot, que substituiu a variável `z` pelo valor 5 como parâmetro da função `inc`, e com isso também eliminando a necessidade da declaração da variável.

Código 2.8: Foo.java

```
1  public class Foo {
2      public static void main(String[] args) throws Exception {
3          Foo f = new Foo();
4          int z = 5;
5          int y = f.inc(z)*2;
6      }
7      public int inc (int x) {return x+1;}
8  }
```

Código 2.9: Foo.jimple

```
1  public class Foo extends java.lang.Object   {
2      public void <init>()    {...}
3
4      public static void main(java.lang.String[]) throws
       ↪  java.lang.Exception  {
5          Foo $r0;
6          int $i1, i2;
7          java.lang.String[] r2;
8          r2 := @parameter0: java.lang.String[];
9          $r0 = new Foo;
10         specialinvoke $r0.<Foo: void <init>()>();
11         $i1 = virtualinvoke $r0.<Foo: int inc(int)>(5);
12         i2 = $i1 * 2;
13         return;                        // ReturnVoidStmt
14     }
15     public int inc(int) {
16         int i0, $i1;
17         Foo r0;
18         r0 := @this: Foo;              // IndentityStmt
19         i0 := @parameter0: int;        // IdentityStmt
20         $i1 = i0 + 1;                  // AssignStmt
21         return $i1;                    // ReturnStmt
22     }
23 }
```

### 2.3.1 Expressões Lambda com *Jimple-Soot*

Para entender como o Soot trata expressões lambda em *Jimple-Soot* o *bytecode*do Código 2.4 é passado para o Soot como entrada da transformação para *Jimple-Soot*, o resultado da transformação são dois arquivos *Jimple-Soot*, um arquivo principal SimpleLambdaExpression.jimple e um arquivo auxiliar SimpleLambdaExpression$lambda_main_0__1.jimple.

Esse arquivo auxiliar implementa uma classe com três métodos `bootstrap\$`, `<init>` e o método específico da interface funcional implementada, no caso do Código 2.4, `compare`. A instrução `invokedynamic` não está presente na representação *Jimple-Soot* do códiogo e foi substituída na linha 15 por uma instrução `staticinvoke`:

```
r0 = staticinvoke <SimpleLambdaExpression$lambda_main_0__1: java.util.Comparator bootstrap$()>();
```

Que atribui a uma variável a referência a instância da classe de *bootstrap* ali criada e posteriormente usa esta variável para fazer `interfaceinvoke` do método `compare`. Que por sua vez chamar o método de implementação do corpo do lambda, nesse caso, `lambda$main$0`, no arquivo principal.

Nesse exemplo em particular, só existe uma expressão lambda, então apenas um arquivo auxiliar e um método lambda serão criados, mas, genericamente, um arquivo de classe de *bootstrap* que implementa uma interface funcional será criado para cada expressão lambda declarada no programa, Código 2.3.1, assim como um método `lambda$` no arquivo principal que implementa o corpo do lambda e é associado a classe de *bootstrap*.

O método `bootstrap$` é chamado apenas uma vez, para instanciar a classe de *bootstrap*, depois disso o programa tem acesso ao método `lambda$` desejado por meio do método especifico da interface funcional implementada na classe de *bootstrap*. O nome da classe de *bootstrap* é derivado do nome da classe principal e o método lambda associado, por exemplo no Código 2.10, `SimpleLambdaExpression` da origem a classe associada ao método `lambda$main$0`, `SimpleLambdaExpression$lambda_main_0__1`.

Código 2.10: SimpleLambdaExpression.jimple

```
1  public class SimpleLambdaExpression extends java.lang.Object    {
2      public void <init>()    {...}
3
4      public static void main(java.lang.String[])    {
5          java.util.Comparator r0;
6          java.io.PrintStream $r1;
7          int $i0;
8          java.lang.String[] r2;
9          r2 := @parameter0: java.lang.String[];
10         r0 = staticinvoke <SimpleLambdaExpression$lambda_main_0__1:
           ↪  java.util.Comparator bootstrap$()>();
11         $r1 = <java.lang.System: java.io.PrintStream out>;
12         $i0 = interfaceinvoke r0.<java.util.Comparator: int
           ↪  compare(java.lang.Object,java.lang.Object)>("hello",
           ↪  "world");
13         virtualinvoke $r1.<java.io.PrintStream: void
           ↪  println(int)>($i0);
14         return;
15     }
16
17     public static int lambda$main$0(java.lang.String, java.lang.String)
        ↪  {
18         java.lang.String r0, r1;
19         int $i0;
20         r0 := @parameter0: java.lang.String;
21         r1 := @parameter1: java.lang.String;
22         $i0 = virtualinvoke r0.<java.lang.String: int
           ↪  compareTo(java.lang.String)>(r1);
23         return $i0;
24     }
25 }
```

Código 2.11: Método bootstrap$

```
1  public static java.util.Comparator bootstrap$()    {
2      SimpleLambdaExpression$lambda_main_0__1 $r0;
3      $r0 = new SimpleLambdaExpression$lambda_main_0__1;
4      specialinvoke $r0.<SimpleLambdaExpression$lambda_main_0__1: void
       ↪  <init>()>();
5      return $r0;
6  }
```

Código 2.12: Método <init>

```
1  public void <init>()    {
2      SimpleLambdaExpression$lambda_main_0__1 $r0;
3      $r0 := @this: SimpleLambdaExpression$lambda_main_0__1;
4      specialinvoke $r0.<java.lang.Object: void <init>()>();
5      return;
6  }
```

Código 2.13: Método específico da interface: compare

```
1  public int compare(java.lang.Object, java.lang.Object)  {
2      SimpleLambdaExpression$lambda_main_0__1 $r0;
3      java.lang.Object $r1, $r2;
4      java.lang.String $r3, $r4;
5      int $i0;
6      $r0 := @this: SimpleLambdaExpression$lambda_main_0__1;
7      $r1 := @parameter0: java.lang.Object;
8      $r2 := @parameter1: java.lang.Object;
9      $r3 = (java.lang.String) $r1;
10     $r4 = (java.lang.String) $r2;
11     $i0 = staticinvoke <SimpleLambdaExpression: int
       ↪  lambda$main$0(java.lang.String,java.lang.String)>($r3, $r4);
12     return $i0;
13 }
```

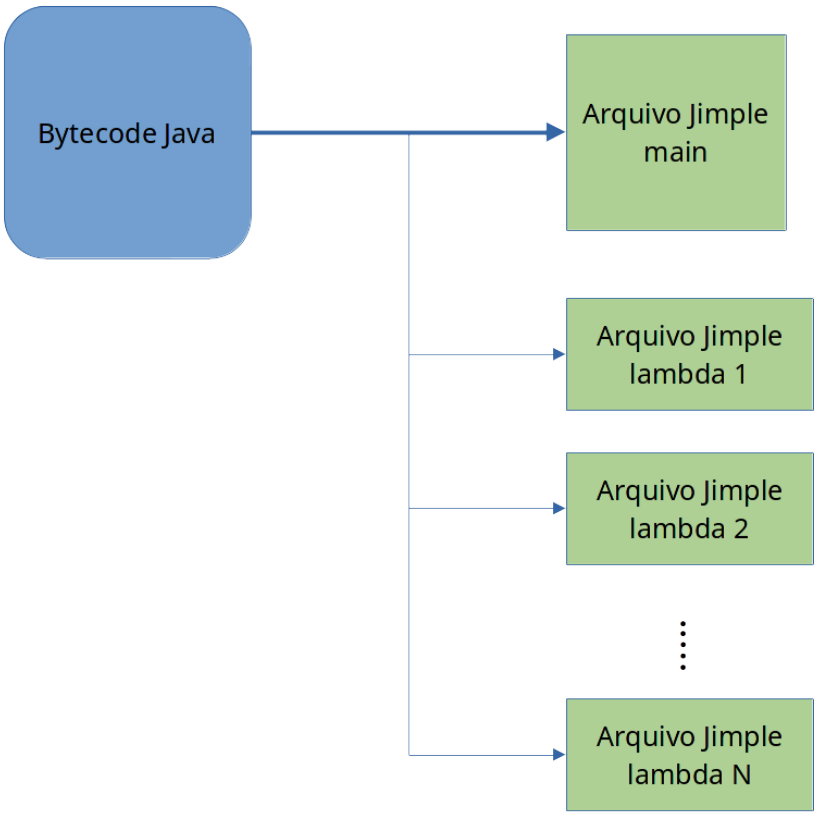


Figura 2.3: Expressões Lambda em Jimple

## 2.4 Rascal Meta-Programming Language

Rascal é uma linguagem de meta-programação voltada para a criação de ferramentas de análise, transformação, geração e visualização de códigos fonte [14], desenvolvida com intuito de tornar a implementação dessas ferramentas rápida, de fácil entendimento e compatível com qualquer linguagem de programação. O núcleo de Rascal é o mesmo qualquer outra linguagem de programação ordinária, com funções, procedimentos e estruturas de controle de fluxo com tratamento de exceções. Possui instruções de laços *foreach*, blocos condicionais e *switches*, entre outras estruturas presentes em qualquer linguagem de programação alto nível, permitindo que novos usuários usem a linguagem facilmente, usando conhecimentos gerais sobre lógica de programação.

Esses recursos de controle básicos conseguem proporcionar a sintaxe e semântica fundamentais para manipular os recursos avançados de meta-programação, como *pattern matching*[3] e *traversal* (travessia).

Alguns recursos úteis para a criação e transformação de códigos:

[3]Casamento de padrões: determina se um determinado padrão se encaixa em um valor ou não.

### Algebraic Data Types

*Algebraic Data Types* são usadas em Rascal para definir novos tipos de dados. Um novo tipo de dado é declarado usando a palavra reservada `data` seguida do nome que do novo tipo, então são passados construtores que representam os possíveis valores que esse tipo de dado pode ter.

### Pattern with Action

Realiza uma ação se o casamento dos padrões for bem sucedido.

### Visit

A expressão `visit` em Rascal faz a travessia de uma árvore visitando seus nós e realizando alguma ação, essa ação pode ser a simples coleta de informações, a aplicação de alguma modificação à árvore ou ambos.

Um `visit` que faz a travessia da árvore coletando informações é chamado de *accumulator*, acumulador. Quando o `visit` percorre uma árvore e aplica modificações sobre ela, dessa forma transformando-a em outra, ele é chamado de *transformer*, transformador. E quando faz ambos, *accumulating transformer*, ou transformador acumulante.

# Capítulo 3

# Jimple Framework

Esse capítulo descreve o funcionamento geral do *Jimple Framework*, mostra partes da *Abstract Syntax Tree (AST)* relevantes para a extração das informações necessárias para o transformação do código Jimple e criação das classes de *bootstrap*.

## 3.1 Funcionamento Geral

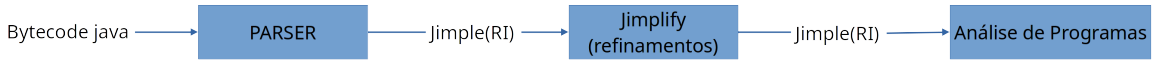


Figura 3.1: Overview Jimple Framework

O *Jimple Framework* permite descompilar *bytecode* Java em uma representação Jimple, oferece mecanismos para refinamento e otimização de código Jimple e disponibiliza ferramentas para análise de fluxo de dados. Seu *parser*, implementado em Java, recebe *bytecode* Java como entrada e retorna a RI Jimple do *Jimple Framework*. O pacote *Jimplify* possui módulos implementados em Rascal que aplicam transformações em código Jimple, refinando sua representação afim de permitir ou facilitar a implementação de análises de fluxo de controle.

No diretório `src/test/java/` o pacote `samples` guarda amostras de códigos Java, as amostras podem estar em diretórios separados por tipo de análise ou outra característica específica que foi queira ser testada. Executando `mvn test` os arquivos Java armazenados no pacote `samples` são compilados e seus arquivos .class armazenados em uma pasta *target* o que facilita a execução de testes que precisam referenciar esses arquivos. O diretório de teste `src/test/rascal/` armazena os *test cases* implementado em Rascal. No diretório `src/main/rascal/` o pacote `lang.jimple.core` guarda o módulo `Syntax.rsc` que define as estruturas da sintaxe Jimple do *Jimple Framework*, essas estruturas que compõe a *Abstract Syntax Tree* da RI Jimple.

## 3.2 Árvore Sintática Abstrata

Um código Jimple será uma *Abstract Syntax Tree (AST)* representada pelo *Algebraic Data Type (ADT)* `ClassOrInterfaceDeclaration`, Código 3.1, possui dois construtores: `classDecl` usado para criar classes e `interfaceDecl` para criar interfaces. Para criar uma ClassOrInterfaceDeclaration (CID) basta chamar o construtor desejado e inserir nos campos argumentos dos tipos especificados na declaração da ADT.

Código 3.1: ADT ClassOrInterfaceDeclaration

```
1  public data ClassOrInterfaceDeclaration
2   = classDecl(Type typeName,
3       list[Modifier] modifiers,
4       Type superClass,
5       list[Type] interfaces,
6       list[Field] fields,
7       list[Method] methods
8     )
9   | interfaceDecl(Type typeName,
10      list[Modifier] modifiers,
11      list[Type] interfaces,
12      list[Field] fields,
13      list[Method] methods
14    );
```

Este trabalho usa o construtor `classDecl` para criar as classes de *bootstrap*, para isso é preciso entender seus campos. O campo `typeName` dará nome à nova classe e tem tipo `Type` (ADT que descreve os tipos aceitos pela linguagem), será um `TObject`[1] do nome da classe sendo criada. A lista `modifiers` contém as palavras-chaves que determinam os modificadores da classe, como `public` ou `abstract`, esses modificadores não recebem argumentos e são valores descritos no ADT `Modifier` da seguinte forma `Public()` e `Abstract()`. A classe vai herdar de uma super classe definida por `superClass`, no caso de classes de *bootstrap* sempre será `TObject()` e a lista `interfaces` contém os tipos implementados pela classe, nesse caso especifico a interface funcional. A lista `fields` descreve os campos da classe, indicando o nome da variável e seu tipo. Por fim a lista `methods` contém os métodos daquela classe.

[1] TObject é o único Type além de TArray que recebe argumentos e serve para criar novos tipos e classes

Código 3.2: ADT Method e MethodBody

```
1  public data Method
2    = method(list[Modifier] modifiers,
3      Type returnType,
4      Name name,
5      list[Type] formals,
6      list[Type] exceptions,
7      MethodBody body);
8
9  data MethodBody
10   = methodBody(list[LocalVariableDeclaration] localVariableDecls,
11                list[Statement] stmts,
12                list[CatchClause] catchClauses)
13   | signatureOnly();
```

O tipo `Method` possui apenas um construtor, `method`. Assim como `classDecl`, seus modificadores são explicitados em `modifiers`. Seu tipo de retorno é `returnType` e seu nome, `name`. A lista `formals` contém o tipo de cada variável esperada como argumento e `exceptions` as possíveis exceções. `body` é o corpo do método, onde são declaradas variáveis, definidas as instruções e clausulas *catch* para exceções.

### 3.2.1 invokedynamic

A *ADT* `InvokeExp` possui um construtor para cada tipo de invocação, o construtor da `indy` é o `dynamicInvoke`, Código 3.3. Do nó `dynamicInvoke(bsmSig, bsmArgs, sig, args)` é possível extrair todas as informações necessárias para criar a classe de *bootstrap* dessa `indy` call e transformar seu nó em um `staticMethodInvoke`, isso será mostrado no Capítulo 4. Primeiro é preciso identificar o que cada argumento de `dynamicInvoke` é, e para que servirá na transformação do código e na criação da classe de *bootstrap*.

Código 3.3: Construtor dynamicInvoke

```
1  dynamicInvoke(MethodSignature bsmSig,
2                list[Immediate] bsmArgs,
3                MethodSignature sig,
4                list[Immediate] args)
```

**bsmSig**

É a assinatura do *BootstrapMethod*, sua única função na transformação é indicar se a `indy` *call* em questão veio realmente de uma expressão lambda.

`methodSignature(_,_,mf,_)`: se mf for igual a metafactory o que gerou a instrução `indy` foi uma expressão lamdbda.

**bsmArgs**

Lista com os BootstrapMethod arguments

A lista contém 3 itens, encapsuladas dentro de `iValue()`, nessas exatas posições:

1. Assinatura do método lambda
   `methodValue(returnType, [formals])`
   Usado como assinatura do método alvo[2] (`targetMethod`)
2. MethodHandle apontando para a implementação final do corpo do lambda
   `methodHandle(methodSignature(className, returnType, methodName, formals))`
   Usado como corpo do método alvo (`targetMethod`) e na transformação de `dynamicInvoke` em `staticMethodInvoke`
3. Assinatura apagada do método lambda
   `methodValue(returnType, [formals])`
   Usado na declaração de variáveis no corpo do método alvo

**sig**

Assinatura do método da interface funcional
`methodSignature(_,returnType, methodName, formals)`

- `returnType`: Interface funcional implementada pela classe de *bootstrap* e tipo de retorno do método `bootstrap$`
- `methodName`: Nome do método alvo, oriundo da interface funcional
- `formals`: Lista dos tipos dos argumentos dos métodos `bootstrap$` e `<init>`

**args**

Lista de variáveis sendo passadas como argumento ao método `bootstrap$`
Passados como argumentos de `staticMethodInvoke`

[2] método da classe de *bootstrap* oriundo da *funcional interface* sendo implementada

Independente do tipo de invocação sendo feita, todas referenciam o método sendo chamado por meio de um `methodSignature`, Código 3.4. `className` é o nome da classe do método, `returnName` o tipo de retorno da função, `methodName` o nome do método e a lista `formals` a lista com os tipos dos argumentos esperados.

Código 3.4: ADT MethodSignature

```
1  data MethodSignature
2    = methodSignature(Name className,
3                      Type returnType,
4                      Name methodName,
5                      list[Type] formals);
```

# Capítulo 4

# Implementação

A partir do entendimento de como o Soot lida com expressões lambda e que a saída não refinada do *parser* traduz métodos lambda para expressões `dynamicInvoke`, o próximo refinamento é aplicar transformações na árvore e criar classes de *bootstrap* para cada ocorrência de `dynamicInvoke` proveniente de uma expressão lambda. O `dynamicInvoke` dá origem a uma classe de *bootstrap* `lambda$X` e é transformado em `staticMethodInvoke`. O `staticMethodInvoke` instancia a classe criada chamando seu método `bootstrap$` e atribui o resultado a uma variável. Esta, então, poderá ser usada para chamar o método alvo (`targerMethod`) quando quiser, que, por fim, chamará o método `lambda$X` que implementa o corpo do lambda, Figura 4.1.

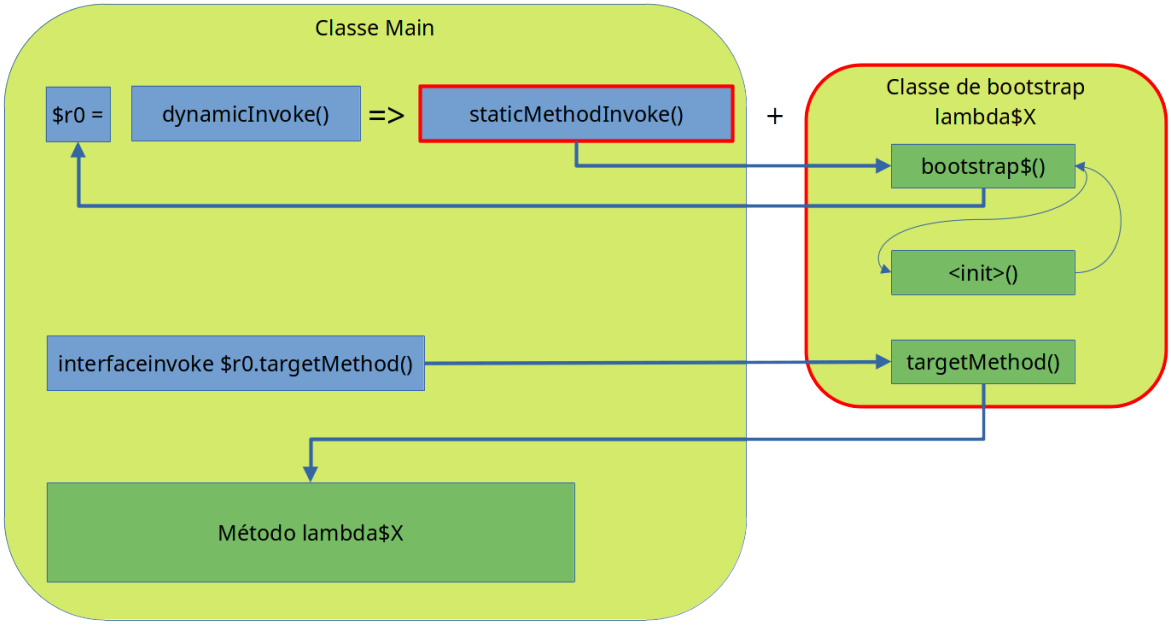


Figura 4.1: Refinamento de LambdaTransformer

## 4.1 LambdaTransformer

`LambdaTransformer` usa os padrões *visiting* e *pattern matching* para implementar transformações de *Term Rewriting* na Abstract Syntax Tree (AST) de códigos Jimple do *Jimple Framework* e a criação das classes de *bootstrap* necessárias para estabelecer a conexão explicita entre a implementação do corpo do lambda e o local da chamada do método.

Código 4.1: Método lambdaTransformer

```
1  alias CID = ClassOrInterfaceDeclaration;
2
3  public list[CID] lambdaTransformer(CID c) {
4    list[ClassOrInterfaceDeclaration] classes = [];
5    c = visit(c) {
6      case dynamicInvoke(_, bsmArgs, sig, args): {
7          classes += generateBootstrapClass(bsmArgs, sig);
8          MethodSignature mh = methodSignature(bsmArgs[1]);
9          insert generateStaticInvokeExp(mh, sig, args);
10     }
11   }
12   classes += c;
13   return classes;
14 }
```

O método *lambdaTransformer* recebe código Jimple no formato de árvore do tipo ClassOrInterfaceDeclaration (CID) e faz a travessia dessa árvore combinando `visit` com *PatternWithAction*. Cada vez que um nó da árvore é visitado é feito *pattern matching* do dado presente naquele nó com o padrão do construtor `dynamicInvoke(_, bsmArgs, sig, args)` do *data type* `InvokeExp`. Caso o padrão não se encaixe, o `visit` apenas passa para o próximo nó, caso contrário, passa a executar as ações definidas no bloco do `case`.

A instrução dentro do bloco, na linha 7, chama o método `generateBootstrapClass` que cria a classe de *bootstrap* referente àquela instrução `indy` e coloca na lista de *CID*, `classes`, que armazenar todas as classes relacionadas ao refinamento sendo executado. Isso inclui a classe original sendo refinada e quaisquer classes criadas no processo de refinamento. Na linha 8, o `methodSignature` não é um construtor, e sim uma função auxiliar, 4.3, que faz *pattern matching* para extrair o `MethodSignature mh` de dentro de um `methodHandle` encapsulado dentro de um `iValue`. O comando `insert` substitui o valor do nó que casou com o padrão pelo retorno do método `generateStaticInvokeExp(mh, sig, args)`. Existem algumas

restrições para o uso do comando `insert`. Ele pode ser usado apenas como a ação de um *PatternWithAction* (PwA), mais especificamente uma expressão `visit` (Visit-Rascal) e o tipo da nova expressão deve ser do mesmo subtipo do valor sendo substituído, isso não é um problema pois ambas as expressões `dynamicInvoke` e `staticMethodInvoke` são valores do mesmo *data type*, `InvokeExp`, Código 4.2. Este comando deve ser o último do bloco de ação, uma vez executado o `visit` parte para o próximo nó.

Código 4.2: ADT Invoke Expression

```
data InvokeExp
  = specialInvoke(Name local, MethodSignature sig,
                  list[Immediate] args)
  | virtualInvoke(Name local, MethodSignature sig,
                  list[Immediate] args)
  | interfaceInvoke(Name local, MethodSignature sig,
                  list[Immediate] args)
  | staticMethodInvoke(MethodSignature sig, list[Immediate] args)
  | dynamicInvoke(MethodSignature bsmSig,
                  list[Immediate] bsmArgs,
                  MethodSignature sig,
                  list[Immediate] args)
  ;
```

Código 4.3: Funções auxiliares com casamento de padrão

```
private Type returnType(methodSignature(_, aType, _, _)) = aType;
private MethodSignature methodSignature(iValue(methodHandle(mh))) = mh;
```

O método `generateStaticInvokeExp`, Código 4.2, deve retornar uma expressão `staticMethodInvoke` do método `bootstrap$`, seus argumentos são simplesmente um `MethodSignature` e a lista de argumentos que o método recebe. Os argumentos são os mesmos da `indy`, o que muda é o *method signature*. O primeiro argumento, o nome da nova classe, é o nome da classe original concatenado[1] com o nome do método lambda associado, o segundo é o tipo de retorno, igual a interface funcional. O terceiro é o nome do método a ser chamado, nesse caso `bootstrap$` e por fim a lista com os tipo dos argumentos esperados.

[1] O operador $ em Rascal concatena duas strings

Código 4.4: Método generateStaticInvokeExp

```
1  private InvokeExp generateStaticInvokeExp(MethodSignature sig1,
   ↪  MethodSignature sig2, list[Immediate] args) {
2    MethodSignature sig = methodSignature(
3          "<split("/", className(sig1))[1]>$<methodName(sig1)>",
4          returnType(sig2),
5          "bootstrap$",
6          formals(sig2));
7
8    return staticMethodInvoke(sig, args);
9  }
```

A construção da classe de *bootstrap* é feita com o método `generateBootstrapClass`, Código 4.5, este método é consideravelmente maior que `generateStaticInvokeExp` e será quebrado em partes para melhor entendimento. Neste método ocorre a construção da CID da nova classe propriamente dita, sua construção deve ser feita de forma *bottom-up*, começando com os itens das folhas, mas para melhor compreender o que deve ser feito o código aqui será demonstrado de cima para baixo a começar pelo ramo inicial, `classDecl`, **??**.

Código 4.5: Método generateBootstrapClass

```
1  private CID generateBootstrapClass(list[Immediate] bsmArgs,
   ↪  MethodSignature bsmSig) {
2      str bsmClassName = "<last(split("/", className(targetSig)))>
3                              $<methodName(targetSig)>";
4      <...>
5      CID bsmClass = classDecl(TObject(bsmClassName),
6                              [Public(), Final()],
7                              object(),
8                              [returnType(bsmSig)],
9                              getFields(bsmSig),
10                             [bsm, initMethod, targetMethod]);
11     return bsmClass;
12 }
```

Como visto anteriormente, Código 3.1, o primeiro argumento da CID é um `TObject` do seu nome[2], seguido da lista de construtores e sua super classe, object()[3]. Depois a lista de interfaces implementadas, `returnType` de `bsmSig` (Seção 3.2.1) retorna a interface funcional sendo implementada. O método `getFields`, Código 4.7, cria a lista de variáveis de classe a partir da lista de tipos que o método `bootstrap$` espera como argumento. Por fim a lista de métodos da classe, `bootstrap$`, `<init>` e `targetMethod`, cada um desses método devem ser criados antes da declaração de `bsmClass`, Código 4.6.

Código 4.6: generateBootstrapClass - targetMethod, initMethod, bsm

```
1  MethodBody bsmBody = methodBody(bsmLocals, bsmStmts, []);
2  MethodBody initBody = methodBody(initLocals, initStmts, []);
3  MethodBody targetBody = methodBody(targetLocals, targetStmts, []);
4
5  Method bsm = method([Public(), Static()],
6                      returnType(bsmSig),
7                      "bootstrap$",
8                      formals(bsmSig),
9                      [],
10                     bsmBody);
11
12 Method initMethod = method([Public(), Static()],
13                     TVoid(),
14                     "<init>",
15                     formals(bsmSig),
16                     [],
17                     initBody);
18
19 Method targetMethod = method([Public(), Static()],
20                       returnType(targetSig),
21                       methodName(bsmSig),
22                       valueFormals(bsmArgs[0]),
23                       [],
24                       targetBody);
```

[2]Nome da classe original concatenada com o nome do método lambda associado
[3]object() = TObject("java.lang.Object")

Os `MethodBody`, Código 3.2, são os corpos de cada método, isso inclui declaração das variáveis locais no primeiro campo e no segundo as instruções que formam o método, o terceiro campo não é necessário para esta implementação.

**bsmBody** : corpo de `bootstrap$`

- bsmLocals - Variáveis locais do método. Incluem uma variável de referência a classe e variáveis para cara um de seus parâmetros;
- bsmStmts - Instruções do corpo método. Incluem instruções:
  1. `idenity`: Instanciam as variáveis.
  2. `invokeStmt`: Invoca o método <init> da classe
  3. `returnStmt`: Retorna uma instância da classe

**initBody** : corpo de <init>

- initLocals - Variáveis locais do método. Incluem uma variável de referência a classe e variáveis para cara um de seus parâmetros;
- initStmts - Instruções do corpo do método. Incluem instruções:
  1. `idenity`: Instanciam as variáveis.
  2. `invokeStmt`: Invoca o método <init> da super classe java.lang.Object
  3. `assign`: Atribuem os valores recebidos como parâmetros aos atributos de instância
  4. `returnEmptyStmt`: Retorna ao método `bootstrap$`

**targetBody** : corpo de `targetMethod`

- targetLocals - Variáveis locais do método. Incluem:
  1. Variável de referência a classe
  2. Variáveis para instanciação dos parâmetros. São do tipo da assinatura do método lambda, contidos em bsmArgs[0].
  3. Variáveis para invocação do método lambda. Usadas, possivelmente, para *type casting* dos parâmetros ou instanciação dos atributos de instância do método. São do tipo da assinatura apagada do método lambda, contidos em tipos de bsmArgs[2].
  4. Variável de retorno. Caso o `returnType` do método lambda não seja `TVoid`
- targetStmts - Instruções do corpo do método. Incluem instruções:
  1. `identity`: Instanciam as variáveis

2. `assign`: Atribuem às variáveis: os valores dos atributos de instância e os valores do *type cast* dos parâmetros
3. Se o `returnType` do `targetMethod` for:

   **TVoid** -

   3. `invokeStmt`: Invoca o método lambda que implementa a interface funcional
   4. `returnEmptyStmt`

   **Caso contrário** -

   3. `assign`: Atribui à variável de retorno a `invokeExp` do método lambda que implementa a interface funcional
   4. `returnStmt`: Retorna o que recebeu do método lambda

Código 4.7: Método getFields

```
1  list[Field] getFields(MethodSignature sig){
2      list[Field] fields = [];
3
4      for(int i <- [0..size(formals(sig))]){
5          fields += field([], formals(sig)[i], "cap<i>");
6      }
7      return fields;
8  }
```

```
1  public data Field
2    = field(list[Modifier] modifiers, Type fieldType, Name name);
```

## 4.2 Verificação

Para a verificação do funcionamento do *LambdaTransformer* foi criado um módulo de testes TestLambdaTransformer.rsc e sete *samples* para testagem. O módulo de teste chama o método do *Decompiler* `decompile` e passa como argumento o *.class* a ser descompilado. O resultado da descompilação é passado como argumento a `lambdaTransformer`. O retorno de `lambdaTransformer` é uma lista com as classes que compõe o sample. A verificação simplificada do módulo de teste é conferir se o número de classes é igual ao esperado. O conteúdo das classes foi comparado com o resultado obtido pelo Soot.

Alguns cenários identificados:

- Quanto às origem das variáveis:
    1. Todas as variáveis de `targetMethod` vem de sua invocação[4]
    2. Todas as variáveis de `targetMethod` vem de atributos de instância
    3. Variáveis de `targetMethod` vem tanto de sua invocação quanto de variáveis da instância
- `returnType` do `targetMethod` é `TVoid` ou não

O exemplo Palindromes.java mostra todos os possíveis cenários de transformação. O cenário 1 é coberto por Palindromes$lambda$0.jimple, o cenário 2 por Palindromes$lambda$1.jimple e o cenário 3 por Palindromes$lambda$2.jimple. Os códigos podem ser encontrados no Apêndice C.

[4]Com exceção da variável de referência a classe

Código 4.8: Palidromes.java

```java
package samples.lambdaExpressions;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Palindromes {
    public static void main(String[] args) throws Exception {

        List<String> words = Arrays.asList("Hannah", "Ana", "Elle",
        ↪  "Bob", "Otto", "Natan", "Luisa", "Andre", "Renata");

        Comparator<String> stringComparator = (s1, s2) ->
        ↪  s1.toLowerCase().compareTo(new
        ↪  StringBuilder(s2).reverse().toString().toLowerCase());

        Runnable r = () -> words.forEach(w -> {
            if (stringComparator.compare(w, w) == 0) {
                System.out.println(w + " is a palindrome.");
            } else {
                System.out.println(w + " is not a palindrome.");
            }
        });

        Thread t = new Thread(r);
        t.start();
        t.join();
    }
}
```

O *sample* envolvendo closures, Código 2.5, foi testado e é coberto pela implementação, refinando a representação Jimple corretamente, dando origem aos códigos 4.9 e 4.10.

Código 4.9: IncClosure.jimple

```
1  public synchronized class IncClosure extends java.lang.Object {
2      public void <init>() {...}
3
4      public static void main(java.lang.String[]) {
5          Incrementable r3;
6          java.lang.String[] r1;
7          int r2;
8          Incrementable $r1;
9          java.io.PrintStream $r2;
10         int $r3;
11         i1 := @parameter0: java.lang.String[];
12         r2 = 1;
13         $r1 = staticinvoke <lambda$0: Incrementable
               ↪ bootstrap$(int)>(r2);
14         r3 = $r1;
15         $r2 = <java.lang.System: java.io.PrintStream out>;
16         $r3 = interfaceinvoke r3.<Incrementable: int inc(int)>(1);
17         virtualinvoke $r2.<java.io.PrintStream: void
               ↪ println(int)>($r3);
18
19         return;
20     }
21     private static TODO int lambda$0(int, int)     {
22         unknown i0;
23         int r1;
24         int $r1;
25         i1 := @parameter0: int;
26         i2 := @parameter1: int;
27         $r1 = r1 + i0;
28         return $r1;
29     }
30 }
```

Código 4.10: IncClosure$lambda$0.jimple

```
1  public final class IncClosure$lambda$0 extends java.lang.Object
   ↪  implements Incrementable  {
2       int cap0;
3      public static Incrementable bootstrap$(int)     {
4          int $r0;
5          IncClosure$lambda$0 $r1;
6          $r0 := @parameter0: int;
7          $r1 = new IncClosure$lambda$0;
8          specialinvoke $r1.<IncClosure$lambda$0: void <init>(int)>($r0);
9          return $r1;
10     }
11     public static void <init>(int)     {
12         IncClosure$lambda$0 $r0;
13         int $r1;
14         $r0 := @this: IncClosure$lambda$0;
15         $r1 := @parameter0: int;
16         specialinvoke $r0.<java.lang.Object: void <init>()>();
17         $r0.<IncClosure$lambda$0: int cap0> = $r1;
18         return;
19     }
20     public static int inc(int)     {
21         IncClosure$lambda$0 $r0;
22         int $r1;
23         int $r2;
24         int $i0;
25         $r0 := @this: IncClosure$lambda$0;
26         $r1 := @parameter0: int;
27         $r2 = $r0.<IncClosure$lambda$0: int cap0>;
28         $i0 = staticinvoke <IncClosure: int lambda$0(int, int)>($r2,
           ↪  $r1);
29         return $i0;
30     }
31 }
```

# Capítulo 5

# Conclusão

Apesar de sua adição relativamente tardia à linguagem, expressões lambda acrescentam aspectos importantes do paradigma de programação funcional ao leque de ferramentas a disposição de programadores Java. Tendo em vista o crescimento em popularidade de linguagens e conceitos de programação funcional, essa adição parecia inevitável. Cada vez mais há um aumento de aceitação e utilização de linguagens funcionais, as mais populares linguagens modernas de programação apresentam elementos fortíssimos de programação funcional. Uma vez presentes na linguagem, expressões lambdas são comumente usadas e, com isso, torna-se imprescindível que ferramentas de análise suportem análise de códigos com expressões lambda.

Neste trabalho, foi implementada uma nova transformação de código Jimple do *Jimple Framework*para o tratamento de expressões lambda utilizando técnicas de travessia de árvore usando *visiting design pattern* e casamento de padrões para identificar e aplicar o refinamento quando necessário. O *Jimple Framework*tem como objetivo facilitar a escrita de análises estáticas de códigos Java, como *data-flow analysis*, para isso é preciso clareza no caminho percorrido pelos dados. A partir de Java 8, a instrução *invokedynamic* passou a ser usada para tratar expressões lambda, porém essa instrução coloca o tratamento das expressões lambda e, consequentemente, o caminho dos dados dentro de uma caixa preta o que impossibilitava análises estática de seus códigos. A solução escolhida pelo Soot *Framework* para tornar o caminho de dados explicito novamente foi transformar o a instrução *invokedynamic* em uma instrução *staticinvoke* comum e o *LambdaMetafactory* em uma classe de *bootstrap* que conecta a implementação de uma interface a chamada do método.

Essa estratégia foi implementada nesse trabalho e incorporada ao *Jimple Framework*que anteriormente, ao encontrar uma chamada `dynamicInvoke` pulava o bloco de código e não analisava a instrução. Agora o caminho dos dados está explicito na Representação Intermediária (RI) Jimple e códigos Java com expressões lambda são refinados com facilidade para que possam ser alvos de análises estáticas. Isso é feito aplicando uma função de travessia à Árvore Sintática Abstrata, em inglês Abstract Syntax Tree (AST), do código Jimple, que com o uso de

casamento de padrões, procura de instruções `dynamicInvoke`. Ao encontrá-las coleta informações para a criação da AST de uma nova classe de *bootstrap*e sobrescreve a instrução com uma `staticMethodInvoke` adequada. Considerando que a função de travessia implementada neste trabalho faz tanto a coleta de informações quando modificações a árvore é possível concluir, também, que o método implementada é um transformador acumulante, *Accumulating Transformer*.

Um dos maiores desafios encontrados foi certificar que o mapeamento dos campos e transposição das variáveis dentro do corpo do método da interface implementada é feito da forma correta para todos os casos de uso de expressões lambda. Um possível ponto de melhoria deste trabalho é a necessidade de desviar o fluxo de código para outra classe toda vez que uma expressão lambda aparece. Em trabalhos futuros seria vantajoso encontrar formas de refinar o código para que o fluxo de dados fique contido em uma só classe.

# Apêndice A

# SimpleLambdaExpression bytecode

SimpleLambdaExpression decompiled heap

```
Classfile /home/luisa/UnB/TCC/Codes/LambdaE/SimpleLambdaExpression.class
  Last modified Jun 3, 2021; size 1201 bytes
  MD5 checksum 91411f34fc624e92f20e9ebf9e06917a
  Compiled from "SimpleLambdaExpression.java"
public class SimpleLambdaExpression
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #10.#21        // java/lang/Object."<init>":()V
   #2 = InvokeDynamic      #0:#27         // #0:compare:()Ljava/util/
       Comparator;
   #3 = Fieldref           #28.#29        // java/lang/System.out:Ljava/io/
       PrintStream;
   #4 = String             #30            // hello
   #5 = String             #31            // world
   #6 = InterfaceMethodref #32.#33        // java/util/Comparator.compare:(
       Ljava/lang/Object;Ljava/lang/Object;)I
   #7 = Methodref          #34.#35        // java/io/PrintStream.println:(I)V
   #8 = Methodref          #36.#37        // java/lang/String.compareTo:(Ljava
       /lang/String;)I
   #9 = Class              #38            // SimpleLambdaExpression
  #10 = Class              #39            // java/lang/Object
  #11 = Utf8               <init>
  #12 = Utf8               ()V
  #13 = Utf8               Code
  #14 = Utf8               LineNumberTable
  #15 = Utf8               main
  #16 = Utf8               ([Ljava/lang/String;)V
  #17 = Utf8               lambda$main$0
  #18 = Utf8               (Ljava/lang/String;Ljava/lang/String;)I
  #19 = Utf8               SourceFile
  #20 = Utf8               SimpleLambdaExpression.java
  #21 = NameAndType        #11:#12        // "<init>":()V
  #22 = Utf8               BootstrapMethods
  #23 = MethodHandle       #6:#40         // invokestatic java/lang/invoke/
      LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
      Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
      MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)
      Ljava/lang/invoke/CallSite;
  #24 = MethodType         #41            //  (Ljava/lang/Object;Ljava/lang/
      Object;)I
  #25 = MethodHandle       #6:#42         // invokestatic
      SimpleLambdaExpression.lambda$main$0:(Ljava/lang/String;Ljava/lang/
      String;)I
  #26 = MethodType         #18            //  (Ljava/lang/String;Ljava/lang/
      String;)I
```

```
36    #27 = NameAndType        #43:#44        // compare:()Ljava/util/Comparator;
37    #28 = Class              #45            // java/lang/System
38    #29 = NameAndType        #46:#47        // out:Ljava/io/PrintStream;
39    #30 = Utf8               hello
40    #31 = Utf8               world
41    #32 = Class              #48            // java/util/Comparator
42    #33 = NameAndType        #43:#41        // compare:(Ljava/lang/Object;Ljava/
          lang/Object;)I
43    #34 = Class              #49            // java/io/PrintStream
44    #35 = NameAndType        #50:#51        // println:(I)V
45    #36 = Class              #52            // java/lang/String
46    #37 = NameAndType        #53:#54        // compareTo:(Ljava/lang/String;)I
47    #38 = Utf8               SimpleLambdaExpression
48    #39 = Utf8               java/lang/Object
49    #40 = Methodref          #55.#56        // java/lang/invoke/
          LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;
          Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
          MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)
          Ljava/lang/invoke/CallSite;
50    #41 = Utf8               (Ljava/lang/Object;Ljava/lang/Object;)I
51    #42 = Methodref          #9.#57         // SimpleLambdaExpression.
          lambda$main$0:(Ljava/lang/String;Ljava/lang/String;)I
52    #43 = Utf8               compare
53    #44 = Utf8               ()Ljava/util/Comparator;
54    #45 = Utf8               java/lang/System
55    #46 = Utf8               out
56    #47 = Utf8               Ljava/io/PrintStream;
57    #48 = Utf8               java/util/Comparator
58    #49 = Utf8               java/io/PrintStream
59    #50 = Utf8               println
60    #51 = Utf8               (I)V
61    #52 = Utf8               java/lang/String
62    #53 = Utf8               compareTo
63    #54 = Utf8               (Ljava/lang/String;)I
64    #55 = Class              #58            // java/lang/invoke/
          LambdaMetafactory
65    #56 = NameAndType        #59:#63        // metafactory:(Ljava/lang/invoke/
          MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;
          Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/
          invoke/MethodType;)Ljava/lang/invoke/CallSite;
66    #57 = NameAndType        #17:#18        // lambda$main$0:(Ljava/lang/String;
          Ljava/lang/String;)I
67    #58 = Utf8               java/lang/invoke/LambdaMetafactory
68    #59 = Utf8               metafactory
69    #60 = Class              #65            // java/lang/invoke/
          MethodHandles$Lookup
70    #61 = Utf8               Lookup
71    #62 = Utf8               InnerClasses
```

```
72    #63 = Utf8               (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/
          String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/
          lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke
          /CallSite;
73    #64 = Class              #66            // java/lang/invoke/MethodHandles
74    #65 = Utf8               java/lang/invoke/MethodHandles$Lookup
75    #66 = Utf8               java/lang/invoke/MethodHandles
76  {
77    public SimpleLambdaExpression();
78      descriptor: ()V
79      flags: ACC_PUBLIC
80      Code:
81        stack=1, locals=1, args_size=1
82           0: aload_0
83           1: invokespecial #1                  // Method java/lang/Object."<
                 init>":()V
84           4: return
85        LineNumberTable:
86          line 3: 0
87
88    public static void main(java.lang.String[]);
89      descriptor: ([Ljava/lang/String;)V
90      flags: ACC_PUBLIC, ACC_STATIC
91      Code:
92        stack=4, locals=2, args_size=1
93           0: invokedynamic #2,  0              // InvokeDynamic #0:compare:()
                 Ljava/util/Comparator;
94           5: astore_1
95           6: getstatic     #3                  // Field java/lang/System.out:
                 Ljava/io/PrintStream;
96           9: aload_1
97          10: ldc           #4                  // String hello
98          12: ldc           #5                  // String world
99          14: invokeinterface #6,  3            // InterfaceMethod java/util/
              Comparator.compare:(Ljava/lang/Object;Ljava/lang/Object;)I
100         19: invokevirtual #7                  // Method java/io/PrintStream.
              println:(I)V
101         22: return
102       LineNumberTable:
103         line 5: 0
104         line 6: 6
105         line 7: 22
106
107   private static int lambda$main$0(java.lang.String, java.lang.String);
108     descriptor: (Ljava/lang/String;Ljava/lang/String;)I
109     flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
110     Code:
111       stack=2, locals=2, args_size=2
```

```
112           0: aload_0
113           1: aload_1
114           2: invokevirtual #8                  // Method java/lang/String.
                 compareTo:(Ljava/lang/String;)I
115           5: ireturn
116         LineNumberTable:
117           line 5: 0
118 }
119 SourceFile: "SimpleLambdaExpression.java"
120 InnerClasses:
121      public static final #61= #60 of #64; //Lookup=class java/lang/invoke/
           MethodHandles$Lookup of class java/lang/invoke/MethodHandles
122 BootstrapMethods:
123   0: #23 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/
       lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/
       MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;
       Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
124     Method arguments:
125       #24 (Ljava/lang/Object;Ljava/lang/Object;)I
126       #25 invokestatic SimpleLambdaExpression.lambda$main$0:(Ljava/lang/String
            ;Ljava/lang/String;)I
127       #26 (Ljava/lang/String;Ljava/lang/String;)I
```

# Apêndice B

# AnonyImplementation bytecode

AnonymousInterfaceImplementation decompiled heap

```
Classfile /home/luisa/UnB/TCC/Codes/LambdaE/AnonymousInterfaceImplementation.class
  Last modified Jun 2, 2021; size 641 bytes
  MD5 checksum 89fb84dbddd8f43a684c0b037204ac04
  Compiled from "AnonymousInterfaceImplementation.java"
public class AnonymousInterfaceImplementation
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #10.#20        // java/lang/Object."<init>":()V
   #2 = Class              #21            // AnonymousInterfaceImplementation$1
   #3 = Methodref          #2.#20         // AnonymousInterfaceImplementation$1."<init>":()V
   #4 = Fieldref           #22.#23        // java/lang/System.out:Ljava/io/PrintStream;
   #5 = String             #24            // hello
   #6 = String             #25            // world
   #7 = InterfaceMethodref #26.#27        // java/util/Comparator.compare:(Ljava/lang/Object;Ljava/lang/Object;)I
   #8 = Methodref          #28.#29        // java/io/PrintStream.println:(I)V
   #9 = Class              #30            // AnonymousInterfaceImplementation
  #10 = Class              #31            // java/lang/Object
  #11 = Utf8               InnerClasses
  #12 = Utf8               <init>
  #13 = Utf8               ()V
  #14 = Utf8               Code
  #15 = Utf8               LineNumberTable
  #16 = Utf8               main
  #17 = Utf8               ([Ljava/lang/String;)V
  #18 = Utf8               SourceFile
  #19 = Utf8               AnonymousInterfaceImplementation.java
  #20 = NameAndType        #12:#13        // "<init>":()V
  #21 = Utf8               AnonymousInterfaceImplementation$1
  #22 = Class              #32            // java/lang/System
  #23 = NameAndType        #33:#34        // out:Ljava/io/PrintStream;
  #24 = Utf8               hello
  #25 = Utf8               world
  #26 = Class              #35            // java/util/Comparator
  #27 = NameAndType        #36:#37        // compare:(Ljava/lang/Object;Ljava/lang/Object;)I
  #28 = Class              #38            // java/io/PrintStream
  #29 = NameAndType        #39:#40        // println:(I)V
  #30 = Utf8               AnonymousInterfaceImplementation
  #31 = Utf8               java/lang/Object
  #32 = Utf8               java/lang/System
```

```
  #33 = Utf8               out
  #34 = Utf8               Ljava/io/PrintStream;
  #35 = Utf8               java/util/Comparator
  #36 = Utf8               compare
  #37 = Utf8               (Ljava/lang/Object;Ljava/lang/Object;)I
  #38 = Utf8               java/io/PrintStream
  #39 = Utf8               println
  #40 = Utf8               (I)V
{
  public AnonymousInterfaceImplementation();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<
            init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=2, args_size=1
         0: new           #2                  // class
            AnonymousInterfaceImplementation$1
         3: dup
         4: invokespecial #3                  // Method
            AnonymousInterfaceImplementation$1."<init>":()V
         7: astore_1
         8: getstatic     #4                  // Field java/lang/System.out:
            Ljava/io/PrintStream;
        11: aload_1
        12: ldc           #5                  // String hello
        14: ldc           #6                  // String world
        16: invokeinterface #7,  3            // InterfaceMethod java/util/
           Comparator.compare:(Ljava/lang/Object;Ljava/lang/Object;)I
        21: invokevirtual #8                  // Method java/io/PrintStream.
           println:(I)V
        24: return
      LineNumberTable:
        line 5: 0
        line 11: 8
        line 12: 24
}
SourceFile: "AnonymousInterfaceImplementation.java"
```

```
84 InnerClasses:
85      static #2; //class AnonymousInterfaceImplementation$1
```

AnonymousInterfaceImplementation$1 decompiled heap

```
Classfile /home/luisa/UnB/TCC/Codes/LambdaE/AnonymousInterfaceImplementation$1.class
  Last modified Jun 2, 2021; size 752 bytes
  MD5 checksum 0582d7532fa523129017c854c8680319
  Compiled from "AnonymousInterfaceImplementation.java"
final class AnonymousInterfaceImplementation$1 extends java.lang.Object implements java.util.Comparator<java.lang.String>
  minor version: 0
  major version: 52
  flags: ACC_FINAL, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#22         // java/lang/Object."<init>":()V
   #2 = Methodref          #3.#23         // java/lang/String.compareTo:(Ljava/lang/String;)I
   #3 = Class              #24            // java/lang/String
   #4 = Methodref          #5.#25         // AnonymousInterfaceImplementation$1.compare:(Ljava/lang/String;Ljava/lang/String;)I
   #5 = Class              #26            // AnonymousInterfaceImplementation$1
   #6 = Class              #28            // java/lang/Object
   #7 = Class              #29            // java/util/Comparator
   #8 = Utf8               <init>
   #9 = Utf8               ()V
  #10 = Utf8               Code
  #11 = Utf8               LineNumberTable
  #12 = Utf8               compare
  #13 = Utf8               (Ljava/lang/String;Ljava/lang/String;)I
  #14 = Utf8               (Ljava/lang/Object;Ljava/lang/Object;)I
  #15 = Utf8               Signature
  #16 = Utf8               Ljava/lang/Object;Ljava/util/Comparator<Ljava/lang/String;>;
  #17 = Utf8               SourceFile
  #18 = Utf8               AnonymousInterfaceImplementation.java
  #19 = Utf8               EnclosingMethod
  #20 = Class              #30            // AnonymousInterfaceImplementation
  #21 = NameAndType        #31:#32        // main:([Ljava/lang/String;)V
  #22 = NameAndType        #8:#9          // "<init>":()V
  #23 = NameAndType        #33:#34        // compareTo:(Ljava/lang/String;)I
  #24 = Utf8               java/lang/String
  #25 = NameAndType        #12:#13        // compare:(Ljava/lang/String;Ljava/lang/String;)I
  #26 = Utf8               AnonymousInterfaceImplementation$1
  #27 = Utf8               InnerClasses
  #28 = Utf8               java/lang/Object
  #29 = Utf8               java/util/Comparator
  #30 = Utf8               AnonymousInterfaceImplementation
```

```
  #31 = Utf8               main
  #32 = Utf8               ([Ljava/lang/String;)V
  #33 = Utf8               compareTo
  #34 = Utf8               (Ljava/lang/String;)I
{
  AnonymousInterfaceImplementation$1();
    descriptor: ()V
    flags:
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<
            init>":()V
         4: return
      LineNumberTable:
        line 5: 0

  public int compare(java.lang.String, java.lang.String);
    descriptor: (Ljava/lang/String;Ljava/lang/String;)I
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=3
         0: aload_1
         1: aload_2
         2: invokevirtual #2                  // Method java/lang/String.
            compareTo:(Ljava/lang/String;)I
         5: ireturn
      LineNumberTable:
        line 8: 0

  public int compare(java.lang.Object, java.lang.Object);
    descriptor: (Ljava/lang/Object;Ljava/lang/Object;)I
    flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=3, locals=3, args_size=3
         0: aload_0
         1: aload_1
         2: checkcast     #3                  // class java/lang/String
         5: aload_2
         6: checkcast     #3                  // class java/lang/String
         9: invokevirtual #4                  // Method compare:(Ljava/lang/
            String;Ljava/lang/String;)I
        12: ireturn
      LineNumberTable:
        line 5: 0
}
Signature: #16                          // Ljava/lang/Object;Ljava/util/
   Comparator<Ljava/lang/String;>;
```

```
84 SourceFile: "AnonymousInterfaceImplementation.java"
85 EnclosingMethod: #20.#21                // AnonymousInterfaceImplementation.
       main
86 InnerClasses:
87      static #5; //class AnonymousInterfaceImplementation$1
```

# Apêndice C

# Palindromes

Palindromes.jimple

```
public synchronized class Palindromes extends java.lang.Object
{

    public void <init>()
    {
        Palindromes r0;


        r0 := @this: Palindromes;

     L2001750510:

        specialinvoke r0.<java.lang.Object: void <init>()>();

        return;

     L533278308:
    }

    public static void main(java.lang.String[]) throws java.lang.Exception
    {
        java.util.List r2;
        java.lang.String[] r1;
        java.lang.Runnable r4;
        java.util.Comparator r3;
        java.lang.Thread r5;
        unknown[] $r1;
        java.util.List $r2;
        java.util.Comparator $r3;
        java.lang.Runnable $r4;
        java.lang.Thread $r5;


        i1 := @parameter0: java.lang.String[];

     L486937144:

        $r1 = newarray (unknown)[9];

        $r1[0] = "Hannah";

        $r1[1] = "Ana";

        $r1[2] = "Elle";

        $r1[3] = "Bob";

```

```
48          $r1[4] = "Otto";
49
50          $r1[5] = "Natan";
51
52          $r1[6] = "Luisa";
53
54          $r1[7] = "Andre";
55
56          $r1[8] = "Renata";
57
58          $r2 = staticinvoke <java.util.Arrays: java.util.List
            ↪  asList(java.lang.Object[])>($r1);
59
60          r2 = $r2;
61
62      L788447014:
63
64          $r3 = staticinvoke <lambda$0: java.util.Comparator bootstrap$()>();
65
66          r3 = $r3;
67
68      L807574510:
69
70          $r4 = staticinvoke <lambda$1: java.lang.Runnable bootstrap$(java.util.List,
            ↪  java.util.Comparator)>(r2, r3);
71
72          r4 = $r4;
73
74      L488167424:
75
76          $r5 = new java.lang.Thread;
77
78          specialinvoke $r5.<java.lang.Thread: void <init>(java.lang.Runnable)>(r4);
79
80          r5 = $r5;
81
82      L1062025338:
83
84          virtualinvoke r5.<java.lang.Thread: void start()>();
85
86      L1086893501:
87
88          virtualinvoke r5.<java.lang.Thread: void join()>();
89
90      L457946492:
91
92          return;
93
```

```
 L1293140103:
}

private static TODO int lambda$0(java.lang.String, java.lang.String)
{
    java.lang.String r2;
    java.lang.String r1;
    java.lang.String $r1;
    java.lang.StringBuilder $r2;
    java.lang.StringBuilder $r3;
    java.lang.String $r4;
    java.lang.String $r5;
    int $r6;


    i1 := @parameter0: java.lang.String;

    i2 := @parameter1: java.lang.String;

 L1733612317:

    $r1 = virtualinvoke r1.<java.lang.String: java.lang.String toLowerCase()>();

 L881864840:

    $r2 = new java.lang.StringBuilder;

    specialinvoke $r2.<java.lang.StringBuilder: void <init>(java.lang.String)>(r2);

    $r3 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder
    ↪  reverse()>();

    $r4 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.String
    ↪  toString()>();

    $r5 = virtualinvoke $r4.<java.lang.String: java.lang.String toLowerCase()>();

    $r6 = virtualinvoke $r1.<java.lang.String: int
    ↪  compareTo(java.lang.String)>($r5);

    return $r6;

 L79761828:
}

private static TODO void lambda$1(java.util.List, java.util.Comparator)
{
    unknown i0;
```

```
        java.util.List i1;
        java.util.Comparator i2;
        java.util.function.Consumer $r1;


        i1 := @parameter0: java.util.List;

        i2 := @parameter1: java.util.Comparator;

     L1111349883:

        $r1 = staticinvoke <lambda$2: java.util.function.Consumer
        ↪  bootstrap$(java.util.Comparator)>(i1);

        interfaceinvoke i0.<java.util.List: void
        ↪  forEach(java.util.function.Consumer)>($r1);

     L1615985432:

        return;
    }

    private static TODO void lambda$2(java.util.Comparator, java.lang.String)
    {
        java.lang.String r1;
        unknown i0;
        int $r1;
        java.io.PrintStream $r2;
        java.lang.StringBuilder $r3;
        java.lang.String $r4;
        java.lang.StringBuilder $r5;
        java.lang.String $r6;
        java.io.PrintStream $r7;
        java.lang.StringBuilder $r8;
        java.lang.String $r9;
        java.lang.StringBuilder $r10;
        java.lang.String $r11;


        i1 := @parameter0: java.util.Comparator;

        i2 := @parameter1: java.lang.String;

     L1865442727:

        $r1 = interfaceinvoke i0.<java.util.Comparator: int compare(java.lang.Object,
        ↪  java.lang.Object)>(r1, r1);

```

```
184         if $r1 != 0 goto L281944022;
185
186     L1146230503:
187
188         $r2 = <java.lang.System: java.io.PrintStream out>;
189
190         $r3 = new java.lang.StringBuilder;
191
192         $r4 = staticinvoke <java.lang.String: java.lang.String
            ↪  valueOf(java.lang.Object)>(r1);
193
194         specialinvoke $r3.<java.lang.StringBuilder: void <init>(java.lang.String)>($r4);
195
196         $r5 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder
            ↪  append(java.lang.String)>(" is a palindrome.");
197
198         $r6 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.String
            ↪  toString()>();
199
200         virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String)>($r6);
201
202     L681791127:
203
204         return;
205
206         goto L304202033;
207
208     L281944022:
209
210         $r7 = <java.lang.System: java.io.PrintStream out>;
211
212         $r8 = new java.lang.StringBuilder;
213
214         $r9 = staticinvoke <java.lang.String: java.lang.String
            ↪  valueOf(java.lang.Object)>(r1);
215
216         specialinvoke $r8.<java.lang.StringBuilder: void <init>(java.lang.String)>($r9);
217
218         $r10 = virtualinvoke $r8.<java.lang.StringBuilder: java.lang.StringBuilder
            ↪  append(java.lang.String)>(" is not a palindrome.");
219
220         $r11 = virtualinvoke $r10.<java.lang.StringBuilder: java.lang.String
            ↪  toString()>();
221
222         virtualinvoke $r7.<java.io.PrintStream: void println(java.lang.String)>($r11);
223
224         return;
225
```

```
      L304202033:

      L1920027660:
    }

}
```

Palindromes$lambda$0.jimple

```
public final class Palindromes$lambda$0 extends java.lang.Object implements
↪  java.util.Comparator
{

    public static java.util.Comparator bootstrap$()
    {
        Palindromes$lambda$0 $r0;


        $r0 = new Palindromes$lambda$0;

        specialinvoke $r0.<Palindromes$lambda$0: void <init>()>();

        return $r0;
    }

    public static void <init>()
    {
        Palindromes$lambda$0 $r0;


        $r0 := @this: Palindromes$lambda$0;

        specialinvoke $r0.<java.lang.Object: void <init>()>();

        return;
    }

    public static int compare(java.lang.Object, java.lang.Object)
    {
        Palindromes$lambda$0 $r0;
        java.lang.Object $r1;
        java.lang.Object $r2;
        java.lang.String $r3;
        java.lang.String $r4;
        int $i0;


        $r0 := @this: Palindromes$lambda$0;

        $r1 := @parameter0: java.lang.Object;

        $r2 := @parameter1: java.lang.Object;

        $r3 = (java.lang.String) $r1;

        $r4 = (java.lang.String) $r2;
```

```
47
48          $i0 = staticinvoke <Palindromes: int lambda$0(java.lang.String,
            ↪  java.lang.String)>($r3, $r4);
49
50          return $i0;
51      }
52
53  }
```

Palindromes$lambda$1.jimple

```
public final class Palindromes$lambda$1 extends java.lang.Object implements
↪  java.lang.Runnable
{


     java.util.List cap0;
     java.util.Comparator cap1;

    public static java.lang.Runnable bootstrap$(java.util.List, java.util.Comparator)
    {
        java.util.List $r0;
        java.util.Comparator $r1;
        Palindromes$lambda$1 $r2;


        $r0 := @parameter0: java.util.List;

        $r1 := @parameter1: java.util.Comparator;

        $r2 = new Palindromes$lambda$1;

        specialinvoke $r2.<Palindromes$lambda$1: void <init>(java.util.List,
        ↪  java.util.Comparator)>($r0, $r1);

        return $r2;
    }

    public static void <init>(java.util.List, java.util.Comparator)
    {
        Palindromes$lambda$1 $r0;
        java.util.List $r1;
        java.util.Comparator $r2;


        $r0 := @this: Palindromes$lambda$1;

        $r1 := @parameter0: java.util.List;

        $r2 := @parameter1: java.util.Comparator;

        specialinvoke $r0.<java.lang.Object: void <init>()>();

        $r0.<Palindromes$lambda$1: java.util.List cap0> = $r1;

        $r0.<Palindromes$lambda$1: java.util.Comparator cap1> = $r2;

        return;
    }
```

```
46
47      public static void run()
48      {
49          Palindromes$lambda$1 $r0;
50          java.util.List $r1;
51          java.util.Comparator $r2;
52
53
54          $r0 := @this: Palindromes$lambda$1;
55
56          $r1 = $r0.<Palindromes$lambda$1: java.util.List cap0>;
57
58          $r2 = $r0.<Palindromes$lambda$1: java.util.Comparator cap1>;
59
60          staticinvoke <Palindromes: void lambda$1(java.util.List,
            ↪  java.util.Comparator)>($r1, $r2);
61
62          return;
63      }
64
65  }
```

Palindromes$lambda$2.jimple

```
public final class Palindromes$lambda$2 extends java.lang.Object implements
↪  java.util.function.Consumer
{


     java.util.Comparator cap0;

    public static java.util.function.Consumer bootstrap$(java.util.Comparator)
    {
        java.util.Comparator $r0;
        Palindromes$lambda$2 $r1;


        $r0 := @parameter0: java.util.Comparator;

        $r1 = new Palindromes$lambda$2;

        specialinvoke $r1.<Palindromes$lambda$2: void
        ↪  <init>(java.util.Comparator)>($r0);

        return $r1;
    }

    public static void <init>(java.util.Comparator)
    {
        Palindromes$lambda$2 $r0;
        java.util.Comparator $r1;


        $r0 := @this: Palindromes$lambda$2;

        $r1 := @parameter0: java.util.Comparator;

        specialinvoke $r0.<java.lang.Object: void <init>()>();

        $r0.<Palindromes$lambda$2: java.util.Comparator cap0> = $r1;

        return;
    }

    public static void accept(java.lang.Object)
    {
        Palindromes$lambda$2 $r0;
        java.lang.Object $r1;
        java.lang.String $r2;
        java.util.Comparator $r3;


```

```
46          $r0 := @this: Palindromes$lambda$2;
47
48          $r1 := @parameter0: java.lang.Object;
49
50          $r2 = (java.lang.String) $r1;
51
52          $r3 = $r0.<Palindromes$lambda$2: java.util.Comparator cap0>;
53
54          staticinvoke <Palindromes: void lambda$2(java.util.Comparator,
            ↪  java.lang.String)>($r3, $r2);
55
56          return;
57      }
58
59  }
```

babelbib: Using the BibTeX style's default font for 'name'. babelbib: Using the BibTeX style's default font for 'lastname'. babelbib: Using the BibTeX style's default font for 'title'. babelbib: Using the BibTeX style's default font for 'jtitle'. babelbib: Using the BibTeX style's default font for 'etal'. babelbib: Using the BibTeX style's default font for 'journal'. babelbib: Using the BibTeX style's default font for 'volume'. babelbib: Using the BibTeX style's default font for 'ISBN'. babelbib: Using the BibTeX style's default font for 'ISSN'. babelbib: Using the BibTeX style's default font for 'url'. babelbib: Using the BibTeX style's default font for 'numeral'.

# Referências


[1] Clausen, L.: *A java bytecode optimizer using side-effect analysis*. Concurrency and Computation: Practice and Experience, 9:1031–1045, 1997.

[2] Evans, Benjamin J., James Gough e Chris Newland: *Optimizing Java: Practical Techniques for Improving JVM Application Performance*. O'Reilly Media, Inc., 1st edição, 2018, ISBN 1492025798.

[3] Vallée-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam e Vijay Sundaresan: *Soot - a java bytecode optimization framework*. Em *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, página 13. IBM Press, 1999.

[4] Vallée-Rai, Raja, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville e Vijay Sundaresan: *Optimizing java bytecode using the soot framework: Is it feasible?* Em *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, página 18–34, Berlin, Heidelberg, 2000. Springer-Verlag, ISBN 354067263X.

[5] Nielson, Flemming, Hanne R. Nielson e Chris Hankin: *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010, ISBN 3642084745.

[6] Lam, Patrick, Eric Bodden, Laurie Hendren e Technische Universität Darmstadt: *The soot framework for java program analysis: a retrospective*.

[7] Do, L.: *User-centered tool design for data-flow analysis*. 2019.

[8] Amme, Wolfram, Marc André Möller e Philipp Adler: *Data flow analysis as a general concept for the transport of verifiable program annotations*. Electronic Notes in Theoretical Computer Science, 176:97–108, julho 2007.

[9] Bodden, Eric: *Invokedynamic support in soot*. Em *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, página 51–55, New York, NY, USA, 2012. Association for Computing Machinery, ISBN 9781450314909. https://doi.org/10.1145/2259051.2259059.

[10] Khedker, Uday, Amitabha Sanyal e Bageshri Karkare: *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., USA, 1st edição, 2009, ISBN 0849328802.

[11] Neward, Ted: *Java 8: Lambdas, part 1*, Jul 2013. https://www.oracle.com/technical-resources/articles/java/architect-lambdas-part1.html.

[12] Evans, Ben: *Behind the scenes: How do lambda expressions really work in java?* Java Magazine, Sep 2020. https://blogs.oracle.com/javamagazine/behind-the-scenes-how-do-lambda-expressions-really-work-in-java.

[13] Nielsen, Janus Dam: *A survivor's guide to java program analysis with soot, 2008. available from http://www.brics.dk/sootguide*.

[14] Klint, Paul, Tijs van der Storm e Jurgen Vinju: *Easy meta-programming with rascal*. páginas 222–289, julho 2009, ISBN 978-3-642-18022-4.