EXECUÇÃO DE FUNÇÕES PARCIAIS EM LINGUAGEM DE PROGRAMAÇÃO C

MARCUS RODRIGUES DE ARAÚJO

EXECUÇÃO DE FUNÇÕES PARCIAIS EM LINGUAGEM DE PROGRAMAÇÃO C

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte

Novembro de 2016

Araújo, Marcus Rodrigues de

A663e Execução de funções parciais em linguagem de programação C / Marcus Rodrigues de Araújo. — Belo Horizonte, 2016
xviii, 83 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Fernando Magno Quintão Pereira

1. Computação — Teses. 2. Compiladores (Computadores) — Teses. 3. Linguagens de programação (Computadores) I. Orientador. II. Título.

CDU 519.6*33(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Execução de funções parciais em linguagem de programação C

MARCUS RODRIGUES DE ARAUJO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. EDSON MARCHETTI DA SILVA
Departamento de Computação - CEFET-MG

PROF. RODOLFO SÉRGIO FERREIRA DE RESENDE
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 30 de Novembro de 2017.

"There is no knowledge,
that is not power."
(Ralph Waldo Emerson)

Resumo

Atualmente a comunidade conta com diversos programas poderosos para a realização de análises dinâmicas. Exemplos desses analisadores são: `Valgrind`, útil para realizar diversos tipos de verificação em relação à execução de programas; e `aprof`, usada para auxiliar na busca por ineficiências assintóticas. Entretanto, essas ferramentas esperam receber como entrada um programa executável. Isso pode dificultar a tarefa de desenvolvedores em analisar trechos de código específicos, como funções. Considerando essa dificuldade, nesta dissertação são apresentadas um conjunto de técnicas que, juntas, possibilitam a execução automática de uma função escrita na linguagem C de forma isolada. Em especial, focamos em métodos que lidam com acessos a arranjos e é tido como objetivo que, ao ser executada, a função alvo não deve gerar acessos de memória inválidos causados pelos dados fornecidos para executá-la. Com as ideias apresentadas aqui, foi construída uma ferramenta, `amazonc`, que possibilita a execução de funções em C de forma isolada. Experimentos realizados com as funções núcleos (*kernels*) do *Polybench* reforçam a eficácia do método apresentado. Juntamente com `aprof`, `amazonc` possibilitou a reconstrução das curvas de complexidade assintóticas de todos os *kernels* considerados. Além disso, as reconstruções das curvas foram feitas sem que `Valgrind` apontasse nenhum erro de acessos inválidos a memória.

Palavras-chave: Execução Automática, Análise Estática, Análise Dinâmica, Linguagem C.

Abstract

Currently the computer science community has several powerful programs for performing dynamic analysis. Examples of these analyzers are: `Valgrind`, useful for performing several types of checks related to the running of programs; and `aprof`, used to aid in the search for asymptotic inefficiencies. However, these tools expect to receive as an input an executable program. This fact complicates the analysis of specific code snippets, such as functions. Considering this difficulty, in this dissertation we present a set of techniques that, together, enable us to execute isolated functions written in C language automatically. In particular, we focus on methods that deal with access to arrays by indexing. Besides that, it is our goal that, when executed, a target function does not contain invalid memory accesses caused by the data automatically generated to test it. With the ideas presented here, we developed a tool, `amazonc`, which enable us to execute functions in C isolated from the rest of the code. Experiments performed with the kernels of the Polybench benchmarks suite reinforce the effectiveness of the method presented. Together with `aprof`, `amazonc` was able to reconstruct the asymptotic complexity curves of all core functions in that benchmark. In addition, the process of reconstructing these curves did not lead to memory errors, as Valgrind has certified.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Neste capítulo é dado ao leitor uma noção do assunto abordado nesta dissertação. Inicialmente será apresentada uma motivação para mostrar a aplicabilidade das ideias discutidas aqui. Em seguida, é reservada uma seção para deixar claro qual é o principal objetivo deste trabalho. Após isso, o leitor é convidado a conhecer melhor os desafios enfrentados no desenvolvimento desta dissertação. Esta apresentação é feita através de uma narrativa simples em que um programador deseja analisar uma função. Depois que os problemas forem apresentados, é dada uma visão geral da técnica de reconstrução de programas apresentada nesta dissertação. Por fim, são apresentadas as contribuições feitas durante o período de pós-graduação.

1.1 Motivação

Atualmente a comunidade conta com diversas ferramentas que analisam a forma como aplicações escritas na linguagem de programação C são executadas. Esse tipo de análise coleta dados durante a execução de programas e costumam fornecer informações valiosas a respeito deles. Exemplos de ferramentas públicas são: `Gcov`, [Best [2003]], usada para obtenção de informações relativas a cobertura de código (*code coverage*); `Valgrind`, [Nethercote & Seward [2007]], que reporta erros decorrentes do uso incorreto da memória dinâmica e acessos a áreas de memória inválidas; e `aprof`, [Coppa et al. [2012]], que é usado para encontrar ineficiências assintóticas em programas. Além disso, na literatura também são encontradas a especificação de muitas outras ferramentas como é o caso de `DART`, [Godefroid et al. [2005]], e `MicroX`, [Godefroid [2014]], que são usadas para realização de testes automáticos.

Essa forma de extrair informação de programas, com base em sua execução, é chamada de análise dinâmica. Apesar de estarem sujeitas a receber uma aplicação

que não termina, essas análises costumam ser mais precisas que suas contrapartes, as análises estáticas. Isso se deve, principalmente, por elas possuírem informações reais sobre os programas que são coletadas durante a sua execução. No entanto, grande parte das ferramentas de análise dinâmica esperam receber como entrada um programa por completo. Isso pode causar certo infortúnio a programadores que queiram analisar, ou testar, partes específicas de um *software*, como uma função.

Outro problema existente com o uso de análises dinâmicas está relacionado com os cuidados que devem ser tomados ao se usar ferramentas desse tipo. Em geral, é inviável executar um programa com todos os tipos de entrada existentes. Dessa forma, para atingir toda eficácia que esse tipo de análise tem a oferecer de forma eficiente, o programa alvo deve ser executado com o menor conjunto de dados de entrada possível que seja representativo o suficiente para o que se pretente analisar.

Além disso, alguém que queira criar um ambiente exclusivo para analisar programas escritos em C tem de lidar com peculiaridades da própria linguagem. Um desses infortúnios acontece devido a falta de mecanismos de introspecção da linguagem. Em Java, por exemplo, é possível obter o tamanho de um arranjo através de um comando similar à "arranjo.length", o que não existe em C. Dessa forma, ao gerar dados para executar funções em C, é importante que o analista fique atento a forma como os dados se relacionam (como vetores e variáveis relacionadas ao seu tamanho) para não introduzir erros que na prática provavelmente não aconteceriam.

Nesta dissertação é mostrado como executar funções isoladas em linguagem de programação C de forma automática. Mais especificamente, é proposto um conjunto de ideias que permitem a geração de entradas para funções que não levem a acessos de memória inválidos. Em geral, é esperado receber como entrada um método que esteja completo o suficiente a ponto de poder ser compilado. Como saída, é retornado um *driver*, em texto de código C, que pode ser compilado junto com a função alvo e usado para executá-la. O processo de reconstrução foca em procedimentos que recebem arranjos multidimensionais como entrada.

A possibilidade de executar funções de forma automática e livre de erros causados por uma má geração de dados podem ser de grande valia para a comunidade como um todo. Em especial, profissionais e pesquisadores que trabalham com teste automático de *software* e com análise de complexidade de algoritmos podem tirar muito proveito das ideias discutidas aqui. Em relação aos primeiros beneficiados, nesta dissertação não é mostrado como realizar teste parcial de funções, mas é dada uma base para mostrar como é possível fazer isso de forma aleatória (como nas estratégias de *fuzzing* [Oehlert [2005]]) e, mais importante, de forma segura. Além disso, analistas que estudam a complexidade de algoritmos podem usar `amazonc`, uma ferramenta criada com as técnicas

discutidas nesta dissertação, juntamente com `aprof` para analisar o comportamento assintótico de funções na prática.

O método descrito aqui para gerar dados para as entradas de uma função pode ser decomposto em quatro passos: a análise do código fonte de entrada, Capítulo 4, a construção do **Grafo de Acessos**, Seção 5.3, troca de informações (restrições) entre os nós do grafo, Seção 5.4, e, finalmente, construir um conjunto de instruções para inicializar cada entrada do programa, Seção 6.2. Esse último passo, exige que uma certa ordem entre os comandos seja respeitada. Para atingir esse fim, é utilizado o **Grafo de Inicialização**, Seção 6.1.

1.2 Objetivo

Neste trabalho é apresentada uma nova maneira de executar, automaticamente, funções isoladas escritas na linguagem C. O que é proposto aqui é similar ao que é mostrado nos trabalhos de Godefroid, [Godefroid et al. [2005]; Godefroid [2014]], e Demontiê, [Demontiê [2016]]. Uma diferença fundamental entre este trabalho e os anteriores é que esses estão focados na execução automática dos programas para fins de teste de *software*, enquanto a técnica que é apresentada nesta dissertação é mais geral e pode ser utilizada tanto para essa finalidade quanto para a realização de outros tipos de análises dinâmicas.

Mais especificamente, lidamos com o problema de gerar dados que não levem a erros causados por acessos fora dos limites de arranjos. Em outras palavras, se uma função é executada e suas entradas interferem no uso, através de indexações, de um arranjo alocado com `n` posições de memória, queremos que os dados gerados para executar esse método nunca acessem posições desse vetor menores do que o índice `0`, limite inferior, e nem maiores do que `n-1`, limite superior.

Esse problema já foi abordado por Demontiê, em sua dissertação de mestrado, [Demontiê [2016]]. Nessa oportunidade, para evitar os problemas com acesso a memória, o autor buscava relacionar arranjos com símbolos existentes no programa que estivessem ligados ao tamanho dos arranjos. Diferente disso, nesta dissertação, foi utilizada uma outra abordagem. O que se optou por fazer foi gerar, primeiramente, arranjos de tamanhos aleatórios e usar os valores de seus tamanhos para adaptar o ambiente no qual eles são usados.

1.3 Exemplos de Desafios Resolvidos

Nesta seção, são apresentados alguns cuidados necessários para tornar possível a execução de uma função isolada em C que não gere acessos inválidos de memória. O processo de reconstrução de código será ilustrado através da saga de Gleison, um programador qualquer, para analisar dinamicamente a função da figura 1.1 com a ferramenta `aprof`. Cada problema mostrado é tratado nesta dissertação. Junto com as apresentações das dificuldades, em forma de código fonte, serão dadas ideias do que pode ser feito para resolver cada problema. Lembrando que `aprof` é uma ferramenta de análise dinâmica que infere complexidade de funções. Ela faz isso relacionando tamanho da entrada de cada método com o custo computacional usado para executá-lo.

Suponha então que Gleison queira analisar o código da figura 1.1 usando `aprof`. O código em questão apresenta uma função que recebe como entrada dois parâmetros: `n`, o tamanho do vetor; e `v`, um ponteiro para a primeira posição do arranjo. Para tal,

```
1 int sumVector(int n, int *v) {
2   int i, sum = 0;
3   for (i = 0; i < n; i++) {
4     sum += v[i];
5   }
6   return sum;
7 }
```

Figura 1.1: Exemplo de função a ser testada por `aprof`

ele terá que criar um *driver* que realize chamadas para essa função. A figura 1.2 mostra uma primeira tentativa de realização dessa tarefa. O primeiro problema que se pode

```
1 int main() {
2   int v[10];
3   sumVector(10, v);
4   return 0;
5 }
```

Figura 1.2: Primeira tentativa de construir um *driver*

apontar com essa função `main` é que `sumVector` está sendo chamada uma única vez. Para que ferramentas de análise dinâmica possam atingir toda sua eficácia ao analisar um programa é necessário que esse seja executado **múltiplas vezes**.

Sendo avisado sobre o problema anterior, Gleison reescreveu sua função principal e chegou ao código da figura 1.3.

```
1 int main() {
2   int v[10];
3   int i, times = 10;
4   for (i = 0; i < times; i++) {
5     sumVector(10, v);
6   }
7   return 0;
8 }
```

Figura 1.3: Segunda tentativa de construir um *driver*

Apesar de executar a função alvo várias vezes, esse *driver* ainda não vai ajudar o programador a analisar o seu método. Ferramentas de análise dinâmica, como `aprof`, esperam que o código a ser analisado seja executados várias vezes com **cargas de trabalho diferentes**.

Em sua terceira tentativa, Gleison escreveu o *driver* da figura 1.4. Agora, a

```
1  int main() {
2    int *v;
3    int i, times = 10, n;
4    for (i = 0; i < times; i++) {
5      n = rand()%100;
6      v = (int*) malloc( (rand()%100) * sizeof(int) );
7      sumVector(n, v);
8      free(v);
9    }
10   return 0;
11 }
```

Figura 1.4: Terceira tentativa de construir um *driver*

cada iteração, `sumVector` será executada com dados diferentes. Infelizmente, Gleison se esqueceu de um detalhe sobre a função que ele está analisando: a variável `n` está relacionado com o tamanho do arranjo `v`. Dessa forma, uma chamada de `sumVector` com um valor para a variável `n` maior que o número de posições alocadas para `v` irá causar acessos indevidos a memória.

Acessos fora dos limites de arranjos são considerados como causadores de comportamentos indefinidos na linguagem C (*undefined behavior*), [Hathhorn et al. [2015]]. Isso vai acarretar em impossibilitar e|ou comprometer a análise que Gleison está fazendo do seu programa.

Para resolver o último problema, Gleison poderia usar uma função `main` como a

que é mostrada na figura 1.5. Note que agora a variável `n` está sendo usada para alocar

```
1  int main() {
2    int *v;
3    int i, times = 10, n;
4    for (i = 0; i < times; i++) {
5      n = rand()%100;
6      v = (int*) malloc( n * sizeof(int) );
7      sumVector(n, v);
8      free(v);
9    }
10   return 0;
11 }
```

Figura 1.5: O *driver* ideal

o tamanho do arranjo `v`.

Essa saga foi mostrada para ilustrar ao leitor alguns dos cuidados que devem ser tomados ao gerar um *driver* para executar uma função em C. Existem mais peculiaridades que devem ser levadas em conta para executar trechos de códigos como que foi mostrado na saga de Gleison. Nesta dissertação, lidamos com os problemas descritos anteriormente e os que são descritos como "Casos Especiais 2 e 3" apresentados a seguir.

1.3.1 Caso Especial 1: O Código que não Compila

Uma vez que o principal alvo deste trabalho são funções isoladas, pode acontecer que, após o processo de extração de um método, não se obtenha um código válido o suficiente para ser compilado. Isso é um problema no caso deste trabalho. As técnicas discutidas nesta dissertação são aplicáveis desde que seja possível gerar uma árvore de sintaxe abstrata *(Abstract Sintax Tree: AST)* para o programa parcial alvo.

Felizmente, completar um trecho de código em C de maneira a torná-lo compilável não é um problema desde a criação de `psyche-c`, [Ribeiro et al. [2016]]. Essa ferramenta é capaz de reconstruir códigos parciais em linguagem de programação C. Ela completa o trecho de programa que é dado como entrada ao ponto de fazer com que ele possa ser compilável. Por causa disso, esse tipo de problema não será discutido aqui.

1.3.2 Caso Especial 2: As Variáveis Globais

Considere o caso em que se tenha a intenção de testar o código da função apresentada na figura 1.6. Esse código não está completo o suficiente para ser compilado. Ao usar

```
1 int foo(int *v) {
2   return v[Global];
3 }
```

Figura 1.6: Função que faz uso de uma variável global (Incompleto).

`psyche-c` para reconstruir esse programa, algo equivalente ao que é mostrado na figura 1.7 é obtido.

```
1 int Global;
2
3 int foo(int *v) {
4   return v[Global];
5 }
```

Figura 1.7: Função que faz uso de uma variável global (Completo).

Para testar esse código é necessário que seja levado em consideração a relação existente entre a variável `Global` e `v`. Em outras palavras, o valor de `Global` na linha 4 da figura 1.7 não pode ser inferior a 0 e nem superior a última posição do arranjo apontado por `v`.

1.3.3 Caso Especial 3: As Funções sem Definição

Um problema similar ao que foi discutido na Subseção 1.3.2 pode acontecer ao se produzir um corpo substituto (*stub*) para uma função. Se no código da figura 1.7 a variável `Global` fosse substituída por uma chamada de função, esse método também deveria respeitar os limites de `v`. Dessa forma, ao gerar um *stub* para uma função é necessário levar em conta as restrições que podem existir sobre suas chamadas.

Neste trabalho, são gerados *stubs* para métodos que possuem somente a sua declaração. Em geral, o que é feito é gerar um valor de retorno aleatório de acordo com o tipo de retorno do método. Os *stubs* criados estão limitados aos tipos primitivos e ponteiros para esses tipos. Nesta dissertação não são gerados dados para estruturas de dados mais complexas como as que são construídas através do especificador de tipos `struct`.

1.4 Visão Geral

`amazonc` é um programa que recebe como entrada uma função em C e retorna, como saída, um *driver* que pode ser usado para executar esse método. O objetivo dessa ferramenta é possibilitar a execução de uma função sem que os dados gerados para executá-la impliquem na existência de acessos à memória inválidos.

Nesta seção, é dada uma ideia geral de como `amazonc` realiza a criação do *driver*. Tal processo acontece considerando que o trecho de código dado como entrada esteja completo o suficiente a ponto de poder ser compilado. Caso este não seja o caso, o usuário pode usar `psyche-c`, [Ribeiro et al. [2016]], para contornar esse problema. Uma vez que tal requisito tenha sido atendido, `amazonc` poderá ser usada.

O processo de criação do *driver* será ilustrado considerando o código exemplo da figura 1.8. Nessa figura, `a`, `b`, `c` e `v` são as entradas da função `arrayAccess`. Um símbolo

```
1  int a;
2  int b();
3
4  int arrayAccess(int c, int *v) {
5    int t1, t2, t3;
6
7    t1 = a * b();
8    t2 = a + c;
9    t3 = b() + c;
10   v[2] = 0;
11
12   return v[t1 + t2 + t3 + 1]; // v[ab + a + b + 2c + 1]
13 }
```

Figura 1.8: Exemplo de código a ser reconstruído.

é considerado como uma entrada se ele é usado antes que algo seja atribuído a ele. Dessa forma, antes de executar um método, como o anterior, devem também ser fornecidos dados para suas entradas. Considerando o objetivo de `amazonc`, é importante que tanto a informação passada para `a` e `c` quanto a que é retornada pela função `b` respeitem o acesso que acontece na linha 12. Em outras palavras, a expressão usada para indexar o arranjo na linha 12 (`t1 + t2 + t3 + 1 = (ab) + (a + c) + (b + c) + 1 = ab + a + b + 2c + 1`) deve resultar em um índice menor que o tamanho do arranjo `v`.

O primeiro passo realizado por `amazonc` é uma Análise de Intervalos Simbólica [Cousot & Cousot [1977]], apresentada no Capítulo 4. Essa análise busca associar cada variável local da função alvo com um intervalo composto por dois valores: um limite

inferior e outro superior, que representam o menor e o maior valor que a variável pode assumir, respectivamente. É importante ressaltar que a análise associa intervalos simbólicos e variáveis a cada ponto do programa. O mapa `R` é a estrutura de dados utilizada para criar os vínculos entre símbolos e intervalos simbólicos. A figura 1.9 apresenta os valores dos intervalos associados a cada variável local da função `arrayAccess` da figura 1.8.

```
1  int a;
2  int b();
3
4  int arrayAccess(int c, int *v) {
5    int t1, t2, t3;
6
7    t1 = a * b(); // R={t1:[ab,ab]}
8    t2 = a + c; // R={t1:[ab,ab],t2:[a+c,a+c]}
9    t3 = b() + c; // R={t1:[ab,ab],t2:[a+c,a+c],t3:[b+c,b+c]}
10   v[2] = 0;
11
12   return v[t1 + t2 + t3 + 1]; // v[ab + a + b + 2c + 1]
13 }
```

Figura 1.9: Código com resultado da Análise de Intervalos.

Juntamente com os intervalos, durante a Análise de Intervalos Simbólica, são coletadas todas as expressões usadas para acessar as dimensões dos arranjos. No caso do exemplo da figura 1.8, existem dois acessos a `v` que acontecem na linha 10, `"v[2]"`, e na 12, `"v[t1 + t2 + t3 + 1]"`. Essas expressões são colocadas em função dos símbolos de entrada do programa a partir dos valores encontrados pela análise anterior. Além disso, se possível, as expressões encontradas são colocadas em sua forma afim (que possuem o formato: $aX + b$, em que a e b são valores numéricos e X é um símbolo do programa).

Uma vez que a análise anterior seja finalizada, e os acessos aos arranjos tenham sido coletados, o **Grafo de Acessos**, detalhado na Seção 5.2, pode ser construído. Esse grafo é composto por nós que representam: arranjos usados no programa parcial, expressões usadas para acessar esses vetores e as entradas da função. Além disso, ele também conta com um conjunto de arestas que associam: (1) expressões aos arranjos; e (2) símbolos às expressões. As arestas de tipo (1) indicam que uma expressão acessou um arranjo. Essa relação possui um rótulo, ou número, que indica a dimensão do vetor que foi acessada. As arestas de tipo (2), por outro lado, ligam símbolos às expressões que as usam. A figura 1.10 apresenta um exemplo de Grafo de Acessos

gerado para o programa exibido na figura 1.8. Note que existe um nó próprio para a multiplicação dos símbolos `a` e `b`. Toda vez que um produto entre dois, ou mais, símbolos aparecem em expressões, são gerados novos símbolos, e nós no Grafo de Acesso, para essas multiplicações.

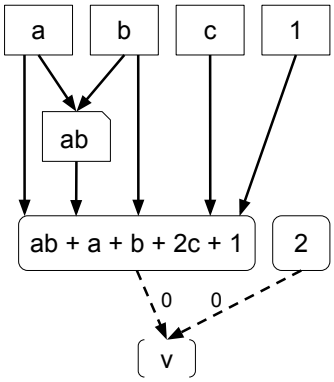

Figura 1.10: Grafo de Acesso para o código da figura 1.8.

Depois que o Grafo de Acessos for construído é dado início a uma etapa de troca de informações entre os seus nós. Esse câmbio é feito em duas etapas. A primeira delas consiste no envio dos valores dos nós literais (números) aos seus vizinhos arranjos. Veja que os vizinhos de um vetor são expressões usadas para acessá-lo. Uma vez que a primeira parte da troca tenha sido realizada, cada arranjo utiliza a informação recebida como restrições de tamanho mínimo para cada uma de suas dimensões. A ideia é a seguinte: se o literal `l`, em que `l` é um número natural, foi, ou pode ser, usado para acessar uma dimensão `d` de um vetor `v`, então `v` deve ter o tamanho da dimensão `d` igual a no mínimo `l`+1. A essa etapa é dado o nome *top-down*, figura 1.11.

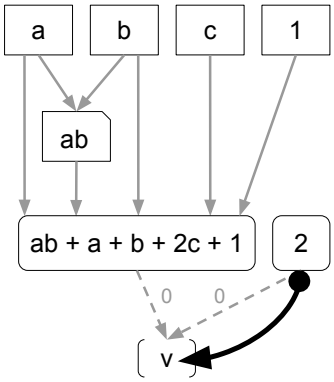

Figura 1.11: Primeira etapa da troca de informações (*top-down*).

A segunda troca de mensagens, *bottom-up*, acontece partindo dos arranjos. Inicialmente são gerados valores simbólicos s_{ij} que representam o tamanho de cada dimensão j no arranjo i. Esse valor tem, associado a ele, suas restrições de tamanho mínimo, geradas na etapa *top-down*. Em seguida, é criado um intervalo simbólico, **Intervalo de Acesso**, que representa toda a dimensão de acesso "disponível" dos arranjos para cada uma de suas dimensões, $[0, s_{ij}]$.

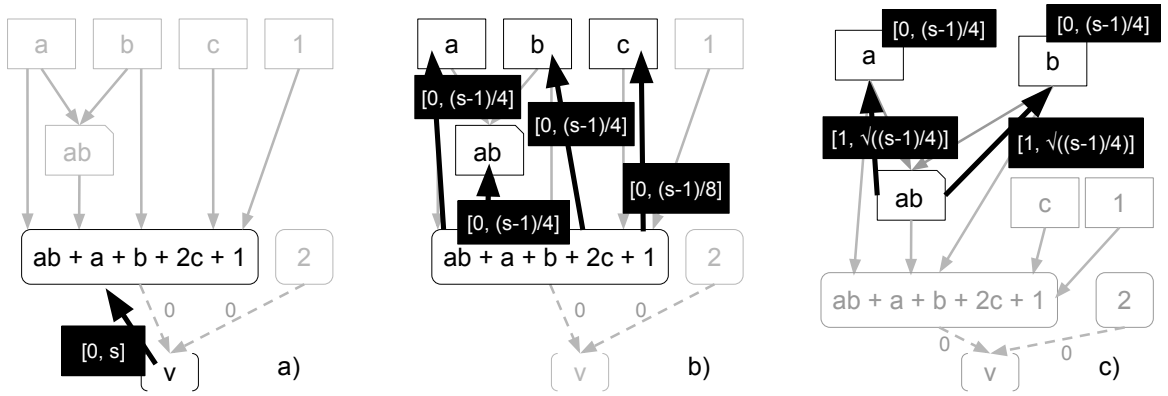

Figura 1.12: Segunda etapa da troca de informações (*bottom-up*)

Após a criação dos Intervalos de Acesso, cada arranjo envia seus espaços de memória disponíveis para as expressões usadas nos acessos às suas dimensões, figura 1.12a). No exemplo da figura anterior está sendo usado o símbolo `s` por simplicidade. As expressões de acesso, por sua vez, dividem o intervalo recebido igualmente por cada uma das parcelas usadas em sua composição, figura 1.12b). É permitidos que cada termo varie entre o valor 0 e uma parte do tamanho total do arranjo.

Um detalhe importante da figura anterior é que o símbolo `c` aparece na expressão multiplicado por 2. Isso faz com que a faixa de valores possíveis de serem assumidos por essa parcela sejam normalizados com o coeficiente em questão, 2, quando uma mensagem é enviada para o nó que representa o símbolo `c`. Essa alteração é feita para garantir que o termo `"2c"` esteja sempre dentro do intervalo de valores passado para ele quando for gerado um valor para `c`.

Por fim, a figura 1.12c) mostra como os nós de produto entre símbolos transmitem informação. Assim como os nós de expressões afins, esses nós dividem a faixa de valores que eles têm igualmente entre os termos que os compõem. Para fazer essa divisão de forma igual, e garantir que essa parcela respeite a faixa de valores associada ao nó `"ab"`, é passado para cada termo um intervalo que vai de 1 até a raiz quadrada do limite superior do intervalo disponível. Isso garante que o valor máximo gerado por essa multiplicação seja justamente o limite superior associado ao nó do produto. Além

disso, fornecendo um limite inferior igual a 1 evita que aconteçam multiplicações por 0.

Um último detalhe a respeito da etapa *bottom-up* é o que acontece quando nós recebem mais de uma mensagem. Esse fato acontece, no exemplo considerado, com os nós `a` e `b`. Na figura 1.12c), os nós em questão já possuíam um intervalo, enviado pela expressão afim, quando receberam uma mensagem do nó de multiplicação. Nesses casos, o intervalo final dos nós que recebem mais de uma mensagem é dado pela interseção dos intervalos recebidos.

Ao final das duas ondas de informação, os intervalos finais associados a cada entrada são conhecidos. A figura 1.13 apresenta esses valores para os símbolos do exemplo em questão. Os intervalos foram gerados de forma a garantir que, para qualquer valor positivo de `s`, as entradas escolhidos respeitam os intervalos de acesso dos vetores. Note que os intervalos estão todos em função do tamanho do arranjo.

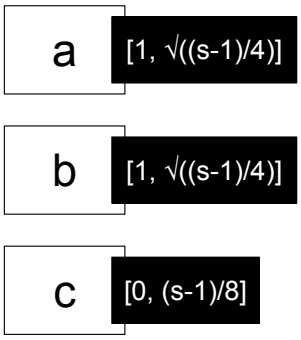

Figura 1.13: Intervalos finais associados com as entradas da função.

Uma vez que as restrições sobre as entradas sejam conhecidas, antes de gerar o *driver*, devem ser consideradas a ordem de definição de cada valor. Considerando as entradas anteriores, por exemplo, o valor de `s` deve ser definido antes das demais entradas. Para garantir essa ordem, é usado o **Grafo de Inicialização**, apresentado na Seção 6.1. Essa estrutura de dados se baseia em um grafo de dependências que assegura a ordem das definição das entradas do programa. Pelo grafo da figura 1.14 é possível ver que os intervalos das entradas da função dependem do valor de `s` que, por sua vez, depende dos valores de `s_lower` e `s_upper`. Os símbolos que compõem o limite superior do intervalo de `s` são parâmetros de `amazonc`. Esse tipo de informação é usada para permitir que o usuário da ferramenta tenha controle sobre os tamanhos máximos das dimensões dos arranjos criados. Já os símbolos usados para compor a

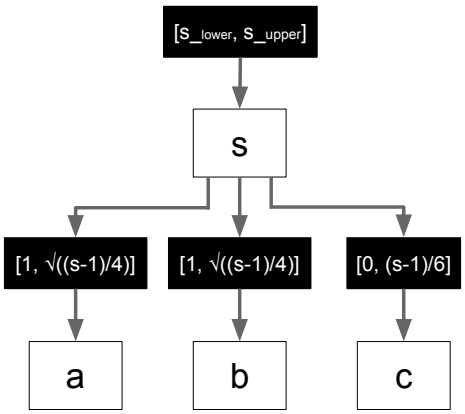

Figura 1.14: Grafo de Inicialização para o exemplo considerado.

expressão do limite inferior do intervalo de `s` são obtidos com as restrições de tamanho mínimo associadas a esse arranjo.

Com as informações obtidas com os dois grafos é possível começar a etapa de criação do *driver*. Esse processo consiste na geração de código C que vai ser usado para executar a função alvo. A figura 1.15 mostra um trecho de código equivalente ao que é gerado por `amazonc`. Note que é gerado código fonte C para executar o programa. Isso facilita que customizações possam ser feitas por usuários da ferramenta, como por exemplo: alteração de parâmetros, modificações nos valores gerados pelos *stubs*, etc.

1.5 Contribuições

O projeto desenvolvido nesta dissertação é fruto do trabalho de um grupo de pessoas. Esses participantes incluem: Marcus Rodrigues de Araújo (autor desta dissertação), Solène Mirliaz e Fernando Magno Quintão Pereira (Orientador). As principais contribuições desta dissertação são:

1. Uma nova técnica para executar funções parciais que lidam com acessos a arranjos e buscam não gerar acessos indevidos à memória;

2. Uma ferramenta capaz de executar trechos de funções que lidam com arranjos multidimensionais;

No geral, tive participação em todas as etapas do desenvolvimento deste projeto. Isso inclui sua concepção, implementação, testes e escrita deste trabalho.

Além do que é apresentado nesta dissertação, no decorrer do mestrado, tive a oportunidade de trabalhar em outros projetos. Desse esforço resultaram quatro artigos.

Três deles publicados no Simpósio Brasileiro de Computação, e um deles publicado em ACM POPL, uma conferência nível A1, de acordo com o Qualis da CAPES. A seguir, segue uma lista das publicações:

1. **Compilação Parcial de Programas Escritos em C**, [Ribeiro et al. [2016]]: esse trabalho apresenta uma máquina usada para reconstruir tipos em linguagem de programação C. Nesse projeto, tive a oportunidade de ajudar em várias etapas do seu desenvolvimento: concepção, implementação, testes e a escrita do artigo.

2. **Inferência de Tipos Dependentes em C**, [de Araujo et al. [2017]]: nesse artigo é mostrado como uma forma restrita de tipos dependentes podem ser inferidos e usados para documentar a linguagem de programação C. Como primeiro autor, tive a responsabilidade principal sobre todas as etapas do projeto.

3. ***The Register Allocation and Instruction Scheduling Challenge***, [Carvalho et al. [2017]]: esse trabalho se baseia em uma revisão sistemática da literatura. Nele, são discutidas as implicações de se considerar o problema da alocação de registradores e o de escalonamento de instruções de forma conjunta, ou seja, como um único problema. Mais uma vez, tive a oportunidade de trabalhar em todas as etapas do desenvolvimento desse projeto. Isso inclui: elaboração da *string* de busca, filtragem dos trabalhos, leitura dos artigos selecionados, responder as questões de pesquisa e escrita do artigo.

4. ***Inference of Static Semantics for Incomplete C Programs***, [Melo et al. [2018]]: nesse artigo é dada uma solução para o problema de reconstruir programas parciais em linguagem C. No caso desse projeto, minhas principais contribuições foram: a criação de uma página web pública, para permitir acesso a ferramenta por parte da comunidade; coleta de dados (programas parciais), para realização dos testes; a realização de alguns testes, os que envolviam a ferramenta `kcc`; e por fim a criação de relatórios com informações a respeito dos experimentos.

```
1  #define NUMERO_DE_TESTES 10
2  #define S_LOWER 10
3  #define S_UPPER 100
4  int a;
5  int b();
6  int arrayAccess(int c, int *v) {
7    int t1, t2, t3;
8
9    t1 = a * b();
10   t2 = a + c;
11   t3 = b() + c;
12   v[2] = 0;
13
14   return v[t1 + t2 + t3 + 1];
15 }
16 int s;
17 int a_upper, b_upper, c_upper;
18 int a_lower, b_lower, c_lower;
19 int b() {
20   return randInt(b_lower, b_upper);
21 }
22 int main(int c, int *v) {
23   int times;
24   for (times = 0; times < NUMERO_DE_TESTES; times++) {
25     int c, s, *v;
26     s = randInt(S_LOWER, S_UPPER);
27     a_lower = 1;
28     a_upper = sqrt((s-1)/4);
29     b_lower = 1;
30     b_upper = sqrt((s-1)/4);
31     c_lower = 0;
32     c_upper = (s-1)/8;
33     a = randInt(a_lower, a_upper);
34     c = randInt(c_lower, c_upper);
35     v = (int*) malloc(s * sizeof(int));
36     arrayAccess(c, v);
37     free(v);
38   }
39   return 0;
40 }
```

Figura 1.15: Driver gerado para executar a função `arrayAccess`.

Capítulo 2

Conceitos Importantes

Neste capítulo é providenciado um *background* teórico com o objetivo de ajudar o leitor a entender melhor este trabalho. Inicialmente será definido o que é uma Árvore de Sintaxe Abstrata (*Abstract Sintax Tree*). Essa é a estrutura de dados onde ocorre a principal análise usada nesta dissertação. Depois disso, serão apresentadas ao leitor as duas principais formas de analisar um programa, as análises estáticas e dinâmicas. Neste trabalho, as duas formas de analisar programas são usadas. Após isso, é dado uma noção do tipo de informação que pode ser obtido com uma Análise de Intervalos. Por fim, é apresentado um tipo de álgebra que lida com símbolos e que é utilizada nesta dissertação.

2.1 Árvore de Sintaxe Abstrata

Um grafo acíclico é aquele que não possui ciclos. Uma árvore é um grafo conexo (existe um caminho entre qualquer par de vértices) e acíclico [Bondy & Murty [1976]]. Uma árvore enraizada é uma árvore em que existe um vértice distinto dos outros que é conhecido como a raiz da árvore. Esse vértice é considerado como um ancestral de todos os outros nós da árvore e serve de ponto de partida para diversos tipos de caminhamentos nessa estrutura de dados. Além disso, temos que uma árvore ordenada é uma árvore orientada em que os filhos de um nó possuem alguma forma de "ordem". Tomando os grafos T1 e T2, apresentados na figura 2.1 a) e b), como exemplo, se T1 e T2 são árvores ordenadas então T1 é diferente de T2.

Em geral, qualquer programa em C válido pode ser representado como uma árvore enraizada e ordenada. A essa árvore se dá o nome de Árvore de Sintaxe Abstrata (AST). Em uma AST cada nó interno representa uma construção sintática da linguagem. Já suas folhas, representam o texto do código fonte que foi digitado em um arquivo. O

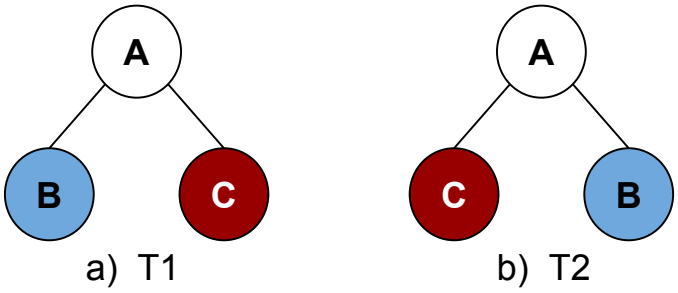

Figura 2.1: Exemplo de grafos ordenados diferentes.

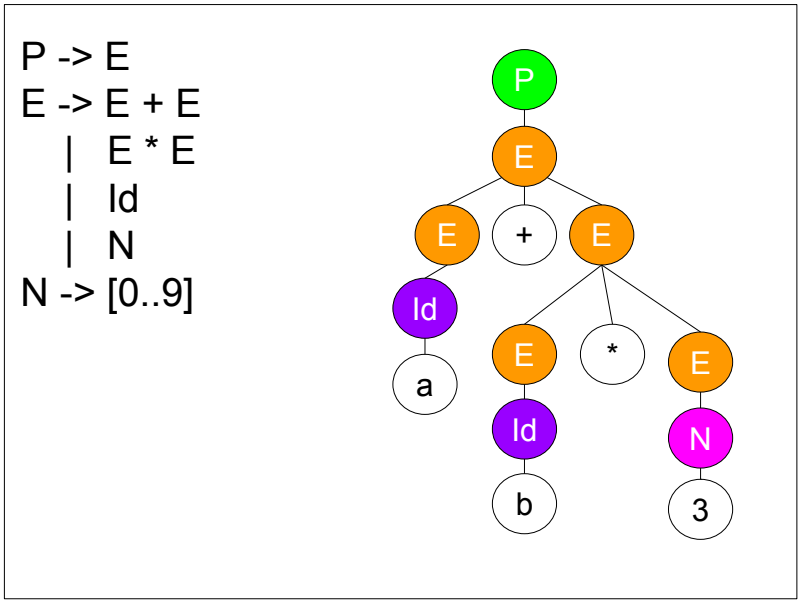

Figura 2.2: Exemplo de AST para a expressão "a+b*3".

`parser` é responsável por transformar código fonte nessa estrutura de dados que é muito utilizada por ferramentas de compilação para abstrair o programa de entrada [Aho et al. [1986]; Appel [2004]]. Esse tipo de estrutura de dados facilita a manipulação e validação do código fonte de diversas formas. Na figura 2.2 é mostrado um exemplo de uma AST para uma dada gramática.

2.2 Análise de Programas

Ao analisar um programa, é possível obter informações valiosas sobre ele. Na literatura é possível encontrar diversos tipos de análises que podem ser utilizadas para diversas finalidades. Exemplos de aplicações são: documentação [de Araujo et al. [2017]], otimi-

zação de código [Nazaré et al. [2014]; Alves et al. [2015]; Quintao Pereira et al. [2013]], segurança [Rodrigues et al. [2016]], inferência da complexidade assintótica [Coppa et al. [2012]; Demontiê et al. [2015]], verificação de problemas relacionados aos acessos a memória [Nethercote & Seward [2007]]. No geral, as maneiras de se realizar análises em um programa são : de forma estática, de forma dinâmica ou por uma combinação das duas.

Uma análise estática é uma forma de analisar programas em que não há a necessidade de executar o programa. Na maioria dos casos, esse tipo de análise é feita sobre alguma versão do código fonte (AST, código intermediário, código binário, etc). Essas análises costumam percorrer a AST do programa, por exemplo, e aplicar regras em construções especificas para gerar ou coletar informação.

Por outro lado, as análises dinâmicas são realizadas durante a execução dos programas. Elas podem ser efetuadas durante uma execução real ou virtual do programa alvo. Para que esse tipo de ferramenta seja eficaz é importante que o programa alvo seja executado diferentes vezes variando os dados de entrada a fim de produzir comportamentos interessantes. Além disso, considerando que as análises dinâmicas podem involver instrumentar o programa alvo, é importante que isso seja feito de forma a minimizar o impacto que isso pode causar na execução do programa alvo.

2.3 Análise de Intervalos

A Análise de Intervalos é uma ferramenta que busca encontrar um limite inferior e outro superior para os valores que variáveis numéricas podem assumir. Essa forma de analisar programas foi introduzida por Cousot, [Cousot & Cousot [1977]], e pode ser feita de forma estática, ou seja, não exige que o programa seja executado. Basicamente, o que ela faz é associar a cada variável numérica um intervalo que contém o menor e o maior valores possíveis que podem ser assumidos por essa entidade.

A seguir é mostrado um exemplo, passo a passo, de como a Análise de Intervalos é feita para um trecho de programa. A demonstração é realizada com auxílio do código da figura 2.3a). Nela, é mostrado o código de uma função, `foo`, que atribui um valor, 0, para determinada posição de um arranjo, a qual é definida a partir do resultado da avaliação de uma condição. O predicado que controla tal atribuição é regido pelo valor da variável `n`, que também é passada para a função como parâmetro.

No Capítulo 4 é descrita uma *Análise de Intervalos Simbólica* que busca aproximar os intervalos de valores que podem ser atribuídos às variáveis do programa. Ao ser submetido para a Análise de Intervalos do Capítulo 4, o código da figura 2.3a) retorna

```
void foo(int n,
         int *v)
{
  int i;
  if (n < 10) {
    i = 0;

  } else {
    i = 10;

  }

  v[i] = 0;
}
a)
```

```
1  void foo(int n,
2           int *v)
3  {
4    int i;
5    if (n < 10) {
6      i = 0;
7  //R(i) = [0, 0]
8    } else {
9      i = 10;
10 //R(i) = [10, 10]
11   }
12 //R(i) = [0, 10]
13   v[i] = 0;
14 }
15 b)
```

Figura 2.3: a) Definição da função `foo`. b) Análise de Intervalos para `foo`.

os intervalos mostrados na figura 2.3b), considerando os resultados para o símbolo `i`. As informações utilizadas ao longo da análise surgem através das atribuições realizadas nas linhas 6, 9 e 13. É interessante notar que o resultado mostrado na linha 12 é obtido através da união dos valores possíveis para `i`, considerando que cada um dos possíveis desvios que podem ser tomados (linhas 6 e 9).

O resultado da Análise de Intervalos nos leva a algumas conclusões interessantes quanto aos valores das variáveis usadas em `foo`. Por exemplo, como mostrado na figura 2.3b), o intervalo de possíveis valores assumidos por `i` varia de 0 a 10. Note que essa análise não infere, necessariamente, todos os números que podem ser atribuídos a uma variável, mas um limite inferior e superior para eles. Durante a análise, não é possível definir qual dos desvios do comando `if` (linha 5) será tomado. Assim, é necessário considerar todas as possibilidades. Essa abordagem leva ao intervalo da variável `i` mostrado na linha 12 na figura 2.3b).

A Análise de Intervalos nos permite, também, tirar conclusões em relação à área de memória apontada por `v`. No código considerado não existem informações explícitas relativas ao número de posições que podem ser acessadas por essa variável. No entanto, considerando que o programa analisado não contém comportamentos indefinidos, `v` deve apontar para uma sequência de posições em memória de tamanho, no mínimo, igual a 11. Isso pode ser inferido uma vez que o arranjo é acessado por uma variável cujo limite superior é 10. Em geral, considera-se que um vetor deve ter tamanho mínimo igual ao maior limite superior das variáveis que são usadas para acessá-lo. Dessa forma, como `v` é indexado somente por `i` (linha 13), conclui-se que, em `foo`, a variável `v` aponta para

no mínimo 11 posições.

2.4 Computação Simbólica

Uma das técnicas fundamentais utilizadas nesta dissertação é a Análise de Intervalos Simbólica. Na Seção 2.3 são dados mais detalhes sobre o tipo de informação retornado por uma análise de intervalos e no Capítulo 4 é mostrada a análise de intervalos simbólica implementada e usada neste trabalho. Um fato fundamental dessa última análise é que ela faz uso de expressões simbólicas, [Cohen [2003, 2002]].

Expressões simbólicas são expressões algébricas compostas por símbolos, cadeias de caracteres, que representam valores concretos. As expressões desse tipo que são utilizadas neste trabalho podem ser definidas de acordo com a gramática abaixo, onde s é um símbolo e $n \in \mathbb{N}$:

$$\begin{aligned} E \;::=\; & n \mid s \mid \min(E,E) \mid \max(E,E) \mid E - E \\ & \mid E + E \mid E/E \mid E \times E \\ & \mid -\infty \mid +\infty \end{aligned}$$

2.4.1 Números Inteiros

Os números inteiros são usados para representar os valores numéricos encontrados no programa. Esses podem ser: constantes atribuídas as variáveis; tamanho de arranjos; limites para a execução de laços, entre outros. É importante salientar que a análise considerada aqui só lida com valores numéricos inteiros. Dessa forma, toda vez que a análise apresentada encontra um número real, esse dado é convertido para `int`.

2.4.2 Símbolos

No contexto deste trabalho, são considerados como símbolos qualquer nome que não tenha associado a ele um intervalo. Eles são usados para construir as expressões simbólicas que serão manipuladas no decorrer das análises apresentadas. Em geral, esses valores representam as entradas das funções. Levando isso em consideração, os símbolos podem aparecer na forma de: variáveis globais, funções não definidas e parâmetro de funções.

2.4.3 Valore Infinitos

Neste trabalho os símbolos infinitos (-∞ e $+\infty$) são considerados como números especiais. Eles são usados para representar valores extremos (o menor, -∞, e o maior número,

$+\infty$) com os quais as expressões simbólicas podem lidar. Levando em conta essas informações, os valores infinitos são manipulados de forma similar às outras expressões simbólicas, estando sujeitos às seguintes regras:

Relação de ordem: $-\infty < e < +\infty$, para toda expressão simbólica.

As operações aritméticas entre expressões simbólicas e valores infinitos considerados são:

1. $e + (+\infty) = +\infty$
2. e + (-∞) = -∞
3. e - ($+\infty$) = -∞
4. e - (-∞) = $+\infty$
5. e/($+\infty$) = 0
6. e/(-∞) = 0
7. e $> 0 \Rightarrow$ e $\times$ ($+\infty$) = $+\infty$
8. e $> 0 \Rightarrow$ e $\times$ (-∞) = -∞
9. e $< 0 \Rightarrow$ e $\times$ ($+\infty$) = -∞
10. e $< 0 \Rightarrow$ e $\times$ (-∞) = $+\infty$

Operações aritméticas entre infinitos consideradas aqui são:

1. $+\infty + (+\infty) = +\infty$
2. -∞ + (-∞) = -∞
3. $+\infty \times (+\infty) = +\infty$
4. -$\infty \times$ (-∞) = $+\infty$

2.4.4 Valores Indefinidos

Uma vez que existe a possibilidade de que computações gerem valores infinitos, é necessário estar preparado para lidar com esses valores. Algumas operações bem definidas considerando símbolos infinitos são mostradas na Subseção 2.4.3. Nesses casos, é possível derivar um resultado para essas computações.

Acontece que existem casos em que a operação com os valores infinitos não são tão simples de serem manipuladas. Quando isso ocorre, dizemos que tal computação é indefinida. Os valores indefinidos são usados para lidar com computações que não estejam bem definidas. Em geral, esse tipo de valor é obtido ao se operar com valores infinitos ou em divisões por zero. Neste trabalho é considerado que um valor indefinido acaba por invalidar toda a expressão ao qual ele pertence. As regras para geração desses valores podem ser vistas a seguir:

1. $0 \times (+\infty)$ = Indefinido
2. $0 \times (-\infty)$ = Indefinido
3. $+\infty + (-\infty)$ = Indefinido
4. $+\infty - (+\infty)$ = Indefinido
5. $(\pm\infty)/(\pm\infty)$ = Indefinido
6. $e/0$ = Indefinido

Capítulo 3

Revisão da Literatura

Neste capítulo, é apresentada uma revisão da literatura com os assuntos mais pertinentes ao que é mostrado nesta dissertação. Primeiramente, são apresentados os trabalhos mais parecidos com o que é feito aqui. Em seguida, são mostrados outros projetos que apresentam ou fazem uso de técnicas parecidas com as que são utilizadas neste trabalho.

3.1 Trabalhos Relacionados

O problema de gerar entradas que não levem a acessos indevidos de memória já foi abordado por Demontiê, em sua dissertação de mestrado, [Demontiê [2016]]. Esse é, inclusive, um dos trabalhos mais próximos do que é mostrado aqui. Na dissertação de Demontiê é desenvolvido um arcabouço de testes para funções escritas em C. Tal ferramental recebe, como entrada, um programa em código intermediário, extrai uma função de interesse e a executa com dados aleatórios (*fuzzing*). Para garantir que acessos fora dos limites dos arranjos não ocorram, Demontiê usa duas análises estáticas. Essas análises realizam inferência dos tamanhos dos arranjos utilizados nos programas. Tais análises buscam associar os tamanhos dos vetores com símbolos encontrados no programa. Com essas informações Demontiê gera um arcabouço de teste similar ao que é feito por DART (*Direct Automated Random Testing*), [Godefroid et al. [2005]]. É importante lembrar que, nesse trabalho, o pesquisador trabalha o tempo todo considerando o código intermediário, produzido pelo LLVM, [Lattner & Adve [2004b]]. Além disso, a geração do *driver* para a execução do programa é transparente aos olhos dos usuários de sua ferramenta. No trabalho apresentado aqui, por outro lado, lidamos o tempo todo (tanto na entrada quanto na saída) com o texto do código fonte. Isso envolve a criação de um *driver*, muito parecido com o que é criado por DART. Outra

diferença entre os trabalhos é que, ao invés de inferirmos o tamanho dos arranjos, o que é feito aqui é **adaptar o ambiente** em que os vetores são usados. Em outras palavras, Demontiê gera arranjos com base em símbolos do programa, nesta dissertação, nós alocamos o arranjo primeiro e adaptamos as entidades que interagem com ele em tempo de execução através dos dados de entrada que serão gerados.

Outro trabalho similar ao que é mostrado nesta dissertação foi desenvolvido por Godefroid et al., [Godefroid et al. [2005]]. *Directed Automated Random Testing* (DART) é uma ferramenta capaz executar testes de unidade de forma automática. De acordo com os autores desse artigo, o funcionamento de DART é apoiado em três técnicas principais: (1) extração de interfaces; (2) geração automática de um *driver* para executar o código; e (3) uma análise dinâmica que auxilia a geração de novos casos de teste. No trabalho apresentado aqui, são executados dois passos exatamente iguais aos dois primeiros usados por DART. A principal diferença do que é mostrado nesta dissertação e no trabalho de Godefroid et al. é que nos preocupamos com a forma como os arranjos serão acessados, ao contrário do que é feito em DART. Como apontado por Demontiê, a principal limitação da técnica de Godefroid et al. é incapacidade de relacionar arranjos com seus tamanhos. Isso, ainda de acordo com Demontiê, pode levar a acessos de memória inválidos que não ocorreriam na prática. Destacamos que o nosso principal objetivo é justamente não gerar entradas para as funções que impliquem em acessos indevidos à memória.

Em outro trabalho, Godefroid apresenta `MicroX` que é uma ferramenta que também é usada para realizar teste de *software*, [Godefroid [2014]]. Nesse artigo o autor apresenta o conceito de *Micro Execution*. *Micro Execution* é a capacidade de executar qualquer trecho de código sem a presença de um *driver* fornecido pelo usuário ou de dados de entrada. A única intervenção que o usuário deve fazer para que essa ferramenta possa executar é identificar a função, ou trecho de código, de interesse em um arquivo binário. Nesta dissertação fazemos algo parecido com isso, no entanto, o fazemos considerando o texto do código fonte de uma única função, em vez de um trecho de código binário. No caso do trabalho apresentado aqui, o usuário também não precisa escrever o *driver* que vai executar o código. No entanto, diferente do que é mostrado por Godefroid, que usa uma máquina virtual para executar um trecho de programa, geramos um *driver*, em código fonte, que realiza chamadas para o método alvo, mais parecido com o que é feito por DART. Isso permite que o usuário de `amazonc` possa realizar alterações no *driver* que é criado, ou em seus parâmetros, antes da execução do programa. Além disso, assim como acontece com DART, MicroX gera entradas para suas execuções sem considerar a interação que pode existir entre diferentes estruturas de dados, como acontece entre arranjos e outras variáveis que representam seus tama-

nhos. Nosso principal objetivo é garantir que ao se executar uma função, com os dados gerados automaticamente, não aconteçam acessos indevidos à memória causados pela má geração de dados.

Os trabalhos discutidos anteriormente são aqueles que possuem um objetivo mais próximo do que apresenteamos nesta dissertação no sentido de executar, automaticamente, programas parciais. A seguir, são apresentados mais estudos que abordam outros assuntos discutidos aqui mas que estão relacionados com técnicas mais específicas.

3.2 Análise de Intervalos

Para atingir os objetivos deste trabalho, a principal ferramenta utilizada é a Análise de Intervalos Simbólica, que é uma Análise de Intervalos em que os limites são dados por expressões simbólicas, [Cohen [2002, 2003]]. A Análise de Intervalos, de uma forma geral, já se mostrou muito útil para toda a comunidade em diversas ocasiões. Um exemplo de aplicação para tal análise é a detecção de *overflow* de inteiros como mostrado no trabalho de Quintao Pereira et al., [Quintao Pereira et al. [2013]]. Nesse trabalho os autores usam essa ferramenta para encontrar os limites de variáveis no código intermediário de C gerado pelo LLVM. Com essa informação os pesquisadores conseguiram reduzir em 50% do tempo de *overhead* gasto com checagens usadas em métodos de detecção de *overflow*.

Outra aplicação prática da Análise de Intervalos pode ser vista através do trabalho de Maas et al., [Maas et al. [2016]]. Esse trabalho, inclusive, é um dos que fazem o uso dessa tecnologia de forma mais semelhante ao que é feito nesta dissertação. A análise estática apresentada por Maas é usada para encontrar o tamanho de arranjos e produzir melhores *Library Bindings* entre as linguagens Python e C. Tanto a análise que é apresentada por Maas quanto a que é mostrada nesta dissertação são feitas de forma simbólica. No entanto, a análise de Maas é feita sobre o código intermediário de C, gerado pelo LLVM, enquanto a que apresentamos é feita sobre a Árvore de Sintaxe do programa. Além disso, a principal finalidade da análise usada nesse trabalho é a inferência de tamanho de arranjos o que se assemelha com o que fazemos aqui.

Alves et al., [Alves et al. [2015]], mostram como usar a Análise de Intervalos Simbólica para implementar um desambiguador de ponteiros. Tal ferramenta é utilizada para fornecer informações sobre diferentes pares de ponteiros que podem ou não referenciar intervalos de posições de memória semelhantes (*aliasing*). Isso, de acordo com os autores, permite alternar o fluxo de execução de programas entre regiões mais otimi-

zadas (devido a ausência de *aliasing*) e as regiões originais (lugares onde a otimização não foi possível devido a possibilidade de haver *aliasing*).

No trabalho de Nazaré et al., [Nazaré et al. [2014]], também são mostrados outras formas de uso da Análise de Intervalos Simbólica. Nesse artigo, tal análise é usada com a finalidade de diminuir o número de checagens feitas por técnicas usadas para prevenir acessos fora dos limites de arranjos de programas em C e para detectar *overflow* de inteiros. As estratégias apresentadas nesse artigo foram incorporados na ferramenta `AddressSanitizer`, [Serebryany et al. [2012]], o que possibilitou uma melhora significativa da ferramenta.

A Análise de Ponteiros é uma ferramenta importante que permite compiladores distinguirem entre diferentes espaços de memória, [Andersen [1994]]. No trabalho de Paisante et al.,[Paisante et al. [2016]], é proposta uma nova análise de *alias* (*Alias Analysis*) que faz uso de informações obtidas com Análise de Intervalos . De acordo com os autores, a combinação das análises possibilitou que uma maior precisão fosse obtida para a desambiguação de diferentes espaços de memória. Além disso, a análise apresentada se mostrou mais rápida do que o estado da arte tinha a oferecer até aquele momento.

Por fim, no artigo de de Araujo et al., [de Araujo et al. [2017]], os autores usam a Análise de Intervalos Simbólica para gerar uma forma restrita de tipos dependentes para a linguagem C. Nesse trabalho, além de propor uma forma restrita de tipos dependentes para tal linguagem, os autores demonstram como a Análise de Intervalos pode ser usada para atingir tal finalidade. O que eles fazem é, basicamente, enriquecer as declarações de tipos com informações extras retiradas da Análise de Intervalos. Com isso, são geradas anotações que melhoram a documentação do código fonte. Nesta dissertação, inclusive, usamos a mesma análise de intervalos que foi usada por de Araujo et al..

3.3 Análise Dinâmica

Acreditamos que a possibilidade de executar uma função de forma isolada pode trazer muitas vantagens para programadores que usam a linguagem C. Como prova de conceito, mostramos, no Capítulo 7, como as ideias discutidas aqui podem ser combinadas com outras ferramentas de análise dinâmica para se extrair informações valiosas a respeito da execução de um programa. Nesse caso, as ferramentas em questão são `Valgrind`, [Nethercote & Seward [2007]], e `aprof`, [Coppa et al. [2012]].

`Valgrind` é um *framework* de instrumentação usado para criação de análises dinâmicas. Dentre as diversas funcionalidades oferecidas por esse arcabouço, citamos:

debugger (`Helgrind`); *profiling* (`Cachegrind` e `Massif`); e detecção automática de erros relacionados ao gerenciamento de memória (`Memcheck`). Neste trabalho, usamos o `Memcheck` para analisar os programas reconstruídos e asegurar que eles não possuem problemas relacionados ao manuseio de memória.

O `aprof` é uma ferramenta construída sobre o `Valgrind` que foi criada com o intuito de ajudar desenvolvedores a encontrar ineficiências assintóticas nos programas. A partir de uma ou mais execuções de um método, `aprof` mede como o desempenho individual das rotinas utilizadas se escalam em função do tamanho de suas entradas. Essa ferramenta é usada aqui para verificar se a execução do código reconstruído, e executado com entradas aleatórias (*fuzzing*) e válidas, nos permite obter a mesma complexidade assintótica das funções que são obtidas quando essas são analisadas manualmente.

3.4 Códigos Parciais e Síntese de Programas

Nesta dissertação nos referimos como sendo um código parcial (ou função isolada) todo trecho de programa, fonte ou binário, que está incompleto de alguma forma. Essa falta implica em não sermos capazes de compilar, analisar ou executar os programas em questão. É possível encontrar vários trabalhos diferentes na literatura que lidam com códigos parciais de alguma maneira.

Um exemplo é o trabalho de Ribeiro et al., [Ribeiro et al. [2016]]. Nesse artigo os autores lidam com o problema de completar as declarações de tipos em um programa de forma a torná-lo compilável. Esse estudo resultou em uma ferramenta, `psyche-c`, que recebe como entrada um trecho de código parcial em C e tem como saída o mesmo texto acrescido de informações suficientes para fazer com que esse programa possa ser compilado. No trabalho apresentado aqui, esperamos como entrada um trecho de código como o que é retornado por `psyche-c`.

Outro trabalho que lida com códigos incompletos é o de Heule et al., [Heule et al. [2015]]. Nesse trabalho os autores atacam o problema de gerar código para código opaco (*opaque code*). Por opaco, os pesquisadores se referem ao trecho de programa que se pode executar mas que por algum motivo o texto do seu código fonte não se encontra disponível. Os motivos de tal indisponibilidade podem ser devido a uma chamada de uma função *built-in* da linguagem ou pela dificuldade de processá-lo (devido a ferramentas de ofuscação de código), por exemplo.

Por fim, como mostrado, os dois trabalhos desenvolvidos por Godefroid [Godefroid [2014]; Godefroid et al. [2005]] também lidam com o problema de executar código parcial de alguma maneira. No caso do `MicroX`, por exemplo, é possível escolher uma

função, ou trecho de código, em um arquivo binário para que uma máquina virtual os execute. DART, por outro lado, recebe uma função no formato de texto, na linguagem C, e cria um *driver* capaz de executá-la.

3.4.1 Stubs

Ao lidar com códigos parciais é possível que exista a falta da definição, ou até mesmo a declaração, de funções usadas ao longo do programa fonte. Se acontecer de termos que lidar com um método em que não possuímos sua definição e nem suas declarações podemos usar `psyche-c`, [Ribeiro et al. [2016]], e reconstruir as suas interfaces (declarações). Acontece que a declaração de um método não é suficiente se o objetivo final é ter a possibilidade de executar um código fonte parcial. Dessa forma, neste trabalho também lidamos com a necessidade de gerar corpos substitutos, ou *stubs*, para as funções. No entanto, queremos ser capazes de fazer isso sem violar restrições existentes no programa relativas ao acesso a arranjos.

É importante lembrar que nesta dissertação são criados *stubs* para métodos que retornam valores e arranjos de elementos com tipo *built-in* da linguagem C. Dessa forma, não consideramos os casos de retornar tipos mais complexos como *structs*. Demontiê , [Demontiê [2016]], mostra em sua dissertação como é possível gerar corpos falsos para funções que devem retornar estruturas de dados mais complexas, como as que são criadas a partir do modificador `struct`.

A criação de *stubs* para funções não definidas não é algo novo, mas a forma como o fazemos e nossos objetivos diferem de muitos trabalho anteriores. Yao et al., [Yao et al. [2014]], descrevem um método para gerar esses corpos com o objetivo de fazer com que a execução do programa siga um caminho determinado. Nossa abordagem é diferente, uma vez que, não focamos na execução de nenhum caminho em particular.

3.4.2 Geração de Dados

Em um de seus trabalhos, [Anand et al. [2013]], Anand et al. apresentam um resumo com várias estratégias para geração de dados para realização de testes automáticos. Acontece que grande parte dos trabalhos tem como foco principal a realização de testes com a finalidade de aumentar a cobertura de código. Diferente dos trabalhos apresentados nesse compilado, o foco desta dissertação é a criação de dados para a execução de funções parciais que não causem acessos inválidos a memória. Em outras palavras, o nosso objetivo não é atingir cobertura de código mas garantir que a execução aconteça de forma segura (em relação a acessos a memória).

Rapps & Weyuker, em um de seus trabalhos [Rapps & Weyuker [1985]], abordam como avaliar a qualidade de dados fornecidos para a realização de testes. Em geral, elas discutem como bons caminhos de execução devem ser selecionados. Por caminhos, elas consideram traços de execução. Critérios como os de *statement coverage* (caminhos que cobrem todos os nós) e *branch coverage* (caminhos que cobrem todas arestas) são colocados à prova. Ao fim, elas mostram quais os tipos de caminhos (classes) podem ser escolhidos para garantir um bom traço de execução para testes e como isso pode ser usado para classificar dados de entradas como bons ou não.

Capítulo 4

Análise de Intervalos Simbólica

Um componente chave usado nesta dissertação é a **Análise de Intervalos Simbólica**. Neste capítulo, são dados detalhes da implementação da análise que é usada neste trabalho. Inicialmente é mostrado o que ela espera como entrada e fornece como saída. Em seguida, detalhes técnicos e teóricos sobre a sua implementação e funcionamento serão discutidos.

A Análise de Intervalos é uma técnica capaz de encontrar os limites que variáveis numéricas podem assumir em diferentes pontos de um dado programa. Essa análise nos permite ter uma ideia melhor sobre a dimensão do universo de valores numéricos que são utilizados no código fonte. Além disso, ela é usada para diversos fins na computação, vide Seção 3.2. Neste trabalho, usamos os resultados dessa análise para analisar como arranjos são acessados.

As implementações da Análise de Intervalo encontradas na literatura, [Quintao Pereira et al. [2013]; Maas et al. [2016]; Alves et al. [2015]; Nazaré et al. [2014]], geralmente funcionam com base no código intermediário gerado pelo LLVM, [Lattner & Adve [2004a]]. Em geral, elas são implementadas junto com as transformações (*pass*) da etapa de otimização do *backend* dos compiladores. A análise utilizada neste trabalho é equivalente à que foi usada por de Araujo et al., [de Araujo et al. [2017]]. Em outras palavras, ela é implementada sobre a Árvore de Sintaxe Abstrata (AST : *Abstract Syntax Tree*) do programa.

4.1 Coleta de Informação

A análise considerada aqui percorre a AST e recolhe informações relacionadas aos usos de variáveis. A coleta de dados é feita com base nas regras da figura 4.1. Nesse esquema é possível verificar os tipos de declarações usadas para obter, restringir e

derivar informação. Seja R um mapa, $R : V \mapsto E^2$, que associa variáveis, V, a intervalos simbólicos, E^2 (definido mais adiante), e S um comando do programa de entrada. Sendo assim, a execução da análise é feita com o auxílio da relação $\mathbf{rg}(R, S)$ que usa as informações contidas em R e S para produzir novas associações entre variáveis e intervalos. Essas faixas de valores são representadas por intervalos fechados, $[l, u]$, com um limite inferior, l, e outro superior, u.

$$\mathbf{rg}(R, \text{skip}) = R \qquad \frac{R' = R \setminus v \mapsto [n, n]}{\mathbf{rg}(R, v = n) = R'} \qquad \frac{R' = R \setminus v \mapsto [s, s]}{\mathbf{rg}(R, v = s) = R'} \qquad \frac{R(v_1) = [l, u] \qquad R' = R \setminus v \mapsto [l, u]}{\mathbf{rg}(R, v = v_1) = R'}$$

$$\frac{R(v_1) = [l_1, u_1] \qquad R(v_2) = [l_2, u_2] \qquad R' = R \setminus v \mapsto ([l_1 + l_2, u_1 + u_2])}{\mathbf{rg}(R, v = v_1 + v_2) = R'} \qquad \frac{\mathbf{rg}(R, S_1) = R_1 \qquad \mathbf{rg}(R_1, S_2) = R_2}{\mathbf{rg}(R, S_1; S_2) = R_2}$$

$$\frac{\begin{array}{c} R(v_a) = [l_a, u_a] \qquad R(v_b) = [l_b, u_b] \qquad R_t = (R \setminus v_a \to [l_a, \min(u_b - 1, u_a)]) \setminus v_b \to [\max(l_a + 1, l_b), u_b] \\ \mathbf{rg}(R_t, S_t) = R'_t \qquad R_f = (R \setminus v_a \to [\max(l_a, l_b), u_a]) \setminus v_b \to [l_b, \min(u_a, u_b)] \qquad \mathbf{rg}(R_f, S_f) = R'_f \qquad R' = R'_t \sqcup R'_f \end{array}}{\mathbf{rg}(R, \text{if}(v_a < v_b)\ S_t \text{ else } S_f) = R'}$$

$$\frac{\mathbf{fp}(R, \text{if}(v_a < v_b)\ S \text{ else skip}) = R'}{\mathbf{rg}(R, \text{while}(v_a < v_b)\ S) = R'} \qquad \frac{\mathbf{rg}(R, S) = R}{\mathbf{fp}(R, S) = R} \qquad \frac{\mathbf{rg}(R, S) = R_1 \qquad R_1 \neq R \qquad \mathbf{fp}(R_1, S) = R'}{\mathbf{fp}(R, S) = R'}$$

Figura 4.1: Regras para coleta de informação da Análise de Intervalos Simbólica.

A figura 4.2 apresenta uma ideia de como é feita a coleta de informações. O processo consiste basicamente em percorrer a AST na ordem com que as declarações aparecem no programa e, cada vez que uma das regras da figura 4.1 é aplicável, as informações contidas nesses nós são recuperadas e o mapa R é atualizado.

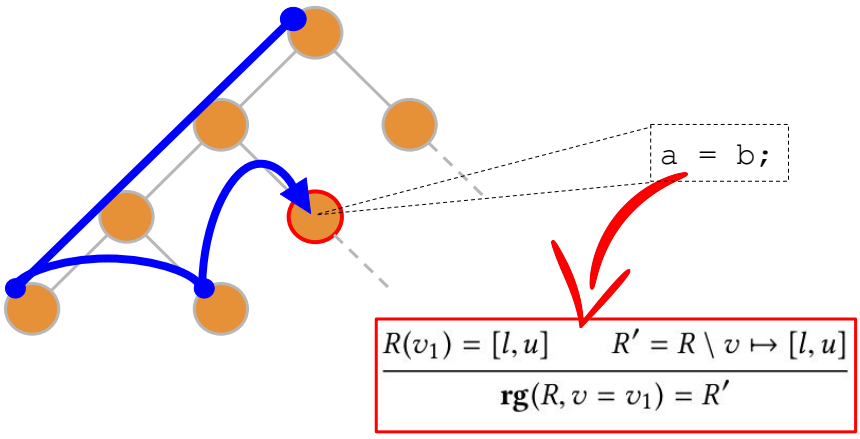

Figura 4.2: Esquema de como é feita a coleta de dados na AST. Após a atribuição "$a = b$;" o intervalo de a deve ser igual ao de b, segundo a regra apresentada na figura.

4.2 Alargando e Estreitando Informações

A obtenção de informação dentro de laços é feita com o auxílio de técnicas de alargamento (*widening*) e estreitamento (*narrowing*). A ideia básica do funcionamento delas é aumentar, ou diminuir, os valores considerados ao máximo possível. Apesar desse método levar a tomadas de decisões muito conservadoras, ele é necessário. Isso deve ser feito para evitar que a execução simbólica de um programa não termine. Em outras palavras, elas são importantes para garantir que o ponto fixo (*fixed point*: **fp**) da execução simbólica do programa seja alcançado.

Inicialmente o corpo do comando de repetição é percorrido e todas as variáveis incrementadas ou decrementadas terão seus valores expandidos para valores extremos, ∞ ou $-\infty$. A verificação da mudança dos valores simbólicos das variáveis é feita considerando os intervalos que elas possuíam antes e depois do laço.

Em seguida, é realizada mais uma travessia no corpo do laço para estreitar os valores previamente alargados. É importante mencionar que, quando o algoritmo se depara com comandos de repetição aninhados, os intervalos relativos a laços mais externos são resolvidos primeiro. Isso faz com que a análise tenha que tomar decisões conservadoras em relação aos limites encontrados. Em troca, ao analisar um laço mais interno, podemos assumir que o contexto no qual ele está inserido já foi resolvido.

4.2.1 Exemplo

O exemplo apresentado nesta seção é feito com base na figura 4.3. Nela é apresento um esquema com a execução da Análise de Intervalos Simbólica em uma função. Nela é dado foco para os possíveis valores assumidos pela variável i. A execução da análise é iniciada partindo do ponto anterior à primeira instrução do corpo da função, (1). Em (2) é mostrado o que acontece com o valor do intervalo de i após uma atribuição. O retângulo tracejado é usado para mostrar o valor corrente do intervalo dessa variável.

Após esse passo, em (3), é iniciado a execução laço. Pelas regras da figura 4.1, ao entrar no laço, o intervalo de i é dado por $[0, min(10-1, 0)]$ que resulta em $[0, 0]$. Em seguida, (4), é mostrado o que acontece com o intervalo de i após um incremento de uma unidade na variável, $R(i) = [1, 1]$. Depois desse passo, a primeira passada pelo corpo do laço termina e é feita uma checagem das variáveis usadas dentro dele. Essa verificação é feita com o intuito de buscar por variáveis numéricas que estão variando, positivamente ou negativamente, com a execução do laço. Isso leva ao alargamento (*widening*) do intervalo da variável i mostrado no passo (5). Repare que o limite superior do seu intervalo nesse momento é dado por ∞.

Seguindo com a análise, em (6), é mostrado o passo de estreitamento (*narrowing*). A condição do laço, "$i < 10$", é usada para corrigir o valor do intervalo previamente expandido. Dessa forma, ao fazer a segunda, e última, passada pelo laço o valor do intervalo de i é igual a $[0, 9]$, como mostrado em (6). Esse último resultado apresenta o menor e o maior valores possíveis de serem assumidos por i ao iniciar a execução do laço. Em (7) é mostrado, mais uma vez, o que acontece com o intervalo de i após um incremento da variável em uma unidade. Esse é o valor final computado para i. Em outras palavras, ao final de uma laço os valores possíveis de serem assumidos por i variam entre 1 e 10.

Em (8), é mostrado o valor corrente de i ao deixar o laço. Esse valor é encontrado a partir da união do valor final de i dentro do laço, $[1, 10]$, e o valor de i antes de passar pelo comando `while`, $[0, 0]$. Isso acontece porque a execução simbólica é feita considerando todos os caminhos que podem ser tomados pelo fluxo de execução, que inclui, nesse caso, a possibilidade de não entrar no laço. Por fim, em (9) é mostrado um mapeamento final dos valores que i pode assumir em diferentes pontos do programa.

4.3 Intervalos Simbólicos

Ao fim do algoritmo de Análise de Intervalos tem-se como saída uma associação de pontos de programa com os intervalos de cada uma das variáveis até aquele ponto. Os limites inferiores e superiores das variáveis são apresentados por meio de **expressões simbólicas**, Seção 2.4. Símbolos de programas são nomes encontrados no código fonte que não podem ser escritos em função de outros nomes. No contexto deste trabalho, consideramos um símbolo como qualquer nome que não tenha associado a ele um intervalo.

Expressões simbólicas formam um semi-reticulado, cuja ordem parcial é dada pela relação $<$ entre símbolos. De modo geral essa ordenação pode ser vista como: $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots < +\infty$. É importante destacar que não existe uma ordem específica entre expressões que envolvem símbolos diferentes. Partindo dessas definições, nosso reticulado é definido como: $\mathcal{R} = \langle E^2, \sqsupseteq, \sqcap, \sqcup, \bot = \emptyset, \top = [-\infty, \infty] \rangle$, onde a ordem parcial é definida como $l_1 \leq l_2 \wedge u_1 \geq u_2 \Rightarrow [l_1, u_1] \sqsupseteq [l_2, u_2]$. O operador *meet* é visto como a interseção dos intervalos: $[l_1, u_1] \sqcap [l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$; e o operador *join* como a união: $[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$.

Considerando as regras da figura 4.1, na figura 4.4 é mostrado um exemplo de como os intervalos simbólicos ficam distribuídos em diferentes pontos de um programa.

Figura 4.3: Esquema com exemplo de como acontecem as operações de *widening* e *narrowing*.

Perceba que a linha 18 desse código fonte é um ponto de encontro de informações. Nesse ponto, são feitas uniões dos intervalos obtidos por cada variável considerando os dois caminhos de execução que poderiam ser tomados pelo condicional da linha 5.

```
1  void exemploIntervalosSimbolicos(int a, int b, int c)
2  {
3    int i, j;
4    // R = {a:[a,a],b:[b,b],c:[c,c]}
5    if (a > b) {
6      // R = {a:[max(a,1+b),a], b:[b,min(a-1,b)],c:[c,c]}
7      i = a;
8      j = 0;
9      // R = {i:[max(a,b+1),a],j:[0,0],
10     //       a:[max(a,1+b),a], b:[b,min(a-1,b)],c:[c,c]}
11   } else {
12     // R = {a:[a,min(a,b)],b:[max(a,b),b],c:[c,c]}
13     i = 2*b;
14     j = 9;
15     // R = {i:[max(2*a,2*b),2*b],j:[9,9],
16     //       a:[a,min(a,b)],b:[max(a,b),b],c:[c,c]}
17   }
18   // Ponto de encontro de informacoes
19   // R = {i:[min(max(a,1+b),max(2*a,2*b)),max(a,2*b)],
20   //       j:[0,9],a:[a,a],b:[b,b],c:[c,c]}
21   int k = j + c;
22   // R = {i:[min(max(a,1+b),max(2*a,2*b)),max(a,2*b)],
23   //       j:[0,9],k:[c,9+c],a:[a,a],b:[b,b],c:[c,c]}
24 }
```

Figura 4.4: Exemplo de como os intervalos simbólicos variam ao longo do código fonte.

Capítulo 5

Geração de entradas para teste

Neste capítulo é apresentado o processo que visa garantir que os acessos aos arranjos, durante a execução automática de um método, nunca ultrapassem seus limites. Essa técnica conta com dois pilares principais. O primeiro deles é o **Grafo de Acesso**. Essa estrutura de dados modela como os acessos são feitos aos arranjos. Tal abstração serve como ponto de partida para adaptar as entradas das funções tratadas por `amazonc`. O segundo pilar é um algoritmo de ondas (*wave algorithms*) que acontece em duas etapas. Em cada etapa são coletados e impostos diferentes tipos de restrições sobre as variáveis do programa.

5.1 Ideia Geral

Durante a Análise de Intervalos Simbólica, são coletados todos os acessos feitos aos arranjos. Na figura 5.1, por exemplo, são coletados os acessos a `v` em dois pontos distintos do texto do programa. O primeiro acontece na linha 10, `"i + 1"`, e o segundo na linha 13, `"2 + get()*n"`. Considerando o primeiro acesso onde o intervalo de `i` é [0, n - 1], para garantir que essa indexação não cause problemas relacionados ao mau uso de memória, `v` deve ser pelo menos de tamanho `n + 1`. Além disso, `n` deve ser escolhido de forma que `n - 1` seja inferior ao tamanho de v.

A partir do código da figura 5.1, é possível ver que os dados podem ser dependentes uns dos outros. Dessa forma, ao gerar dados para executar uma função automaticamente, como a anterior, é necessário descrever e resolver adequadamente essas dependências. Ainda assim, nesse exemplo, resta uma dúvida: qual das variáveis, `n` ou `v`, deve se adaptar ao valor da outra? No caso desta dissertação, a opção escolhida foi a de adaptar o valor `n` com base no tamanho do arranjo `v`.

```
1  int* init(int);
2  int get();
3
4  int function(int n, int* v) {
5    v = init(n);
6    int i;
7
8    for (i = 0; i < n; i++) {
9      // R(i) = [0, n-1]
10     v[i + 1] = i; // acesso
11   }
12   // R(i) = [0, n]
13   return v[2 + get()*n]; // acesso
14 }
```

Figura 5.1: Código usado como exemplo no capítulo.

Para resolver esse problema, inicialmente é criado um grafo com as dependências entre os arranjos e as expressões usadas para acessá-los. Após saber quais entidades se relacionam umas com as outras, as restrições de cada uma, dadas como intervalos de valores possíveis a serem assumidos por variáveis numéricas, são propagadas por essa estrutura de dados. A esse grafo, é dado o nome de **Grafo de Acessos**. Quando todas as variáveis recebem suas restrições e todas podem ser satisfeitas, é possível começar a etapa de geração de valores para elas.

5.2 O Grafo de Acessos

Para que seja possível gerar valores corretos é necessário identificar quais variáveis precisam ser definidas e quais são as restrições sobre elas. Nos casos de tipos numéricos, essas restrições são representadas como intervalos simbólicos que representam limites sobre os valores que elas podem assumir. Esses intervalos precisam ser derivados de forma a garantir que qualquer expressão usada para acessar um vetor seja menor ou igual ao tamanho associado a essa estrutura de dados. No caso dos arranjos, suas restrições indicam o tamanho mínimo que cada uma de suas dimensões deve possuir.

Definição 1 (`Entrada`): uma variável é considerada como uma entrada se ela precisa de uma definição.

Variáveis globais, argumentos dos métodos e funções que só possuem suas de-

clarações são entradas. É importante notar que no caso de procedimentos sem suas definições, o que precisa ser definido é o valor a ser retornado pelo *stub* criado para eles. Entradas podem ser vistas como o oposto de uma variável local, já que sua definição é fixa pelo texto do código fonte.

Definição 2 (`Variável em Aberto`): uma variável em aberto é uma entrada numérica.

No exemplo da figura 5.1, `i` é uma variável local, `v` e `init` são arranjos de entrada e `n` e `get` são variáveis em aberto. Para saber as restrições sobre as variáveis de entrada, é necessário saber quais expressões acessam os arranjos. Expressões enviam aos arranjos associados a elas (através de acessos) quais são seus intervalos de valores possíveis de serem acessados. Se uma dada expressão não depende de variáveis de entrada, então seu intervalo é fixo e o arranjo precisa adaptar ao seu tamanho a esse valor. Quando um arranjo sabe seu tamanho ele impõe que as expressões usadas para acessá-lo devam estar em um intervalo de $[0, size - 1]$, onde $size$ é o tamanho do vetor considerado.

Para permitir o câmbio de informações, é utilizado uma estrutura de dados especial, o Grafo de Acessos.

Definição 3 (`Grafo de Acessos`): um Grafo de Acessos é um grafo orientado $G = ((N \cup V), (D \cup A), L)$ tal que:

- N é o conjunto de nós *numéricos*. Um nó numérico é dado por uma *expressão* (como, por exemplo: $n + 2$ ou $n \times m$, veja a Definição 4) ou uma variável em aberto.
- V é o conjunto de nós de *arranjos*. Um nó de arranjo é usado para representar um ou mais arranjos de entrada. Existem casos em que mais de um vetor compartilham as mesmas restrições. Nesses casos, eles são associados a um mesmo nó do Grafo de Acessos.
- D é o conjunto de arestas entre nós numéricos. Se $(n_1, n_2) \in D$, então a expressão dada por n_2 depende de n_1. Como exemplo, se $n_1 = a$ e $n_2 = 2 + a$ então $(n_1, n_2) \in D$.
- A é o conjunto de arestas que ligam nós numéricos à nós de arranjos. Eles representam acessos feitos aos arranjos. Se $(n, v) \in A$, então n é usado para acessar v.

- $L : A \to \mathbb{N}$ é um mapa que associa arestas de A à números naturais. Ele é usado para registrar qual é a dimensão (começando de 0) acessada na aresta que liga um nó numérico à um nó de arranjo. Se $(n, v) \in A$ e $L(n, v) = i$, n acessa v em sua i-ésima dimensão.

Não é permitido que as expressões usadas para acessar os arranjos tenham qualquer formato. Veja que, se uma expressão tem um intervalo associado a ela (ou seja, seu valor resultante precisa ser maior ou igual a um limite inferior e menor ou igual a um limite superior), essa informação tem que ser propagada para os símbolos que a compõem. Para que isso seja possível, é necessário que sejam desenvolvidas formas de compartilhar essas restrições. Neste trabalho, são considerados dois formatos possíveis para essas expressões: se elas estiverem na forma *afim* ($a_0 + a_1S_1 + \ldots + a_nS_n$) ou na forma de um *produto* ($S_1 \times S_2$).

Definição 4 (`Nós de Expressão`): Um nó *afim* representa uma expressão que possui a forma $a_0 + a_1S_1 + \ldots + a_nS_n$, em que $(a_i)_{i\in[0,n]}$ são valores literais e os $(S_i)_{i\in[1,n]}$ são símbolos. Se $n = 0$, então o nó representa uma constante literal, a_0. Um nó de *produto* introduz um símbolo novo que representa a multiplicação de dois outros símbolos: $P = S_1 \times S_2$.

- *Observação 1.* Tanto os nós de produtos quanto os de variáveis em aberto definem símbolos e, por isso, são chamados de nós de *símbolos*. (Nós afins não produzem símbolos novos.)

- *Observação 2.* Um símbolo gerado a partir de um nó de produto pode ser usado dentro da expressão de um nó afim.

No exemplo da figura 5.1, 0, $i + 1$ e $2 + get() \times n$ são expressões válidas. No entanto, um novo símbolo $P_{get()n} = get() \times n$ é criado e a expressão $2 + get() \times n$ se torna $2 + P_{get()n}$ que é afim.

5.3 A Construção do Grafo de Acessos

Nesta seção, é descrito o algoritmo responsável por construir o Grafo de Acessos. A figura 5.2 apresenta um diagrama de classes simplificado com os tipos de nós considerados nessa estrutura de dados. Primeiramente, são extraídas as entradas (variáveis em aberto e arranjos) da função alvo. Cada entrada cria um novo nó no Grafo de Acessos. Em seguida, são criados novos nós com todas as expressões usadas para acessar

arranjos. Os nós dessas expressões são ligados aos nós dos arranjos que elas acessam e aos nós de entrada que elas dependem. A seguir, são dados detalhes sobre os passos realizados no processo de construção de um Grafo de Acessos:

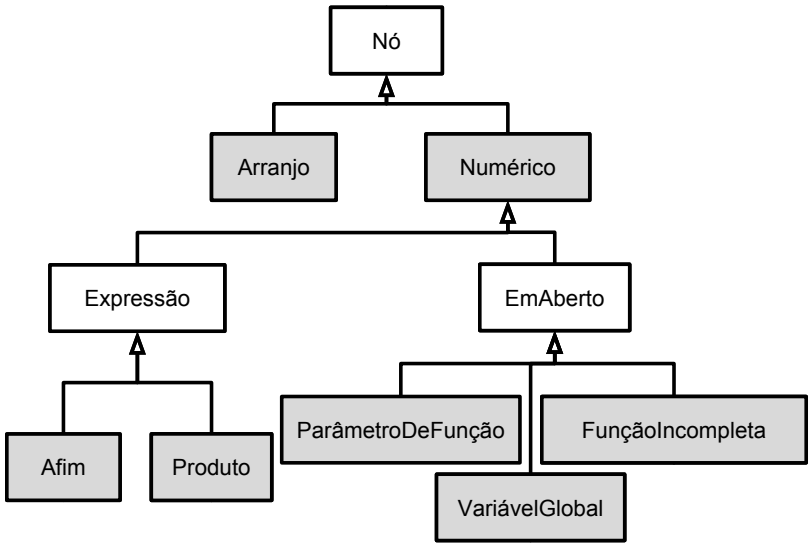

Figura 5.2: Tipos de nós no Grafo de Acessos.

Passo 1: Extração das variáveis abertas e arranjos. Esse passo também realiza uma travessia na AST do programa para coletar informações. Todas as entradas (símbolos de funções indefinidas, variáveis globais e argumentos do método alvo) são extraídas e, para cada uma, são gerados novos nós no Grafo de Acesso. Todos os arranjos, incluindo os que são locais, são coletados.

- *Observação 1.* Se a entrada é uma variável numérica, então um nó de variável aberta é criado. Caso contrário, se esta é um arranjo, então um nó de arranjo é criado.

No exemplo da figura 5.1, *get*() e n produzem dois nós de variável em aberto. v e *init*(), por outro lado, produzem dois nós de arranjos.

No entanto, note que por causa da atribuição $v = init(n)$, se v é acessado por uma expressão e então *init*() deve gerar um arranjo cujo tamanho é maior do que e. Em outras palavras, v e *init*() compartilham as mesmas restrições de acesso. Nesses casos, quando nós de vetores compartilham restrições, eles são fundidos e ficam sujeitos às mesmas restrições.

Fundindo nós de Arranjos. Se uma variável de tipo arranjo a é atribuída a outra variável de arranjo b ($b := a$), então o nó de arranjo referente a cada uma das variáveis é fundido. O nó de arranjo resultante representa as duas variáveis, a e b.

Uma vez que as variáveis em aberto e as de arranjo tenham sido coletadas, o próximo passo é lidar com as expressões que acessam os arranjos.

Passo 2: Ligando expressões e arranjos. Para cada expressão simbólica e que acessa um arranjo v, é aplicado a essas entidades um algoritmo que cria arestas entre essas duas entidades. Primeiramente, esse algoritmo percorre a AST correspondente a expressão e para produzir um conjunto de expressões simbólicas que dependam somente de variáveis em aberto. Para tal, os símbolos que representam as variáveis locais são trocados pelos valores de seus intervalos produzindo, dessa forma, duas expressões simbólicas (uma para o limite inferior e outra para o superior). Considerar os limites dos arranjos é suficiente: se um arranjo aceita acessos feitos por esses extremos, ele irá aceitar também qualquer valor entre eles. Uma vez que o caminhamento na AST tenha terminado, cada valor simbólico é transformado em uma expressão afim. Se necessário, nós de produtos são criados e ligados aos símbolos que os constituem. Quando é obtido sucesso em transformar uma expressão para a sua forma afim, um novo nó, de expressão afim, é criado e adicionado no grafo. Todos os nós de símbolos cujos símbolos são usados no nó afim se tornam pai do novo nó. O nó de expressão afim recentemente criado se torna pai do nó do arranjo v.

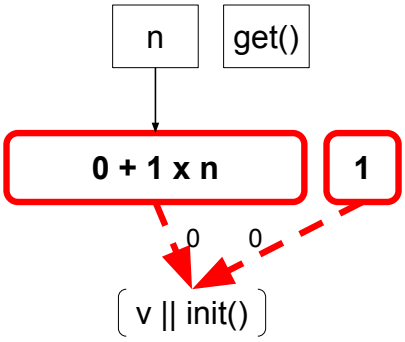

Figura 5.3: Resultado da análise das expressões (Passo 2).

As arestas entre nós de expressões afins e nós de arranjos possuem um rótulo indicando o número da dimensão que foi acessada pela expressão. Nas figuras de grafos mostrados até agora, esse rótulo corresponde ao 0 que aparece do lado das arestas tracejadas (indicadas aqui para representas arestas do conjunto A). No exemplo anterior, figura 5.1, $i+1$ é usado para fazer um acesso a v. Após algumas checagens, i é detectado como uma variável local de intervalo $[0, n-1]$. Isso produz então dois valores simbólicos, $\{0, n-1\}$. 1 produz um único valor simbólico $\{1\}$. São realizadas operações com os valores simbólicos encontrados e as expressões resultantes são colocadas na forma afim $\{1, 0 + 1 \times n\}$. Dois nós afins são criados. 1 não possui nós pais mas se

torna um nó pai do nó de v. $(0+1\times n)$ tem a variável em aberto n como pai e também se torna pai de v. As modificações resultantes no Grafo de Acesso são destacadas em negrito na figura 5.3.

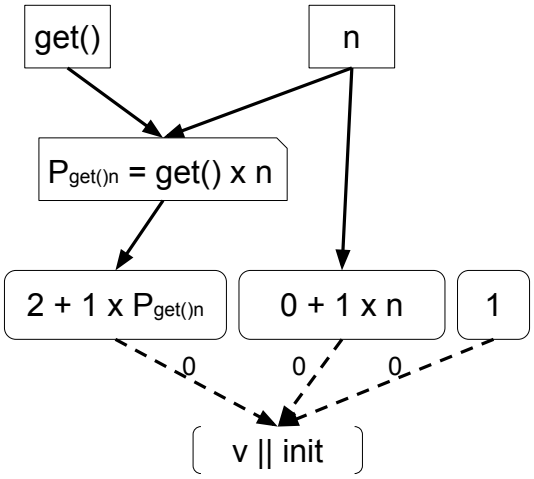

Figura 5.4: Exemplo de Grafo de Acesso para o código da figura 5.1.

Em seguida, a expressão "$2 + get() \times n''$ passa pelo mesmo processo que "$i + 1''$. Contudo, será gerado um nó de produto para $get() \times n$. O novo símbolo criado, com o nó de produto, irá substituir a multiplicação na expressão que vai passar a ser afim $2 + 1 \times P_{get()n}$.

A partir deste ponto, o Grafo de Acessos está construído. O grafo associado com o código da figura 5.1 é mostrado na figura 5.4.

5.4 Propagação de Informação

Uma vez que o Grafo de Acessos da função alvo esteja construído, as informações podem ser distribuídas. Existem duas possibilidades para lidar com os dados existentes em cada nó: (1) derivar o tamanho do vetor partindo das variáveis em aberto ou (2) impor valores sobre as variáveis em aberto com base no tamanho do arranjo. A segunda opção foi escolhida no caso deste trabalho. Isso aconteceu porque os intervalos das entradas não são conhecidos, a princípio (o que é equivalente a considerar $[-\infty, +\infty]$). Além disso, é sabido que os acessos devem ser valores maiores do que 0 e, se existir acessos usando constantes literais, é possível obter esses valores analisando o texto do código fonte. Dessa forma, nós de arranjos impõem restrições sobre os valores dos nós numéricos.

A abordagem usada para realizar a propagação de informação é feita com base no algoritmo descrito a seguir. São consideradas duas ondas de informação. A primeira delas consiste nos arranjos que recebem os seus requisitos mínimos de tamanho (são considerados os números literais usados para acessar esse vetor). Em seguida, arranjos enviam os Intervalos de Acessos que podem ser usados pelas expressões usadas para acessá-los. As expressões, representadas pelos nós numéricos, dividem os intervalos disponíveis para cada uma das parcelas, ou termos, que as compõem. Os nós das variáveis em aberto (entradas numéricas) são os últimos a receber as informações. Se um nó recebe mais de uma mensagem, são feitas interseções das mensagens para se chegar a um valor final.

Primeira onda: Literais (top-down). Cada nó que representa um valor literal enviam seus valores para os nós de arranjos. Quando um nó de vetor recebe um literal de uma aresta com rótulo d, ele atualiza seu valor de acesso máximo, por um literal, naquela dimensão.

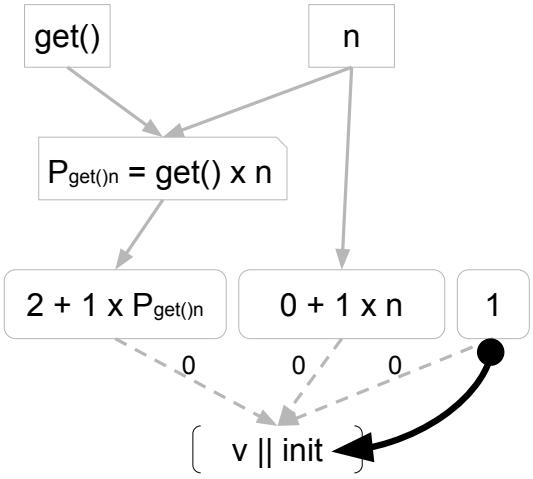

Figura 5.5: Primeira onda: *top-down*.

A figura 5.5 mostra um exemplo do funcionamento da primeira fase da troca de informações para o grafo da figura 5.4. Nesse caso, só existe um valor literal, 1, que pode ser usado para acessar $v|init()$, na primeira dimensão. Dessa forma, a primeira dimensão de $v|init()$ deve ter um tamanho estritamente maior que 1.

Depois dessa etapa, os arranjos conhecem quais devem ser os tamanhos mínimos de cada uma de suas dimensões. Com essa informação, eles produzem um novo símbolo, $size_i$, para cada dimensão, i.

Segunda onda: Propagação do tamanho dos arranjos (bottom-up). O caminhamento da segunda onda funciona de forma similar ao que acontece em uma busca em

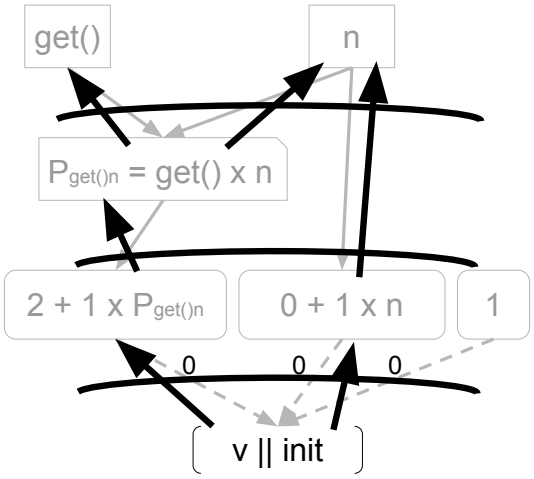

Figura 5.6: Segunda onda (*bottom-up*).

largura. Nessa etapa, cada nó possui um comportamento predefinido para dois tipos de ações possíveis: recebimento e envio de mensagens. Em geral, receber uma mensagem implica em saber sobre quais informações o nó recebente está. Enviar uma mensagem, por outro lado, está relacionado em restringir o universo de valores disponíveis pelos pais dos nós. Inicialmente, todos os nós numéricos são inicializados com o intervalo $[-\infty, +\infty]$. A seguir são dados detalhes sobre o comportamento de cada nó ao receber e enviar mensagens:

1. **Arranjos**

   a) *Ao receber uma mensagem:* Esses nós não recebem mensagens nessa etapa (eles já receberam informações na etapa anterior). Por outro lado, o algoritmo começa a sua execução partindo deles.

   b) *Ao enviar uma mensagem:* Durante essa etapa, os nós de arranjo enviam seus intervalos de acessos disponíveis para cada um de seus nós pais que não são constantes literais. Essas mensagens, correspondem aos índices que as expressões estão liberadas para acessar. O valor da mensagem consiste no intervalo $[0, size_i - 1]$ (em que $size_i$ corresponde ao tamanho da dimensão i).

2. **Nós Afins**

   a) *Ao receber uma mensagem:* Quando um nó afim recebe uma mensagem, com um intervalo de acesso, ele faz a interseção desse valor com as restrições que ele já possui, seus limites de acesso são inicializados com $[-\infty, +\infty]$.

b) *Ao enviar uma mensagem:* Ao enviar uma mensagem, o primeiro passo realizado por um nó afim é se livrar da sua constante, se ela existir. Para tal, a constante em questão é desconsiderada e seu valor é subtraído do limite superior do espaço de acessos permitido ao nó. Em seguida, o intervalo resultante é divido igualmente entre cada uma das parcelas existentes na expressão afim. Como exemplo, suponha que uma dada expressão possa variar entre $[0, size_i - c]$, em que c é a constante que existe na expressão. Suponha ainda que a expressão possua n termos. Dessa forma, cada um dos símbolos que compõem a expressão recebem um intervalo referente a $[0, (size_i - c)/n]$. É possível que uma das parcelas anteriores esteja sendo multiplicada por uma constante a, por exemplo. Nesse caso, o intervalo passado ao nó do símbolo relativo ao termo anterior seria $[0, (size_i - c)/an]$.

3. **Nós de Produto**

a) *Ao receber uma mensagem:* Os nós que representam os produtos, ao receber uma mensagem, fazem uma interseção do seu valor corrente com a informação recebida.

b) *Ao enviar uma mensagem:* Da mesma forma que os nós representantes das expressões afins, esses nós dividem sua informação igualmente entre os seus nós pais. Dessa forma, suponha que um nó relativo a multiplicação dos símbolos a e b, ab, esteja submetido ao intervalo $[l, u]$. Para transmitir a informação, inicialmente é considerado a divisão, em parcelas iguais, do limite superior do intervalo. Para tal, é usada a operação de raiz quadrada, $\sqrt{u}$. Isso garante que o maior valor a ser atingido pela multiplicação seja u. Em seguida, o valor o limite inferior do intervalo produzido é dado por $max(1, \sqrt{l})$. Forçar o limite inferior em 1 é feito para garantir que a multiplicação relativa ao nó não resulte no valor 0. Uma vez que o intervalo seja construído, $[min(1, \sqrt{l}), \sqrt{u}]$, ele é enviado para cada um dos nós pais do nó produto em questão.

4. **Variáveis em Aberto**

a) *Ao receber uma mensagem:* Assim como os outros nós, quando uma variável em aberto recebe uma mensagem ela atualiza seu intervalo fazendo a interseção dos valores recebidos.

b) *Ao enviar uma mensagem:* Esses nós não possuem nós pais, eles são os extremos do grafo. Por esse motivo, eles não enviam informação.

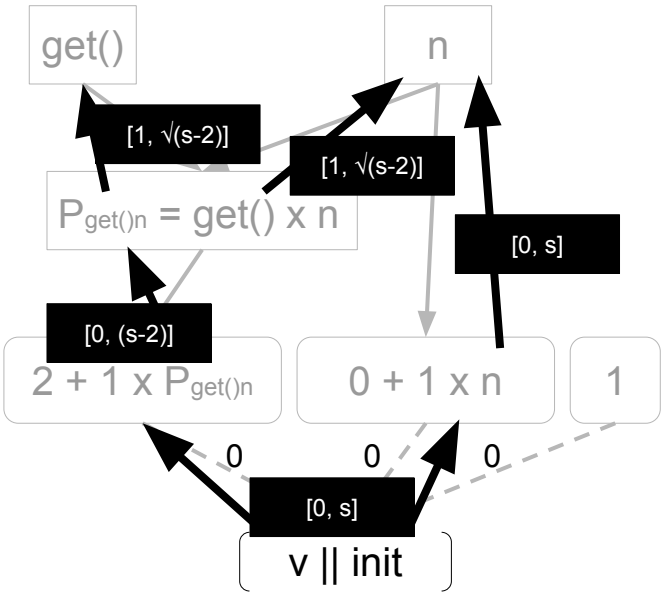

Figura 5.7: Mensagens enviadas durante a segunda onda.

A figura 5.6 apresenta um esquema com a ideia de como a segunda troca de informações funciona. Na figura 5.7, é possível ver como a troca de mensagens é realizada para o exemplo de grafo mostrado na figura 5.4. Na figura 5.8 é possível ver os valores dos intervalos finais de $get()$ e n após a segunda onda.

Uma vez que a troca de informações seja finalizada, as restrições sobre as variáveis abertas ficam em função dos símbolos de tamanhos dos arranjos. Dessa forma, seus valores dependem que os valores de tamanhos dos arranjos sejam definidos. A avaliação de fato das expressões são feitas em tempo de execução, com valores concretos, depois que são criados dados para as variáveis de tamanho dos vetores. A ordem com que as entradas devem ser definidas, que garante que as restrições sejam respeitadas, é determinada pelo Grafo de Inicialização, Seção 6.1.

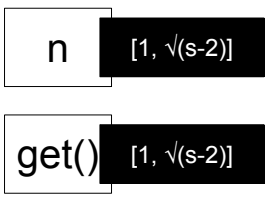

Figura 5.8: Resultado final da segunda onda.

Capítulo 6

Geração de Código

Uma vez que todas as entradas saibam as restrições que devem atender já é quase possível gerar código para executar a função alvo. O detalhe a ser considerado é que a criação do código não pode ser feita sem considerar as dependências existentes entre os símbolos. Isso acontece porque, como mostrado no capítulo anterior, as entradas estão em função dos símbolos dos tamanhos dos arranjos. Neste capítulo são dados detalhes da forma como esse problema é tratado por `amazonc`. Inicialmente, será apresentada a estrutura de dados que auxilia a estabelecer a ordem de inicialização das entradas, Seção 6.1. Em seguida, é dado uma breve descrição de como o *driver* final é gerado.

6.1 O Grafo de Inicialização

Para cada nó de arranjo e variáveis em aberto existentes no Grafo de Acessos, seus intervalos e valores numéricos são definidos em tempo de execução. As restrições sobre essas entradas já foram definidas anteriormente, Seção 5.4. No entanto, tanto os arranjos quanto as variáveis em aberto estão em função de outros símbolos (relacionados as tamanhos dos vetores). Isso implica que os símbolos de tamanho devem ser definidos antes dos seus dependentes. A ordem de definição correta das entradas é atingida graças ao Grafo de Inicialização. Para cada símbolo que representa uma entrada ou o tamanho de um arranjo, são criados dois nós nesse grafo. Um deles representa o intervalo do símbolo, *nó de intervalo*, e o outro representa o seu valor, que é usado para a realização da chamada da função alvo, *nó de valor*. É importante destacar que arranjos sobre as mesmas restrições continuam sendo representados pelos mesmos nós no Grafo de Inicialização. Uma aresta do nó A para o nó B, (A, B), implica que A deve ser definido antes de B.

Existem 2 tipos de regras usadas para a criação de arestas entre os nós do Grafo de Inicialização, GI, são elas:

1. **O Mesmo Símbolo**: todo nó de símbolo tem associado a ele um nó de intervalo. Dessa forma, sempre vai existir uma aresta que sai de um nó de intervalo em direção à um nó de símbolo, $(IntervaloDoSimbolo, Simbolo)$. Com essa regra, fica garantido que um valor não pode ser gerado antes que seu intervalo tenha sido definido.

2. **Os Requisitos do Intervalos**: seja S o conjunto de nós relativos aos símbolos que aparecem nas expressões simbólicas de um nó de intervalo, r. Então, cada $s \in S$ deve ser um nó pai de r, $\forall s \in S \ . \ \exists(s, r) \in GI$. De fato, um intervalo não pode ser computado se os símbolos usados em sua composição não forem definidos a priori.

O Grafo de Inicialização equivalente aos resultados apresentados na figura 5.8 é mostrado na figura 6.1. Nesse grafo é possível notar que o intervalo de todos os símbolos de entrada do programa dependem do valor do símbolo s, representante do tamanho associado a v e $init$. Esse símbolo, por sua vez, depende somente de suas restrições de valores mínimos e máximos.

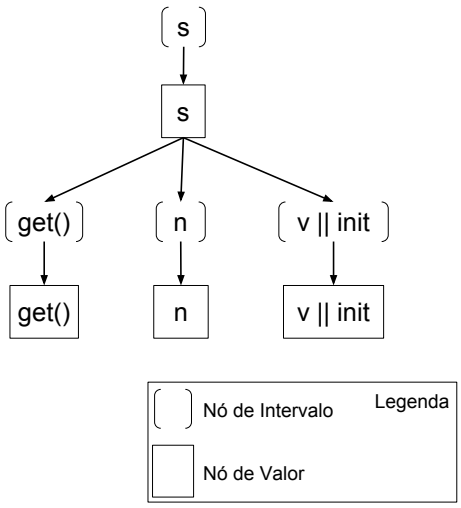

Figura 6.1: Grafo de Inicialização para os resultados encontrados na figura 5.8.

As restrições de valores mínimos relacionadas a um símbolo que representa o tamanho de um vetor são aquelas encontradas na primeira fase do algoritmo de ondas (*top-down*). O valor máximo, por outro lado, é um parâmetro de `amazonc`. Esse valor pode ser configurado antes que a ferramenta seja executada, através dos parâmetros de linha de comando, e também pode ser modificado no texto do programa que é gerado

para o *driver*. Caso não sejam fornecidos parâmetros ao executar a ferramenta, um valor padrão, igual 500, é usado como limite máximo.

6.2 A Criação do Driver

Depois que o Grafo de Inicialização, Seção 6.1, estiver pronto, o arquivo principal, que contém a função `main`, responsável por definir todas as entradas e realizar chamadas ao método alvo, pode ser criado. Um nó do Grafo de Inicialização é tido como pronto (para ser escrito no texto do arquivo do *driver*) quando ele não possui nenhuma dependência (nó pai). Quando um nó (de valor ou intervalo) atinge esse estado sua definição é adicionada ao arquivo principal e, em seguida, ele é removido do Grafo de Inicialização, possivelmente permitindo que outros nós atinjam o estado de prontidão.

Considerando que $x\_\_lower$ e $x\_\_upper$ são respectivamente os valores mínimo e máximo encontrados como intervalo final da variável em aberto x, o código das figuras 6.2, 6.3 e 6.4 são usadas para dar uma ideia de como é feita a definição de x dependendo do seu papel no programa:

```
int main() {
  //...
  for (...) {
    //...
    int x = rand(x__lower, x__upper);
    //...
    funcaoAlvo(..., x, ...);
    //...
  }
}
```

Figura 6.2: se x é um parâmetro de tipo inteiro do procedimento que será executado.

O valor a ser gerado como tamanho da dimensão de um arranjo é escolhido aleatoriamente com base no intervalo $[(maiorAcesso) + 1, TamanhoMaximo]$, se ele não possui tamanho fixo, e $[n, n]$, se seu tamanho é fixo e igual a n. O valor de $TamanhoMaximo$, pode ser passado como parâmetro para `amazonc`. Caso ele não seja fornecido, como discutido anteriormente, um valor padrão, 500, é usado. O valor de $maiorAcesso$, por outro lado, consiste no valor que é obtido em uma das etapas

```
int a, b;
int x() { return rand(a, b);}
int main() {
  //...
  for (...) {
    //...
    a = x__lower;
    b = x__upper;
    //...
    funcaoAlvo(...);
    //...
  }
}
```

Figura 6.3: se x for uma função usada por $funcaoAlvo$ (considerando que esse método retorna um valor inteiro e não recebe parâmetros).

```
int x;
int main() {
  //...
  for (...) {
    //...
    x = rand(x__lower, x__upper);
    //...
    funcaoAlvo(...);
    //...
  }
}
```

Figura 6.4: se x for uma variável global usada por $funcaoAlvo$.

(*top-down*) de troca de informação.

Considerando que $v\_\_lower$ e $v\_\_upper$ são respectivamente os valores mínimo e máximo gerados para abstrair o tamanho do arranjo v, o código das figuras 6.5, 6.6 e 6.7 são usadas para dar uma ideia de como é feita a definição desse vetor, dependendo do seu papel no programa:

É importante lembrar que o *driver* final é criado na forma de texto. Dessa forma, os usuários desse arquivo podem alterar seus valores diretamente no documento. Depois que cada entrada é devidamente inicializada, a função `main` é completada com código

```
int main() {
  int i;
  //...
  for (...) {
    //...
    int size = rand(v__lower, v__upper);
    int *v = (int*) malloc(sizeof(int) * size);
    for (i = 0; i < size; i++) v[i] = rand(0, 100);
    //...
    funcaoAlvo(..., v, ...);
    free(v);
    //...
  }
}
```

Figura 6.5: se v é um parâmetro do procedimento que será executado.

para para realizar chamadas ao método que será executado e mais algumas instruções para liberar memória alocada de forma dinâmica, ex.: “free(v)”.

```
int a, b;
int v() {
  int i;
  int size = rand(a, b);
  int *array = (int*) malloc(sizeof(int) * size);
  for (i = 0; i < size; i++) array[i] = rand(0, 100);
  return array;
}
int main() {
  //...
  for (...) {
    //...
    a = v__lower;
    b = v__upper;
    //...
    funcaoAlvo(...);
    //...
  }
}
```

Figura 6.6: se v for uma função usada por $funcaoAlvo$ (considerando que esse método retorna um valor ponteiro para `int` e não recebe parâmetros).

```
int *v;
int main() {
  int i;
  //...
  for (...) {
    //...
    int size = rand(v__lower, v__upper);
    v = (int*) malloc(sizeof(int) * size);
    for (i = 0; i < size; i++) v[i] = rand(0, 100);
    //...
    funcaoAlvo(..., v, ...);
    free(v);
    //...
  }
}
```

Figura 6.7: se v for uma variável global usada por $funcaoAlvo$.

Capítulo 7

Resultados

A fim de avaliar as ideias apresentadas neste trabalho, foram realizados dois experimentos com `amazonc`. O objetivo do primeiro experimento é checar se a ferramenta é capaz de executar funções que usam arranjos sem gerar dados que levem a acessos indevidos à memória. Já no segundo, feito com `aprof`, tem-se a intenção de verificar se a execução automática gerada é útil para analisar métodos isolados. Visando esses propósitos, `amazonc` foi aplicada sobre todos os programas disponíveis no *Polybench* e sobre os seguintes algoritmos de ordenação: *bubbleSort*, *insertionSort*, *quickSort* e *selectionSort*, mais detalhes no Apêndice C. O *Polybench* contém um conjunto de programas utilizados em álgebra linear, mineração de dados, processamento de imagens, entre outras áreas da matemática. Dessa forma, para realizar os experimentos de `amazonc`, foram utilizados um total de 30 funções núcleo (*kernel*) dos programas encontrados no *Polybench* e mais 5 algoritmos de ordenação que totalizam 35 funções escritas em linguagem C. O principal motivo para a escolha dessa coleção de métodos é que todos eles recebem, como parâmetro, arranjos e inteiros, estes relacionados aos tamanhos daqueles.

Antes que os experimentos com *Polybench* acontecessem de fato, ouve uma etapa de preparação para os testes a serem realizados. A metodologia usada para essa etapa prévia consistiu em:

1. pré-processar os arquivos do *Polybench*;
2. extrair o texto do *kernel* de cada programa em um arquivo diferente;
3. se necessário, `psyche-c` era usado para reconstruir o trecho de código obtido;
4. substituir as dimensões dos arranjos, conhecidas em tempo de compilação, passados como parâmetros, ex. "`T f(T A[1][2])`", por uma versão que usa ponteiros,

Kernel	LoC	Nro. Vet	Maior Dim.
kernel_2mm	86	5	2
kernel_3mm	98	7	2
kernel_adi	102	4	2
kernel_atax	65	4	2
kernel_bicg	73	5	2
kernel_cholesky	56	1	2
kernel_correlation	103	4	2
kernel_covariance	75	3	2
kernel_deriche	130	4	2
kernel_doitgen	64	3	3
kernel_durbin	72	2	1
kernel_fdtd_2d	81	4	2
kernel_floyd_warshall	49	1	2
kernel_gemm	68	3	2
kernel_gemver	103	9	2
kernel_gesummv	77	5	2
kernel_gramschmidt	73	3	2
kernel_heat_3d	68	2	3
kernel_jacobi_1d	54	2	1
kernel_jacobi_2d	56	2	2
kernel_lu	53	1	2
kernel_ludcmp	91	4	2
kernel_mvt	70	5	2
kernel_nussinov	64	2	2
kernel_seidel_2d	50	1	2
kernel_symm	70	3	2
kernel_syr2k	74	3	2
kernel_syrk	63	2	2
kernel_trisolv	55	3	2
kernel_trmm	53	2	2

Tabela 7.1: Programas usados nos experimentos. LoC: tamanho do programa em linhas de código; Nro. Vet: quantidade de arranjos recebidos como parâmetro; Maior Dim.: maior dimensão dentre os arranjos.

ex. "`T f(T **A)`";

5. usar `amazonc` para gerar um *driver* para cada núcleo considerado;

6. compilar cada um dos `drivers` construídos;

Após essa sequencia de passos, foram obtidos 30 programas executáveis do *Polybench*. Para os algoritmos de ordenação, foram necessários seguir apenas os dois últimos passos (usar `amazonc` para gerar um *driver* e compilar o resultado). Cada programa obtido nos passoas anteriores consistem em uma função `main` que realizava uma certa quantidade de chamadas para cada método extraído com dados diferentes.

A tabela 7.1 apresenta um resumo das funções extraídas do *Polybench* que foram usadas nesse experimento. No total, foram analisados 99 arranjos de 30 *kernels* diferentes. Alguns parâmetros foram variados na realização de cada experimento. Detalhes sobre essas adaptações são descritas a seguir. Todos os algoritmos de ordenação usados recebem, como parâmetro de entrada, apenas um arranjo com uma dimensão.

7.1 Experimentos com Valgrind

Os experimentos realizados com o `Valgrind` foram feitos com o intuito de verificar se o *driver* construído por `amazonc` realmente é capaz de executar programas que usam arranjos sem introduzir acessos errôneos a memória. Nesse experimento, um sucesso é contabilizado toda vez que a execução de uma função reconstruída (dentre o total de 35) é realizada sem que `Valgrind` aponte erros causados por acessos à posições de memória não alocadas. Os experimentos foram realizados seguindo os seguintes parâmetros (configuráveis a partir de `amazonc`):

1. cada função foi executada um total de 100 vezes;
2. cada execução foi realizada com dados diferentes;
3. o tamanho máximo a ser alocado para uma dada dimensão de um arranjo foi limitado em 500;

Nesse caso, a ferramenta foi capaz de executar todos os programas sem problemas decorrentes de acesso a posições de memória não alocadas.

Durante a execução de um dos *kernels*, `kernel_lu`, `Valgrind` reportou erros causados devido a divisão por 0. Isso aconteceu porque em algum momento da execução desse núcleo acontece uma divisão por um dos elementos do arranjo. Como mostrado anteriormente, os valores criados para inicializar os elementos dos arranjos são feitos a partir de um número aleatório que esteja no intervalo: $[0, 100]$. Neste trabalho, dados que não estejam relacionados aos tamanhos de arranjos não são considerados. Além disso, não é foco desta dissertação desenvolver um método de criar dados para a execução de funções que estejam livres de divisões por zero.

Por causa desses motivos, foi realizado um segundo teste com `kernel_lu`. Nessa ocasião, foi feita uma simples modificação no trecho de código que gera valores para os elementos do arranjo bi-dimensional `A`. Em vez de gerar qualquer número no intervalo $[0, 100]$ na inicialização desse vetor, esse intervalo foi adaptado para $[1, 100]$. Com essa modificação, o núcleo anterior foi executado novamente 100 vezes e não apresentou nenhum erro, apontado por `Valgrind`, durante a sua execução.

7.2 Experimentos com aprof

Os testes realizados com `aprof` foram feitos para verificar se a habilidade de executar programas de forma automática, e com dados aleatórios, é realmente útil. O experimento consiste em executar as 35 funções várias vezes com dados diferentes e checar se as curvas construídas por `aprof` se aproximam das que seriam geradas pelas funções de complexidade esperadas, quando analisadas visualmente.

O experimento foi realizado limitando o tamanho máximo das dimensões geradas para os arranjos em 500. Além disso, cada função foi executada 2000 vezes. Com essas informações foram coletados e analisados: o gráfico de custo, que associa o tamanho dos dados de entrada ao custo para executar a função; e o gráfico com a curva de complexidade (gerado pela regra "*guess ratio rule*"). As duas visualizações são criadas pela interface gráfica de `aprof`.

Houveram alguns programas, 7 no total, que tiveram seus *drivers* modificados manualmente. A seguir é feita uma lista das modificações:

1. **kernel_adi**:
   a) o intervalo de `tsteps` foi alterado para $[50, 500]$. Isso teve que ser feito porque essa variável não é restringida o suficiente pelo código;
2. **kernel_heat_3d**:
   a) o intervalo de `tsteps` foi alterado para $[50, 500]$. Isso é necessário porque essa variável não é restringida o suficiente pelo código;
   b) o limite máximo para a dimensão dos arranjos foi alterado para 80. Isso foi necessário por causa de problemas do `aprof` com falta de memória;
3. **kernel_doitgen**:
   a) o limite máximo para a dimensão dos arranjos foi alterado para 100. Isso foi necessário por causa de problemas do `aprof` com falta de memória;
4. **kernel_lu**:
   a) alteração no intervalo de valores gerados para inicializar os arranjos. Alteração de $[0, 100]$ para $[1, 100]$. Isso foi necessário por causa da existência de problemas relacionados com a divisão por 0.
5. **kernel_jacobi_1d**:

a) o intervalo de `tsteps` foi alterado para $[50, 500]$. Isso teve que ser feito porque essa variável não é restringida o suficiente pelo código;

6. **kernel_jacobi_2d**:

a) o intervalo de `tsteps` foi alterado para $[50, 500]$. Isso teve que ser feito porque essa variável não é restringida o suficiente pelo código;

7. **kernel_seidel_2d**:

a) o intervalo de `tsteps` foi alterado para $[50, 500]$. Isso teve que ser feito porque essa variável não é restringida o suficiente pelo código;

Considerando as modificações apresentadas anteriormente, foi possível analisar e chegar as complexidades esperadas de todos os programas analisados. As figuras 7.2, 7.4 e 7.6 são exemplos de saídas retornadas por `aprof` para alguns núcleos do *Polybench*. Esses gráficos foram gerados a partir das execuções dos programas reconstruídos por `amazonc` considerando como entrada os códigos das figuras 7.1, 7.3 e 7.5, respectivamente.

As informações retornadas por `aprof` são apresentadas aqui em duas partes. A primeira, gráfico da esquerda, consiste na razão de curvas de complexidade considerada pelo método *Guess Ratio Rule*, ou Regra da Divisão pela Intuição, em tradução livre. Esse método consiste na avaliação da taxa de crescimento de uma função $T(n)$ quando ela é dividida por outra $H(n)$, $T(n)/H(n)$. A função $H(n)$ representa a intuição, ou suposição, do que seria a complexidade teórica esperada do método analisado e $T(n)$ é dado pelos valores da Função de Custo (gráfico da direita) encontrados por `aprof`. Se $T \in O(H)$, então em algum momento a razão deve estabilizar em uma constante não negativa, enquanto que se $T \notin O(H)$ o resultado da operação eventualmente irá crescer.

Já a segunda parte é composta pelo Gráfico de Custos da função analisada (a direita nas imagens das análises). Essa visualização representa a relação existente entre o tamanho da entrada, dado em RMS (*Read Memory Size*), e o custo necessário à sua computação (dado pelo número de blocos básicos executados). Esse gráfico costuma ser útil quando sua curva é uma reta, ou seja, o custo de executar o método analisado cresce proporcionalmente com a carga de dados usada em sua execução. Se este não for o caso, a visualização esquerda é mais útil para analisar os resultados.

```
1  void kernel_trisolv(int n, float **L, float *x, float *b)
2  {
3    int i, j;
4    for (i = 0; i < n; i++)
5    {
6        x[i] = b[i];
7        for (j = 0; j <i; j++)
8          x[i] -= L[i][j] * x[j];
9        x[i] = x[i] / L[i][i];
10   }
11 }
```

Figura 7.1: Núcleo do *Polybench* (*kernel trisolv*).

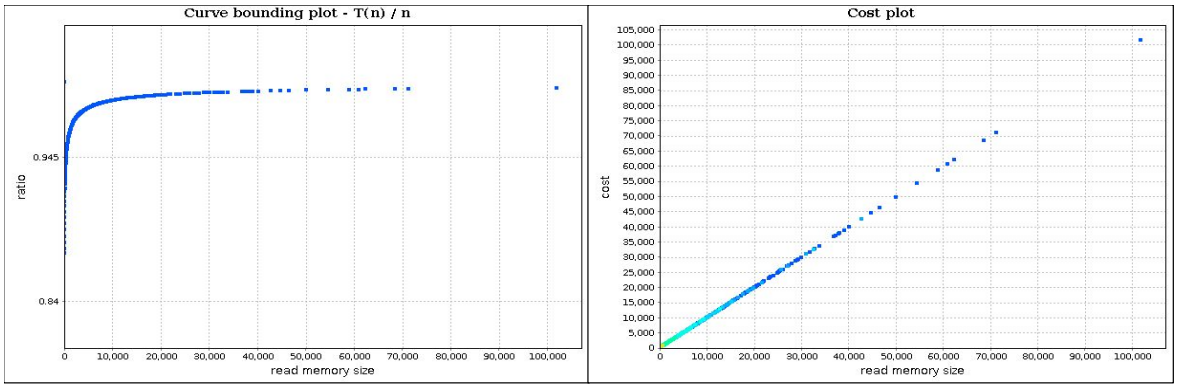

Figura 7.2: Resultado da análise do *kernel trisolv* por `aprof`.

```
1  void kernel_cholesky(int n, float **A)
2  {
3    int i, j, k;
4    for (i = 0; i < n; i++) {
5       for (j = 0; j < i; j++) {
6          for (k = 0; k < j; k++) {
7             A[i][j] -= A[i][k] * A[j][k];
8          }
9          A[i][j] /= A[j][j];
10      }
11      for (k = 0; k < i; k++) {
12         A[i][i] -= A[i][k] * A[i][k];
13      }
14      A[i][i] = sqrt(A[i][i]);
15   }
16 }
```

Figura 7.3: Núcleo do *Polybench* (*kernel cholesky*).

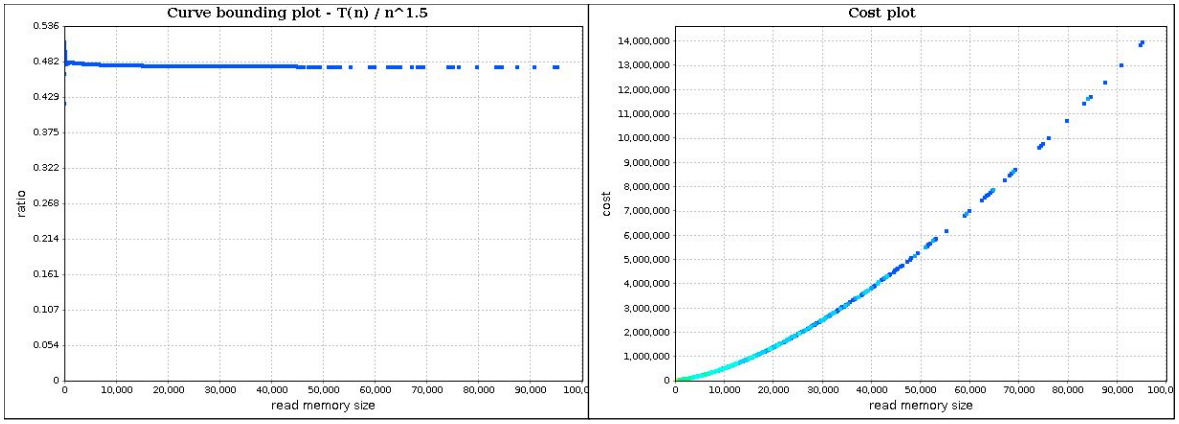

Figura 7.4: Resultado da análise do *kernel cholesky* por `aprof`.

```
1  void kernel_durbin(int n, float *r, float *y)
2  {
3    float z[n];
4    float alpha;
5    float beta;
6    float sum;
7    int i,k;
8    y[0] = -r[0];
9    beta = 1.0;
10   alpha = -r[0];
11   for (k = 1; k < n; k++) {
12     beta = (1-alpha*alpha)*beta;
13     sum = 0.0;
14     for (i=0; i<k; i++) {
15       sum += r[k-i-1]*y[i];
16     }
17     alpha = - (r[k] + sum)/beta;
18     for (i=0; i<k; i++) {
19       z[i] = y[i] + alpha*y[k-i-1];
20     }
21     for (i=0; i<k; i++) {
22       y[i] = z[i];
23     }
24     y[k] = alpha;
25   }
26 }
```

Figura 7.5: Núcleo do *Polybench* (*kernel durbin*).

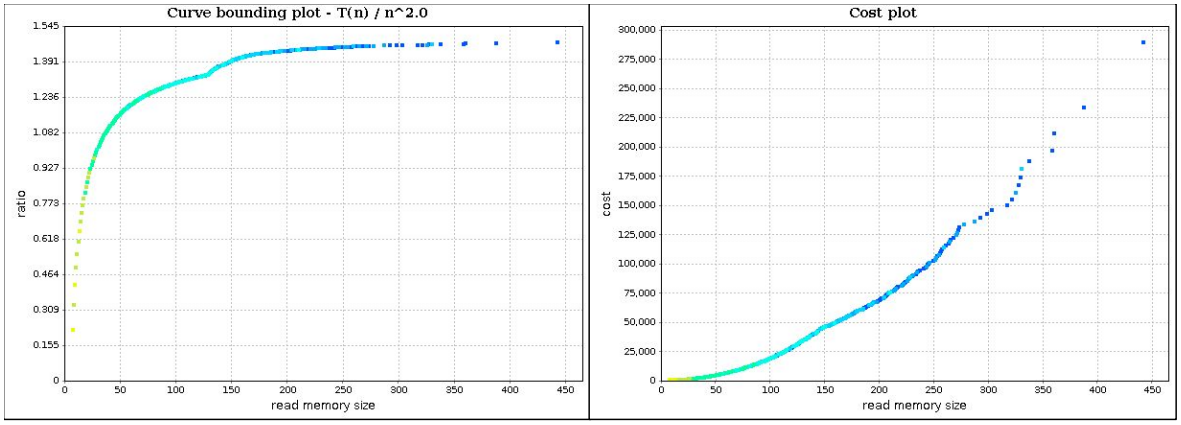

Figura 7.6: Resultado da análise do *kernel durbin* por `aprof`.

Capítulo 8

Considerações Finais

Neste capítulo são dadas as considerações finais a respeito deste trabalho. Inicialmente são apontadas as limitações teóricas e práticas de `amazonc`. Em seguida, são mostradas quais as formas possíveis de expandir e melhorar essa ferramenta. Por fim, é apresentado uma conclusão para esta dissertação.

8.1 Limitações da Ferramenta

Como apresentado no Capítulo 7, `amazonc` foi capaz de executar automaticamente diversas funções não triviais em C. Apesar disso, ela conta com algumas limitações. Uma delas está relacionada a forma como a análise é feita. Para relacionar uma variável de tipo ponteiro com um tamanho é feita uma análise de como essa entidade é utilizada. Neste trabalho, um arranjo é analisado, e considerado como tal, se no texto do programa são encontradas sintaxes de indexação como "`v[E]`", em que `v` tem tipo ponteiro e `E` possui tipo numérico. Isso implica que situações envolvendo outras maneiras de dereferenciar ponteiros, como "`*(v + E)`", não são consideradas aqui. Tal omissão pode fazer com que `amazonc` deixe arranjos escaparem de serem analisados.

Além disso, todo o trabalho é fortemente dependente dos resultados da Análise de Intervalos Simbólica, descrita no Capítulo 4. A definição de limites exatos nesse tipo de análise é um problema indecidível. Tal resultado é conclusão imediata do Teorema de Rice, [Rice [1953]]. Sendo assim, o funcionamento `amazonc` depende diretamente dos resultados retornados por essa análise.

Por fim, neste trabalho são considerados que os códigos recebidos como entrada não possuem estruturas de dados como as criadas com o modificador `struct`. Por esse motivo, a análise apresentada não possui regras para lidar com estruturas de dados

mais complexas. Isso implica também que a ferramenta não é capaz de criar *stubs* para funções que retornem `structs`, ou ponteiros para esse tipo de dado.

8.2 Trabalhos Futuros

Este trabalho pode ser expandido de muitas formas. Uma alternativa é integrar as ideias discutidas aqui com uma, ou mais, técnicas existentes na literatura, Anand et al. [2013], que buscam gerar dados para melhorar *code coverage*. Com isso, seria possível criar um *framework* para realização testes automáticos mais seguro em relação aos dados gerados para executar o código.

Além disso, como mostrado, atualmente são considerados um conjunto restrito de expressões usadas para acessar os arranjos (expressões afins e produtos de símbolos). Outra forma de expandir este trabalho é considerar mais forma de acesso, como divisão de símbolos por exemplo. O principal desafio em expandir o trabalho seguindo essa vertente é a criação de formas de espalhar as informações entre os nós do Grafo de Acesso de forma a garantir as restrições de Intervalos de Acessos.

Outra forma de expandir as ideias discutidas aqui é implementando ou melhorando os pontos apontados na seção de limitações, Seção 8.1. Nesse sentido, uma forma de aprimorar `amazonc` é considerar outras formas de aritmética de ponteiros, diferente da sintaxe de indexação. Atualmente, a análise descrita aqui funciona considerando exclusivamente a forma como os vetores do código fonte são indexados. Além disso, outra limitação apontada na seção 8.1 está relacionada às limitações da Análise de Intervalos Simbólica. Dessa forma, se a análise de intervalos apresentada aqui for melhorada (com regras para coleta de informação em `structs` por exemplo), ou substituída por outra com melhores resultados, a ferramenta também será melhorada.

Por fim, no trabalho de Araujo et al., [de Araujo et al. [2017]], é dado uma ideia de como tipos dependentes poderiam ser usados para melhorar a geração de casos de teste para ferramentas que fazem testes automáticos. Essas ideias poderiam ser incorporadas ao que é descrito aqui para analisar trechos de código de forma inter-procedural (entre funções) e executar métodos considerando mais informações sobre o seu contexto.

8.3 Considerações Finais

Neste trabalho foi apresentado uma forma de executar funções isoladas em linguagem C de forma automática. Foi demonstrado uma abordagem de como tal objetivo pode ser alcançado sem que acessos inválidos a memória sejam feitos. Em especial, nesta

dissertação, é considerado o caso especial em que os programas parciais sejam compostos de funções compiláveis que manipulem arranjos através do uso de indexações. As ideias discutidas aqui foram usadas para a implementação de `amazonc`. Essa ferramenta foi capaz de gerar código que executa todas as funções núcleos do *Polybench* automaticamente sem gerar nenhum acesso inválido a memória.

Referências Bibliográficas

Aho, A. V.; Sethi, R. & Ullman, J. D. (1986). *Compilers, Principles, Techniques*. Addison wesley.

Alves, P.; Gruber, F.; Doerfert, J.; Lamprineas, A.; Grosser, T.; Rastello, F. & Pereira, F. M. Q. (2015). Runtime pointer disambiguation. Em *ACM SIGPLAN Notices*, volume 50, pp. 589--606. ACM.

Anand, S.; Burke, E. K.; Chen, T. Y.; Clark, J.; Cohen, M. B.; Grieskamp, W.; Harman, M.; Harrold, M. J.; Mcminn, P.; Bertolino, A. et al. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978--2001.

Andersen, L. O. (1994). *Program analysis and specialization for the C programming language*. Tese de doutorado, University of Cophenhagen.

Appel, A. W. (2004). *Modern compiler implementation in C*. Cambridge university press.

Best, S. (2003). Analyzing code coverage with gcov. *Linux Magazine*, pp. 43--50.

Bondy, J. A. & Murty, U. S. R. (1976). *Graph theory with applications*, volume 290. Macmillan London.

Carvalho, J. F. N.; Sousa, B. L.; Araújo, M. R. & Bigonha, M. A. S. (2017). The register allocation and instruction scheduling challenge. Em *Simposio Brasileiro de Linguagens de Programacao – SBLP 2017*. SBC.

Cohen, J. S. (2002). *Computer algebra and symbolic computation: elementary algorithms*. Universities Press.

Cohen, J. S. (2003). *Computer algebra and symbolic computation: Mathematical methods*. Universities Press.

Coppa, E.; Demetrescu, C. & Finocchi, I. (2012). Input-sensitive profiling. Em *PLDI*. ACM.

Cousot, P. & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Em *POPL*, pp. 238--252. ACM.

de Araujo, M. R.; Melo, L. T. C. & Pereira, F. M. Q. (2017). Inferência de tipos dependentes em c. Em *Simposio Brasileiro de Linguagens de Programacao – SBLP 2017*. SBC.

Demontiê, F. (2016). Geração de casos de testes para linguagens com aritimética de ponteiros. Dissertação de mestrado, Universidade Federal de Minas Gerais.

Demontiê, F.; Cezar, J.; Bigonha, M.; Campos, F. & Pereira, F. M. Q. (2015). Automatic inference of loop complexity through polynomial interpolation. Em *Brazilian Symposium on Programming Languages*, pp. 1--15. Springer.

Godefroid, P. (2014). Micro execution. Em *ICSE*, pp. 539--549. ACM.

Godefroid, P.; Klarlund, N. & Sen, K. (2005). Dart: directed automated random testing. Em *PLDI*, pp. 213--223. ACM.

Hathhorn, C.; Ellison, C. & Roşu, G. (2015). Defining the undefinedness of c. Em *ACM SIGPLAN Notices*, volume 50, pp. 336--345. ACM.

Heule, S.; Sridharan, M. & Chandra, S. (2015). Mimic: Computing models for opaque code. Em *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 710--720. ACM.

Lattner, C. & Adve, V. (2004a). Llvm: A compilation framework for lifelong program analysis & transformation. Em *CGO*, pp. 75--86. IEEE.

Lattner, C. & Adve, V. S. (2004b). LLVM: A compilation framework for lifelong program analysis & transformation. Em *CGO*, pp. 75–88. IEEE.

Maas, A. J.; Nazaré, H. & Liblit, B. (2016). Array length inference for c library bindings. Em *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pp. 461--471. IEEE.

Melo, L. T. C.; de Araujo, M. R.; Ribeiro, R. G. & Pereira, F. M. Q. (2018). Inference of static semantics for incomplete c programs. Em *POPL*. ACM Press.

Nazaré, H.; Maffra, I.; Santos, W.; Barbosa, L.; Gonnord, L. & Quintão Pereira, F. M. (2014). Validation of memory accesses through symbolic analyses. Em *ACM SIGPLAN Notices*, volume 49, pp. 791--809. ACM.

Nethercote, N. & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. Em *ACM Sigplan notices*, volume 42, pp. 89--100. ACM.

Oehlert, P. (2005). Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58--62.

Paisante, V.; Maalej, M.; Barbosa, L.; Gonnord, L. & Quintão Pereira, F. M. (2016). Symbolic range analysis of pointers. Em *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pp. 171--181. ACM.

Quintao Pereira, F. M.; Rodrigues, R. E. & Sperle Campos, V. H. (2013). A fast and low-overhead technique to secure programs against integer overflows. Em *CGO*, pp. 1--11. IEEE.

Rapps, S. & Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE transactions on software engineering*, (4):367--375.

Ribeiro, R. G.; Melo, L. T. C.; de Araujo, M. R. & Pereira, F. M. Q. (2016). Compilacao parcial de programas escritos em C. Em *SBLP*, pp. 16--31.

Rice, H. G. (1953). Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358--366.

Rodrigues, B.; Quintão Pereira, F. M. & Aranha, D. F. (2016). Sparse representation of implicit flows with applications to side-channel detection. Em *Proceedings of the 25th International Conference on Compiler Construction*, pp. 110--120. ACM.

Serebryany, K.; Bruening, D.; Potapenko, A. & Vyukov, D. (2012). Addresssanitizer: A fast address sanity checker. Em *USENIX Annual Technical Conference*, pp. 309--318.

Yao, C.; wen Wang, Y.; Li, F. & zhan Gong, Y. (2014). A method of function modeling in accurate stub generation. *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, pp. 1022–1026.

Apêndice A

Glossário

amazonc	Ferramenta desenvolvida para executar funções que recebem arranjos como entrada sem gerar acessos inválidos à memória.
Análise Estática	Forma de analisar um programa sem que ele tenha que ser executado.
Análise Dinâmica	Análise que coleta informações sobre um programa com a partir de dados obtidos com a sua execução.
Análise de Intervalos	Análise que busca associar variáveis numéricas de um programa a limites inferiores e superiores.
Análise de Intervalos Simbólica	É um tipo de Análise de Intervalos que considera expressões simbólicas para representar os limites.
Aprof	Ferramenta de análise dinâmica desenvolvida por Coppa et al. [Coppa et al.[2012]] capaz de encontrar ineficiências assintóticas em programas.
Arranjo	Qualquer sequência de células alocadas contiguamente na memória. Usado como sinônimo para vetor.
AST	Árvore de Sintaxe Abstrata (do inglês Abstract Syntax Tree) é uma estrutura de dados que representa o código fonte de um programa.

Código Parcial	Todo trecho de programa, fonte ou binário, que está incompleto de alguma forma. Essa falta implica em não sermos capazes de compilar, analisar ou executar os programas em questão.
DART	DART (Direct Automated Random Testing), [Godefroid et al. [2005]] é uma ferramenta capaz executar testes de unidade de forma automática.
Driver	Método principal usado para inicializar variáveis e organizar as chamadas necessárias aos para execução correta no método alvo.
Expressão Afim	Expressões que possuem o formato aX + b, em que a e b são valores numéricos e X é uma variável ou, no caso deste trabalho, um símbolo do programa.
Expressão Simbólica	Expressão composta por números e símbolos (p.e.: "$2*a+1$").
Fuzzing	Técnica usada para testar programas através do fornecimento de entradas aleatórias.
Grafo de Acessos	Grafo utilizado para representar expressões simbólicas que acessam arranjos e os símbolos que compõem essas expressões.
Grafo de Inicialização	Grafo de dependências usado para assegurar a ordem de definição das entradas do programa.
Library Bindings	Library Bindings é a capacidade de integrar programas implementados em uma linguagem de programação com serviços de outros programas através de uma API bem definida.
LLVM	LLVM (do inglês Low Level Virtual Machine) é um arcabouço de ferramentas usadas para tarefas relacionadas a compilação de programas.
MicroX	MicroX [Godefroid [2014]] é uma ferramenta usada para realização de teste de software através de Micro Execution.

Micro Execution	Micro Execution é a capacidade de executar qualquer trecho de código sem a presença de um driver ou de dados de entrada fornecidos pelo usuário.
Polybench	Conjunto de programas usados para avaliação da performance de programas, disponíveis em: http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/.
psyche-c	Ferramenta desenvolvida por Ribeiro et al. [Ribeiro et al. [2016]] usada para reconstruir programas incompletos na linguagem C ao ponto de torná-los compiláveis.
Símbolo	Identificadores encontrados no programa. Neste trabalho, é dado foco aos símbolos que são entrada de métodos. São consideradas como entradas os símbolos que são lidos antes de terem valores atribuídos a eles.
Stub	Método criado artificialmente a partir da declaração de uma função.
Valgrind	Valgrind é um software livre que auxilia o trabalho de depuração de programas criado por Julian Seward.
Vetor	Qualquer sequência de células alocadas contiguamente na memória. Usado como sinônimo para arranjo.

Apêndice B

Tutorial: Usando amazonc

Neste Apêndice serão dados detalhes de como `amazonc` pode ser obtida, construída e usada pela comunidade. Essa ferramenta se baseia em um programa escrito em C++ (padrão C++11). Além de um compilador para essa linguagem, o usuário deverá ter acesso ao repositório `psyche`, em `http://cuda.dcc.ufmg.br:8080/`, e, preferencialmente, um programa utilizado para construir arquivos *Makefile*, como o `cmake`.

Abaixo, seguem os passos necessários para compilação e execução de `amazonc` em um computador com o sistema operacional Linux (o passo-a-passo foi testado em um ambiente Ubuntu 16.04.4 LTS):

1. primeiro é necessário baixar o repositório `psyche`, disponível em `http://cuda.dcc.ufmg.br:8080/`;
   a) `$ git clone` *url-do-projeto-psyche*
   b) é importante ter as permissões necessárias para acessar o `gitlab` presente no *link* anterior.
2. uma vez que o repositório tenha sido baixado, mude o diretório corrente para o seguinte endereço `pasta-do-repositorio/pcc/src/psychec/`;
   a) `$ cd pasta-do-repositorio/pcc/src/psychec/`
3. use `cmake` para gerar o *Makefile* do projeto;
   a) `$ cmake .`
4. o *Makefile* obtido deve ser usado para compilar e gerar o executável de `amazonc`.
   a) `$ make`

5. na pasta onde foi gerado o programa anterior existe também um *script* em `bash`, `testFile.sh`, que deve ser usado para executar `amazonc`.

6. Para tal, primeiramente, coloque o arquivo a ser testado (código de uma função compilável) dentro da pasta `stubTests` e em seguida use `testFile.sh` para testá-lo.

   a) `$ ./testFile.sh stubTests/foo.c`

7. na pasta `stubTests` irão ficar todos os arquivos gerados no processo. Isso inclui: arquivos `.dot` (com os grafos de acesso e de inicialização gerados), o arquivo principal que faz chamadas para a execução do método alvo e arquivos `.csv` com informações a respeito das entradas geradas para a execução das funções.

8. é possível modificar os *drivers* (arquivos com a função `main` que é responsável por realizar chamadas ao método sendo testado testado) para customizar a forma como serão geradas as entradas para as funções.

Apêndice C

Algoritmos de ordenção usados nos experimentos

Os algoritmos listados nesta seção foram usados nos experimentos deste trabalho. Eles foram tirados do site `http://www.geeksforgeeks.org/sorting-algorithms/` em 30 de Setembro de 2017.

```
1  void bubbleSort(int *arr, int n)
2  {
3      int i, j, temp;
4      for (i = 0; i < n-1; i++)
5          for (j = 0; j < n-i-1; j++)
6              if (arr[j] > arr[j+1]) {
7                  temp = arr[j];
8                  arr[j] = arr[j+1];
9                  arr[j+1] = temp;
10             }
11 }
```

Figura C.1: Bubble Sort

```
1  void heapSort(int *arr, unsigned int N)
2  {
3      if(N==0)
4        return;
5
6      int t;
7      unsigned int n = N, parent = N/2, index, child;
8
9      while (1) {
10         if (parent > 0) {
11             t = arr[--parent];
12         } else {
13             n--;
14             if (n == 0) {
15                 return;
16             }
17             t = arr[n];
18             arr[n] = arr[0];
19         }
20         index = parent;
21         child = index * 2 + 1;
22         while (child < n) {
23             if (child+1 < n && arr[child+1] > arr[child]) {
24                 child++;
25             }
26             if (arr[child] > t) {
27                 arr[index] = arr[child];
28                 index = child;
29                 child = index * 2 + 1;
30             } else {
31                 break;
32             }
33         }
34         arr[index] = t;
35     }
36 }
```

Figura C.2: Heap Sort

```
1  void insertionSort(int *arr, int n)
2  {
3      int i, key, j;
4      for (i = 1; i < n; i++)
5      {
6          key = arr[i];
7          j = i-1;
8
9          while (j >= 0 && arr[j] > key)
10         {
11             arr[j+1] = arr[j];
12             j = j-1;
13         }
14         arr[j+1] = key;
15     }
16 }
```

Figura C.3: Insertion Sort

```
1  void quickSort(int *arr, int low, int high)
2  {
3      if (low < high)
4      {
5          int temp;
6          int pivot = arr[high];
7          int i = (low - 1), j;
8
9          for (j = low; j <= high- 1; j++)
10         {
11             if (arr[j] <= pivot)
12             {
13                 i++;
14                 temp = arr[i];
15                 arr[i] = arr[j];
16                 arr[j] = temp;
17             }
18         }
19
20         temp = arr[i+1];
21         arr[i+1] = arr[high];
22         arr[high] = temp;
23
24         int pi = i + 1;
25         quickSort(arr, low, pi - 1);
26         quickSort(arr, pi + 1, high);
27     }
28 }
```

Figura C.4: Quick Sort

```
1  void selectionSort(int *arr, int n)
2  {
3      int i, j, min_idx, temp;
4
5      for (i = 0; i < n-1; i++)
6      {
7          min_idx = i;
8          for (j = i+1; j < n; j++)
9            if (arr[j] < arr[min_idx])
10             min_idx = j;
11
12         temp = arr[min_idx];
13         arr[min_idx] = arr[i];
14         arr[i] = temp;
15     }
16 }
```

Figura C.5: Selection Sort