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

Integrando a Representação Static Single Assignment no Rascal Jimple Framework

Mateus Luiz Freitas Barros

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifacio de Almeida

Brasília
2021

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

Integrando a Representação Static Single Assignment no Rascal Jimple Framework

Mateus Luiz Freitas Barros

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof. Dr. Edna Dias Canedo
CIC / UnB

Aluno de Doutorado Pedro Henrique Teixeira Costa
CIC / UnB

do Bacharelado em Ciência da Computação

Brasília, 19 de julho de 2021

Dedicatória

Este trabalho é dedicado aos meus colegas e amigos que estão cursando, que se formaram e que abandonaram Ciência da Computação e cursos relacionados da Universidade de Brasília. Alunos esses que passam por problemas financeiros, sociais e emocionais.

Eu senti parte do que vocês passam e passaram. Esse trabalho é uma conquista coletiva, dedicado a vocês.

Agradecimentos

Esse trabalho foi realizado graças ao suporte acadêmico e emocional do professor Rodrigo Bonifácio, um professor humano a quem sou muito grato por ter tido a oportunidade de ser orientado não só no TCC, mas também no curso.

Também agradeço a toda minha família que garantiu a base e a educação para eu conseguir alcançar um curso superior e ter acesso oportunidades que nunca seriam possíveis sem eles.

Eu agradeço a todos os meus amigos que conheci no ambiente da universidade. Pessoas que me inspiram até hoje e sempre fazem eu tentar ser o melhor de mim. Assim como meus amigos da empresa bxblue, que me deram todo o apoio, espaço e oportunidade para eu conseguir me desenvolver como um bom desenvolvedor de software enquanto busco minha formação.

E por último, mas não menos importante, um agradecimento especial para o Uriel, Daniels, Marcola, Wlad, Rebores, Rods, Kilmer e todos os amigos da Central/CJR que deram suporte em várias áreas diferentes durante esses anos.

Eu sou grato por vocês todos.

Resumo

Este trabalho apresenta a implementação de um algoritmo de transformação de programas em Jimple para a representação *Static Single Assingment* (SSA). A representação SSA objetiva a simplificação de análises e transformações de programas em códigos de instruções de 3 endereços. Exemplos de análises e transformações que se beneficiam da representação SSA incluem os algoritmos de *data-flow analysis* para otimização de código e algoritmos de *taint-analysis* usados na área de segurança de software. A implementação proposta abrange as principais etapas descritas na literatura para a geração de SSA, como inserção de *phi-functions* e renomeação de variáveis.

Palavras-chave: SSA, Static Single Assignment, Rascal

Abstract

This work presents an implementation of the Static Single Assignment transformation algorithm for Jimple programs. The SSA representation aims to simplify the analysis and transformations for 3 address instruction codes. Examples of analysis and transformation algorithms that takes benefits from it are data-flow algorithms for code optimization and taint-analysis algorithms used in software security. The implementation covers the main steps described by the literature for the SSA transformation, such as phi-function insertion and variable renaming.

Keywords: SSA, Static Single Assignment, Rascal Jimple Framework

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O processo de otimização de programas ao nível de código intermediário é uma prática comum em compiladores e interpretadores [1]. Tais otimizações visam eliminar código redundante ou modificar trechos de código para serem mais performáticos [2]. Tipicamente, compiladores e interpretadores se beneficiam de algoritmos que analisam fluxos de dados, com o objetivo de identificar as oportunidades de otimização. Essas análises são realizadas a partir de um *Control-Flow Graph* (CFG), uma representação do código com os possíveis fluxos em um programa [1]. O vértices de um CFG representam os *statements* de um programa; enquanto que as arestas entre dois vértices $v_1 \rightarrow v_2$ indicam a existência de uma transição de v_1 para v_2.

Análises e otimizações de programa também se beneficiam do uso de representações intermediárias mais propícias que o código fonte e que o código binário. Por exemplo, a linguagem Java é compilada para um formato de *bytecode* (*classfile*), cujas instruções são interpretadas pela *Java Virtual Machine* (JVM), uma *stack-based machine*. O formato de *bytecode* não favorece a análise e transformação de programas, uma vez que não deixa o fluxo de controle explícito. Para mitigar esse problema, frameworks de análise e transformação de programas para Java (como o Soot) fazem uso de representações intermediárias em *three-address code* (como o JIMPLE [2]).

A aplicação do formato proposto pelo JIMPLE abre espaço para o uso de diferentes técnicas de análise como a verificação detalhada do uso de ponteiros ou então a análise da quantidade de atribuições para uma variável [3] [1], isso porque essas técnicas tiram proveito do formato de 3 endereços. A representação *Static Single Assignment* (SSA) extende o formato de 3 endereços e torna ainda mais direta a implementação de alguns tipos de análises. Por exemplo, a representação SSA torna explícita quais definições de variáveis (atribuições) chegam em um determinado *statement*. Na representação SSA, o CFG é modificado para representar um código que garante que cada variável declarada possui apenas uma única atribuição. A representação SSA facilita a implementação de

vários algoritmos de análise e transformação de programas [4]. Conforme mencinonado, a representação SSA leva a uma forte restrição: só deve existir apenas uma atribuição a uma variável. Obedecer essa regra requer técnicas de resolução de ambiguidades, particularmente quando dois *branches* de código atualizam a mesma variável. A geração da representação intermediária SSA, a partir de um código JIMPLE, por exemplo, envolve dois estágios. No primeiro a inserção de um novo conceito para lidar com a ambiguidade, chamada inserções de *phi-functions*. O segundo estágio envolve a renomeação de variáveis [4]. Cada uma dessas etapas são acompanhadas estruturas de dados auxiliares criadas em tempo de execução, como a árvore de dominância e as fronteiras de dominância do CFG [5, 4, 6].

O *JimpleFramework* é um projeto de pesquisa e desenvolvimento que objetiva facilitar a implementação de algoritmos de análise e transformação de programas para linguagens compiladas para a JVM [7]. O *JimpleFramework*, implementado na linguagem de metaprogramação Rascal, já suporta uma série de algoritmos de análise estática, incluindo a geração de *Call Graph*, *Control Flow Graph*, *Data Flow Analysis* e *Points-to Analysis*. Por outro lado, até a realização deste projeto, o *JimpleFramework* não suportava o formato SSA. Desta forma, o principal objetivo deste trabalho de graduação foi explorar diferentes algoritmos para a geração da representação intermediária SSA e integrar a geração SSA no *JimpleFramework*. O algoritmo de transformação SSA aqui apresentado reproduz as etapas necessárias para se realizar a transformação assim como a construção das estruturas de dados auxiliares.

Este documento é dividido em quatro capítulos, com o primeiro capítulo sendo esta Introdução. O segundo capítulo de Fundamentação Teórica apresenta os conceitos essenciais para o entendimento e realização do projeto. O terceiro capítulo apresenta a implementação da transformação de JIMPLE pura para a representação SSA, assim como as limitações do programa, listagem de testes, resultados e os trabalhos futuros. E por último é apresentada a conclusão e as considerações finais do trabalho.

Capítulo 2

Fundamentação Teórica

Este capitulo apresenta a fundamentação necessário para um melhor entendimento deste trabalho. Inicialmente é feita uma introdução sobre análise de programas na Seção 2.1. A Seção **??** aborda a representação *Static Single Assignment* e os algoritmos usados para gerar tal representação. Por fim, a Seção **??** descreve a representação intermediária JIMPLE, e como ele se relaciona com os conceitos apresentados anteriormente.

2.1 Análise de programas

A análise estática de programas consiste em um conjunto de técnicas, tipicamente aplicadas em tempo de compilação, para extrair fatos sobre os programas. Os resultados das análises podem ser usados para atingir diferentes objetivos, como, por exemplo, realizar otimização para remover código redundante ou identificar vulnerabilidades em software. Para decidir quais otimizações podem ser aplicadas em um programa, devemos aplicar técnicas que buscam insumos no fluxo do programa. Para realizar tal tarefa, podemos aplicar a análise de fluxo chamada *Data-flow*. Com as informações recuperadas e estruturadas, é possível realizar novas análises usando as informações do grafo gerado pelo *Data-flow*, como por exemplo *reaching definitions* e *using definitions* [1].

Programas antes da compilação ou interpretação, tendem a conter estruturas e declarações que não são necessariamente escritas da forma mais otimizada, podendo ser muitas vezes até redundantes. Para lidar com esse fato de forma automatizada, podemos aplicar diversas análises e melhorias no código em tempo de compilação ou até mesmo de execução [1][8].

Entre as etapas possíveis do processo de análise do programa, a que avalia os fluxos de dados de um programa é chamada *Data-flow analysis*. Comumente a representação do *Data-Flow* é na forma de um grafo [8], onde os nós e as relações entre si definem o fluxo de dados pelo qual o programa poderá percorrer durante sua execução. Uma vez o caminho

do programa devidamente mapeado, é possível aplicar uma série de transformações e otimizações no código a ser executado [1], como a propagação de constantes ou a remoção de código morto.

Para maximizar a obtenção de informações acerca da estrutura de um programa, a representação de grafo se mostra adequada, dado que seus formalismos matemáticos definem bem a relações entre os nós. Neste estudo, o grafo que representa os caminhos de um programa será chamado *Control-Flow Graph*, ou na sua forma abreviada, CFG. Os nós de um CFG são representações de blocos do programa (ou *statements*), em que esses blocos podem ser as instruções a serem executadas [1]. Já as relações entre esses blocos são representadas pelas arestas do grafo. Apresentamos a seguir um exemplo de código e seu respectivo *Control-Flow Graph*:

Algorithm 1: Código exemplo

```
bt ← true;
if bt == then
    v1 ← 1;
else
    v1 ← 2;
print v1;
```

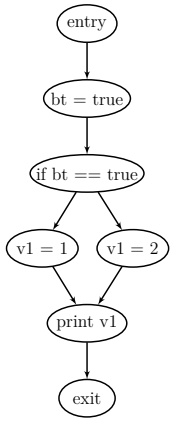

Figura 2.1: Grafo do código de exemplo

Com o CFG construído, é possível executar as transformações para se obter insumos do código e realizar uma eventual otimização. Entre as possíveis análises que podemos fazer, as comumente citadas são *points-to analysis* e *constant propagation* [3] [1]. A primeira consiste em descobrir todos os objetos e referências que uma única variável pode apontar [5] além de outras informações, já a segunda, visa transformar as variáveis que não mudam de valor em apenas uma constante. Este trabalho não contempla em detalhes nenhum tipo de otimização no CFG, é explorado apenas a transformação de código que viabiliza esses e outros tipos de otimizações.

Já análise de *reaching definiton* é uma técnica que pode ser aplicada diretamente no CFG gerado na *Data-flow analysis*. Ela consiste em avaliar o programa para buscar definições de variáveis feitas em cada ponto do fluxo do programa [2] [1]. O algoritmo *Single Static Assignment* usa esse tipo de análise de *reaching definition* para buscar fluxos específicos que uma dada variável é manipulada. São explicados na próxima seção os detalhes do algoritmo e como a *reaching defintions* é usada no mesmo.

2.2 Static Single Assignment

Após a execução de algoritmos para construção do CFG e a computação das relações de *def-use* e *use-def*, a partir de um algoritmo *reaching definitions*, é possível realizar análises e transformações de programas. Uma transformação de particular interesse é a geração da representação intermediária *Static Single Assignment*. A representação SSA requer que todas as variáveis que apareçam em um dado programa recebam uma única atribuição. Para obter tal propriedade, os algoritmos de geração SSA envolvem a inclusão de *phi-function* e *variable renaming* [4]. Algumas estruturas precisam ser definidas e computadas, sendo elas *Immediate Dominators* de vértices, *Dominance Frontier* e *Dominance Tree* [5][6]:

- Quando todos os caminhos para chegar em um nó B precisam passar por um dado nó A, é dito que A domina B.
- Se A domina B e ambos diferem, então A domina estritamente B.
- O *immediate dominator* de um nó B é o vértice mais próximo que o domina estritamente em qualquer caminho do início até B.
- É dito que a *dominance frontier* de um nó A é um conjunto de nós Z, se A dominar um antecessor de Z, mas não dominar Z de forma estrita.
- A *dominance tree* de um CFG é uma árvore que representa a relação de dominância entre os nós. O nó raiz corresponde ao ponto de entrada do programa, e os filhos são os nós onde seu pai é seu dominador imediato.

Uma vez que temos essas estruturas computadas, é possível aplicar algoritmos [6] que se utilizam das mesmas para transformar o CFG, assim satisfazendo as propriedades do SSA. O primeiro algoritmo objetiva a inserção de *phi-functions*. Essa função especial chamada *phi-function*, é responsável por escolher qual variável foi definida e qual valor usado ao executar em um dado fluxo do grafo. Pa ra ilustrar melhor, é apresentado o CFG da Figura 2.2 depois da inserção das *phi-functions*, assim como o algoritmo de cálculo da *Dominance Frontier* e inserção da *phi-function* do *Single Static Assignment Book* [4]:

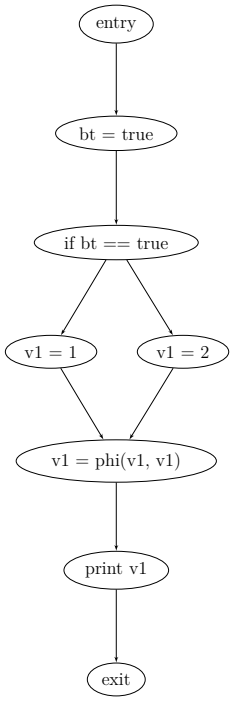

Figura 2.2: CFG do código com inserção de phi-functions

Algorithm 2: Cálculo da *Dominance Frontier* [4]

let DominanceFrontier be a set of frontier nodes for a given node x
for $(a, b) \in CFGedges$ **do**
    $x \leftarrow a$
    while *x does not strictly dominate b* **do**
        $DominanceFrontier(x) \leftarrow DominanceFrontier(x) \bigcup b$
        $x \leftarrow immediateDominator(x)$
return DominanceFrontier

Algorithm 3: Inserção de *Phi-function* [4] [6]

```
for v: variable names in original program do
    F ← {}; // set of blocks where the φ is added
    W ← {}; // set of blocks that contain definitions of a variable
    for d ∈ Defs(v) do
        let B be the basic block containing d;
        W ← W ⋃{B};
    while W ≠ {} do
        remove a basic block X from W;
        for Y : basic block in DominanceFrontier(X) do
            if Y ∉ F then
                add v ← φ(...) at entry of Y in the correspondent CFG;
                F ← F ⋃{Y};
                if Y ∉ Defs(v) then
                    W ← W ⋃{Y};
```

O algoritmo de cálculo da *dominance frontier* define inicialmente um mapa onde a chave são nós e o valor para cada chave é um conjunto de vértices que pertencem à fronteira de dominância. Para cada relação de um nó A para B, guardamos uma variável temporária X com o valor de A e verificamos se X não domina estritamente B. Se X não domina B estritamente, adiciona B ao conjunto de nós da fronteira de X e procura pelo próximo dominador imediato de X, isso é feito para verificar se existem mais vértices em que precisamos adicionar B ao conjunto da fronteira de dominância, esse processo é feito até que X domine estritamente B [4][6]. No fim teremos a *dominance frontier* para cada nó formada pronta para ser usada para inserção de funções phi.

Visto que todas as *phi-functions* forem devidamente inseridas, podemos aplicar a segunda transformação para satisfazer as propriedades *Static Single Assignment*. Essa etapa é chamada *Variable Renaming*, ela consiste em garantir que todas as variáveis sejam únicas no programa, satisfazendo assim a propriedade de que todas as variáveis só devem ter uma única atribuição [4]. É apresentado o CFG da figura 2.3 depois de aplicado a renomeação de variáveis, assim como o algoritmo de renomeação descrito por Cytron et al. [6]:

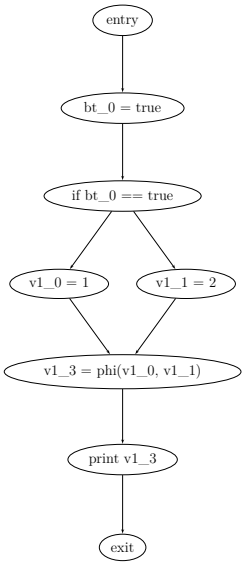

Figura 2.3: CFG do código com a renomeação de variáveis

Algorithm 4: Algoritmo de renomeação de variáveis [6]

$C(*) \leftarrow 0$;
$S(*) \leftarrow EmptyStack$;
$SEARCH(X)$:
  for *A: assignment in X* **do**
    if *A is an ordinary assignment* **then**
      for *assignment* $V \in RHS(A)$ **do**
        Replace use of V by use of V_i where $i = Top(S(V))$;
    let V be $LHS(A)$
      $i \leftarrow C(V)$;
      replace V by V_i as LHS(A);
      push i onto $S(V)$;
      $C(V) \leftarrow i + 1$;
  for $Y \in Succ(X)$ **do**
    $j \leftarrow WhichPred(Y, X)$;
    for $\phi function\ F \in Y$ **do**
      Replace the *j-th* operant V of F by V_i where $i = Top(S(V))$;
  for $Y \in Children(X)$ **do**
    call SEARCH(Y);
  for *assingment* $A \in X$ **do**
    pop S(oldLHS(A));

Para se ter sucesso na transformação proposta pelo SSA, é esperado que o programa esteja estruturado em um formato de três endereços. Muitas linguagens de programação, como linguagens baseadas na *JVM*, não são estruturadas dessa forma por padrão [9], então é necessário realizar uma transformação antes de aplicarmos o *Single Stagic Assignment*. O JIMPLE é uma das especificações que propõe esse formato para linguagens compiladas para *bytecode*.

2.3 JIMPLE

O formato *classfile* representa o código intermediário *bytecode* usado pela máquina virtual Java para tornar o código portável para outros sistemas e dispositivos [9]. A JVM (Java Virtual Machine) é uma máquina virtual que executa as instruções de forma empilhada, deste modo, as instruções do código intermediário são organizadas para obedecer essa execução em pilha. Devido a sua documentação bem definida e estrita, o *bytecode* precisa apenas de uma JVM compilada para que o sistema em questão que o execute.

Representar o código em *bytecode* padrão do Java tem suas vantagens, como o fato de ser uma especificação bem documentada e além der ser um código final, ou seja, não precisamos aplicar mais transforações para obtermos o código em pilha [10]. Porém, para serem feitas otimizações, o processo se torna mais complexo, dado que os algoritmos maduros para essa categoria de tarefa trabalham em um código de instruções de 3 operadores. Além disso, temos outros fatores que podem inviabilizar a transformação, como a verbosidade para representar expressões em pilha. Abaixo é apresentado um exemplo de código Java e sua representação em JIMPLE:

```
1   public class Fibonacci {
2     public void run(int maxNumber) {
3       int previousNumber = 0;
4       int nextNumber = 1;
5
6       for (int i = 1; i <= maxNumber; ++i) {
7         int sum = previousNumber + nextNumber;
8         previousNumber = nextNumber;
9         nextNumber = sum;
10      }
11    }
12  }
```

Listing 2.1: Código Fibonacci em Java

```
    Fibonacci r0;
    int $r1, r1, r2, r3, r5, r4;

    r0 = @this
    i1 = @parameter0
    r2 = 0
    r3 = 1
    r4 = 1
    goto label2

    label 1
    $r1 = r2 + r3
    r5 = $r1
    r2 = r3
    r3 = r5
    r4 = r4 + 1

    label2
    if r4 <= r1
      goto label1

    return
```

Listing 2.2: Código Fibonacci em formato JIMPLE

O JIMPLE tenta tirar proveito das transformações e otimizações possíveis em código de 3 operadores, como *point-to analysis* e *constant propagation* [10]. Esta representação pretende apresentar o código em pilha do *bytecode* na estrutura de 3 operadores, podendo assim, tomar proveito das técnicas de transformações maduras e previamente estudadas para nesse tipo de formato. Uma dessas técnicas é a própria aplicação proposta pelo *Single Static Assignment*, descrita anteriormente.

Capítulo 3

Implementação

O formato JIMPLE é uma representação mais adequada para realizar análises e transformações de programa, quando comparado com o código no formato de bytecode[10]. Visando tirar proveito das vantagens da representação, o projeto *JimpleFramework* adereça esses pontos através de uma implementação da especificação JIMPLE na linguagem de programação Rascal, projeto inspirado pela implementação Java no *framework* Soot [10]. Como uma forma de contribuir com o JimpleFramework, este estudo busca implementar uma transformação de código JIMPLE para o formato SSA (*Static Single Assignment*) [4], usando a linguagem de programação Rascal.

3.1 JimpleFramework

O projeto *JimpleFramework* é um arcabouço para análise de programas Java desenvolvido na linguagem de metaprogramação chamada Rascal [7]. A linguagem Rascal disponibiliza uma série de abstrações e estruturas de dados que facilitam a manipulação de linguagens de programação. As ferramentas criadas no projeto e as facilidades do Rascal são usadas em conjunto para a realização da transformação *SSA* apresentada nesse trabalho.

JimpleFramework é um projeto que possuí vários módulos que visam em conjunto transformar um código em pilha *bytecode* em uma estrutura em três endereços. Os quatro módulos principais são: *analysis*, *core*, *decompiler* e *toolkit*. O módulo de *analysis* é responsável por executar operações de *data-flow* [1]. Já o *core* possui recursos relacionados à construção e estrutura do código de 3 endereços, incluindo a sintaxe abstrata da linguagem JIMPLE. O *decompiler* contem as funcionalidades relacionadas à decompilação do *classfile* para um formato abstrato de uso interno. E por último o *toolkit*, que agrega uma série de funções que aplicam alguma transformação na estrutura do código, como por exemplo a geração de um *Control-Flow Graph*.

Rascal é uma linguagem de programação com foco em metaprogramação, permitindo analisar, transformar e exportar código fonte com mais facilidade, pois toda a integração desses componentes estão implementados de forma nativa na linguagem [11]. Os recursos de *visitors* e *pattern matching* da linguagem Rascal foram usados de forma extensiva no projeto, além das estruturas de dados em grafo, que facilitam a travessia e o batimento de padrão em nós que o formam e representam a execução de um programa.

O principal módulo do JimpleFramework usado é o *toolkit*, mais especificamente o componente responsável por criar o *CFG*, pois com o código representado nessa estrutura podemos realizar as operações descritas nos algoritmos de transformação para SSA [6]. Além dele, também é usado a sintaxe abstrata para fazer o reconhecimento das instruções. A linguagem Rascal permite ainda expressar os algoritmos de forma bem similar a definição formal, além de disponibilizar de forma nativa estruturas de dados que facilitam todas as operações feitas em cima do *CFG*. Na próxima seção será explicado com mais detalhes os componentes relacionados diretamente a transformação SSA.

3.2 Algoritmo de transformação *SSA*

O algoritmo de transformação *SSA* é baseado em duas definições formais presentes no livro *SSA Book* [4] e no artigo *An Efficient Method of Computing Static Single Assignment Form* [6]. A implementação é divida em módulos que representam cada etapa de transformação de um código JIMPLE para o formato *SSA*, são elas: criação da fronteira de dominância, inserção das *phi-functions* e renomeação das variáveis, sendo que cada uma das etapas possuem algum tipo de etapa intermediária internamente e todas são chamadas em sequência em um módulo chamado *Generator*. O resultado das implementações são um conjunto dos algoritmos explorados na literatura, além de possuírem limitações relacionadas a instruções que possam aparecer no programa de entrada. A Figura 3.1 apresenta uma visão de alto nível da implementação cuja arquitetura segue um *pipeline*. Cada etapa do *pipeline* é explorada ao longo deste capítulo.

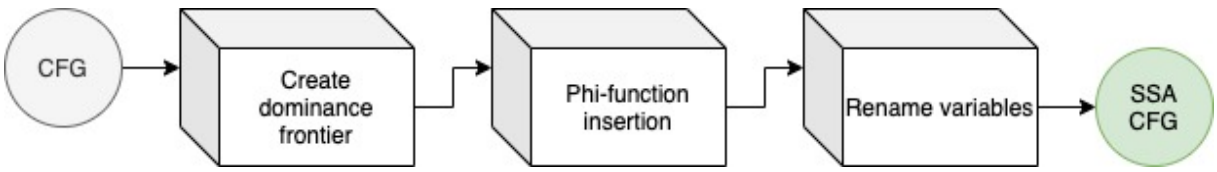

Figura 3.1: Arquitetura do módulo de transformação em *SSA*

A criação da fronteira de dominância se baseia na implementação presente no livro *SSA Book* [4]. Esse algoritmo foi escolhido por conta da simplicidade e expressividade do conceito. A implementação também usa o conceito de árvore de dominância, em que

os filhos de um dado nó representam os nós imediatamente dominados [4]. A rotina que retorna essa árvore está disponível como uma biblioteca em Rascal [12]. Para computar a fronteira de dominância em si, passamos como entrada para o algoritmo o CFG e a árvore de dominância já computados, e o retorno final é uma estrutura de dados em mapa que associa um nó do grafo (**chave**) a uma lista de nós presentes na fronteira. A implementação em Rascal é apresentada a seguir:

```
1  public map[Node, set[Node]] createDominanceFrontier(FlowGraph flowGraph,
       map[&T, set[&T]] dominanceTree) {
2    allNodes = { origin | <origin, _> <- flowGraph } + { destination | <_,
       destination> <- flowGraph };
3
4    dominanceFrontiers = ();
5
6    for(graphNode <- allNodes) {
7      for(predecessor <- predecessors(flowGraph, graphNode)) {
8        tempPredecessor = predecessor;
9        while(tempPredecessor != findIdom(dominanceTree, graphNode)) {
10         dominanceFrontiers[tempPredecessor] = tempPredecessor in
       dominanceFrontiers ? dominanceFrontiers[tempPredecessor] : {};
11         dominanceFrontiers[tempPredecessor] = dominanceFrontiers[
       tempPredecessor] + {graphNode};
12         tempPredecessor = findIdom(dominanceTree, tempPredecessor);
13       };
14     };
15   };
16
17   return dominanceFrontiers;
18 }
```

Listing 3.1: Implementação do cáculo da fronteira de dominância

A função de cálculo da *dominance frontier* apresentada no *SSA Book* é inspirada na definição formal apresentada no artigo *An Efficient Method of Computing Static Single Assignment Form* [4][6]. Essa versão foi escolhida para a implementação em Rascal devido a simplicidade da mesma, porém ela não é a mais performática, pois sua complexidade pode ser quadrática ao calcular todas as fronteiras comparado nó a nó [6]. Existe uma versão do algoritmo [6] que permite o cálculo em tempo linear, em que são utilizados *dominance frontiers* intermediárias e uma estratégia de travessia *bottom-up* da árvore de dominância. Essa versão otimizada não será abordada neste trabalho, apenas a simplificada.

Uma vez que a criação da fronteira de dominância já foi computada, a inserção das *phi-functions* é feita na segunda etapa. Essa implementação é baseada no algoritmo apresentado por Cytron et al. [6], junto com rotinas de suporte para fazer as computações.

O algoritmo recebe como entrada o CFG e a *dominance frontier* e retorna um novo CFG com as *phi-functions* inseridas. A fronteira de dominância é usada especialmente para encontrar as junções onde as *phi-functions* devem ser inseridas. A implementação em Rascal da função é apresentada a seguir:

```
1
2  public FlowGraph insertPhiFunctions(FlowGraph flowGraph, map[&T, set[&T
       ]] dominanceFrontier) {
3    variableList = { getStmtVariable(graphNode) | <graphNode, _> <-
       flowGraph, isVariable(graphNode) };
4
5    for(V <- variableList) {
6      DomFromPlus = ();
7      Work = ();
8      W = {};
9
10     for(X <- blocksWithVariable(flowGraph, V)) {
11       Work[X] = 1;
12       W = W + {X};
13     };
14
15     while(W != {}) {
16       <X, newSet> = takeOneFrom(W);
17       W = newSet;
18
19       if(X in dominanceFrontier) {
20         for(Y <- dominanceFrontier[X]) {
21           if(!(Y in DomFromPlus)) {
22             flowGraph = insertPhiFunction(flowGraph, Y, V); // add
       v    (...) at entry of Y
23             DomFromPlus[Y] = 1;
24             if(!(Y in DomFromPlus)) {
25               Work(Y) = 1;
26               W = W + {Y};
27             };
28           };
29         };
30       };
31     };
32   };
33
34   return flowGraph;
35
36 }
```

Listing 3.2: Implementação do cálculo da inserção das phi-functions

Como pontuado por Cytron et al. [6], o tempo para a execução do algoritmo para uma única variável depende da quantidade de atribuições e de *phi-functions*, mais o número das relações válidas presentes na *dominance frontier*. Com o *CFG* modificado para conter as *phi-functions*, é possível aplicar a etapa de renomeação de variáveis em seguida.

Ou seja, após a inserção das *phi-functions*, temos que garantir que cada variável recebe uma única atribuição, o que requer a renomeação das variáveis. Assim como o módulo de *phi-function*, a nossa implementação é baseada no algoritmo de Cytron et al. [6], com ajuda de várias funções e estruturas de dados internas para auxiliar a execução das substituições e operações no CFG. O algoritmo recebe como entrada o CFG anotado com as *phi-functions* e retorna um novo grafo com as variáveis renomeadas. A implementação em Rascal do algoritmo de renomeação é apresentado a abaixo:

```
1
2  public FlowGraph applyVariableRenaming(FlowGraph flowGraph) {
3    VARIABLE_VERSION_STACK = (); // Stack for each variable that holds the
       next variable to ve replaced
4    VARIABLE_ASSIGNMENT_COUNT = (); // Counts how many assingments have
      been processed for a given variable
5    REPLACED_NODES = {}; // Keep tracking of all nodes replaced in one
      execution
6    LAST_VERSION_REPLACED = (); // Keep tracking of the new version of a
      given node
7
8    ADJACENCIES_MATRIX = createAdjacenciesMatrix(flowGraph); // Mainly
      used to rebuild the renamed flow graph
9    IDOM_TREE = createIdomTree(createDominanceTree(flowGraph)); // Used to
       traverse the flow graph
10
11   map[Node, list[Node]] newBlockTree = replace(entryNode()); // Start
      algorithm
12
13   // Rebuild renamed flowGraph
14   FlowGraph newFlowGraph = {};
15   for(fatherNode <- newBlockTree) {
16     newFlowGraph = newFlowGraph + { <fatherNode, nodeChild> | nodeChild
      <- ADJACENCIES_MATRIX[fatherNode]};
17   };
18
19   return newFlowGraph;
20 }
21
22 public map[Node, list[Node]] replace(Node X) {
23   if((X == exitNode())) return ADJACENCIES_MATRIX;
```

```
24
25  Node oldNode = X;
26
27  // Deal with all nodes that aren't assigments bug uses a variable in
     some way
28  if(isNonAssignmentStatementToRename(X)) {
29    statementBody = returnStmtNodeBody(X);
30    Node renamedStatement = stmtNode(replaceImmediateUse(statementBody))
     ;
31    renameNodeOcurrecies(X, renamedStatement);
32    X = renamedStatement;
33  };
34
35  // Deal will all nodes that are an assigment and are not renamed
36  if(isOrdinaryAssignment(X) && !isRenamed(X)) {
37
38    // Replace right hand side variables
39    list[Immediate] rightHandNodeImmediates =
     returnRightHandSideImmediates(X);
40    for(rightHandSideImmediate <- rightHandNodeImmediates) {
41      newAssignStmt = replaceRightVariableVersion(ADJACENCIES_MATRIX,
     rightHandSideImmediate, X);
42
43      renameNodeOcurrecies(X, newAssignStmt);
44
45      X = newAssignStmt;
46    };
47
48    // Replace left hand side variables
49    if(isLeftHandSideVariable(X)) {
50      Variable V = returnStmtVariable(X);
51      Immediate localVariableImmediate = local(V[0]);
52
53      int assingmentQuantity = returnAssignmentQuantity(
     localVariableImmediate);
54      newAssignStmt = replaceLeftVariableVersion(ADJACENCIES_MATRIX, X,
     assingmentQuantity);
55
56      renameNodeOcurrecies(X, newAssignStmt);
57
58      stackVariableVersion(localVariableImmediate, assingmentQuantity);
59      iterateAssignmentQuantity(localVariableImmediate);
60      findAndAddPhiFunctionArgs(oldNode, newAssignStmt); // Search
     variable uses of replaced variable in phi-function and rename it
61
```

```
62        X = newAssignStmt;
63      };
64    }
65
66    for(child <- IDOM_TREE[X]) {
67      // We need to check the last version replaced because the renamed
       version was not beign reflected in the current ‘for‘ statement
       iteration,
68      // so we need to check the last version replaced
69      nodeToRename = LAST_VERSION_REPLACED[child]? ? LAST_VERSION_REPLACED
       [child] : child;
70      replace(nodeToRename);
71    };
72
73    if(!ignoreNode(oldNode) && isVariable(oldNode) && !isRenamed(X))
74      popOldNode(oldNode);
75
76    return ADJACENCIES_MATRIX;
77  }
```

Listing 3.3: Implementação da renomeação de variáveis

Algumas funções auxiliares foram omitidas na apresentação da implementação por facilidade da leitura, mas podem ser encontradas no repositório do projeto [7]. Outro ponto a se notar na implementação é o uso de estruturas de dados auxiliares que facilitam a travessia, consulta e recriação do *CFG* e de sua *dominance tree*.

A implementação atual do algoritmo de renomeação também possui algumas limitações. Por exemplo, não são suportadas dois tipos de instrução: instruções do tipo *invoke* e *field reference* [9]. Ambas necessitam manipulações mais profundas que vão serão feitas em trabalhos futuros.

3.3 Testes

Para garantir estabilidade na implementação, foram feitos testes unitários e testes de integração. Os unitários são responsáveis por testar cenários diferentes de cada módulo de forma mais detalhada. Já os testes de integração usam códigos de *classfiles* obtidos de programas Java, entre esses estão programas que fazem operações matemáticas, operações com vetores e ordenação.

Os testes unitários buscam garantir que os módulos implementados estão coerentes com a especificação. Os casos de teste no formato JIMPLE foram criados manualmente para se adequarem aos cenários que devem ser contemplados. Uma alternativa seria usar

métodos gerados pelo decompilador de *classfiles* presente no projeto, mas foi escolhido a criação manual como uma forma de simplificar os testes e facilitar a reutilização.

Já os testes de integração têm como objetivo garantir que a utilização em conjunto dos módulos atinjam o resultado sem erros. Diferente dos testes unitários, os de integração necessitam códigos decompilados de *classfiles*. Os exemplos usados para os testes foram implementações de algoritmos de ordenação, operações matemáticas e manipulação de vetores. Não foram utilizados códigos que possam gerar instruções do tipo *invoke* e *field reference* [9], devido a limitação da implementação. Os arquivos java usados e os fontes de cada um são:

Algoritmo	Fonte
ApplyPower	https://www.w3resource.com/java-exercises/math/java-math-exercise-16.php
BabylonianSquareRoot	https://www.w3resource.com/java-exercises/math/java-math-exercise-14.php
BinomialCoefficient	https://www.w3resource.com/java-exercises/math/java-math-exercise-24.php
Fibonacci	https://www.guru99.com/fibonacci-series-java.html
FindArrayAverageValue	https://www.w3resource.com/java-exercises/math/java-math-exercise-17.php
GetFractionalPart	https://www.w3resource.com/java-exercises/math/java-math-exercise-2.php
ReverseInteger	https://www.w3resource.com/java-exercises/math/java-math-exercise-6.php
RoundUpDivision	https://www.w3resource.com/java-exercises/math/java-math-exercise-4.php
TaylorExponential	https://www.w3resource.com/java-exercises/math/java-math-exercise-25.php
ValidateDoubleIsInteger	https://www.w3resource.com/java-exercises/math/java-math-exercise-3.php
GeneratePrimes	https://www.w3resource.com/java-exercises/math/java-math-exercise-20.php
SimpleException	Criado para o teste
BubbleSort	https://www.javatpoint.com/bubble-sort-in-java
QuickSort	https://www.geeksforgeeks.org/java-program-for-quicksort/

Tabela 3.1: Tabela de algoritmos java usados para o teste e suas fontes.

Os testes foram feitos para garantir uma maior confiabilidade na implementação. Ambos testes unitários e de integração tentam buscar uma maior integridade no código de formas diferentes. Os arquivos de testes não se utilizam das instruções não suportadas pela implementação do *SSA*. Foram feitos testes unitários em 14 algoritmos diferentes, com 8 deles executando sem erros e 6 com erros. Os arquivos de teste que apresentaram erros são relacionados a falhas no decompilador. A correção desses erros foge do escopo desse trabalho e foram repassados para os mantenedores do JimpleFramework. Os arquivos de teste que mostraram os erros estão listados abaixo, totalizando 5 arquivos de teste, sendo 3 relacionados a erros na criação do *CFG* e 2 de decompilação.

Foram feitos testes de desempenho para medir quanto tempo em média cada arquivo do teste de integração consome, tirando apenas os arquivos que expuseram algum erro

Arquivo de teste	Erro relacionado
src/test/rascal/ssa/SSAIntegrationTests/sort/TestBubbleSort.rsc	Erro de criação do CFG
src/test/rascal/ssa/SSAIntegrationTests/sort/TestQuickSort.rsc	Erro de criação do CFG
src/test/rascal/ssa/SSAIntegrationTests/math_problems/TestReverseInteger.rsc	Erro de criação do CFG
src/test/rascal/ssa/SSAIntegrationTests/math_problems/TestValidateDoubleIsInteger.rsc	Erro de criação do CFG
src/test/rascal/ssa/SSAIntegrationTests/math_problems/TestGeneratePrimes.rsc	Erro de decompilação
src/test/rascal/ssa/SSAIntegrationTests/other/TestSimpleException.rsc	Erro de decompilação

Tabela 3.2: Tabela de testes que expuseram erros do framework.

relacionado a decompilação ou criação do CFG. Para medir o tempo médio de execução, foi utilizado uma funcionalidade nativa da linguagem Rascal para executar *benchmarks*. O código usado para a execução é apresentado abaixo:

```
1
2  import IO;
3  import List;
4  import util::Benchmark;
5
6  void main() {
7    list[num] result = [ benchmark(("result" : void() {testToRunBenchmark
       ();}))["result"] | i <- [ 1 .. 30] ];
8    print(result);
9    print("\n");
10   num add(num x, num y) { return x + y; };
11   num sum = reducer(result, add, 0);
12   print(sum / 30);
13 }
```

Listing 3.4: Implementação da renomeação de variáveis

Para a execução do teste de desempenho foi utilizado um computador com a seguinte configuração:

- Modelo: MacBook Pro (13 polegadas, Mid 2012)
- Processador: 2,5 GHz Dual-Core Intel Core i5
- Memória: 16 GB 1600 MHz DDR3
- Placa gráfica: Intel HD Graphics 4000 1536 MB
- Sistema operacional: macOS Catalina 10.15.7

Cada caso de teste foi executado 30 vezes para garantir uma menor diferença entre as médias. O tempo pode variar dependendo das condições do computador. A tabela que contém o tempo gasto na conversão para SSA em cada execução está disponível no Github [13]. Os testes de *benchmark* foram feitos apenas nos algoritmos que aplicam operações

matemáticas, e naqueles que não tiveram erro durante a criação do CFG. A Tabela ?? apresenta o tempo médio de execução para cada caso de teste:

Nome do arquivo de teste	Tempo médio de execução (ms)
TestApplyPower	605,7333333
TestBabylonianSquareRoot	854,3333333
TestBinomialCoefficient	1144,366667
TestFibonacci	642,8
TestFindArrayAverageValue	3057,9
TestGetFractionalPart	261,8333333
TestRoundUpDivision	349,6666667
TestTaylorExponential	677,9666667

Tabela 3.3: Tabela com tempo médio de execução para cada teste

A seguir é apresentado um exemplo de código Java e seu CFG antes e depois da transformação. O algoritmo corresponde ao de cálculo da série de Fibonacci usado nos testes:

```
1
2  public class Fibonacci {
3    public void run(int maxNumber) {
4      int previousNumber = 0;
5      int nextNumber = 1;
6
7      for (int i = 1; i <= maxNumber; ++i) {
8        int sum = previousNumber + nextNumber;
9        previousNumber = nextNumber;
10       nextNumber = sum;
11     }
12   }
13 }
```

Listing 3.5: Algoritmo em Java da série de Fibonacci

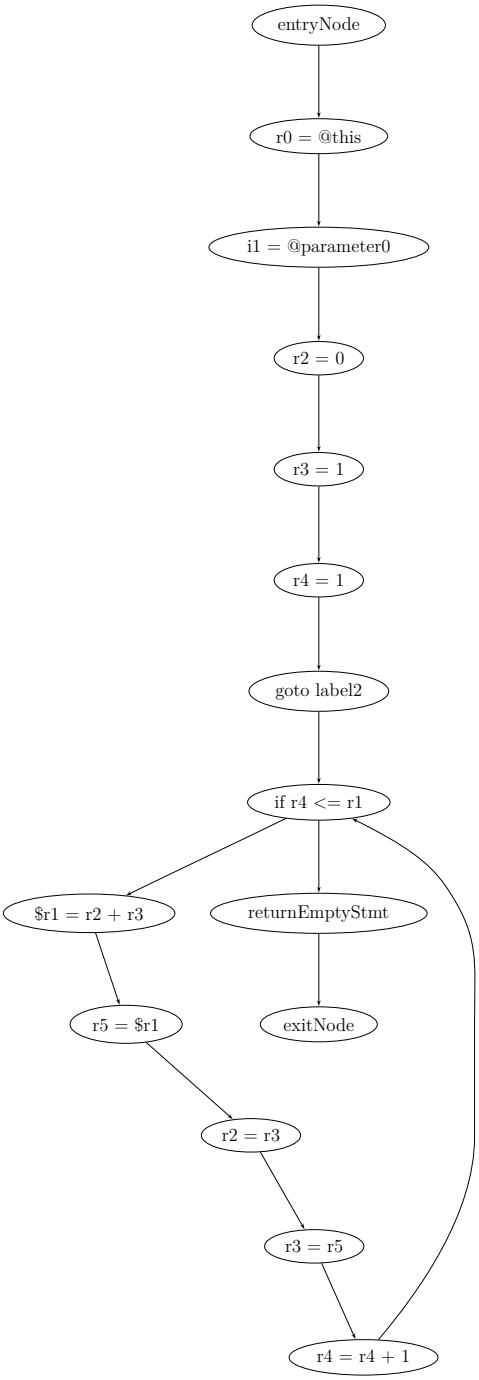

Figura 3.2: CFG do algoritmo Fibonacci

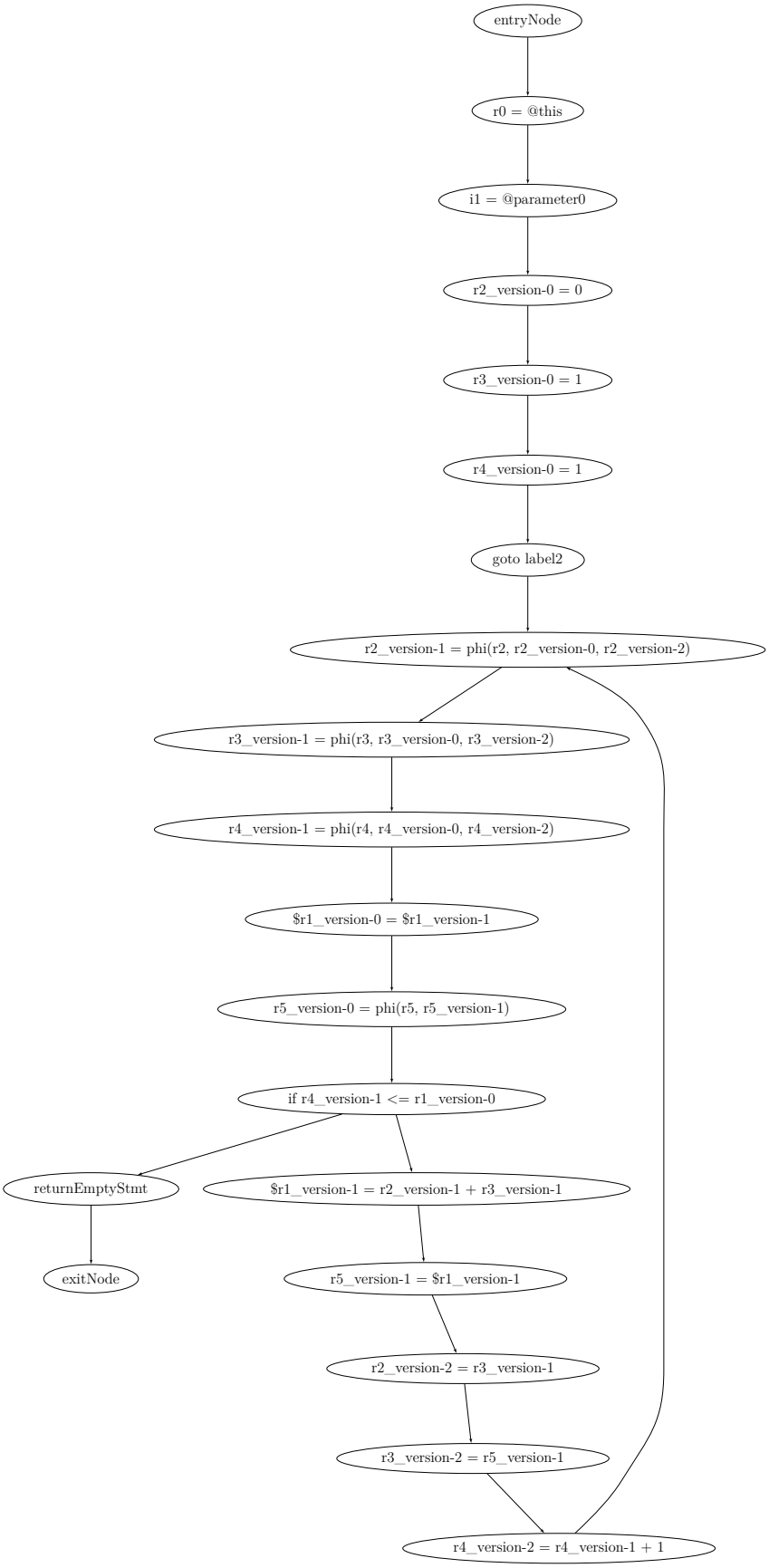

Figura 3.3: CFG do algoritmo Fibonacci ápos transformação SSA

Capítulo 4

Conclusão

Quando comparado com uma representação de três endereços, a representação em pilha do *classfile* se torna mais complexa quando queremos aplicar algoritmos de análise e transformação de programas [10]. Como uma forma de tentar se utilizar das soluções de otimização já existentes, o JIMPLE surge como um formato de três endereços para códigos fonte da JVM [10] [2]. Uma vez o código representado neste novo modelo, é possível realizar transformações que viabilizam o uso de nova otimizações, sendo uma delas a transformação no formato SSA (*Static Single Assingment*).

O SSA é um formato que define um formalismo em cima do *Control-Flow Graph*, uma estrutura de dados usada para representar o fluxo de um programa. O formato proposto tem como principal regra garantir que cada variável tenha uma única atribuição. Ou seja, uma mesma variável não pode ter mais de dois valores durante a execução do código. Com a representação SSA, as relações de *def-use* e *use-def* se tornam explícitas, tornando mais simples a implementação de otimizações de código [6].

O projeto *JimpleFramework* feito pela Universidade de Brasília, tem como objetivo fornecer um arcabouço de ferramentas de transformação de códigos *classfiles* no formato de três endereços e de análises e otimização. O software é implementado na linguagem de meta programação Rascal [11], que conta com diversas facilidades para análise e transformação de códigos de forma nativa. Este trabalho propõe uma versão inicial da implementação da transformação SSA como parte do *JimpleFramework*.

O algoritmo de transformação SSA proposto nesse trabalho é uma implementação feita em Rascal. O software utiliza algoritmos propostos de diferentes trabalhos [4] [5] [6] para realizar a computação de estruturas de dados auxiliares, como árvore e fronteira de dominância, e para implementar as etapas ta transformação, sendo elas a inserção de *phi-functions* e renomeação de variáveis. A implementação é uma versão simplificada, chamadas de funções e referência a atributos de classe e instância não são contempladas. O suporte para essas operações pode ser implementado como uma próxima interação do

algoritmo.

Bibliografia

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Raja Vallee-rai e Laurie Hendren. "Jimple: Simplifying Java Bytecode for Analyses and Transformations". Em: (jan. de 2004).

[3] Ondrej Lhoták. "Spark: A flexible points-to analysis framework for Java". Em: (ago. de 2009).

[4] Lots of authors. "Static Single Assignment Book". Em: *Static Single Assignment Book*. 2018, pp. 1–318.

[5] Navindra Umanee. "Shimple: An Investigation of Static Single Assignment Form". Em: (2005), pp. 1–33.

[6] R. Cytron et al. "An Efficient Method of Computing Static Single Assignment Form". Em: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 25–35. ISBN: 0897912942. DOI: 10.1145/75277.75280. URL: https://doi.org/10.1145/75277.75280.

[7] Rodrigo Bonifácio. *JimpleFramework*. https://github.com/PAMunb/JimpleFramework. 2020.

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

[9] Raja Vallee-rai e Laurie Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Jan. de 2004.

[10] Raja Vallee-rai et al. "Soot - a Java Bytecode Optimization Framework". Em: *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (out. de 1999). DOI: 10.1145/1925805.1925818.

[11] "Rascal Metaprogramming Language". Em: (2021). URL: https://www.rascal-mpl.org/.

[12] Paul Klint, Tijs van der Storm e Jurgen Vinju. “EASY Meta-programming with Rascal”. Em: jul. de 2009, pp. 222–289. ISBN: 978-3-642-18022-4. DOI: 10.1007/978-3-642-18023-1_6.

[13] Mateus Luiz Freitas Barros. *ssaRascalBenchmarkResults*. https://github.com/mateusluizfb/ssaRascalBenchmarkResults. 2021.