UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

GABRIEL ARAÚJO SALDANHA

DESENVOLVIMENTO DE UMA FERRAMENTA DE ANÁLISE ASSINTÓTICA AUTOMATIZADA DE PROGRAMAS

Ouro Preto, MG
2026

GABRIEL ARAÚJO SALDANHA

DESENVOLVIMENTO DE UMA FERRAMENTA DE ANÁLISE ASSINTÓTICA AUTOMATIZADA DE PROGRAMAS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro

Ouro Preto, MG
2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Gabriel Araújo Saldanha

DESENVOLVIMENTO DE UMA FERRAMENTA DE ANÁLISE ASSINTÓTICA AUTOMATIZADA DE PROGRAMAS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 03 de Março de 2026

Membros da banca

Dr. Rodrigo Geraldo Ribeiro - Orientador(a) - Universidade Federal de Ouro Preto
Dr. - Elton Máximo Cardoso - Universidade Federal de Ouro Preto
MSc. Guilherme Augusto Anício Drummond do Nascimento - Universidade Federal de Ouro Preto

Rodrigo Geraldo Ribeiro, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 04/03/2026

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR 3 GRAU**, em 04/03/2026, às 07:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1068898** e o código CRC **B6B6504E**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho aos meus pais Suelândio e Ivanice.

Agradecimentos

Agradeço, primeiramente, aos meus pais, Suelândio e Ivanice, por todo o amor, incentivo e apoio incondicional. Vocês são os pilares fundamentais na construção de quem sou e dos valores que carrego comigo.

Ao meu orientador, que me acompanhou durante grande parte da graduação, minha eterna gratidão pela paciência, pela didática e por me apresentar a esta área da computação que tanto me cativou. Agradeço também pelas conversas dentro e fora do ambiente acadêmico, que tanto enriqueceram minha trajetória.

Aos amigos Heitor, Nader e Thiago, sou grato por estarem ao meu lado nos momentos em que mais precisei de companhia. Obrigado por me inspirarem e me motivarem a ser uma pessoa melhor.

Agradeço ao Vinícius e à pequena Liz, e de modo especial à Luíza Aparecida, pelas conversas e conselhos ao longo deste percurso. A presença e a amizade de vocês foram muito importantes.

Aos amigos André, Guilherme, Hugo, Maria, Rayane, Sabrina e Sara, meu muito obrigado por tornarem esses anos mais leves e por todo o apoio.

Estendo meus agradecimentos aos colegas de curso, em especial Bruno, Felipe, Jéssica, Kézia, Lucas, Matheus, Nicolas, Pedro Henrique, Pedro Morais e Vitor. Sou profundamente grato por tudo que fizeram por mim ao longo da graduação, pela companhia, pelos ensinamentos e por contribuírem tanto para meu crescimento dentro e fora da academia.

“No fim tudo dá certo, e se não deu certo é porque ainda não chegou ao fim.”

Sabino (1998)

Resumo

Esta monografia apresenta o desenvolvimento de uma ferramenta para a automação da análise assintótica de programas, motivada pela complexidade e pela possibilidade de erros inerentes à análise manual de algoritmos. O trabalho fundamenta-se na Ciência da Computação teórica, unindo a Análise de Algoritmos à Verificação Formal com o objetivo de fornecer uma solução confiável e automatizada para o auxílio de profissionais e estudantes. Para atingir esse objetivo, foram explorados os fundamentos da Lógica de Hoare, incluindo conceitos de correção e completude, adaptando-a para a análise de custos computacionais com base nos trabalhos de Silva (2022). A metodologia proposta resultou na implementação de um Gerador de Condições de Verificação, que utiliza o algoritmo de pré-condição mais fraca de custo para traduzir especificações de programas em metas de prova rigorosas. Os resultados demonstram a viabilidade e eficácia da abordagem. A ferramenta, validada através de exemplos práticos, mostrou-se capaz de garantir a correção lógica e validar limites superiores de custo de execução, permitindo a inferência segura da complexidade assintótica a partir de custos absolutos. Assim, o estudo estabelece o núcleo prático de uma ferramenta que integra verificação formal e análise de desempenho, mitigando a necessidade de contagem manual de operações e elevando a precisão das análises.

Palavras-chave: Análise Assintótica. Lógica de Hoare. Automação de Programas. Verificação Formal.

Abstract

This monograph presents the development of a tool for the automation of asymptotic analysis of programs, motivated by the complexity and the possibility of errors inherent in the manual analysis of algorithms. The work is based on theoretical Computer Science, uniting Algorithm Analysis with Formal Verification with the goal of providing a reliable and automated solution to assist professionals and students. To achieve this goal, the foundations of Hoare Logic were explored, including concepts of correctness and completeness, adapting it for the analysis of computational costs based on the work of Silva (2022). The proposed methodology resulted in the implementation of a Verification Condition Generator, which uses the cost weakest precondition algorithm to translate program specifications into rigorous proof goals. The results demonstrate the feasibility and effectiveness of the approach. The tool, validated through practical examples, proved capable of ensuring logical correctness and validating execution cost upper bounds, allowing for the safe inference of asymptotic complexity from absolute costs. Thus, the study establishes the practical core of a tool that integrates formal verification and performance analysis, mitigating the need for manual operation counting and increasing the accuracy of the analyses.

Keywords: Asymptotic Analysis. Hoare Logic. Program Automation. Formal Verification.

Lista de Abreviaturas e Siglas

VC	Condições de Verificação
VCG	Gerador de Condições de Verificação
wlp	Pré-Condição Liberal Mais Fraca
wpc	Pré-Condição Mais Fraca de Custo

Lista de Símbolos

$\wedge$	Conjunção lógica (Operador "e")
$\vee$	Disjunção lógica (Operador "ou")
$\iff$	Equivalência lógica (Operador "se e somente se")
$\implies$	Implicação lógica (Operador "implica")
$\neg$	Negação lógica (Operador "não")
$\vdash$	Sequente (Símbolo de derivação)
$\cup$	União

1 Introdução

A Análise de Algoritmos é uma área fundamental da Ciência da Computação, sendo crucial para o projeto de soluções eficientes. Um tópico relevante é a Análise de Complexidade, que consiste na mensuração dos recursos de tempo e espaço consumidos durante a execução de programas Ziviani (2004). Essa análise permite prever o comportamento de um algoritmo para diferentes volumes de dados de entrada por meio da análise assintótica, que descreve o desempenho do algoritmo quando o tamanho da entrada tende ao infinito.

Em geral, o processo de análise de algoritmos é repetitivo e realizado de maneira predominantemente manual, exigindo um sólido conhecimento teórico do desenvolvedor. A ausência de ferramentas automatizadas e confiáveis para essa finalidade pode acarretar em falhas, resultando na identificação equivocada de gargalos de desempenho ou em comparações imprecisas entre diferentes algoritmos.

Nesse contexto, surge a motivação para este trabalho: desenvolver uma ferramenta capaz de auxiliar desenvolvedores e estudantes na realização de operações de análises assintóticas de algoritmos. O objetivo do sistema é atuar como um verificador formal, capaz de receber uma estimativa de complexidade proposta pelo usuário e atestar, com rigor matemático, se a função informada constitui um limite superior válido e seguro para o tempo de execução do algoritmo em questão.

1.1 Justificativa

A capacidade de prever o comportamento de um programa em relação ao volume de dados processados permite a criação de soluções escaláveis, um requisito fundamental no cenário tecnológico atual. No entanto, a execução manual dessa análise consome tempo considerável e está sujeita a falhas que impactam diretamente a eficiência e a qualidade dos sistemas desenvolvidos.

Este trabalho justifica-se pela busca em preencher essa lacuna, apresentando uma ferramenta automatizada que amplie a precisão, a confiabilidade e a acessibilidade da análise assintótica atuando como um validador formal de limites assintóticos informados pelo usuário. Além de representar um avanço prático para desenvolvedores, tal ferramenta pode desempenhar um papel relevante no ensino de algoritmos, facilitando a compreensão de conceitos teóricos que, por natureza, são abstratos.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver uma ferramenta capaz de validar se uma análise assintótica passada pelo usuário é um limite superior para um determinado trecho de código escrito em uma linguagem imperativa própria.

Como objetivos específicos, destacam-se:

- Investigar os fundamentos teóricos que sustentam a análise assintótica;
- Revisar a literatura e identificar abordagens correlatas para a automação da análise;
- Propor um modelo conceitual para a implementação da ferramenta;
- Desenvolver a ferramenta validadora de limites superiores.

1.3 Organização do Trabalho

Este trabalho está dividido nos seguintes capítulos: o Capítulo 2 apresenta uma revisão bibliográfica detalhada, discutindo e analisando trabalhos relacionados e os conceitos essenciais para a compreensão deste estudo; o Capítulo 3 detalha a metodologia e o desenvolvimento da ferramenta proposta; no Capítulo 4, os resultados encontrados são apresentados e discutidos; e, por fim, o Capítulo 5 traz as conclusões obtidas e trabalhos futuros.

2 Revisão Bibliográfica

O objetivo deste capítulo é apresentar os conceitos necessários para o melhor entendimento desta monografia, sendo constituído pelas seguintes seções: a Seção 2.1 apresenta os principais conceitos abordados nesta monografia; a Seção 2.2 apresenta os trabalhos de outros autores que embasaram o desenvolvimento desta monografia.

2.1 Fundamentação Teórica

Nesta seção, apresentaremos conceitos e definições fundamentais para o para a compreensão do trabalho, onde cada subtópico será direcionada para um tópico específico. Inicialmente, na Seção 2.1.1 apresentado os conceitos referentes a Análise Assintótica; na Seção 2.1.2 apresentaremos a sintaxe da linguagem que utilizaremos para a ferramenta e para os exemplos presentes nesta obra; na Seção 2.1.3 aprofundaremos na Lógica de Hoare; na Seção 2.1.4, será apresentado a definição de Correção e Completude da Lógica de Hoare; por fim, na Seção 2.1.5 apresentaremos uma adaptação feita na estrutura da lógica de Hoare para realizarmos uma contagem de custos de cada operação.

2.1.1 Análise Assintótica

Como projetista, ao desenvolver algoritmos para áreas como otimização, teoria dos grafos, pesquisa operacional, dentre outras, é extremamente importante considerar aspectos de tempo de execução e espaço ocupado.

Conforme Ziviani (2004), em muitas situações podem existir vários algoritmos para resolver o mesmo problema, sendo pois necessário escolher aquele que é o melhor. Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado para resolver o problema em questão.

Para realizar uma análise mais assertiva e justa de um algoritmo, devemos executar o estudo em um ambiente na qual não há o favorecimento pela arquitetura de um compilador ou que os resultados não sejam influenciados pela arquitetura de um *hardware* específico. O intuito é analisar um algoritmo observando apenas suas características mais intrínsecas, sem depender do local onde será executado. Foi com essa intenção que o computador MIX foi proposto na obra *The Art of Computer Programming* Knuth (1997).

O MIX é uma idealização de um computador, feito para medir custos de algoritmos a partir de modelos matemáticos. Por meio dele, examinamos os conjuntos de operações e o custo associado. Por costume, ignoramos os custos de algumas operações com o intuito de analisar

apenas as mais significativas. Por exemplo, em algoritmos de ordenação, o foco recai no número de comparações, desprezando operações aritméticas.

Definimos como função de complexidade, frequentemente denotada como $f(n)$, a operação responsável por analisar algoritmos. Esta função expressa a quantidade de memória ou o número de operações resultante da execução do algoritmo de tamanho n. A expressão resultante para cada algoritmo é derivada da contagem de vezes que um tipo de operação ocorre durante a execução do código.

A seguir, apresentamos um exemplo de análise assintótica para a função fatorial, onde determinaremos a sua complexidade de tempo.

Exemplo: Análise assintótica da função fatorial

```
Function fatorial(n):
  x = 1            // (1) atribuição
  i = 1            // (1) atribuição
  while (i <= n):  // (1) comparação que será realizada (n + 1) vezes
    x = x * i      // (1) multiplicação que será realizada (n) vezes
    i = i + 1      // (1) soma que será realizada (n) vezes
  return x         // (1) retorno
```

As instruções são executadas da seguinte forma:

- 2 atribuições iniciais: **x = 1**, **i = 1**
- O teste do *while* ocorre **n+1** vezes
- As multiplicações e somas ocorrem **n** vezes cada
- 1 instrução de retorno

Portanto, a função executa aproximadamente:

$$f(n) = 2 + (n + 1) * (1) + (n) * (1 + 1) + 1 = 3n + 4$$

2.1.2 A linguagem considerada

Para definirmos de maneira rigorosa a análise assintótica de programas, vamos considerar um fragmento de uma linguagem de programação imperativa que contém construções presentes na maioria das linguagens de programação imperativas atuais.

- Os símbolos **V** representam variáveis arbitrárias. Todas as variáveis possuem escopo global.

- Os símbolos **E** representam expressões ou termos arbitrários.
- Os símbolos **S** representam *statements* arbitrários (operações booleanas).
- Os símbolos **C** representam comandos da nossa linguagem de programação.

Possuímos as seguintes regras para a nossa gramática:

$$
\begin{array}{rcl}
\mathit{prog} & \rightarrow & \text{C} \\
C & \rightarrow & \text{V = E} \\
 & | & \text{C; C} \\
 & | & \textbf{IF S THEN } C_1 \textbf{ ELSE } C_2 \\
 & | & \textbf{WHILE S DO C} \\
E & \rightarrow & V \\
 & | & \text{E + E} \\
 & | & \text{E - E} \\
 & | & \text{E * E} \\
 & | & (E) \\
S & \rightarrow & \text{E < E} \\
 & | & \text{E} \leq \text{E} \\
 & | & \text{E > E} \\
 & | & \text{E} \geq \text{E} \\
 & | & \text{E} \wedge \text{E} \\
 & | & \text{E} \vee \text{E} \\
 & | & \text{E == E} \\
 & | & \text{E} \implies \text{E} \\
 & | & \neg\, \text{E} \\
 & | & (E)
\end{array}
$$

2.1.3 Lógica de Hoare

Segundo Aldrich (2006), o objetivo da lógica de Hoare é fornecer um sistema formal para o raciocínio sobre a correção de um programa. A lógica de Hoare baseia-se na ideia de uma especificação como um contrato entre a implementação de uma função e seus clientes. A especificação é composta por uma pré-condição e uma pós-condição. A pré-condição é um predicado que descreve a condição da qual a função depende para operar corretamente; o cliente deve atender a essa condição. A pós-condição é um predicado que descreve a condição que a função estabelece após ser executada corretamente.

2.1.3.1 Notação

C.A.R. Hoare introduziu a partir do paper *An axiomatic basis for computer programming* Hoare (1969) o conceito denotado como Tripla de Hoare:

$$\{P\}\ C\ \{Q\}$$

Nesta tripla:

- P representa a **pré-condição**, uma afirmação que deve ser verdadeira antes da execução do programa.
- C corresponde ao **programa** (ou comando) de uma linguagem de programação (no nosso caso, a pseudolinguagem definida na Seção 2.1.2).
- Q simboliza a **pós-condição**, uma afirmação que se espera que seja verdadeira após a execução do programa, desde que a pré-condição P fosse satisfeita.

As condições P e Q são expressas utilizando notações matemáticas em conjunto com operadores lógicos, tais como $\wedge$, $\vee$, $\neg$ e $\implies$.

É necessário as seguintes condições para que uma tripla de Hoare seja verdadeira:

1. As condições de P devem satisfazer o estado de C, permitindo-o que execute.
2. Caso a execução de C termine, o estado após sua execução deve satisfazer as condições de Q.

Exemplo: Análise de uma tripla de Hoare válida

Considere a seguinte tripla de Hoare:

$\{X = 1\}\ X = X + 2\ \{X = 3\}$

Neste caso, a tripla é considerada válida. Se iniciarmos a execução em um estado na qual a pré-condição é válida (onde a variável X armazena o valor 1), a operação de atribuição incrementará 2 a este valor. Ao término da instrução, o novo estado do programa garantirá que a pós-condição seja verdadeira, com X assumindo o valor 3.

Exemplo: Análise de uma tripla de Hoare inválida

Considere a seguinte tripla de Hoare:

$\{X = 1 \wedge Y = 2\}\ X = Y;\ Y = X\ \{X = 2 \wedge Y = 1\}$

Neste caso, a tripla é considerada inválida pois o programa falha em realizar a troca dos valores corretamente. Ao iniciar a execução em um estado que satisfaz a pré-condição ($X = 1$ e $Y = 2$), a primeira instrução ($X = Y$) sobrescreve o valor original de X, resultando em um estado intermediário onde ambas as variáveis valem 2 ($X = 2$ e $Y = 2$). Consequentemente, a segunda instrução ($Y = X$) apenas atribui a Y o valor atual de X, que já é 2. Ao término da execução, o estado final do programa é $X = 2$ e $Y = 2$, o que não satisfaz a pós-condição exigida de que Y deveria ser igual a 1.

Contudo, a tripla $\{P\}\ C\ \{Q\}$ é uma especificação de correção parcial, pois nela pode conter *loops* e por este motivo, C pode ser executado, mas não necessariamente ele irá parar e chegar na pós-condição Q.

Um tipo mais forte de especificação é uma especificação de correção total. Iremos representá-la da seguinte forma: $[P]\ C\ [Q]$. Uma especificação de correção total é verdadeira se, e somente se, as duas condições a seguir se aplicarem:

1. Se C for executado em um estado que satisfaça P, então C termina.
2. Após o término, Q é válido.

A correção total é o que nos interessa, mas geralmente é mais fácil prová-la estabelecendo a correção parcial e o término separadamente.

2.1.3.2 Axiomas e Regras

Embora a semântica das triplas de Hoare formalize a corretude de um programa, a verificação prática requer um método dedutivo sistemático para a derivação dessas propriedades. Para isso, a Lógica de Hoare utiliza um sistema dedutivo composto por axiomas e regras de inferência. Esses mecanismos permitem que a verificação de um programa complexo seja decomposta na verificação de seus comandos individuais, seguindo a estrutura sintática da linguagem.

Para realizar a prova formal de uma especificação $\{P\}\ C\ \{Q\}$, utiliza-se um conjunto de regras de inferência e axiomas que operam sobre a estrutura dos comandos. Formalmente, essas regras são apresentadas no formato:

$$\frac{\text{premissas}}{\text{conclusão}}$$

onde, se as premissas forem demonstradas, pode-se concluir a validade da tripla abaixo da linha.

Além desse formalismo, é necessário optar por um modelo de notação para o cálculo das triplas. Dentre as abordagens existentes, destaca-se a notação clássica de Hoare, fundamentada no raciocínio de trás para frente, e a notação de Floyd-Hoare, que acompanha o fluxo natural de execução do programa. Como ambas são equivalentes na capacidade de provar a corretude de

um programa, a escolha da notação torna-se uma decisão de projeto, guiada pela conveniência do método de verificação que se deseja implementar.

Axioma de Atribuição

$$\vdash \{P[E/V]\}\ V = E\ \{P\} \qquad\qquad \vdash \{P\}\ V = E\ \{P[E/V]\}$$

Notação de Hoare *Notação de Floyd-Hoare*

Na notação clássica de Hoare para garantir que uma determinada pós-condição P seja verdadeira após a execução do comando, a pré-condição necessária deve ser a própria fórmula P, substituindo-se todas as ocorrências livres da variável V pela expressão E.

Já na notação de Floyd-Hoare, ao partir de um estado inicial onde a pré-condição P é verdadeira, a execução da instrução $V = E$ modifica diretamente o estado da memória. Consequentemente, a pós-condição reflete esse novo estado, indicando que a variável V agora detém o valor avaliado da expressão E. Desta maneira, o pós-condição é formalmente representada pela expressão $P[E/V]$

Exemplo:

$$\vdash \{P\}\ X = 1\ \{P[1/X]\}$$

Neste caso, a tripla é considerada válida pois o programa executa a atribuição do valor 1 à variável X. Ao iniciar a execução com as propriedades lógicas definidas em P, o comando atualiza o estado da memória. Assim, a pós-condição gerada reflete exatamente o estado original P, com a propriedade recém-adquirida de que a variável X foi substituída pelo valor 1, formalizando com rigor a transição de estado promovida pelo comando.

Fortalecimento de pré-condição

$$\frac{\vdash P \Rightarrow P' \quad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

P é uma condição que implica P'. Se P é verdadeira, P' também deve ser verdadeira. Portanto, se o programa C garante Q quando iniciado em um estado que satisfaz a condição P', ele certamente garantirá Q quando iniciado em um estado que satisfaz a condição P, pois todo estado que satisfaz P também satisfaz P'.

Exemplo:

$$\frac{\vdash X = 2 \Rightarrow X \geq 0 \quad \vdash \{X \geq 0\}\ X = X + 1\ \{X > 0\}}{\vdash \{X = 2\}\ X = X + 1\ \{X > 0\}}$$

Neste caso, sabemos previamente que se X for maior ou igual a zero (P'), incrementaremos X em 1, garantindo que o resultado seja maior que zero (Q). A propriedade $X = 2$ (P) é uma condição muito restrita. Como todo número igual a 2 é, por definição, maior ou igual a zero ($X = 2 \Rightarrow X \geq 0$), podemos utilizar o fortalecimento para deduzir que iniciar o programa com $X = 2$ também é perfeitamente seguro e satisfará a pós-condição $X > 0$.

Enfraquecimento de pós-condição

$$\frac{\vdash \{P\}\ C\ \{Q'\} \quad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

Q é uma condição menos restritiva que Q', ou seja, Q' implica Q. Desta forma, se Q' é verdadeira, Q também deve ser verdadeira. Portanto, se o programa C garante a condição Q' na terminação, ele certamente garantirá a condição Q, pois todo estado que satisfaz Q' também satisfaz Q.

Exemplo:

$$\frac{\vdash \{True\}\ X = 2\ \{X = 2\} \quad \vdash X = 2 \Rightarrow X > 0}{\vdash \{True\}\ X = 2\ \{X > 0\}}$$

Neste caso, a operação de atribuição garante uma pós-condição exata e mais restrita: ao final da execução, independentemente do estado inicial, X será exatamente 2 (Q'). No entanto, para a lógica de um programa maior, pode ser suficiente saber apenas que o valor final é positivo. Como o estado de ser igual a 2 implica logicamente em ser maior que zero ($X = 2 \Rightarrow X > 0$), a regra permite enfraquecer a especificação. Assim, a tripla inferior torna-se validamente provada, garantindo a pós-condição $X > 0$ (Q).

Conjunção de Especificação

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\} \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Essa regra permite combinar duas especificações de correção parcial para o mesmo comando. Caso P_1 resulte em Q_1 e P_2 resulte em Q_2 sendo que ambos executam o comando C, então quando C for executado em um estado que satisfaz $P_1 \wedge P_2$ resultará em um estado que satisfaz $Q_1 \wedge Q_2$.

Exemplo:

$$\frac{\vdash \{X > 0\}\ X = X + 1\ \{X > 1\} \quad \vdash \{Y = 5\}\ X = X + 1\ \{Y = 5\}}{\vdash \{X > 0 \wedge Y = 5\}\ X = X + 1\ \{X > 1 \wedge Y = 5\}}$$

Neste caso, a premissa à esquerda prova uma propriedade sobre a variável X. A premissa à direita prova uma propriedade sobre a variável Y. A regra da conjunção nos permite unir essas duas verdades independentes em uma única tripla válida, demonstrando o comportamento do comando C sobre o estado combinado de X e Y.

Disjunção de Especificação

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\} \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

De forma semelhante à regra de conjunção, a disjunção de especificação permite combinar duas especificações para o mesmo comando. Caso P_1 resulte em Q_1 ou P_2 resulte em Q_2, onde ambos utilizam o comando C, então é equivalente a dizer que quando C for executado em um estado que satisfaça $P_1 \vee P_2$ resultará em um estado que satisfaça $Q_1 \vee Q_2$.

Exemplo:

$$\frac{\vdash \{X = 0\}\ X = X + 1\ \{X = 1\} \quad \vdash \{X = 5\}\ X = X + 1\ \{X = 6\}}{\vdash \{X = 0 \vee X = 5\}\ X = X + 1\ \{X = 1 \vee X = 6\}}$$

Neste caso, o estado exato da variável X antes da instrução é incerto, mas sabemos que ela só pode assumir o valor 0 ou o valor 5. A regra da disjunção avalia os dois cenários separadamente. Como $X = 0$ resulta em $X = 1$ e $X = 5$ resulta em $X = 6$, a conclusão une as duas possibilidades de forma rigorosa. Ao final da execução, o estado de incerteza é propagado corretamente: a variável X valerá 1 ou 6, dependendo de qual condição inicial era verdadeira.

A regra de sequenciamento

$$\frac{\vdash \{P\}\ C_1\ \{Q\} \quad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

Por meio desta regra, podemos derivar uma especificação para uma sequência de comandos. Utilizamos de condições intermediárias para provar a validade de nossos comandos.

Exemplo:

$$\frac{\vdash \{X = 1\}\ X = X + 1\ \{X = 2\} \quad \vdash \{X = 2\}\ Y = X\ \{Y = 2\}}{\vdash \{X = 1\}\ X = X + 1;\ Y = X\ \{Y = 2\}}$$

Neste caso, queremos provar que a sequência de comandos resultará em $Y = 2$. O primeiro comando incrementa X, garantindo a condição intermediária $X = 2$. O segundo comando utiliza essa garantia como ponto de partida para atribuir o valor a Y.

A regra condicional

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\} \quad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \textbf{IF}\ S\ \textbf{THEN}\ C_1\ \textbf{ELSE}\ C_2\ \{Q\}}$$

A regra condicional é usada para provar a especificação de comandos de bifurcação de fluxo. Para que a tripla completa seja válida, é necessário provar que ambos os caminhos de execução convergem para a mesma pós-condição Q. Se a condição S for avaliada como verdadeira, o estado inicial P é fortalecido para $P \wedge S$ e o comando C_1 é executado. Se for falsa, o estado é fortalecido para $P \wedge \neg S$ e o comando C_2 é executado. Como o programa garante Q independentemente do caminho tomado, a estrutura inteira é provada correta.

Exemplo:

$$\frac{\vdash \{True \wedge X < 0\}\ Y = -X\ \{Y \geq 0\} \quad \vdash \{True \wedge \neg(X < 0)\}\ Y = X\ \{Y \geq 0\}}{\vdash \{True\}\ \textbf{IF}\ X < 0\ \textbf{THEN}\ Y = -X\ \textbf{ELSE}\ Y = X\ \{Y \geq 0\}}$$

Este exemplo ilustra o cálculo do valor absoluto de X. A partir de um estado desconhecido ($True$), a ramificação impõe restrições locais. No caminho verdadeiro, sabemos que X é negativo ($X < 0$), logo, invertê-lo ($-X$) resultará em um número positivo para Y. No caminho falso, deduzimos que X já é positivo ou zero ($\neg(X < 0)$), então a atribuição direta também garante que Y será positivo. Como ambos os ramos terminam satisfazendo $Y \geq 0$, a tripla do IF inteiro é validada.

A regra while

$$\frac{\vdash \{I \wedge S\}\ C\ \{I\}}{\vdash \{I\}\ \textbf{WHILE}\ S\ \textbf{DO}\ C\ \{I \wedge \neg S\}}$$

Esta regra é usada para comandos de loop. É necessário um predicado I que é verdadeiro antes do loop começar e permanece verdadeiro após cada execução do corpo do loop C. A regra afirma que, se I é um invariante do corpo do loop C sempre que a condição do teste S for verdadeira, então I também é um invariante do comando *WHILE* completo. Além disso, a regra incorpora o fato de que, quando o loop termina, a condição de teste S deve ser falsa. Portanto, a regra permite concluir que, se o comando *WHILE* termina, a pós-condição $I \wedge \neg S$ é verdadeira.

Exemplo:

$$\frac{\vdash \{X \geq 0 \wedge X > 0\}\ X = X - 1\ \{X \geq 0\}}{\vdash \{X \geq 0\}\ \textbf{WHILE}\ X > 0\ \textbf{DO}\ X = X - 1\ \{X \geq 0 \wedge \neg(X > 0)\}}$$

Neste exemplo, o objetivo é decrementar X até que ele zere. O invariante escolhido é $X \geq 0$, afirmando que X nunca ficará negativo. A premissa prova que se X não é negativo e é maior que zero, subtrair 1 dele manterá o valor maior ou igual a zero, mantendo o invariante verdadeiro. Pela regra do laço, ao término da execução, teremos a certeza de que $X \geq 0$ por meio do invariante e $X \leq 0$ pela negação da condição $X > 0$. Por meio da resolução deste sistema matemático de conjunção é possível determinar que o estado final do programa será $X = 0$.

2.1.4 Correção e Completude

A lógica de Hoare é correta, no sentido de que, se a tripla de Hoare $\{P\}\ C\ \{Q\}$ é derivável no sistema dedutivo, então ela é semanticamente válida: a execução do comando C, em qualquer estado que satisfaça P, não leva a um erro e resulta em um estado que satisfaz Q. Esta propriedade garante que não sejam provadas propriedades falsas sobre programas, fornecendo uma base confiável para raciocínio formal.

Contudo, a completude de um sistema lógico de Hoare refere-se à sua capacidade de derivar todas as triplas semanticamente válidas, ou seja, todas aquelas que são verdadeiras em todas as execuções do programa. Devido às limitações impostas pela indecidibilidade do Problema da Parada Turing (1936), é impossível construir um sistema completamente automatizado que seja ao mesmo tempo correto e completo para todas as linguagens de programação Turing-completas.

Exemplo: Dada a tripla $\{True\}\ C\ \{False\}$, a única maneira dela ser verdadeira seria caso o comando C não terminasse. Contudo, realizar a prova de um problema que entra em looping infinito é indecidível, como demonstrado no Problema da Parada Turing (1936).

Para lidarmos com problemas que possuem esta condição, devemos utilizar do conceito da completude relativa. Por meio dela abordamos as limitações inerentes de decidibilidade e completude da Lógica de Hoare. A ideia central é separar a "lógica de programação"da "lógica de especificação". Qualquer tripla semanticamente válida pode ser derivada no sistema, desde que se tenha acesso a uma teoria de expressões suficientemente poderosa para provar as pré e pós-condições necessárias. Assim, a completude relativa desloca a dificuldade para o raciocínio sobre as expressões aritméticas e booleanas, o que é uma limitação prática, mas não teórica, da lógica de Hoare.

Um conceito chave para essa prova é a pré-condição liberal mais fraca (wlp), que é a pré-condição mais fraca que garante que a pós-condição Q seja verdadeira se o comando C

terminar. O operador $wlp(C, Q)$ constrói uma declaração sintática a partir de um comando e uma pós-condição. Ele tem as seguintes propriedades:

- $\forall P.(\{P\}\ C\ \{Q\}) \Rightarrow (P \Rightarrow wlp(C, Q))$
- $\vdash \{wlp(C, Q)\}\ C\ \{Q\}$

A definição de $wlp(C, Q)$ é feita recursivamente com base na estrutura do comando C:

- Para um comando de atribuição: $wlp((V = E), Q) = Q[E/V]$
- Para uma sequência: $wlp((C1; C2), Q) = wlp(C1, wlp(C2, Q))$
- Para condicionais: $wlp((\textbf{IF}\ S\ \textbf{THEN}\ C1\ \textbf{ELSE}\ C2), Q) = (S \wedge wlp(C1, Q)) \vee (\neg S \wedge wlp(C2, Q))$
- Para comandos de repetição: $wlp((\textbf{WHILE}\ S\ \textbf{DO}\ C), Q) = \bigwedge n.\textbf{iterwlp}\ n\ S\ C\ Q$

Onde o $iterwlp$ é uma função definida recursivamente que calcula a pré-condição para um número *n* de iterações do *loop*.

- Para $n = 0$, a pré-condição é $\neg S \Rightarrow Q$. Isso significa que se o *loop* não executa nenhuma vez, pois a condição S é falsa, então a pós-condição Q deve ser verdadeira.
- Para $n > 0$, a pré-condição é $S \Rightarrow wlp(C, (iterwlp\ (n-1)\ S\ C\ Q))$. Isso indica que, se a condição S é verdadeira, o corpo do *loop* C deve ser executado e a pré-condição resultante para as $n - 1$ iterações restantes deve ser satisfeita.

A noção de completude relativa, fundamentada na utilização da wlp, permite contornar essas limitações, garantindo que todo comportamento correto de um programa possa ser capturado desde que se disponha de uma teoria expressiva o suficiente para manipular as expressões subjacentes.

2.1.5 Custo das expressões

Com base na tese Silva (2022), é possível desenvolver uma adaptação na Lógica de Hoare para estimar o custo total de um algoritmo. Esta extensão não apenas nos permite provar a correção de um programa, mas também quantificar o tempo de sua execução. Ao adicionarmos um componente de custo às triplas de Hoare, transformamos a semântica operacional do programa para que ela não apenas avalie o significado de uma instrução, mas também calcule o custo exato de sua execução. Começamos esta análise formal definindo a semântica para o custo de avaliação de expressões e, em seguida, estendemos esse conceito para as principais regras da lógica.

Nossa semântica não apenas avaliará o significado de um programa, mas também calculará o custo exato de sua execução. Para isso, começamos definindo a semântica para o custo de avaliação de expressões.

Para avaliar o custo de uma expressão, precisamos estabelecer o custo de operações atômicas em nossa linguagem, como ler da memória ou realizar operações aritméticas e lógicas. Para isso, definiremos a constante c_V, para o custo de avaliação de uma variável.

Sendo assim, definimos as seguintes operações:

$$\mathcal{T}[V] = c_V$$
$$\mathcal{T}[Z] = c_Z$$
$$\mathcal{T}[S\ (a)] = c_a$$

$$\mathcal{T}[E_1 + E_2] = \mathcal{T}[E_1] + \mathcal{T}[E_2] + c_+$$
$$\mathcal{T}[E_1 - E_2] = \mathcal{T}[E_1] + \mathcal{T}[E_2] + c_-$$
$$\mathcal{T}[E_1 * E_2] = \mathcal{T}[E_1] + \mathcal{T}[E_2] + c_*$$
$$\mathcal{T}[S_1 \leq S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\leq$$
$$\mathcal{T}[S_1 < S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_<$$
$$\mathcal{T}[S_1 \geq S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\geq$$
$$\mathcal{T}[S_1 > S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_>$$

$$\mathcal{T}[S_1 \wedge S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\wedge$$
$$\mathcal{T}[S_1 \vee S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\vee$$
$$\mathcal{T}[S_1 == S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_{==}$$
$$\mathcal{T}[S_1 \implies S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\implies$$
$$\mathcal{T}[S_1 \iff S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\iff$$
$$\mathcal{T}[S_1 \neg S_2] = \mathcal{T}[S_1] + \mathcal{T}[S_2] + c_\neg$$

Por motivo de simplificação, será considerado todos os custos atômicos como 1.

Para definirmos estas operações no formato de tripla de Hoare, devemos utilizar do formato $\{P\}C\{Q|\mathcal{T}\}$. Sendo assim, podemos representar as seguintes operações:

Regra da Atribuição

$$\{P[\mathcal{T}[E]/V]\}\ V = E\ \{P \mid \mathcal{T}[E] + c_{\text{assign}}\}$$

Regra da Sequência

$$\frac{\{P\}\ C_1\ \{Q \mid \mathcal{T}_1\} \quad \{Q\}\ C_2\ \{R \mid \mathcal{T}_2\}}{\{P\}\ C_1; C_2\ \{R \mid \mathcal{T}_1 + \mathcal{T}_2\}}$$

Regra Condicional

$$\frac{\{P \wedge S\}\ C_1\ \{Q \mid \mathcal{T}_1\} \quad \{P \wedge \neg S\}\ C_2\ \{Q \mid \mathcal{T}_2\}}{\{P\}\ \textbf{IF}\ S\ \textbf{THEN}\ C_1\ \textbf{ELSE}\ C_2\ \{Q \mid max(\mathcal{T}_1, \mathcal{T}_2) + \mathcal{T}[S]\}}$$

Regra do Enfraquecimento

$$\frac{\{P'\}\ C\ \{Q' \mid \mathcal{T}'\} \quad P \implies P' \quad Q \implies Q' \quad \mathcal{T}' \leq \mathcal{T}}{\{P\}\ C\ \{Q \mid \mathcal{T}\}}$$

Regra do Laço de Repetição

$$\frac{\{P \wedge S\} \implies f \leq \mathcal{N} \quad \forall k.\{P \wedge S \wedge f = k\}\ S\ \{P \wedge f > k \mid \mathcal{T}[C] \cdot k\}}{\{P \wedge f \geq 0\}\ \textbf{WHILE}\ S\ \textbf{DO}\ C\ \{P \wedge \neg S \mid \sum_{i=0}^{N-1} \mathcal{T}[C] + (N+1) \cdot \mathcal{T}[S]\}}$$

Na regra do WHILE, as variáveis f, N e $t(k)$ são valores fornecidos por um oráculo. Estas variáveis representam:

- f é uma função de término, que começa como um valor positivo e aumenta a cada iteração, mas permanecer menor que N.
- N é o número máximo de iterações.
- k é responsável por indicar a quantidade de interações que o copo do $while$ irá realizar.

Assim como na lógica de Hoare, a completude do sistema depende de hipóteses externas. No caso da análise de custo, a introdução de um oráculo é necessária para guiar a definição da função de término e da função de custo em laços iterativos. Esse oráculo fornece limites superiores tanto para o número de iterações quanto para o custo associado ao corpo do laço, permitindo que a dedução prossiga de forma consistente.

Portanto, a lógica de custo proposta deve ser entendida como relativamente completa, onde todas as propriedades verdadeiras podem ser demonstradas, desde que as funções de término e de custo sejam adequadamente fornecidas.

Dessa forma, para sequências de código reconhecidas como corretas, torna-se possível correlacionar o tempo estimado de execução de uma função com a notação assintótica *Big O*, possibilitando a realização de uma análise automatizada de complexidade para esse conjunto de algoritmos.

Exemplo: Algoritmo de Divisão

Neste exemplo, demonstramos como a lógica de verificação com custos pode ser aplicada para provar a correção, a terminação e o limite superior de tempo de execução de um programa. Usaremos o algoritmo de divisão para ilustrar este processo.

O objetivo é provar a seguinte asserção de correção total:

$\{r = x \wedge q = 0 \wedge y > 0 \wedge x \geq 0\}\ S\ \{x = r + y \cdot q \wedge r < y \mid 20x + 10\}$

Onde S representa o laço de repetição while

```
while (y <= r) do
    r = r - y
    q = q + 1
```

e $20x + 10$ é o limite de tempo de execução que queremos provar.

Para provar esta asserção, precisamos de informações adicionais, fornecidas por um oráculo. Essas informações são essenciais para a verificação de laços. Sendo assim, definimos as seguintes variáveis:

- A invariante do laço I como $x = q \cdot y + r \wedge y \geq 0 \wedge r \geq 0$.
- A função de variante f como $x - r$.
- O Número máximo de iterações N como x.
- A função de custo do corpo do laço $t(k)$ como 10.

Passo 1: Prova do Corpo do Laço

Primeiro, aplicamos as regras de atribuição e da sequência para provar a correção e o custo do corpo do laço, S_w. Assumindo que o custo de cada operação atômica é 1.

Começamos com a segunda atribuição do corpo do laço: $q = q + 1$. A regra de atribuição exige a substituição da pós-condição ($I \wedge f > k$) pela expressão atribuída ($q + 1$), resultando na seguinte tripla:

$$\vdash \{I \wedge f > k\}\ q = q + 1\ \{(I \wedge f > k)[q + 1/q] \mid T[q + 1] + c_{\text{assign}}\}$$

O custo desta operação é o custo de avaliar a expressão $q + 1$ (que é 3, já que $T[q + 1] = T[q] + T[1] + C_+ = 1 + 1 + 1 = 3$), além da realização da operação de atribuição. Assim, obtemos:

$$\vdash \{(I \wedge f > k)[q + 1/q]\}\ q = q + 1\ \{I \wedge f > k \mid 3 + 1\}$$

Agora, aplicamos o mesmo processo para a primeira atribuição do corpo do laço, $r = r - y$, usando a pré-condição que acabamos de derivar como pós-condição. A nova pós-condição será $(I \wedge f > k)[q + 1/q][r - y/r]$.

O custo desta instrução é o custo de avaliar a expressão $r - y$ (3) mais o custo da atribuição, totalizando 4.

Com a regra de sequência, podemos combinar as duas provas parciais para obter a prova do corpo do laço (S_w), somando os custos (4 + 4 = 8).

$$\vdash \{I \wedge f > k\}\ S_w\ \{(I \wedge f > k)[q + 1/q][r - y/r] \mid 8\}$$

Passo 2: Verificação do Invariante e Variável de Terminação

O próximo passo é verificar se o invariante e a função de variante são mantidos. É preciso provar que a pré-condição do laço antes de uma iteração $(I \wedge y \leq r \wedge f = k)$ implica a pré-condição do corpo do laço que acabamos de derivar.

$$(I \wedge y \leq r \wedge f = k) \rightarrow (I \wedge f > k)[q + 1/q][r - y/r]$$

Além disso, como o custo derivado (8) é menor que o custo que queremos provar (10), podemos usar a regra do enfraquecimento para obter um limite mais relaxado para o corpo do laço, de 10.

$$\vdash \{I \wedge y \leq r \wedge f = k\}\ S_w\ \{I \wedge f > k \mid 10\}$$

Passo 3: Aplicação da Regra do Laço de Repetição

Agora que o corpo do laço foi verificado, aplicamos a regra do Laço de Repetição para provar a asserção completa. Esta regra combina os custos de todas as iterações e a verificação final da condição do laço.

O custo total do laço é a soma do custo do corpo do laço para cada iteração e o custo de avaliar a condição do laço $(y \leq r)$. Como o laço executa no máximo x vezes, a condição é avaliada $x + 1$ vezes (sendo a última vez falsa). Assumindo que o custo de avaliar a expressão booleana $y \leq r$ é 3, o custo total é:

$$\sum_{i=0}^{x-1} 10 + (x + 1) \cdot 3 = 10x + 3x + 3 = 13x + 3$$

Aplicando a regra do Laço de Repetição, obtemos:

$$\vdash \{I \wedge f \geq 0\}\ S\ \{I \wedge \neg(y \leq r) | 13x + 3\}$$

Finalmente, a regra do enfraquecimento é usada novamente para provar que a pré-condição inicial e o custo desejado são válidos. A pré-condição inicial $(r = x \wedge q = 0 \wedge y > 0 \wedge x \geq 0)$ implica a pré-condição do laço $(I \wedge f \geq 0)$. A pós-condição final $(x = r + y \cdot q \wedge r < y)$ é

implicada pelo invariante e a negação da condição do laço. Por fim, o custo provado $(13x + 3)$ é menor que o custo desejado $(20x + 10)$, o que valida a asserção final.

$$20x + 10 \geq 13x + 3$$

Essa prova demonstra que a lógica de verificação de custos é uma ferramenta poderosa para garantir a correção e analisar a complexidade de tempo de execução de programas, transformando um processo intuitivo em uma prova matematicamente rigorosa.

2.2 Trabalhos Relacionados

Esta seção apresenta uma análise crítica dos trabalhos que serviram como base teórica e metodológica para o desenvolvimento desta monografia. O objetivo é posicionar esta pesquisa no contexto acadêmico existente, destacando as contribuições predecessoras e, simultaneamente, explicitando o avanço proposto.

A fundamentação teórica deste trabalho assenta-se em dois pilares principais: 1) os princípios clássicos de verificação formal de programas e 2) as extensões modernas para raciocínio sobre custo computacional.

O artigo *Background reading on Hoare Logic* Gordon (2016) traz conceitos fundamentais para esta monografia. Gordon não apenas relembra os axiomas e regras de inferência da lógica de Hoare, mas também demonstra sua relevância para a verificação formal moderna. Um dos aspectos mais valiosos deste trabalho para nossa pesquisa é a sua exploração do refinamento de programas, um processo que permite derivar sistematicamente uma implementação concreta a partir de uma especificação abstrata, metodologia que inspira a nossa abordagem para a análise de custo.

Para lidar com a complexidade inerente à verificação de programas que manipulam estruturas de dados dinâmicas e ponteiros, este trabalho recorre à Lógica de Separação. Esta formulação foi estabelecida por Reynolds (2002) no seminal *Separation Logic: A Logic for Shared Mutable Data Structures*. A introdução dos operadores de conjunção separadora e de implicação separadora por Reynolds providencia um mecanismo para compor e decompor asserções sobre a *heap*, permitindo um raciocínio local e modular. É importante notar que as ideias fundamentais da Lógica de Separação têm raízes em trabalhos anteriores, principalmente na obra de Burstall (1972), que propôs técnicas pioneiras para provar a correção de programas que alteram estruturas de dados, introduzindo a noção de raciocínio sobre subestados independentes da memória.

O trabalho mais diretamente relacionado à contribuição central desta monografia é a tese de Silva (2022), chamada *Formal Verification of Resource Usage*. Este trabalho é fundamental por propor uma estratégia formal de adaptação da Lógica de Hoare para incorporar a contagem de operações e a derivação de custos relativos de execução de algoritmos. A tese estabelece as

bases para uma análise de complexidade mais precisa e integrada ao processo de verificação formal, que é exatamente o ponto de partida que adotamos.

Embora a tese de Silva (2022) forneça o mecanismo básico para a contagem de passos, ainda possui a lacuna da falta de uma metodologia formal e automatizável para inferir a complexidade assintótica a partir destes custos absolutos. Por esta razão, nosso objetivo não é apenas contar operações, mas construir os formalismos necessários para que uma ferramenta seja capaz de, automaticamente, analisar o custo de um programa em relação ao tamanho da entrada n e derivar a sua classe de complexidade assintótica. O formalismo desenvolvido nesta monografia pretende ser o núcleo de uma ferramenta que, integrando estes contributos históricos, feche o ciclo entre a verificação formal e a análise de desempenho, automatizando a prova de que um programa não apenas está correto, mas também é eficiente.

Por fim, a estruturação prática deste trabalho foi significativamente influenciada pelo tutorial de Sitnikovski (2021), que apresenta uma implementação detalhada da lógica de Hoare utilizando a linguagem funcional Haskell. Esta obra foi fundamental para o delineamento da linguagem imperativa base utilizada no problema, especialmente no desenvolvimento das rotinas de avaliação de expressões aritméticas e booleanas. Além disso, a abordagem de Sitnikovski para representar o cálculo proposicional e a teoria dos números forneceu o modelo metodológico para a representação da lógica de Hoare em nível de valores no Haskell, permitindo que as triplas e regras de inferência fossem codificadas de forma a facilitar a dedução de fatos sobre programas sem a necessidade de execução plena.

3 Desenvolvimento

Este capítulo tem como objetivo principal apresentar a metodologia, os formalismos teóricos e especificar a ferramenta desenvolvida para a análise da complexidade de programas de maneira automatizada. Também será apresentado a implementação do código em Haskell da ferramenta desenvolvida, que está presente no seguinte repositorio do Github: HoareCost.

Partindo dos conceitos fundamentais da Lógica de Hoare, o capítulo detalha na Seção 3.1 a estruturação da linguagem imperativa desenvolvida. A Seção 3.2 descreve o núcleo da ferramenta: o algoritmo de Pré-condição Mais Fraca de Custo (*wpc*), responsável por extrair simultaneamente a corretude lógica e o custo temporal. Para viabilizar a análise de laços, a Seção 3.3 apresenta o mecanismo de Oráculo, que fornece os metadados necessários para superar problemas de indecidibilidade. Na Seção 3.4, demonstra-se a Geração de Condições de Verificação (VCG), que traduz as análises em metas de prova rigorosas submetidas ao provador SMT Z3. Por fim, a Seção 3.5 sintetiza as considerações finais sobre o desenvolvimento realizado.

3.1 Desenvolvimento da Linguagem

Nesta seção, será apresentada a linguagem desenvolvida para a ferramenta proposta.

3.1.1 Sintaxe

A construção da linguagem fundamenta-se em estruturas básicas que estabelecem a base lógica da ferramenta. Por meio destas estruturas, será possível descrever triplas de Hoare para a nossa linguagem.

A seguir, definem-se tais estruturas, seguindo as regras formais descritas previamente neste trabalho.

Os operadores aritméticos são representados pela seguinte estrutura gramatical:

Algoritmo 3.1: Estruturação dos operadores aritméticos

```
data Arith a =
  Var a
  | Z
  | S (Arith a)
  | Plus (Arith a) (Arith a)
  | Minus (Arith a) (Arith a)
  | Mult (Arith a) (Arith a)
```

Os operadores booleanos são definidos pela gramática:

Algoritmo 3.2: Estruturação dos operadores booleanos

```
data PropCalc a =
  PropVar a
  | Not (PropCalc a)
  | And (PropCalc a) (PropCalc a)
  | Or (PropCalc a) (PropCalc a)
  | Imp (PropCalc a) (PropCalc a)
```

Adicionalmente, apresentam-se os componentes da lógica de primeira ordem:

Algoritmo 3.3: Estruturação dos componentes da lógica de primeira ordem

```
data FOL a =
  Eq (Arith a) (Arith a)
  | Lt (Arith a) (Arith a)
  | Gt (Arith a) (Arith a)
  | Le (Arith a) (Arith a)
  | Ge (Arith a) (Arith a)
  | ForAll a (PropCalc (FOL a))
  | Exists a (PropCalc (FOL a))
```

Por fim, os comandos da linguagem são representados da seguinte forma:

Algoritmo 3.4: Estruturação dos comandos da linguagem

```
data Command a =
  CAssign a (Arith a)
  | CSequence (Command a) (Command a)
  | CIfElse (PropCalc (FOL a)) (Command a) (Command a)
  | CWhile String (PropCalc (FOL a)) (Command a)
```

3.2 Análise de custos de pré-condições

A partir da sintaxe e de conceitos definidos em capítulos anteriores, podemos desenvolver uma estrutura responsável por analisar assintoticamente nossos programas de entrada. Para isso, é necessário desenvolver a função responsável por analisar cada comando e atribuir um valor correspondente.

Implementaremos o algoritmo de Pré-Condição Mais Fraca, que recebe um comando C e uma pós-condição Q e retorna uma tupla que contém a pré-condição mais fraca e uma expressão que representa o custo de execução do programa: (wp, t_C).

Deste modo, os comandos da nossa linguagem pode ser representada da seguinte forma:

Algoritmo 3.5: Algoritmo de Cálculo da Pré-condição Mais Fraca de Custo (*wpc*)

```
wpc :: Command String → PropCalc (FOL String) →
IO (PropCalc (FOL String), Arith String)

wpc (CAssign v e) Q = (Q[e/v], (costAExpr e) + (S Z))

wpc (CSequence S1 S2) Q = (wp1, t1 + t2)
  where
    (wp2, t2) = wpc S2 Q
    (wp1, t1) = wpc S1 wp2

wpc (CIfElse cond S1 S2) Q = (wp_if, cost_if)
  where
    (wp1, t1) = wpc S1 Q
    (wp2, t2) = wpc S2 Q
    wp_if     = (cond ⟹ wp1) ∧ (¬ cond ⟹ wp2)
    cost_if   = max(t1, t2) + 𝒯[cond]

wpc (CWhile nome cond S) Q = do
  OracleData inv variant n costFun <- getOracle nome
  let wp = And inv (PropVar (Ge variant Z))

  (_, bodyCost) <- wpc S inv
  let totalCondCost = Mult (Plus (S Z) (costBExpr cond)) (Plus n (S Z))
  let totalBodyCost = Mult bodyCost n
  let cost = Plus totalCondCost totalBodyCost

  return (wp, cost)
```

Um detalhe presente neste trecho está no operador $While$, onde há a necessidade de utilizarmos uma estrutura denominada de Oráculo. Esta estrutura fornecerá informações referente a termos presentes nos laços de repetição. Esta estrutura será melhor detalhada na próxima seção.

3.3 Oráculo

O Oráculo consiste em um conjunto de funções e estruturas auxiliares para os operadores de laço de repetição. O conceito de Oráculo foi definido na tese Silva (2022), que apresenta este operador responsável por informar variáveis de estado, invariantes e funções de custo que não podem ser trivialmente inferidas de forma estática.

No contexto desta ferramenta, o Oráculo resolve o problema da análise de laços (`CWhile`)

ao fornecer o suporte necessário para o cálculo da Pré-condição Mais Fraca e da complexidade temporal. Sem o auxílio dessas informações, a automação da análise seria limitada pela indecidibilidade do problema da parada e pela dificuldade de gerar invariantes válidos automaticamente.

Abaixo, apresenta-se a estrutura de dados que define as informações providas pelo Oráculo:

Algoritmo 3.6: Estruturação dos tipo de dado Oráculo

```
data OracleData = OracleData
    {
        invariant :: PropCalc (FOL String),
        variant :: Arith String,
        numIterations :: Arith String,
        costFunction :: Arith String
    }
```

Para a sua utilização, há a necessidade de utilização de uma interface com o usuário, onde ele informará estes dados. Estes dados passados são repassados para o nosso mecanismo principal, o wpc.

3.4 Geração de Condições de Verificação

Para automatizarmos as aplicações da lógica de Hoare de custo, é necessário um mecanismo que converta as triplas em condições matemáticas que podem ser provadas por um sistema externo, como um provador de teoremas. Este processo é realizado por um Gerador de Condições de Verificação (VCG), que se baseia no algoritmo de pré-condição mais fraca. O objetivo é gerar uma série de metas de prova que, se validadas, garantem que a tripla de Hoare original seja válida.

As principais condições de verificação geradas são:

- $P \rightarrow wp$: A pré-condição inicial do programa deve implicar a pré-condição mais fraca calculada pelo wpc. Se essa condição for verdadeira, a correção parcial do programa é garantida.
- $VC(C, Q)$: Esta função lida com a prova de laços de repetição. Para cada laço while, o VCG exige a prova de que o invariante de laço é mantido, que a função de terminação diminui a cada iteração, garantindo a terminação, e que a pós-condição é satisfeita quando o laço termina.
- $P \rightarrow T \geq t_C$: O custo total do programa, fornecido pela anotação do usuário (T), deve ser maior ou igual ao custo mínimo necessário para executar o programa, derivado do cálculo

da pré-condição mais fraca (t_C). Essa condição garante que a anotação de custo fornecida é um limite superior válido para a execução do programa.

O processo do VCG é semi-automático. Para programas sem laços, a geração das VCs é direta. No entanto, para programas com laços, o usuário deve fornecer informações adicionais que serão capturadas pelo sistema do oráculo. Essas informações, que são difíceis de inferir automaticamente, são essenciais para que o algoritmo gere condições que possam ser provadas. Uma vez que todas as VCs são geradas, elas podem ser enviadas para um provador de teoremas automático, como o EasyCrypt Barbosa et al. (2023) ou o Z3 Moura e Bjørner (2008), para validação.

Ao aplicar este método, o VCG traduz a tripla de Hoare inicial em uma série de condições lógicas, como as discutidas na seção anterior. A prova de cada uma dessas condições garante que o programa não só funcione como esperado, mas também que sua complexidade de tempo de execução esteja dentro do limite superior especificado. Isso transforma o processo de prova, que seria manual e propenso a erros, em um procedimento mecânico e sistemático.

Exemplo:

Para ilustrar o funcionamento do VCG, podemos analisar o algoritmo de divisão já discutido anteriormente.

A tripla de Hoare para o algoritmo é:

$$\{r = x \land q = 0 \land y > 0 \land x \geq 0\}\ C\ \{x = r + y \cdot q \land r < y \mid 20x + 10\}$$

Para aplicar o VCG, a tripla é decomposta em um conjunto de condições de prova. O VCG utiliza do wpc para calcular a pré-condição e o custo do programa.

O VCG gera três tipos principais de condições de verificação:

1. **Condição de Correção:** $P \rightarrow wp$. A pré-condição inicial, P, deve implicar a pré-condição mais fraca, wp, calculada para o programa C e a pós-condição Q. Para o exemplo do algoritmo de divisão, isso se traduz em:

$$(r = x \land q = 0 \land y > 0 \land x \geq 0) \rightarrow (I \land x - r \geq 0)$$

2. **Condição de Custo:** $P \rightarrow T \geq t_S$. Para o algoritmo de divisão, isso resulta em:

$$(r = x \land q = 0 \land y > 0 \land x \geq 0) \rightarrow (20x + 10 \geq 13x + 3)$$

3. **Condições Adicionais para Laços ($VC(S, Q)$):** Como o algoritmo de divisão contém um laço *while*, o VCG gera um conjunto adicional de VCs para garantir a correção e a terminação do laço.

- **Invariante de Laço:** A condição $(I \wedge y \leq r \wedge x - r = k) \rightarrow wp_S$ garante que o invariante de laço (I) é mantido a cada iteração, e a função de variante f aumenta.
- **Terminação e Custo do Corpo:** A condição $(I \wedge y \leq r \wedge x - r = k) \rightarrow t(k) \geq t_S$ verifica se o custo do corpo do laço, $t(k)$, é um limite superior válido para o custo real, t_S.
- **Pós-condição Final:** A condição $(I \wedge \neg(y \leq r)) \rightarrow Q$ garante que o invariante e a negação da condição do laço (*while*) implicam a pós-condição desejada Q.
- **Limite de Iterações:** A condição $(I \wedge y \leq r) \rightarrow x - r \leq x$ assegura que a função de variante está limitada e o laço terminará.

Ao provar cada uma dessas VCs, é possível demonstrar formalmente que o algoritmo de divisão não só está correto, mas também que seu tempo de execução é, no máximo, o especificado $20x + 10$. Este processo transforma uma prova manual e propensa a erros em um procedimento sistemático e mecânico.

$$\text{VG}(\{P\}\ C\ \{Q \mid T\}) = \{P \rightarrow wp\} \cup \{P \rightarrow T \geq t\}\ \cup\ \text{VC}(C, Q)$$

3.4.1 Implementação das condições de verificação (VCs)

Na arquitetura da ferramenta, a geração de VCs ocorre de forma integrada ao cálculo da análise assintótica. Enquanto a função wpc propaga as pré-condições de trás para frente, a função vc identifica as obrigações de prova, especialmente em estruturas onde a WP não pode ser calculada de forma puramente sintática sem auxílio externo, como nos laços de repetição.

Para um comando C, uma pré-condição P e uma pós-condição Q, a condição de verificação fundamental é expressa pela implicação: $P \implies WP\ (C,\ Q)$. A implementação da coleta dessas condições na ferramenta é apresentada a seguir:

Algoritmo 3.7: Estruturação da função de Condições de verificação (VCs)

```
vc :: Command String → PropCalc (FOL String) →
IO [PropCalc (FOL String)]

vc (CAssign _ _) _ = pure []

vc (CSequence c1 c2) q = do
  (wp2, _) <- wpc c2 q
  vcs1 <- vc c1 wp2
  vcs2 <- vc c2 q
  pure (vcs1 ++ vcs2)

vc (CIfElse b c1 c2) q = do
  vcs1 <- vc c1 q
  vcs2 <- vc c2 q
  pure (vcs1 ++ vcs2)

vc (CWhile loopId b c) q = do
  OracleData inv variant n costFun <- getOracle loopId

  let k = Var "k"
  let postBody = And inv (PropVar (Gt variant k))
  (wpS, tS) <- wpc c postBody
  vcsBody <- vc c inv

  let
    vc1 = PropVar
        (ForAll "k" (Imp (And (And inv b)
        (PropVar (Eq variant k))) wpS))
    vc2 = Imp (And inv (Not b)) q
    vc3 = Imp (And inv b) (PropVar (Le variant n))
    vc4 = PropVar
        (ForAll "k" (Imp (And inv b)
        (PropVar (Ge costFun tS))))

  pure ([vc1, vc2, vc3, vc4] ++ vcsBody)
```

A complexidade do comando $CWhile$ destaca-se por exigir dados do Oráculo para fundamentar suas análises. Além dos critérios clássicos de Hoare, a ferramenta introduz verificações de custo ($vc3$ e $vc4$), assegurando que as estimativas temporais sejam consistentes com a execução real do corpo do laço.

Dessa forma, a ferramenta não apenas estima a complexidade temporal, mas também assegura que o comportamento lógico do algoritmo é consistente. Se todas as VCs geradas forem tautologias, o algoritmo é considerado correto em relação às suas especificações.

3.4.2 Implementação do Gerador de Condições de Verificação (VCG)

A última etapa do processo de análise lógica e temporal é consolidada pelo Gerador de Condições de Verificação. Esta função é responsável por unificar a pré-condição inicial fornecida pelo usuário, o cálculo da pré-condição mais fraca e as obrigações de prova coletadas durante a travessia do programa.

O VCG produz um conjunto de fórmulas lógicas que representam a corretude total do algoritmo. A implementação da função vcg é detalhada a seguir:

Algoritmo 3.8: Estruturação do Gerador de Condições de Verificação (VCG)

```
vcg :: PropCalc (FOL String) → Command String →
PropCalc (FOL String) → Arith String → IO [PropCalc (FOL String)]

vcg p s q t = do
  (wp, ts) <- wpc s q
  vcs <- vc s q
  let
    c1 = Imp p wp
    c2 = Imp p (PropVar (Ge t ts))
  pure (c1 : c2 : vcs)
```

Nesta estrutura, as condições $c1$ e $c2$ garantem, respectivamente, que a especificação lógica seja respeitada e que o limite superior de custo informado ou esperado seja consistente com a complexidade extraída do código.

Os dados produzidos pela função vcg consistem em uma lista de fórmulas da Lógica de Primeira Ordem. Para a validação final, essas condições são processadas por meio da biblioteca **SBV** (*Symbolic Boolean Variables*) Erkök (2015), que as traduz e as submete ao provador SMT (*Satisfiability Modulo Theories*) **Z3** Moura e Bjørner (2008). O provador atua verificando a validade de cada implicação; caso todas as VCs sejam provadas satisfatíveis, confirma-se que o algoritmo atende tanto às restrições lógicas de estado quanto aos requisitos de desempenho assintótico definidos pelo Oráculo.

3.5 Conclusão

Neste capítulo, detalhou-se o processo de desenvolvimento da linguagem imperativa e o funcionamento interno da ferramenta de análise assintótica. A estrutura apresentada partiu

da definição de uma sintaxe formal robusta, fundamentada em tipos algébricos de dados, que permitiu a representação não apenas de comandos de fluxo, mas também de propriedades da Lógica de Primeira Ordem.

A implementação do algoritmo de wpc demonstrou ser o núcleo da ferramenta, unificando a verificação de corretude lógica com a extração automática de expressões de complexidade temporal. A introdução da figura do Oráculo revelou-se uma solução estratégica para contornar a indecidibilidade inerente à análise de laços de repetição, permitindo que informações externas (invariantes e variantes) guiem a geração de provas lógicas sem comprometer o rigor formal.

Por fim, VCG consolidou o pipeline de análise ao transformar programas anotados em obrigações de prova matemáticas. A integração com a biblioteca SBV e o provador SMT Z3 automatiza a etapa final de validação, permitindo que o desenvolvedor tenha uma confirmação matemática de que seu algoritmo não apenas funciona conforme o esperado, mas também respeita os limites assintóticos especificados.

4 Resultados

Neste capítulo serão apresentados alguns exemplos utilizados como testes e seus respectivos resultados encontrados sobre sua execução na ferramenta desenvolvida.

Iremos analisar algumas implementações na nossa linguagem e suas respectivas condições geradas pelo VCG.

4.1 Troca

Nosso primeiro exemplo será de um algoritmo de troca de valores entre variáveis. Sua implementação será apresentada a seguir:

Algoritmo 4.1: Troca de variáveis

```
{ ((A == 1 and B == 2) and C == 0) }
    C = A;
    A = B;
    B = C
{ (A == 2 and B == 1) | 6 }
```

Diferentemente de algoritmos iterativos, este programa é composto puramente por comandos sequenciais e de atribuição. Por não conter laços de repetição, a execução dispensa a interação com o componente Oráculo, dependendo inteiramente do cálculo automático da pré-condição mais fraca por meio de substituições sucessivas de baixo para cima.

Para este algoritmo, o VCG produziu duas VCs, as quais foram avaliadas com sucesso pelo provador Z3:

1. **VC #1 (Correção Lógica):** Verifica se a pré-condição declarada satisfaz a pré-condição mais fraca calculada a partir da pós-condição. Ao aplicar a regra de atribuição de trás para frente, substituindo as variáveis (B por C, A por B e C por A), o sistema obteve uma tautologia ($1 == 1 \wedge 2 == 2 \implies 2 == 2 \wedge 1 == 1$). O solver retornou *Q.E.D.*, confirmando que as trocas de valores garantem o estado final exigido.

2. **VC #2 (Validade do Custo Exato):** Verifica se o custo alvo estabelecido na especificação ($T = 6$) é suficiente para cobrir o custo sintático real das operações calculado pela ferramenta. Como a linguagem atribui o custo de 2 unidades para cada operação de atribuição de variável simples (1 para o comando e 1 para avaliar a variável), os três comandos totalizam um custo exato de 6. A ferramenta gerou a prova $6 \geq 6$, retornando *Q.E.D.*.

Este exemplo, embora simples, demonstra o correto funcionamento da ferramenta de verificação conforme a semântica definida.

4.2 Divisão

Neste exemplo será de um algoritmo de divisão por subtrações sucessivas. Sua implementação será apresentada a seguir:

Algoritmo 4.2: Divisão por subtrações sucessivas

```
{ ((R == X and Q == 0) and (Y > 0 and X >= 0)) }
    while "divLoop" (Y <= R) do
        R = R - Y;
        Q = Q + 1
    end
{ (X == R + Y * Q and R < Y) | 20 * X + 10 }
```

Durante a execução foi fornecido os seguintes dados para o oráculo:

- **Invariante (I):** $(X == Q * Y + R) \wedge (Y > 0) \wedge (R \geq 0)$.
- **Variante (f):** $X - R$.
- **Limite de Iterações (N):** X.
- **Função de Custo do Corpo (k):** 8.

Para este algoritmo, o VCG produziu seis VCs, as quais foram avaliadas com sucesso pelo provador Z3:

1. **VC #1 (Segurança Lógica Inicial):** Verifica se a pré-condição implica o invariante antes da primeira iteração. O resultado foi *Q.E.D.*, confirmando que as inicializações $R = X$ e $Q = 0$ satisfazem o invariante.
2. **VC #2 (Validade do Custo Global):** Verifica se o custo alvo $20X + 10$ é suficiente para cobrir o custo calculado pelo wpc ($12X + 8$ aproximadamente). O resultado foi *Q.E.D.*, validando o limite superior de recursos.
3. **VC #3 (Preservação e Progressão):** Garante que, para cada iteração, o invariante é mantido e o variante $X - R$ cresce estritamente. Com o invariante estabelecendo $Y > 0$, o solver provou (*Q.E.D.*) que $X - (R - Y) > X - R$.
4. **VC #4 (Correção Pós-Loop):** Verifica se a saída do loop ($I \wedge \neg(Y \leq R)$) implica a pós-condição lógica. O Z3 confirmou que $R < Y$ satisfaz a especificação de correção da divisão.

5. **VC #5 (Limitação do Variante):** Verifica se o variante está contido no limite de iterações N. O resultado foi *Q.E.D.*.
6. **VC #6 (Validade do Custo do Oráculo):** Verifica se o custo fornecido pelo especialista para o corpo do loop (8) cobre o custo real das operações de atribuição e aritmética. O resultado foi *Q.E.D.*.

A execução demonstrou que a ferramenta é capaz de integrar com sucesso os dados passados pelo oráculo com a prova automática de limites de recursos, garantindo que o algoritmo de divisão opera dentro dos limites de tempo e correção lógica especificados.

4.3 Somatório

Este terceiro exemplo implementa um somatório utilizando uma estrutura condicional presente dentro de um laço de repetição.

Algoritmo 4.3: Somatório dos N primeiros números

```
{ ((N >= 0 and I == 1) and T == 0) }
    while "sumLoop" (I <= N) do
        if (I <= N) then
            T = T + I;
            I = I + 1
        end
        else
            skip
        end
    end
{ T <= N * N | 20 * N + 15 }
```

A análise deste algoritmo é fundamental para demonstrar a capacidade da ferramenta de lidar com o cálculo do custo em estruturas que geram bifurcações no fluxo de controle.

Durante a execução foi fornecido os seguintes dados para o oráculo:

- **Invariante (I):** $(T \leq (I-1) * N) \wedge (I \geq 1) \wedge (I \leq N+1)$.
- **Variante (f):** I.
- **Limite de Iterações (N):** N.
- **Função de Custo do Corpo (k):** 12.

Para este algoritmo, o VCG produziu seis VCs, as quais foram avaliadas com sucesso pelo provador Z3:

1. **VC #1 (Segurança Lógica Inicial):** Verifica se o estado de inicialização ($N \geq 0 \wedge I == 1 \wedge T == 0$) satisfaz o invariante estipulado. Como $0 \leq (1 - 1) * N$, a proposição é trivialmente verdadeira, e o provador retornou *Q.E.D.*.

2. **VC #2 (Validade do Custo Global):** Confirma se o limite fornecido na especificação ($20N + 15$) cobre o custo real acumulado da execução calculado pelo wpc. O retorno foi *Q.E.D.*.

3. **VC #3 (Preservação e Progressão):** Comprova que o corpo do laço (incluindo o desvio condicional) mantém a validade do invariante de laço na transição de estados e que a variável de variante I progride estritamente ($I_{novo} > I_{antigo}$). A prova obteve *Q.E.D.*.

4. **VC #4 (Correção Pós-Loop):** O solver infere se a terminação do laço (ou seja, quando $I > N$) combinada com o invariante implica logicamente a pós-condição $T \leq N \times N$. A prova foi validada com *Q.E.D.*, garantindo que o algoritmo calcula um valor que satisfaz sua especificação matemática de limite.

5. **VC #5 (Limitação do Variante):** Certifica que o crescimento da variável de controle do variante está confinado ao limite de iterações do laço (N), resultando em *Q.E.D.*.

6. **VC #6 (Validade do Custo do Oráculo):** Verifica se a estimativa estática (12) declarada pelo especialista abstrai de forma segura o custo real do bloco interno, que contém o comando $CIfElse$. O resultado foi *Q.E.D.*.

A obtenção de validade de todas as condições demonstra a robustez da ferramenta.

5 Considerações Finais

Neste capítulo, na Seção 5.1 será feito um resumo geral do trabalho evidenciando os resultados e objetivos alcançados. Já na Seção 5.2 definirá os caminhos a serem traçados para a continuidade da pesquisa.

5.1 Conclusão

O objetivo geral deste trabalho foi estudar os formalismos e realizar a construção de uma ferramenta de análise assintótica automatizada de programas. Para atingir esse objetivo, os fundamentos teóricos da Lógica de Hoare, os conceitos de análise assintótica, e os trabalhos relacionados foram investigados. E a partir de uma metodologia baseada na Lógica de Hoare de custo culminou na criação de um modelo conceitual para um Gerador de Condições de Verificação.

A automação do processo consolidou o VCG proposto e desenvolvido. A ferramenta demonstrou-se plenamente funcional ao traduzir as especificações de programas em um conjunto de obrigações de prova estritas, que foram submetidas e validadas com sucesso pelo provador de teoremas Z3, por meio da biblioteca SBV.

Os resultados obtidos na bateria de testes atestaram a viabilidade da ferramenta. O sistema foi capaz de verificar e provar limites superiores de custo para algoritmos produzidos na linguagem proposta. Ao obter a resposta de validade lógica em todas as condições geradas, o trabalho atingiu seu objetivo principal: integrar com êxito a verificação formal de corretude funcional com a análise quantitativa de desempenho. Dessa forma, o estudo estabelece um núcleo teórico e prático confiável, superando a necessidade de contagem manual de operações e elevando a precisão da análise de algoritmos.

5.2 Trabalhos Futuros

Embora a ferramenta desenvolvida tenha comprovado a viabilidade da verificação simultânea de custo e corretude, o campo da análise formal oferece amplas oportunidades de aprimoramento. Algumas das frentes possíveis para trabalhos futuros:

- **Expansão da Linguagem e Estruturas de Dados**: A linguagem imperativa base utilizada neste estudo pode ser estendida para suportar chamadas de funções, recursão e tipos de dados compostos, como arranjos (arrays) e ponteiros.

- **Uso de notação assintótica ao invés de custos exatos**: Atualmente, a ferramenta exige que o usuário forneça um polinômio limitante com constantes explícitas (como $20X + 10$) para a verificação do custo. Uma evolução natural para a ferramenta seria a incorporação de um mecanismo algébrico capaz de abstrair automaticamente os coeficientes multiplicativos e os termos de menor grau. Isso permitiria que a especificação e a validação fossem realizadas diretamente por meio da notação Big-O (como $O(N)$ ou $O(N^2)$), consolidando o propósito da análise assintótica.

- **Desenvolvimento de Interface Gráfica e Integração**: Para cumprir o objetivo de auxiliar no ensino e no desenvolvimento prático, a criação de uma interface amigável (como um plugin para IDEs ou uma aplicação web) facilitaria a interação do usuário com a ferramenta desenvolvida, abstraindo a necessidade de escrever fórmulas em Lógica de Primeira Ordem diretamente no código-fonte.

Referências

ALDRICH, J. *Lecture Notes: Hoare Logic*. 2006. Carnegie Mellon University, Course 15-654/15-854.

BARBOSA, M.; BARTHE, G.; GRéGOIRE, B.; KOUTSOS, A.; STRUB, P.-Y. Mechanized proofs of adversarial complexity and application to universal composability. *ACM Trans. Priv. Secur.*, Association for Computing Machinery, New York, NY, USA, v. 26, n. 3, jul. 2023. ISSN 2471-2566. Disponível em: <https://doi.org/10.1145/3589962>.

BURSTALL, R. M. Some techniques for proving correctness of programs which alter data structures. In: MELTZER, B.; MICHIE, D. (Ed.). *Machine Intelligence 7*. Edinburgh, Scotland: Edinburgh University Press, 1972. p. 23–50.

ERKöK, L. *SBV: SMT Based Verification in Haskell (Version 5.0)*. 2015. <https://hackage.haskell.org/package/sbv-5.0/docs/Data-SBV.html>. Accessed: 2026-02-05.

GORDON, M. *Background reading on Hoare Logic*. [S.l.]: University of Cambridge, 2016. Learning Guide for the CST Part II course.

HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, v. 12, p. 576–583, 1969.

KNUTH, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Boston, MA, USA: Addison-Wesley, 1997. ISBN 9780201896831.

MOURA, L. de; BJøRNER, N. Z3: an efficient smt solver. In: . [S.l.: s.n.], 2008. v. 4963, p. 337–340. ISBN 978-3-540-78799-0.

REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 2002. p. 55–74.

SABINO, F. *No fim dá certo*. Rio de Janeiro: Editora Record, 1998.

SILVA, A. C. F. d. *Formal Verification of Resource Usage*. Dissertação (MSc Thesis) — Universidade do Porto, Faculdade de Ciências, 2022.

SITNIKOVSKI, B. *Tutorial implementation of Hoare logic in Haskell*. 2021. Disponível em: <https://arxiv.org/abs/2101.11320>.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, Oxford University Press, v. 2, n. 1, p. 230–265, 1936.

ZIVIANI, N. *Projeto de Algoritmos: Com Implementações em Pascal e C*. 2ª. ed. [S.l.]: Cengage Learning, 2004. ISBN 8522103909.