UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Heitor de Paula Santos Damasceno

AnyEater: uma Ferramenta de Código Aberto Para Analisar o Uso do Tipo Genérico "any" em Projetos Typescript

Belo Horizonte
2024

Heitor de Paula Santos Damasceno

AnyEater: uma Ferramenta de Código Aberto Para Analisar o Uso do Tipo Genérico "any" em Projetos Typescript

Versão Final

Relatório Final Técnico da matéria de Projeto Orientado a Computação II, do curso de Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: André Hora

Belo Horizonte
2024

Resumo

O Typescript tem ganhado cada vez mais espaço no mercado, entre outros motivos, por ser o sucessor da linguagem Javascript que já era muito popular e oferecer ferramentas para promover a programação segura e a legibilidade de código, como a definição estática de tipos. Apesar disso, a linguagem também oferece um tipo genérico, o "any", que desativa as medidas de segurança do compilador e aparentemente contradiz os princípios do Typescript. Segundo os criadores da linguagem, esse tipo genérico existe como um tipo temporário a ser usado no processo de conversão de código Javascript para Typescript, mas ele acaba sendo usado como solução permanente em alguns projetos, o que pode trazer consequências inesperadas. Na etapa anterior, buscamos explorar com que frequência o uso do "any" ocorria e por que ele ocorria. Nessa etapa, desenvolvemos uma ferramenta que detecta o uso do "any" em código Typescript, aponta o lugar exato onde ele ocorre e apresenta outras informações úteis para a sua substituição. A ferramenta está disponível publicamente em: `https://github.com/HEITORPS123/anyeater`.

Palavras-chave: Computação, Engenharia de Software, Typescript

Abstract

Typescript has been gaining more and more space in the market, among other reasons, because it is a successor to the Javascript language, which was already very popular and because it offers tools to promote safe programming and code readability, such as static type definition. Despite this, the language also offers a generic type, “any”, which disables the compiler’s security measures and contradicts the principles of Typescript. According to the language’s creators, this generic type exists as a temporary type to be used in the process of converting Javascript code to Typescript, but it ends up being used as a permanent solution in some projects, which can have unexpected consequences. In the previous step, we sought to explore how frequently the use of “any” occurred and why it occurred. At this stage, we developed a tool that detects the use of “any” in Typescript code, points out the exact place where it occurs and presents other useful information for its replacement. The tool is publicly available at: `https://github.com/HEITORPS123/anyeater`.

Keywords: Computing, Software Engineering, Typescript

Lista de Figuras

Sumário

*

1. Introdução

A linguagem de programação Javascript foi criada para uso em navegadores e códigos de aplicações web, tendo como uma de suas características principais o fato de ser uma linguagem dinamicamente tipada, ou seja, as variáveis não tem tipo definido e seu tipo pode variar conforme a execução do programa. Já o Typescript, criado pela Microsoft, tinha o objetivo de ser um Javascript mais seguro em relação aos tipos de suas variáveis, sendo uma linguagem "fortemente tipada", onde todas as veriáveis devem ter seu tipo definido antes da execução.

Isso contribuiu para o aumento de sua adoção entre os programadores nos últimos anos, já que aumentou a demanda por código mais limpo e compreensível, com a evolução dos paradigmas de engenharia de software e a disseminação de padrões de projeto nas aplicações. A definição explícita de tipos no geral faz com que trechos de código sejam mais previsíveis e fáceis de se compreender à primeira vista.

Dessa maneira, a linguagem de programação Typescript vem ganhado cada vez mais espaço no mercado como sucessor do Javascript, e uma forma de maior qualidade de utilizar uma linguagem dinâmica. Em uma pesquisa ampla que envolveu quase 30.000 desenvolvedores, a empresa Jetbrains [1] revelou que em 2022, o Typescript foi a linguagem que mais cresceu em uso, tendo seu uso triplicado nos últimos 6 anos e ultrapassando C, C++ e PHP entre outras.

Apesar disso, o Typescript ainda possui opções que podem ser utilizadas para desativar a checagem de tipos do compilador, aparentemente contrariando os princípios que levaram à sua criação. Uma delas, é o tipo genérico "any", cujo uso é recomendado apenas durante o processo de conversão de código Javascript já existente para código Typescript [2], e considerado uma má prática de programação em outros casos.

A primeira parte do trabalho [3] buscou minerar repositórios de código aberto cuja linguagem de programação principal é o Typescript, construir uma base de dados robusta de projetos de código Typescript e documentar o uso do tipo genérico "any" nos projetos minerados. Ao fim dela, foi possível determinar que o tipo genérico "any"era frequentemente empregado por desenvolvedores, tendo 100% de presença nas amostras, aparecendo mais de 10 vezes em 70% dos repositórios e mais de 100 vezes em 18% dos repositórios. Também foi possível mapear os casos de uso mais frequentes deste tipo, sendo eles como parâmetro de função e conversão de tipo de variáveis.

1.1 Objetivos Específicos

Nesta segunda parte, o objetivo específico será utilizar os conhecimentos previamente adquiridos para construir uma ferramenta de código aberto que consiga analisar um repositório de código Typescript e gerar relatórios referentes ao uso de tipos genéricos que identifiquem o local exato e a função sintática de cada "any". Os objetivos gerais dessa etapa serão propor uma maneira prática de identificar e eliminar instâncias de uso de tipos genéricos em projetos e avaliar a eficácia das medidas propostas em casos práticos com o desenvolvimento e aplicação

da ferramenta de código aberto.

2. Referencial Teórico

A tipagem estática, oferecida pelo Typescript, significa que quando variáveis são declaradas, o seu tipo deve também ser declarado, podendo ser por exemplo “number”, “string” ou “boolean”. Ela é diferente da tipagem dinâmica do Javascript, onde o tipo das variáveis não é declarada mas inferida pelo interpretador.

As características específicas do Typescript e os benefícios que provém de seu uso em comparação com o Javascript foram explorados em [4], onde se estudou o impacto da tipagem especificamente na usabilidade de APIs, e concluiu-se que a tipagem estática contribui positivamente nesse quesito. Outro trabalho que abordou o tema foi [5], que fez um estudo extensivo de 17 linguagens de programação estáticas e dinâmicas, entre elas Javascript e Typescript, e concluiu que a tipagem estática auxilia programadores em algumas tarefas importantes no processo de desenvolvimento, como o processo de apurar *bugs*.

Como explicitado anteriormente, há estudos sobre os benefícios da utilização da tipagem estática. Contudo, no que se refere ao uso de Typescript em conjunto com tipos genéricos, mais especificamente o tipo any”, que desabilita a checagem de tipos do compilador e é considerado uma má prática de programação pelos próprios desenvolvedores da linguagem [2], existem poucos trabalhos publicados.

A influência do uso do tipo genérico “any” na qualidade do código foi notoriamente abordada em [6], onde estudou-se a correlação de sua presença com o número de code smells por linhas de código. Nessa linha de pensamento, a etapa anterior deste trabalho explorou estritamente o Typescript e o uso do “any” em repositórios de código aberto, e essa última etapa busca propor uma ferramenta para análise de código que possa ajudar usuários a diminuir esse uso, fornecendo uma análise da função sintática que o tipo genérico ocupa, como listado no apêndice A 2.

No que se refere ao uso de ferramentas de software para analisar código e gerar relatórios que podem ser usados para tomar decisões com o intuito de melhorá-lo, o programa Lizard tem notoriedade entre programadores. Com mais de 1800 estrelas no github [7], é um analisador de complexidade ciclomática de códigos C, C++ e python, que oferece também diversas features e um arcabouço robusto para apoiar o processo de refatoração de código. Procuramos aqui propor algo parecido em funcionalidade, mas focando na análise do uso de tipos genéricos.

3. Metodologia

Esse trabalho procurou desenvolver uma ferramenta capaz de analisar repositórios de código Typescript e elaborar relatórios sobre o uso do tipo genérico “any”, exibindo um panorama estatístico do uso desse tipo genérico no projeto, fornecendo a localização deles e a função sintática que ocupam. A seguir estão delimitados os passos que foram seguidos para cumprir o objetivo do projeto.

3.1 Estudo das Soluções de Mercado

Primeiramente, foi realizado um processo amplo de revisão bibliográfica com o intuito de entender quais são as técnicas mais populares e efetivas para a solução de problemas similares, e guiar a implementação do projeto. A tipagem estática em um código se refere a uma questão sintática, ou seja, os tipos são explicitados no código e resolvidos durante a transformação deste em sua árvore de sintaxe abstrata. Dessa forma, adotou-se a técnica de análise de árvore sintática para a extração das informações buscadas.

Aliado a isso, foi feito o estudo de outros programas de código aberto que apresentam soluções similares ao que foi aqui proposto. O exemplo mais pertinente hoje é o analizador de código Lizard [7], que utiliza uma interface de linha de comando para fazer a interação com o usuário, aliado a técnicas de análise síntatica de linguagens como a construção e análize da árvore de sintaxe abstrata do código. Outra ferramenta que foi estudada foi o JDeodorant, um add-on para Eclipse que identifica code-smells em código Java.

O Lizard é apresentado primariamente como uma analizador de complexidade ciclomática, mas também possui features que o permitem identificar duplicação de código e realizar outras formas de análise estática de código. Com 1800 estrelas no GitHub, o projeto busca valorizar a simplicidade, escolhendo uma interface de linha de comando simples, que imita os comandos conhecidos do Unix com a possibilidade de customização com flags. A análise de código do Lizard é feita através da interpretação da árvore de sintaxe abstrata do código.

O JDeodorant é descrito como um ”detector”de code-smells [8], que são problemas de design convencionados por diversos autores no campo de engenharia de software. Ele é distribuído como plugin da ide Eclipse, e atualmente conta com mais de 50 milhões de downloads.

3.2 Levantamento de Requisitos

Após a etapa anterior, houve o levantamento de requisitos do projeto, onde foram decididas quais seriam as features a ser implementadas nele, com base no que seria útil para os potenciais usuários. Foram valorizadas opções comuns em interfaces de linha de comando como Lizard, junto de algumas opções mais específicas quanto a aplicação do software.

3.3 Desenvolvimento

O código foi desenvolvido com a linguagem Typescript, valorizando o detalhamento dos tipos utilizados, que ficam como uma espécie de "documentação" para possíveis forks. Ele está hospedado em um repositório aberto ao público no GitHub, e utiliza arquitetura limpa como base, para garantir boa legibilidade e manutenebilidade.

A parte principal do projeto pode ser determinada como o algoritmo que utiliza a API de compilador do Typescript [9] para analizar arquivos de código que recebe como input e determinar a especificação de tipos genéricos nestes, assim como o seu papel na lógica do código. O pseudocódigo deste algoritmo está detalhado na listagem 1.

```
1  // Define a funcao recursiva com a logica principal
2  generateAst(node: ts.Node, ...) {
3      // Recebe o pai do no atual
4      var parentNode = node.parent
5      // Se o tipo sintatico for 'any'
6      if (node.syntaxKind == 'anyKeyword') {
7          // Pega a linha e caractere onde o 'any' aparece
8          let { line, character } =  sourceFile.getLineAndCharacterOfPosition
               (node.getStart())
9          // Pega o tipo do pai do no
10         var parentType = ts.SyntaxKind[parentNode?.kind]
11         // Verifica se o 'any' faz parte de um tipo composto
12         // Se sim, sobe mais um nivel na arvore sintatica
13         // ate achar o real uso daquele 'any'.
14         // Por ex: Parametro de funcao ou conversao de tipo
15         while (parentType == 'ArrayType' || parentType == 'UnionType') {
16             currentNode = currentNode?.parent
17             parentType = ts.SyntaxKind[currentNode.kind]
18         }
19         //
20         Salva as informacoes encontradas
21     }
22     Chama generateAst para o proximo  n
23 }
```

Listing 1: Algoritmo principal simplificado

3.4 Testes e Relatório Final

O fim do processo de desenvolvimento foi sucedido pela realização dos testes, escritos através de frameworks como o Jest, junto de ferramentas de CI/CD configuradas com o GitHub Actions. Houve também a realização de testes manuais com a utilização da base de dados construída na última etapa do trabalho, assim como a elaboração de testes de sistema, de maneira a garantir maior confiança aos resultados da aplicação. Alguns desses testes com dados reais foram

retratados na seção de ”Estudo de Caso”.

O framework de CI/CD construído para a manutenebilidade do projeto é detalhado na figura 1. Esse framework conta com duas ações: build e publish. Na ação build, é carregada a imagem mais recente do sistema operacional Ubuntu, que condiz com o ambiente no qual foi desenvolvido o projeto, seguido da configuração da versão do node, 18 nesse caso, junto com a instalação das dependências exigidas. Após isso, os testes de unidade do framework Jest são executados, sendo que uma falha em algum desses testes significa a falha de todo o pipeline. A ação publish efetua os mesmos passos até a instalação das dependências, diferenciando-se após isso com a ”construção”do pacote e eventual publicação na NPM.

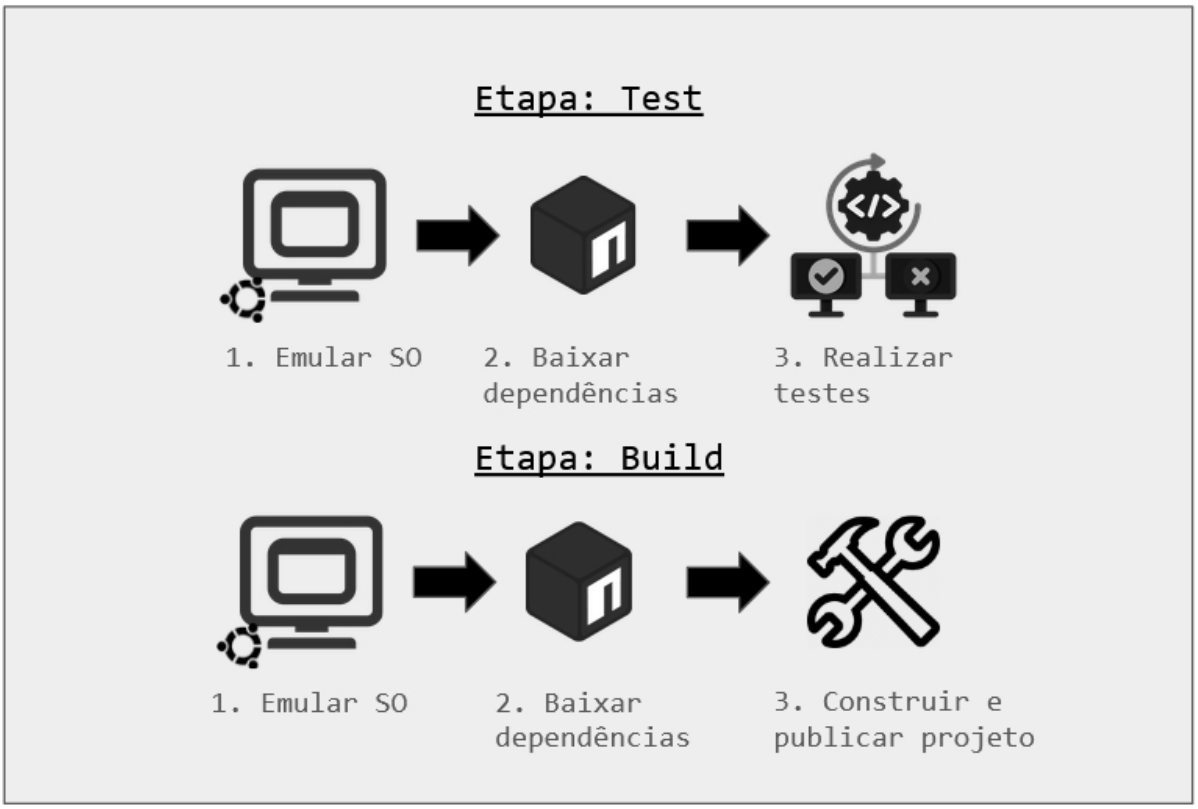

Figura 1: Diagrama de Etapas de CI/CD do Projeto

4. Resultados

Esta seção apresenta os resultados obtidos a partir da condução dos processos descritos na metodologia.

4.1 Distribuição

Atualmente, o projeto compilado e seu código fonte está disponível na plataforma de controle de versões GitHub, assim como no gerenciador de pacotes oficial do node NPM, onde já conta com mais de 150 downloads. Além do código, as plataformas também trazem uma página inicial chamada *readme*, com informações a respeito do projeto e como ele deve ser utilizado, assim como representado na figura 2, e um contrato de licença do tipo MIT.

O *readme* contém uma descrição do projeto, uma seção com explicações de como instalá-lo, um guia de utilização do AnyEater através da linha de comando, a lista de opções da aplicação e um exemplo de uso real da aplicação.

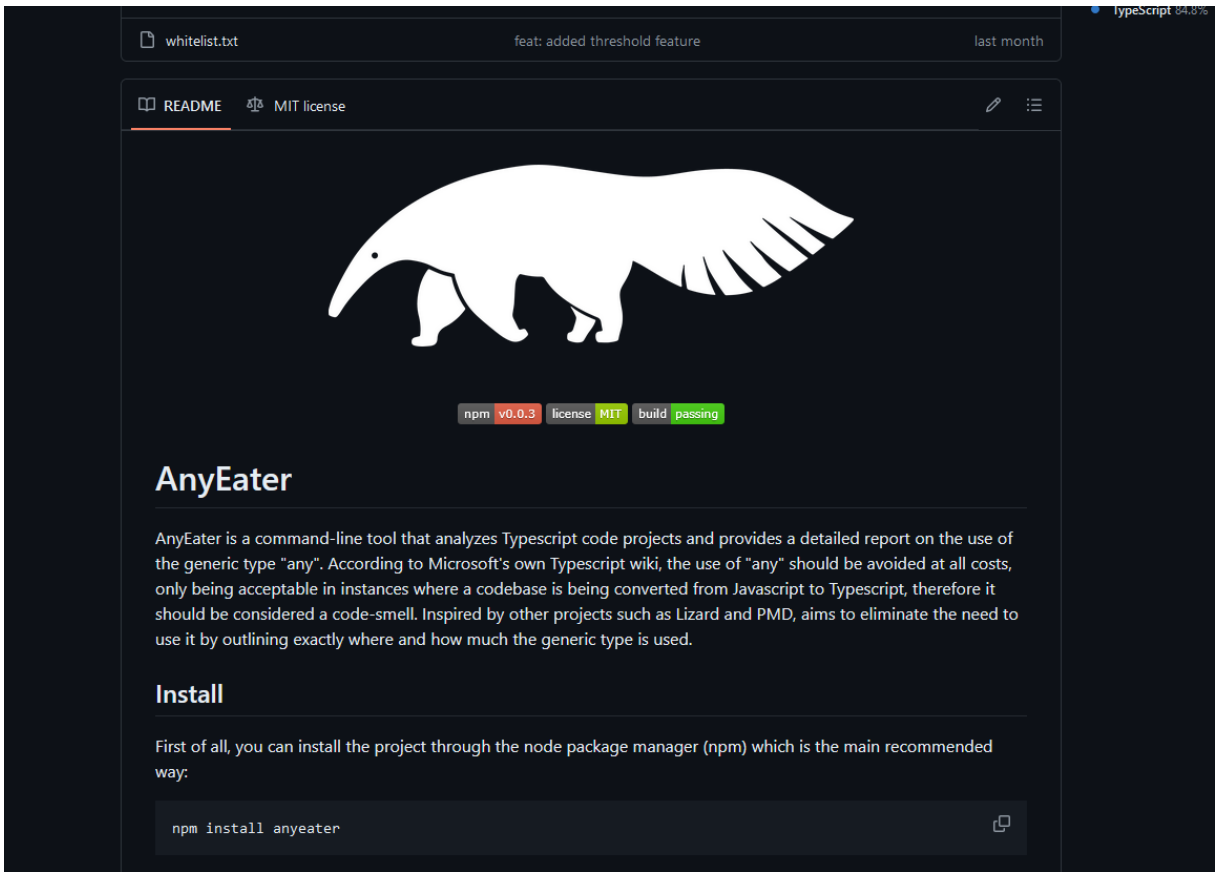

Figura 2: Página Principal do Projeto no Github

4.2 Recursos da Ferramenta

Após a instalação do AnyEater através de um do gerenciador de pacote do node, ou em algum dos repositórios, e a resolução de suas dependências, torna-se possível utilizá-lo. A execução do programa pode ser feita através das linhas de comando, da seguinte maneira:

`anyeater <diretorio-alvo>[opções]`

Ou, se o pacote não estiver registrado na variável PATH:

`npx anyeater <diretorio-alvo>[opções]`

Até a data de escrita deste relatório, a ferramenta possui uma funcionalidade principal: detectar e classificar os tipos genéricos em arquivos Typescript. Além disso, o *AnyEater* possui diversas flags que possibilitam opções de execução diferentes para o programa. Essas flags são descritas abaixo:

-h, –help: exibe todas as opções de linha de comando possíveis para o anyeater.

–version: imprime o número da versão atual.

-V, –verbose: imprime todas as saídas de modo detalhado (modo verboso).

-t, –threshold <number>: define o número de tipos genéricos usados em um arquivo a partir do qual deveria ser gerado um alerta.

-w, –whitelist <whitelist-path>: caminho para um arquivo com uma lista de arquivos que o programa deveria ignorar quando analisa um projeto.

-f, –format <type>: define o tipo de formato no qual a saída deve ser impressa (ex: csv ou standard).

4.3 Output

O output padrão do programa é uma tabela que indica o nome do arquivo analizado, o número de tipos genéricos encontrado no arquivo, o número do tipos genéricos com a função sintática mais prevalente e o número do tipos genéricos com a segunda função sintática mais prevalente. Além disso, o uso de opções podem alterar o output padrão para incluir também uma separação entre arquivos que superam o limite de tipos genéricos definidos, uma lista de toda classificação sintática de cada tipo genérico encontrada, assim como um output em formato de valores separados por vírgulas (csv). No output padrão, o programa termina com uma tabela com a soma total das estatísticas apresentadas.

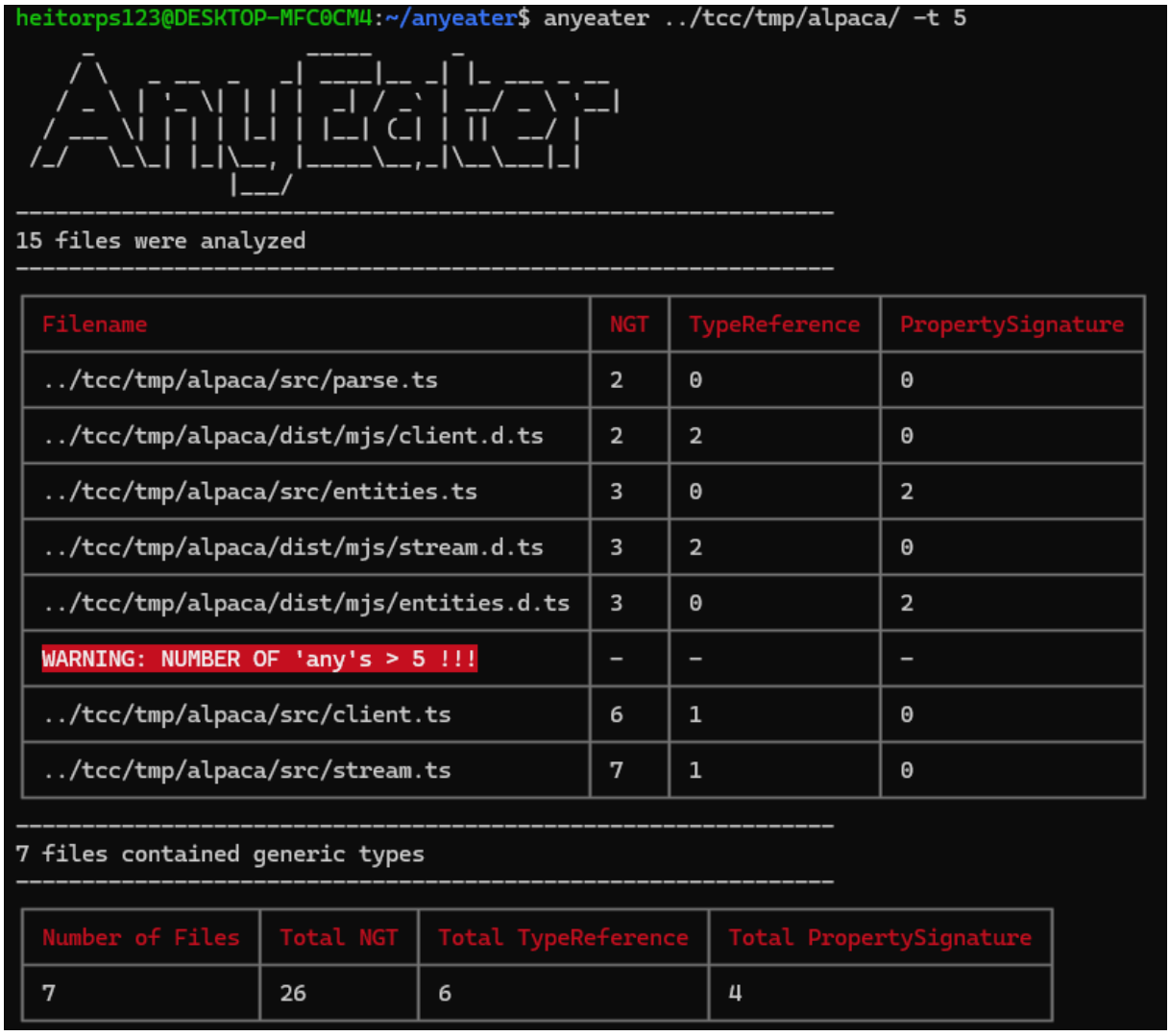

Figura 3: Output do Projeto

5. Estudo de Caso

Para demonstrar a utilização da ferramenta, propomos nesta seção a geração de relatórios para 10 projetos distintos, expondo as estatísticas gerais encontradas na tabela 1. Foram escolhidos projetos Typescript de código aberto disponíveis no github de maneira aleatória, e a ferramenta foi instalada no ambiente node global, e executada com a opção de formatação CSV para facilitar a junção dos dados. O projeto que mais apresentou tipos genéricos foi **ccf-asct-reporter**, com os dois usos mais prevalentes sendo *Parameter* e *TypeReference*.

Tabela 1: Análise de 10 Projetos com o Anyeater

Projeto	Número de ”any”s
learnwithjason/learnwithjason.dev	17
atlassian/stricter	35
tracer-protocol/perpetual-pools-contracts	35
vyquocvu/anystate	2
tc9011/awesome-nest	9
zgz682000/itms-cms	16
railmapgen/rmg	67
lxsmnsyc/venatu	22
hubmapconsortium/ccf-asct-reporter	96
nikersify/pico	4

Essa análise é só uma aplicação do *AnyEater* e não é o objetivo desta etapa, portanto não é tão profunda em sua exploração de dados estatísticos. Uma análise mais completa foi realizada na primeira parte do projeto [3].

6. Conclusão

Neste trabalho, buscamos desenvolver uma ferramenta de linha de comando capaz de analisar projetos de código Typescript e prover um relatório detalhado do uso do tipo genérico “any” nestes. Buscamos expandir nas descobertas apresentadas na primeira etapa da disciplina de projeto orientado a computação, onde foram exploradas estatísticas reais a respeito do uso do tipo genérico em projetos de código aberto e revelou-se sua prevalência na rotina dos desenvolvedores. Através de um estudo de ferramentas que se propõe a resolver questões similares e um estudo das necessidades dos possíveis usuários, um meticuloso processo de desenvolvimento e um período extenso de testes, conseguimos publicar uma ferramenta gratuita e de fácil acesso que está disponível atualmente no npm e no github.

Em trabalhos futuros pode-se expandir as capabilidades da ferramenta aqui apresentada expandindo a quantidade de opções disponíveis para a interação com o usuário, como por exemplo aumentando os tipos diferentes de outputs que podem ser gerados pelo *AnyEater*. Além disso, pode-se focar em melhorias referentes à ampliação do escopo do projeto, como a introdução de um sistema que fornece sugestões de tipos para substituição dos tipos genéricos encontrados, por exemplo através de técnicas de aprendizado de máquina.

Referências

[1] Jetbrains. The state of developer ecosystem 2022. https://www.jetbrains.com/lp/devecosystem-2022/, 2022. Accessed: 2024-06-10.

[2] Typescript.org. Do's and don'ts. https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html, 2024. Accessed: 2024-06-10.

[3] Heitor de Paula Santos Damasceno. Analisando como desenvolvedores typescript utilizam o tipo genérico "any", 2023. POC I.

[4] Lars Fischer and Stefan Hanenberg. An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. *SIGPLAN Not.*, 51(2):154–167, oct 2015.

[5] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, sep 2017.

[6] Justus Bogner and Manuel Merkel. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github, 2022.

[7] Terry Yin. Lizard. https://github.com/terryyin/lizard, 2024. Accessed: 2024-06-10.

[8] Nikolaos Tsantalis. jdeodorant. https://github.com/tsantalis/JDeodorant, 2024. Accessed: 2024-06-10.

[9] Josh Goldberg. Using the compiler api. https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API, 2023. Accessed: 2023-10-03.

A. Tabela de Possíveis Funções Sintáticas do “any”

Nome do Tipo	Explicação do Significado do Tipo
Parameter	Define o tipo do valor que é passado para a função.
AsExpression	Usado junto a uma expressão que efetua uma conversão de tipo explícito.
TypeReference	Variável que faz referência a um tipo.
VariableDeclaration	Usado na declaração de uma variável.
PropertySignature	Define o tipo de uma propriedade de um objeto.
PropertyDeclaration	Usado junto da declaração da propriedade de um objeto
IndexSignature	Tipifica o índice de algum objeto.
TypeAssertion Expression	Explicita o tipo de alguma variável que já foi declarada.
TypeParameter	Uma expressão que indica um tipo que pode ser substituído.
MethodDeclaration	Explicita o tipo de um método declarado.
FunctionType	Define o tipo de uma função declarada.
FunctionDeclaration	Explicita o tipo de uma função declarada.
NewExpression	Utilizado dentro de uma expressão de criação de objeto dinâmico (New).
CallExpression	Utilizado dentro da chamada de uma função.
ArrowFunction	Define o tipo de uma ”ArrowFunction”, um tipo específico de função do Typescript.
TupleType	Define uma tupla de tipos.
MethodSignature	Tipificação da assinatura de um método.
TypeAliasDeclaration	Define uma espécie de ”apelido”para outro tipo.
ExpressionWith TypeArguments	Um tipo passado para uma expressão como o próprio argumento.
TypeOperator	Usado junto de operadores que atuam em cima de tipos, como o operador KeyOf.
ConditionalType	Usado dentro de um condicional que checa o tipo (ex: if(TypeOf string === ...)).
MappedType	Maneira de criar tipos a partir de outros tipos, como por exemplo um tipo que é a variante parcial de outro.
GetAccessor	Usado com o operador ”Get”do Typescript.
FunctionExpression	Define o tipo de uma função designada a uma variável.
ConstructorType	Tipagem de um construtor.
CallSignature	Tipagem da assinatura de uma chamada.
ParenthesizedType	Tipo envolto de parênteses para evitar ambiguidade.
FirstTypeNode	Tipos para contagem em enums.
ConstructSignature	Tipagem da assinatura de um construtor.
LastTypeNode	Tipos para contagem em enums.
TaggedTemplate Expression	Chamada de função específica com uma Tag (<T>) de tipo.
OptionalType	Declaração de um tipo opcional com o operador ’?’
RestType	Tipagem de uma variável que resume múltiplas possíveis variáveis, faz uso do operador ’...’.

Tabela 2: Explicação dos Tipos Sintáticos