



UNIVERSIDADE DE SÃO PAULO  
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES  
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

JEAN PIERRE DE BRITO

**Estudo sobre a eficacia do desenvolvimento orientado a tipo para evitar bug**

São Paulo

2022

JEAN PIERRE DE BRITO

**Estudo sobre a eficácia do desenvolvimento orientado a tipo para evitar bug**

Versão original

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Orientador: Prof. Dr. Daniel Cordeiro

São Paulo

2022

## Lista de figuras

Figura 1 – Cronograma das atividades . . . . .	11
Figura 2 – Figura 1 - Ciclo do desenvolvimento orientado para o teste – TDD . . .	15
Figura 3 – Tipos de erros . . . . .	34
Figura 4 – Tipo de erro . . . . .	35
Figura 5 – Estrutura da composição . . . . .	37
Figura 6 – Programa erro divisão por zero em python . . . . .	40
Figura 7 – Programa erro divisão por zero em Java . . . . .	40
Figura 8 – Programa erro divisão por zero em Haskell . . . . .	41

## Lista de tabelas

Tabela 1 – Cruzamento dos temas dos artigos . . . . .	38
---	----

## Sumário

<b>1</b>	<b>Introdução</b>	<b>6</b>
1.1	<i>Motivação/Justificativa</i>	7
1.2	<i>Problema de pesquisa</i>	8
1.3	<i>Objetivos</i>	8
1.3.1	Objetivos específicos	9
1.4	<i>Metodologia</i>	9
1.5	<i>Estrutura da pesquisa</i>	10
1.6	<i>Cronograma</i>	11
<b>2</b>	<b>Fundamentação teórica</b>	<b>12</b>
2.1	<i>O código limpo e suas diretrizes</i>	12
2.2	<i>O ciclo de desenvolvimento orientado a testes funcionais (TDD)</i>	14
2.3	<i>Programação orientada a testes funcionais na prática</i>	16
2.4	<i>Programação Puramente Funcional</i>	18
2.5	<i>Programação Orientada a Tipos e suas diretrizes</i>	18
2.6	<i>Bugs: Compreendendo, caracterizando e classificando os tipos de bugs</i>	22
2.7	<i>reformular capitulos section</i>	28
2.8	<i>Fundamentação teórica</i>	28
2.9	<i>Introdução</i>	28
2.10	<i>A linguagem de programação Haskell</i>	30
2.10.1	Tipos com parâmetros	30
2.11	<i>Descrição de tipos</i>	33
2.11.1	Checagem de erros de tipos	34
2.11.2	Erros por bugs causados por código ausente	35
<b>3</b>	<b>Estado da Arte</b>	<b>36</b>
3.1	<i>Uma base que compõe os elementos de software</i>	36
3.2	<i>Cruzamento dos artigos e dissertações analisadas</i>	38
3.3	<i>Algumas considerações sobre uma estrutura de aplicação de tipos de erros</i>	39
3.4	<i>Conclusão</i>	41

REFERÊNCIAS . . . . .	44
-----------------------	----

## 1 Introdução

Encontramos no mercado atualmente, diversas linguagens de programação, que mesmo estando no auge de sua arquitetura e evoluindo de forma dinâmica, mas que contudo, não se adaptam e por vezes não se tornam flexíveis e dinâmicas para que possam serem utilizadas pelos programadores. O que se faz necessário, é que evoluam e utilizem tipos de dados que se tornem mais compreensíveis e que ofereçam aos programadores uma certa facilidade de adaptação para que possam utilizar largamente todas as suas funcionalidades sem que seja preciso que fiquem restritos aos tipos de dados. Dessa forma, a linguagem ideal ofereceria uma tipagem de dados dinâmica. Assim, a utilização de sistemas se daria de forma automática e haveria uma inferência dos tipos de dados que seria algo extremamente de ser utilizado pelos programadores e ao contrário de outros sistemas se tornaria um padrão a ser seguido por outras linguagens, e então teríamos uma grande concorrência pelo desenvolvimento de sistemas mais intuitivos.

Com relação aos sistemas que possuem proteção à prova de falhas, isso provavelmente origina-se de sistemas fracamente tipados, e que por essa razão estão constantemente sujeitos à erros e enganos no momento da programação. Com relação à sistemas sujeitos à falhas, [Arora et al. \(2012\)](#), realizaram uma intensa pesquisa para a criação de sistemas menos vulneráveis, sendo que o fundamento elementar dessa abordagem é que a funcionalidade comum que reaparece com regularidade suficiente e deve ser escrita uma vez, e esses softwares comuns deveriam ser montados por meio da reutilização em vez de serem reescritos de forma repetida.

Desta maneira essa abordagem na verdade é baseada em uma transformação passo a passo de um programa que fosse intolerante a falhas em um software tolerante a falhas, deveríamos adicionar elementos de software que pudessem tolerar um tipo específico de falha. E assim, dependendo do tipo e nível de tolerância a falhas que necessitaríamos alcançar, os componentes de tolerância a falhas são responsáveis por garantir que o programa seja tolerante a falhas lide com falhas, de forma a não mais desempenhar nenhum papel em assegurar que o programa funcione corretamente na falta de falhas.

Quando abordamos uma sintaxe semântica de teorias de tipos simples, a álgebra com suas operações de ligações ordenadas, com estrutura em si não vinculativa algebricamente, que abrangem exemplos comuns das teorias algébricas, incluindo aí cálculos computacionais

e lógica de predicados. Sabemos que no passado as extensões semelhantes à álgebra universal foram bastante exploradas. Assim, a álgebra universal é uma base que serve para realizar uma descrição de uma classe de estruturas matemática: que poderíamos precisar como sendo aquelas que são equipadas com operações matemáticas e de operações algébricas mono classificadas e que satisfaçam leis equacionais. Segundo [Arkor e Fiore \(2020\)](#) Apesar de que essas estruturas sejam bastante predominantes, existem ainda várias outras estruturas que interessam às ciências da computação que mesmo não se encaixando nessa estrutura, possuem suas particularidades e aplicações bastante interessantes. Assim esses tipos podem por vezes ainda estar sujeitos à erros muito comuns e dependendo da linguagem utilizada, um erro em sua declaração causaria bug's indesejados. Sendo assim, o que [Sinha e Hanumantharya \(2005\)](#), nos induz a ter em mente é que devemos tomar extremo cuidado ao utilizar essas expressões e fazer o máximo para ter uma sintaxe correta destes tipos de dados.

É bastante interessante e tem-se muito desejado por diversos indivíduos em entender a álgebra universal e assim, visto que de uma perspectiva da teoria da linguagem de programação, está estrutura é muito conveniente para a sintaxe abstrata: pois a estrutura das linguagens de programação, se descartarmos os detalhes superficiais da sintaxe concreta, e tendo um ponto de vista categórico, a teoria algébrica dos tipos nos oferece uma correspondência precisa entre a estrutura sintática e a semântica. De acordo com [Sinha e Hanumantharya \(2005\)](#):

A abordagem baseada em componentes envolve a transformação passo a passo de um programa intolerante a falhas em um programa tolerante a falhas, adicionando componentes de software que podem tolerar um tipo específico de falha. Dependendo do nível de tolerância a falhas que precisa ser alcançado, os componentes de tolerância a falhas são responsáveis por garantir que o programa tolerante a falhas lide com as falhas, não desempenhando nenhum papel em garantir que o programa funcione corretamente na ausência de falhas. ([SINHA; HANUMANTHARYA, 2005](#), p.366).

### 1.1 Motivação/Justificativa

Nossa real motivação para a confecção deste trabalho foi o imenso desejo de apresentar uma questão que se mostra de uma forma bastante conflitante para diversos programadores no desempenho de suas atividades do cotidiano; escrever um código limpo; livre de erros; que proporcione um desempenho hábil da aplicação e que traga resultados satisfatórios para a empresa ou clientes para os quais prestam serviço remunerado. A



maioria dos profissionais que lidam no dia a dia com a análise e desenvolvimento de sistemas, sempre estão à procura de sistemas com uma tipagem e linguagem de alto nível para que a rentabilidade de seu trabalho lhes proporcione maior agilidade e eficácia, ansiamos em apresentar um trabalho que venha a trazer opções e soluções que reduzam e agreguem valor a este anseio da comunidade.

Contudo, há algo mais profundo envolvido nisso, a exemplo disso, grandes organizações necessitam soluções que atendam rapidamente às suas expectativas e que resolvam as necessidades de seus clientes/consumidores, bancos, entidades governamentais e não governamentais, consumidores comuns, empresários anseiam por soluções que atendam às suas necessidades e que possuem ansiedade de ver suas aplicações e sistemas prontos em um período de tempo em que por vezes não é um prazo bastante hábil para que os desenvolvedores possam executar as suas tarefas, apesar de terem suas equipes se dedicando à isso. Essas soluções e sistemas precisam estar livre de falhas e erros, e é aí que entra os sistemas e linguagens à prova de falhas e de fácil criação. [Arora et al. \(2012\)](#) encontraram um framework que possui dois elementos tolerantes a falhas justamente para projetos de programas tolerantes a falhas que podem ser projetados fazendo uso de um elemento, que abordaremos em nosso trabalho. Temos como problema de pesquisa analisar a linguagem altamente tipada para se evitar tipos de erros em programação que onerem tempo e execução de sistemas.

## 1.2 Problema de pesquisa

Nosso problema de pesquisa consiste em avaliar uma linguagem de programação que seja altamente tipada de forma a oferecer e reduzir os tipos de erros em programação, bem como outras possíveis falhas que venham a ocorrer.

## 1.3 Objetivos

Tenho como objetivo geral avaliar uma linguagem de programação que seja altamente tipada de forma a oferecer uma redução dos tipos de erros em programação, bem como as possíveis falhas que venham a ocorrer durante o desenvolvimento de programas.

### 1.3.1 Objetivos específicos

Tendo ainda em vista o objetivos geral, podemos desdobrá-lo em três objetivos específicos, a saber:

1 - avaliar formas de programação orientada a tipos de categorias exibidas e categorias de tipos de dados para que possa dessa forma se mostrar uma solução de nosso objetivo geral;

2 - Averiguar a sintaxe abstrata tolerantes a falhas de maneira que se possa tolerar os diferentes erros de forma automática;

3 - Disponibilizar catálogo que incentivem ao mínimo a ocorrência de tipos de erros ocorridos nas linguagens de programação, demo modo que o objetivo geral atenda ao objetivo específico de número 3.

## 1.4 Metodologia

A metodologia em um trabalho acadêmico é uma das partes mais relevantes em que apresentamos a forma como o trabalho foi conduzido e de que maneira coletamos as informações para concluirmos o que nos propusemos a realizar com a pesquisa. Aqui definimos o tipo de pesquisa, se é uma revisão da literatura, ou uma pesquisa de campo, se ela é qualitativa ou quantitativa, etc.

Em nosso caso e tendo em vista ainda a metodologia. foi realizada uma revisão da literatura com o intuito de averiguar junto à autores renomados no assunto o que há de mais recente e que atenda às strings de nossa pesquisa. De acordo com :

O objetivo de uma pesquisa exploratória é familiarizar-se com um assunto ainda pouco conhecido ou explorado. Assim, se constitui em um tipo de pesquisa muito específica, sendo comum assumir a forma de um estudo de caso. Nesse tipo de pesquisa, haverá sempre alguma obra ou entrevista com pessoas que tiveram experiências práticas com problemas semelhantes ou análise de exemplos análogos que podem estimular a compreensão.

(GIL, 2008, p.35).

Nossa pesquisa é ainda do tipo qualitativa e trata-se de um estudo de caso, empírico e aplicada pois vamos fazer a programação exata do sistema em que serão analisados artefatos do Hit-hub para um melhor aprofundamento dos dados analisados, e especialmente vai se tratar de uma análise documental em artigos publicados no Scopus.

O método empírico é algo que se apoia apenas em experiências vividas, por meio da observação de elementos, e não se baseia apenas na teoria e métodos científicos. Assim sendo, o empírico é o conhecimento que se adquire durante toda a vida, no dia-a-dia, e ele não é baseado em comprovação científica alguma.

O método empírico é um método criado para testar a validade de teorias e hipóteses em um contexto de experiência. O método empírico gera evidências, uma vez que aprendemos fatos através das experiências vividas e presenciadas, para obter conclusões.

Já o conhecimento empírico ou senso comum é o conhecimento baseado em uma experiência vulgar, ou imediata, não metódica e que não foi interpretada e organizada de forma racional.

Estamos dessa forma com uma pesquisa de abordagem empírica e com objetivos exploratórios e procedimentos técnicos experimentais, forma utilizados experimentos laboratoriais com linguagens de programação, de acordo com o que foi descrito no capítulo 3: Estado da arte. A análise documental será realizada de forma documental de artefatos do git hub / logs.

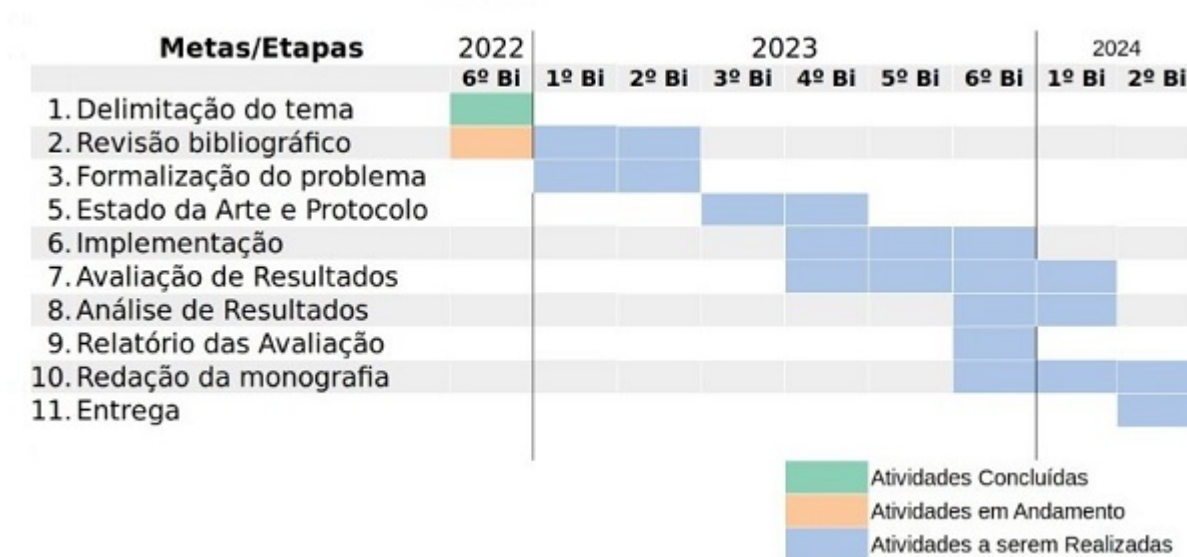
### *1.5 Estrutura da pesquisa*

A estrutura de nossa pesquisa está dividida em quatro partes principais, a saber: introdução, onde apresentamos uma breve descrição do que será abordado na pesquisa, citando alguns dos principais autores envolvidos no estado da arte e algumas abordagens iniciais do problema de pesquisa, objetivos gerais e específicos, bem como a justificativa e metodologia além da estrutura do documento; capítulo 2 - referencial teórico, onde abordamos as características da linguagem halkell, altamente tipada por inferência de tipos de dados, além de algumas outras considerações, no capítulo 3 realizamos a finco e de forma referenciada como os mais diversos autores que versam sobre o tema de nossa dissertação, averiguando o que realmente nos fez ver como pesquisas e periódicos, artigos e dissertações nos apresentam soluções para o problema de erros e falhas nos sistemas tipados e na conclusão, apresentamos a que conclusão chegamos de nossa pesquisa, como ela contribuiu para a comunidade acadêmica e ainda para futuros trabalhos de outros estudantes.

### 1.6 Cronograma

A estimativa para elaboração do projeto é de a partir do 6º bimestre de 2022 até o segundo bimestre de 2024 sendo a primeira parte a delimitação do tema e o planejamento de execução. Na fase de levantamento bibliográfico será realizada a definição da fonte de pesquisa, delimitação dos artigos e livros que serão observados para a revisão sistemática, previsto para o ano letivo de 2022 e no primeiro e segundo bimestre de 2023. Nessa fase, será elaborada a pesquisa para identificação dos fatores críticos de sucesso com a delimitação dos critérios de pesquisa de campo e critérios de coleta de dados, a formalização do problema de pesquisa. Na sequência será elaborado e apresentado o anteprojeto de pesquisa seguido da coleta de dados tanto do questionário como também da revisão sistemática, fase da análise de dados e organização do roteiro do projeto. Ao final será realizada a revisão, seguida da entrega e defesa do projeto conforme Figura 1, apresentada logo abaixo:

Figura 1 – Cronograma das atividades



Fonte:Elaborada pelo autor (2022)

Baseado na metodologia apresentada, podemos observar que de um ponto de vista metodológico a pesquisa pode ser qualificada ainda como sendo quantitativa e qualitativa, visto que apresentamos no estado da arte elementos que fundamentam o que estamos afirmando nessa seção da dissertação.

## 2 Fundamentação teórica

### 2.1 O código limpo e suas diretrizes

Código limpo significa que o código é fácil de ler, fácil de entender e fácil de mudar (TAKAGI, 2021). É tão simples quanto ter variáveis, funções e classes descritivamente nomeadas para que, ao ler, saiba o que cada um contém ou faz sem adivinhação (HAZIN, et. al., 2017).

Martim (2019), menciona sobre o emprego desses princípios-chave: KISS, que significa Keep It Simple, Stupid, e DRY, Don't Repeat Yourself. Pergunte a si mesmo se há uma solução melhor para resolver problemas ou complexidade dentro do seu código. Comece a construir hábitos que sejam propícios para limpar códigos, tais como:

- Verificar suas convenções de nomeação. Mantenha-os limpos e conciso;
- Ser consistente. Usar os mesmos nomes em funções semelhantes (Get vs Fetch, Set vs Update, Add vs Append);
- Se copiar o código várias vezes, considere melhores maneiras de tornar seu código mais eficiente;
- Para solicitações de banco de dados, mantenha-o o mais simples e produtivo possível.

Se achar que um pedaço de código é muito complexo ou não é a melhor maneira de fazê-lo, então pare. Dê a si mesmo alguns momentos para repensar o processo e ver se pode chegar a uma maneira melhor. Realizar uma revisão com colegas, pode ser uma boa opção (MANGRICH, et. al., 2021). Sempre tenha em mente a importância de se comparar o código de sua equipe com o seu e esteja aberto a sugestões, pois assim a empresa sempre trabalhará focada e com códigos mais robustos.

A maioria dos códigos não é escrito e esquecido, é escrito e depois modificado ao longo do tempo para mantê-lo atualizado e eficiente no que diz respeito ao projeto (TAKAGI, 2021). Provavelmente, não é a única pessoa que precisará entender o que seu código faz, e é por isso que o que ele faz precisará ser óbvio. É possível usar “comentários” no código?

Sim, mas se essa é sua única forma de comunicação dentro do seu código, então deve-se estar escrevendo código sujo. Ter certeza de que seu código pode falar por si mesmo também irá ajudá-lo a desenvolver seu vocabulário técnico. Isso beneficia elevando sua

qualidade de código, quando se pode discutir o código claramente, pode-se pedir ajuda e sugestões mais eficazes sobre como melhorar o que se escreve. Código limpo não se escreve sozinho, é preciso um foco dedicado em apresentar o que se quer transmitir (MARTIN, 2019).

Mas o porquê é importante considerar estas diretrizes? (HAZIN, et. al., 2017), explica que uma equipe não estará no campo de desenvolvimento de software por muito tempo antes de se juntar a outro projeto que é uma colaboração em tempo real ou contém código legado. Trabalhar com código que os outros têm ou trabalharão torna evidente que o código limpo é importante.

Saltar em código que não tem uma direção clara é demorado para todas as partes envolvidas, porque quando não é aparente o que o código faz ou por que ele faz, se desperdiça recursos tentando descobrir seu propósito (MANGRICH, et. al., 2021). É reutilizável? Quanto mais limpo for o código, maior a probabilidade de se reutilizá-lo! Escrever código limpo faz de um programador, mais eficiente porque está escrevendo código que faz uma coisa clara, que não leva tempo extra para escrever. O código limpo também torna a manutenção mais fácil e rápida (CARVALHAES, et. al., 2021).

Pedir ajuda de um colega e depois ter que percorrer um código mal escrito para encontrar o problema nunca é um impulso para sua confiança. Escrever código limpo permitirá que se produza código de qualidade que se orgulha de compartilhar com a equipe, sabendo que eles terão mais facilidade em trabalhar nele também (MARTIN, 2019).

Normalmente, nem sempre se escreve um código limpo (MANGRICH, et. al., 2021). E preciso ter a rotina da escrita, “treinar” junto a equipe para levar a prática da perfeição. Quando se sabe melhor, pode-se fazer melhor. Se tirar um tempo para escrever código limpo agora, colherá os benefícios da legibilidade e da depuração mais fácil para frente.

Se achar que passa a maior parte do tempo escrevendo código, está errado. (CARVALHAES, et. al., 2021), menciona que se gasta mais tempo lendo códigos, descobrindo bugs, encontrando problemas e descobrindo como resolvê-los. Código limpo torna todas essas coisas mais fáceis. Se há uma coisa em que todos os programadores concordam, é que código limpo é melhor (MANGRICH, et. al., 2021).

## 2.2 O ciclo de desenvolvimento orientado a testes funcionais (TDD)

O conceito de Desenvolvimento Orientado por Testes (TDD) foi introduzido em 2003 por Kent Beck (WAGNER; FANTONI, 2021). Não há definição formal, mas Beck dá abordagens e exemplos de TDD. O objetivo do TDD é "escrever código limpo que funcione" (HAZIN, et. al., 2017).

No TDD, siga apenas uma regra de ouro: Só altere o código de produção se algum teste falhar. Caso contrário, apenas refator para otimizar o código. Para os requisitos atualizados, converta-os em casos de teste, adicione esses testes e, só então, escreva um novo código (MEIRELES; DE RESENDE COSTA, 2021).

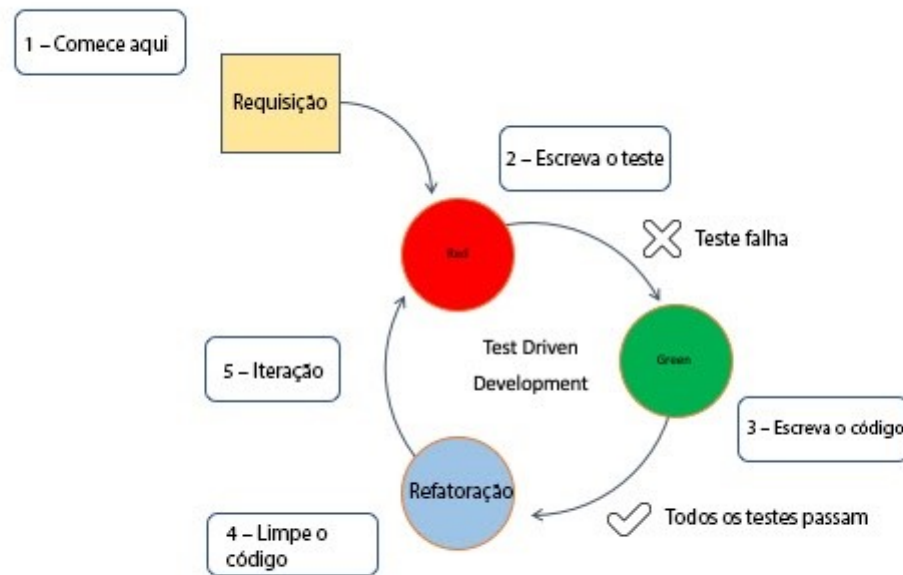
TDD é um ciclo de desenvolvimento muito curto e repetitivo. Os requisitos do cliente são transformados em casos de teste altamente específicos e o software é escrito e melhorado para passar nos novos testes (WAGNER; FANTONI, 2021).

O Desenvolvimento Orientado por Testes está relacionado a conceitos de programação em programação extrema, defendendo atualizações/lançamentos de software frequentes em ciclos curtos de desenvolvimento e promovendo extensas revisões de código, testes de unidade e adição incremental de recursos (MEIRELES; DE RESENDE COSTA, 2021).

Um conceito intimamente relacionado ao TDD é o AtDD (Acceptance Test-Driven Development, desenvolvimento orientado para testes de aceitação), onde o cliente, o desenvolvedor e o testador participam do processo de análise de requisitos (HAZIN, et. al., 2017). O TDD é tanto para desenvolvedores de aplicativos móveis quanto para aplicativos web, enquanto o ATDD é uma ferramenta de comunicação para garantir que os requisitos sejam bem definidos (WAGNER; FANTONI, 2021).

De Matos et. al., (2018), menciona que o ciclo TDD, também conhecido como processo Vermelho-Verde-Refator, pode ser observado na Figura 1 abaixo, demonstra que:

Figura 2 – Figura 1 - Ciclo do desenvolvimento orientado para o teste – TDD



Fonte: (De Matos et. al., 2018) – adaptado pelo autor

1. Adicione um teste, que certamente falhará. (Vermelho): No TDD, cada recurso em um software é adicionado pela primeira vez em termos de casos de teste. Um teste é criado para uma função nova ou atualizada. Para escrever os testes, os desenvolvedores devem entender as especificações e requisitos do recurso. Essa prática separa o TDD dos métodos tradicionais de desenvolvimento de software, onde os testes de unidade são escritos após a escrita do código-fonte. Desta forma, o TDD faz com que o desenvolvedor se concentre nos requisitos antes de escrever o código.
2. Execute todos os testes. Veja se algum teste falha: A execução de testes valida que o arreio de teste está funcionando corretamente e simultaneamente prova que, à medida que novos testes adicionados estão falhando com o código existente, um novo código é necessário.
3. Escreva apenas código suficiente para passar em todos os testes. (Verde): O novo código escrito nesta fase pode não ser perfeito e pode passar no teste de forma irrelevante. O único requisito nesta fase é que todos os testes sejam aprovados. Uma maneira possível de começar adicionando as declarações é retornar uma constante e adicionar gradualmente blocos lógicos para construir a função.



4. Execute todos os testes. Se algum teste falhar, volte para o passo 3. Caso contrário, continue: Se todos os testes forem aprovados, pode-se dizer que o código atende aos requisitos de teste e não degrada quaisquer recursos existentes. Se algum teste falhar, o código deve ser editado para garantir que todos os testes passem.
5. Refatorar o código. (Refator): À medida que a base de código cresce, ela deve ser limpa e mantida regularmente. Como? Há algumas maneiras:
  - Um novo código que poderia ter sido adicionado por conveniência para passar em um teste pode ser movido para seu lugar lógico no código.
  - A duplicação deve ser eliminada.
  - Definições de objeto e nomes devem ser definidos para representar seu propósito e uso.
  - À medida que mais recursos são adicionados, as funções se tornam longas. Pode ser benéfico dividir e nomear cuidadosamente para melhorar a legibilidade e a manutenção.
  - Como todos os testes são reexecutados ao longo da fase de refatoração, o desenvolvedor pode estar confiante de que o processo não altera nenhuma funcionalidade existente.
6. Se um novo teste for adicionado, repita a partir do passo 1. Dê pequenos passos, mirando apenas 1 a 10 edições entre cada teste executado. Se o novo código não satisfaz rapidamente um novo teste, ou outros testes não relacionados falharem inesperadamente, então desfaça/reverta para um código de trabalho, em vez de fazer depuração extensiva.

Ao usar bibliotecas externas, é importante não fazer incrementos tão pequenos que apenas testem a biblioteca em si, a menos que seja para testar se a biblioteca está desatualizada/incompatível, buggy ou não completa de recursos.

### 2.3 Programação orientada a testes funcionais na prática

De Mattos et. al., (2018), menciona que para sistemas de grande porte, o teste funcional é desafiador e requer uma arquitetura modular com componentes bem definidos. Alguns requisitos-chave que devem ser cumpridos são:

- A alta coesão garante que cada módulo forneça um conjunto de capacidades relacionadas, facilitando a manutenção dos testes correspondentes.
- O acoplamento baixo permite testes isolados de módulos.

Na Modelagem de Cenários, um conjunto de gráficos de sequência é construído, cada gráfico focado em um único cenário de execução em nível de sistema. Ele fornece um excelente veículo para criar estratégias de interação em resposta a uma entrada. Cada Modelo de Cenário serve como um conjunto de requisitos para os recursos que um componente fornecerá. A modelagem de cenários pode ser útil na construção de testes TDD em sistemas complexos (SARCINELLI, 2018).

É importante diferenciar o código entre teste funcional e a produção (DE SOUZA, JHONATA LIMA, 2022). O conjunto de testes funcionais da unidade deve ser capaz de acessar o código para testar. No entanto, o desenho de critérios como ocultação de informações e encapsulamento e separação de módulos não deve ser comprometido. No design orientado para objetos, os testes funcionais ainda não poderão acessar membros e métodos de dados privados e exigir uma codificação extra. Alternativamente, uma classe interna pode ser usada dentro do código fonte para conter os testes da unidade. Tais hacks de teste não devem permanecer no código de produção. Os praticantes de TDD frequentemente argumentam se os dados privados devem mesmo ser testados (DE MATTOS, et. al., 2018).

Barros et. al., (2019), cita que com base em linguagens de programação únicas, existem várias estruturas que suportam o desenvolvimento orientado a testes, como:

1. csUnit e NUnit – Ambos são estruturas de teste de unidades de código aberto para projetos .NET.
2. PyUnit e DocTest: Estrutura de teste da Unidade Popular para Python.
3. Junit: Ferramenta de teste de unidade amplamente utilizada para Java
4. TestNG: Outra popular estrutura de testes Java. Este quadro supera as limitações do Junit.
5. Rspec: Uma estrutura de testes para projetos Ruby

O processo de entrega de produtos de qualidade exige não apenas depuração, mas também exige otimização no processo de desenvolvimento. Quando incorporada corretamente, a abordagem TDD proporciona inúmeros benefícios, particularmente em termos de trazer

custo-benefício no longo prazo e fornecer verdadeiro valor para as empresas (BARROS, et. al., 2019).

#### *2.4 Programação Puramente Funcional*

#### *2.5 Programação Orientada a Tipos e suas diretrizes*

A programação orientada a tipos é um paradigma que abrange abordagens funcionais e orientadas a objetos para escrever código tipado estaticamente (SILVA, et al., 2019). A maioria das linguagens de programação nesta categoria faz uso extensivo de subtipagem e polimorfismo paramétrico.

Esta seção introduz os conceitos básicos, a maioria deles conhecidos de linguagens funcionais e orientadas a objetos, e depois passa a explicar algumas técnicas mais envolvidas usadas em linguagens com poderosos sistemas de tipos.

Muito trabalho tem sido investigado com paradigmas de programação (CLEMENTE, 2021). Paradigmas comuns incluem imperativo, funcional, orientado a objetos e orientado a aspectos. No entanto, no desenvolvimento de ideias de programação orientada a tipos, toma-se conceito familiar de um tipo, associa-se em profundidade a semântica de tempo de execução a tal, de modo que o comportamento do uso da variável (ou seja, acesso e atribuição) pode ser determinado pela análise do tipo específico (ROCHA; NETO, 2003).

Em muitas linguagens existe a necessidade de combinar um número de atributos com uma variável, para este fim permitimos que o programador combine tipos juntos para formar um supertipo (cadeia de tipos).

Uma cadeia de tipos é uma coleção de tipos, combinados pelo programador (VEDOVATTO; COSTA, 2013). É essa cadeia de tipo que determinará o comportamento de uma variável específica. A precedência na cadeia de tipos é da direita para a esquerda (ou seja, o último tipo adicionado substituirá o comportamento dos tipos adicionados anteriormente). Essa precedência permite que o programador adicione informações adicionais, seja de forma terminante ou para uma expressão específica, à medida que o código progride (ANDRADE, et al., 2004).

As variáveis de tipo são um conceito interessante. Semelhante às variáveis normais do programa, elas são declaradas para conter uma cadeia de tipos. Ao longo da execução

do programa, elas podem ser tratadas como variáveis normais do programa e podem ser verificadas por meio de condicionais, atribuídas e modificadas (SILVA, et al., 2019).

Rocha e Neto (2003), mencionam que há uma série de vantagens na programação orientada a tipos, incluindo:

- Eficiência - A rica quantidade de informações permite que o compilador execute muita análise estática e otimização, resultando em maior eficiência.
- Simplicidade - Ao fornecer uma biblioteca de tipos limpa, o programador pode usar tipos bem documentados para controlar muitos aspectos de seu código.
- Linguagem mais simples - Ao tirar a maior parte da complexidade da linguagem e colocá-la em uma biblioteca de tipos fracamente acoplada, a linguagem é mais simples do ponto de vista do design e da implementação (do compilador). Adicionar inúmeras palavras-chave de linguagem geralmente resulta em um design frágil, usando programação orientada a tipos, isso é evitado
- Facilidade de manutenção - Ao alterar o tipo, pode-se ter um efeito considerável na semântica do código, abstraindo o programador isso torna o código mais simples, mais flexível e mais fácil de manter.

Todas as linguagens paralelas atuais sofrem com o comprometimento da simplicidade versus eficiência. Ao abstrair o programador longe dos detalhes de baixo nível, dá-lhes uma linguagem simples de usar, mas o alto nível de informações fornecidas ao compilador permite que muita análise seja realizada durante a fase de compilação. A partir de linguagens de baixo nível (como C), é difícil para o compilador entender como o programador está usando o paralelismo, portanto, a otimização de tal código é limitada (ROCHA; NETO, 2003).

Fornece-se ao programador a escolha entre programação explícita e implícita - eles podem confiar nos padrões de linguagem embutidos e seguros ou, alternativamente, usar tipos adicionais para obter mais controle (e desempenho). Portanto, a linguagem é aceitável tanto para o programador paralelo novato quanto para o especialista (CLEMENTE, 2021).

Vedovatto e Costa (2013), mencionam que a programação orientada a tipos pode ser usada em:

- Programação GUI - A programação GUI pode ser bastante cansativa e repetitiva (daí o uso de IDEs de design gráfico). Usando tipos, isso abstrairia o programador de muitas das questões repetitivas.

- Retrofit Existing Languages - A abordagem de tipo poderia ser aplicada a linguagens existentes onde um retrofit poderia ser realizado, mantendo o programador em sua zona de conforto, mas também lhes dando o poder da programação orientada a tipos.
- Numerosos sistemas de tipo - O sistema de tipos é completamente separado da linguagem real, seria possível fornecer um número de sistemas de tipo para uma única linguagem, como um sistema paralelo, um sistema sequencial etc.

Sendo assim, Maguire (2018), relata que a programação orientada a tipos é uma das abordagens mais populares e fundamentais na construção de softwares modernos. É uma técnica que permite criar estruturas de dados complexas, seguras e eficientes por meio do uso de tipos estáticos (THOMPSON, 1991). Tipos estáticos são definidos em tempo de compilação e verificam a validade das operações realizadas com os objetos de uma classe, evitando assim erros e reduzindo a complexidade do código. Nesse sentido, a programação orientada a tipos é considerada uma das melhores práticas para construir softwares confiáveis e escaláveis (SCOTT, 2000).

Uma das vantagens da programação orientada a tipos é a sua capacidade de detectar erros de tipagem em tempo de compilação (PIERCE, 2002). Isso significa que o compilador verifica se as operações realizadas em uma variável são compatíveis com o tipo declarado. Caso contrário, o compilador exibe uma mensagem de erro e impede a compilação do programa. Essa verificação estática ajuda a evitar erros de tempo de execução e facilita a detecção de erros de programação (MAGUIRE, 2018). Abaixo, segue um pequeno exemplo de um código fonte, desenvolvido em Haskell (que será abordado na seção XX), evidenciando a estrutura de um pequeno programa orientado a tipos. Exemplo 01: exemplo de programação orientada a tipos, tendo como o uso do tipo de dados 'Person para definir uma pessoa com nome e idade, onde:

```
data Person = Person String Int
personName :: Person -> String
personName (Person name _) = name
personAge :: Person -> Int
personAge (Person _ age) = age
```

Nesse exemplo, definimos um novo tipo de dados Person, que representa uma pessoa com um nome e uma idade. Em seguida, definimos duas funções personName e personAge, que acessam o nome e a idade de uma pessoa, respectivamente. Observe que o tipo de dados Person é definido usando a palavra-chave data, que é usada para definir tipos de dados personalizados em Haskell. Para criar uma pessoa usando o tipo de dados Person, pode-se usar o seguinte código: let john = Person "John Doe" 30.

Nesse exemplo, criamos uma nova pessoa chamada john com nome "John Doe" e idade 30. Podemos então acessar o nome e a idade de john usando as funções `personName` e `personAge`, como mostrado abaixo:

```
print (personName john) – Output: "John Doe" print (personAge john) – Output:
30
```

Além disso, a programação orientada a tipos ajuda a reduzir a complexidade do código, tornando-o mais fácil de ler, entender e manter (THOMPSON, 1991). Isso ocorre porque a definição de tipos estáticos permite que o programador saiba exatamente quais operações podem ser realizadas com um objeto de uma determinada classe. Dessa forma, a programação orientada a tipos ajuda a evitar erros de lógica e torna o código mais fácil de depurar (PIERCE, 2002).

A programação orientada a tipos é importante para o desenvolvimento de software por várias razões. Em primeiro lugar, ela ajuda a garantir a segurança e a confiabilidade do código, permitindo que os desenvolvedores capturem erros de tempo de compilação em vez de erros de tempo de execução. Isso pode reduzir significativamente o tempo e o esforço necessários para depurar e testar o software (SCOTT, 2000).

Além disso, a programação orientada a tipos permite que os desenvolvedores criem abstrações de software mais robustas e reutilizáveis (MAGUIRE, 2018). Ao definir tipos de dados personalizados, os desenvolvedores podem encapsular dados e comportamentos relacionados em um único objeto, tornando o código mais modular e fácil de entender e manter (PIERCE, 2003).

A programação orientada a tipos também pode ajudar a melhorar a eficiência do código, permitindo que o compilador otimize o código com base nas informações de tipo (THOMPSON, 1991). Por exemplo, o compilador pode usar informações de tipo para evitar a necessidade de conversões de tipo desnecessárias ou para otimizar a alocação de memória (PIERCE, 2003).

A programação orientada a tipos é uma técnica de programação bem estabelecida e amplamente utilizada, com muitas bibliotecas e estruturas de programação disponíveis para os desenvolvedores (MAGUIRE, 2018). Isso significa que, ao usar a programação orientada a tipos, os desenvolvedores podem aproveitar a experiência e os recursos da comunidade de desenvolvimento de software para acelerar o desenvolvimento de seus próprios projetos (SCOTT, 2000).

Outra vantagem da programação orientada a tipos é a sua capacidade de suportar polimorfismo e herança. Polimorfismo é a capacidade de um objeto de uma classe se comportar como outro objeto de uma classe diferente (MENEZES, 2012). Herança é a capacidade de criar uma nova classe que é uma extensão de uma classe existente, herdando suas propriedades e métodos. Esses recursos são importantes para a reutilização de código e a criação de hierarquias de classes (COSTA, 2013).

Embora a programação orientada a tipos tenha sido criticada por ser excessivamente restritiva e burocrática, muitos programadores ainda consideram essa abordagem como essencial para a construção de softwares seguros e confiáveis (DEJEU, 2013). Na verdade, a programação orientada a tipos é tão importante que muitas linguagens de programação modernas, como Java, C, Python e Ruby, a suportam nativamente. Essas linguagens oferecem recursos avançados de programação orientada a tipos, como interfaces, classes abstratas e polimorfismo.

Por fim, a programação orientada a tipos também pode ajudar a melhorar a comunicação e a colaboração entre os membros da equipe de desenvolvimento de software. Ao usar tipos de dados bem definidos e documentados, os desenvolvedores podem se comunicar mais claramente sobre o comportamento e os requisitos do código, facilitando a colaboração e a solução de problemas em equipe (FONSECA; SANTOS, 2016).

## *2.6 Bugs: Compreendendo, caracterizando e classificando os tipos de bugs*

Um bug é um problema inesperado com software ou hardware (BELLER, et al., 2018). Problemas típicos são muitas vezes o resultado de interferência externa com o desempenho do programa que não foi antecipado pelo desenvolvedor (LUNA, 2022). Pequenos bugs podem causar pequenos problemas, como telas congeladas ou mensagens de erro inexplicáveis que não afetam significativamente o uso (SILVA, et al., 2020). Os principais bugs podem não apenas afetar o software e o hardware, mas também podem ter efeitos não intencionais em dispositivos conectados ou software integrado e podem danificar arquivos de dados (BELL, et al. 2018).

(Aqui pode ser mencionando a tabela do Excel, relacionando alguns tipos de bugs, coloque uns 10).

A palavra bug originou-se na engenharia. O uso do termo na computação é atribuído à programadora pioneira Grace Hopper. Em 1944, Hopper era um jovem oficial da Reserva Naval que trabalhou no computador Mark I em Harvard. Hopper descreveu mais tarde um incidente em que um técnico teria puxado um bug real - uma mariposa, na verdade - entre dois relés elétricos em um computador Mark II. A Marinha teve a mariposa em exibição por muitos anos. O Smithsonian agora o tem em suas propriedades (BELLER, et. al., 2015).

Embora os bugs normalmente causem falhas irritantes no computador, seu impacto pode ser mais sério (SILVA, et al., 2020). Um artigo de 2005 da Wired sobre os 10 piores bugs de software da história relatou que os bugs causaram grandes explosões, aleijaram sondas espaciais e mataram pessoas. Por exemplo, em 1982, um sistema - supostamente implantado pela Agência Central de Inteligência - controlando o gasoduto Transiberiano causou a maior explosão não nuclear da história (WIRED, 2005).

O artigo também disse que, entre 1985 e 1987, um bug, chamado de condição de raça, em um dispositivo de radioterapia resultou na entrega de doses letais de radiação, matando cinco pessoas e ferindo outras. Em 2005, a Toyota fez um recall de 160.000 Priuses porque um bug fez com que as luzes de advertência acendessem e o motor parasse sem motivo (WIRED, 2005).

Um bug é apenas um tipo de problema que um programa pode ter (BELLER, et al., 2018). Os programas podem ser executados sem bugs e ainda ser difíceis de usar ou falhar em algum objetivo principal. Esse tipo de falha é mais difícil de testar. Um programa bem projetado desenvolvido usando um processo bem controlado resulta em menos bugs por milhares de linhas de código. É por isso que é importante incluir a usabilidade nos testes (LUNA, 2022).

Bell, et al. (2018), relata sobre diferentes tipos de bugs que causam mau funcionamento dos computadores, como: Aritmética. Às vezes referidos como erros de cálculo, bugs aritméticos são erros matemáticos no código que fazem com que ele não funcione. Interface. Um bug de interface ocorre quando sistemas incompatíveis estão conectados ao computador. O problema pode vir de uma peça de hardware ou software. Uma interface de programação de aplicativos pode ser um exemplo de um bug de interface. Lógica. Esses erros acontecem quando a lógica do script faz com que o programa produza as informações erradas ou fique preso e não forneça nenhuma saída. Um exemplo de um erro lógico é um loop infinito onde uma sequência de código é executada continuamente. Sintaxe. Esses bugs



vêm de código escrito com os caracteres errados. Linguagens de programação diferentes têm sintaxes diferentes, portanto, usar a sintaxe de uma pode causar um bug em outra. Equipe. Este é um bug que surge quando há falta de comunicação entre os programadores. Um exemplo é quando há diferenças entre a documentação do produto e o produto. Outro exemplo é quando os comentários descrevem incorretamente o código do programa.

Já Beller et. al., (2015), cita que outra maneira simples de categorizar bugs é da perspectiva do usuário. Esses tipos de bug incluem o seguinte:

Visual. Um usuário pode concluir a função escolhida, mas algo parece errado com o aplicativo. Isso pode ser um problema com o design responsivo do aplicativo. Funcional. Um bug funcional significa que o programa não funciona como pretendido. Por exemplo, um usuário clica no botão Salvar, mas os dados não são salvos. Breu et. al., (2010), cita que os bugs também podem ser classificados pelo nível de dificuldade que causam ao usuário: Bugs de baixo impacto têm um efeito mínimo na experiência do usuário. Os de alto impacto afetam algum nível de funcionalidade, mas o aplicativo ainda é utilizável. Bugs críticos impedem a funcionalidade principal do aplicativo. Cendron e Annes (2014), cita outra abordagem para a classificação de bugs que é observar onde eles ocorrem: Bugs no nível da unidade são bugs de software simples contidos em uma unidade de código. Eles são tipicamente devido a erros de cálculo ou lógica e lidam com um pedaço de software. Eles geralmente são fáceis de consertar. Bugs no nível do sistema são bugs mais complexos causados por vários softwares interagindo de maneiras que causam problemas.

Silva et al., (2020), menciona que os bugs fora de limite surgem quando o usuário interage com o programa de uma maneira inesperada. Por exemplo, isso acontece quando um usuário insere um parâmetro em um campo de formulário que o programa não foi projetado para manipular. Bugs fora da conexão podem ser usados para explorar software (BREU, et. al., 2010). Por exemplo, os agentes de ameaças usam a vulnerabilidade Infra: Halt para realizar ataques de envenenamento de cache do sistema de nomes de domínio na tecnologia operacional.

Segundo Cendron e Annes (2014), várias maneiras de resolver bugs, dependendo do tipo de bug e onde e quando eles são encontrados. A melhor maneira de resolver erros de programação é através da prevenção (LEITÃO, 2016). O uso de um processo de desenvolvimento de software sólido, como as metodologias Agile e DevOps, pode impedir que bugs aconteçam. O teste de qualidade está embutido nessas metodologias de desenvolvimento.

Uma dessas práticas de desenvolvimento, é o desenvolvimento orientado a testes (ROCHA, et. al., 2017). Os testes devem ser criados antes que um recurso seja codificado para fornecer um padrão contra o qual codificá-lo. Outra prática recomendada é usar o desenvolvimento orientado por comportamento, que incentiva os desenvolvedores a codificar um aplicativo e documentar o processo com base em como um usuário deve interagir com ele (LEITÃO, 2016).

Os desenvolvedores podem impedir que os bugs cheguem aos usuários testando com antecedência e frequência. Juntamente com o teste de software, uma revisão de código por pares com outros desenvolvedores, um desenvolvedor sênior ou uma equipe de garantia de qualidade (QA) pode ser útil. (ROCHA, et. al., 2017), cita que o benchmarking ou o teste de benchmark estabelecem as expectativas de desempenho de linha de base para o software em diferentes tipos de cargas de trabalho. Os testes de benchmark podem avaliar a estabilidade, a capacidade de resposta, a velocidade e a eficácia do software.

Bugs que podem ficar dormentes sob um conjunto de condições podem causar um problema sério em outros (DO AMARAL; SIRQUEIRA, 2022). O teste de benchmark pode ajudar a identificar esses bugs. Alguns tipos de benchmarking são os seguintes: O benchmarking de carga avalia os sistemas de software sob uma carga específica, que geralmente é a quantidade usual de tráfego esperada para um aplicativo (COMIS, et. al., 2022).

O benchmarking Spike avalia o desempenho do software durante um aumento súbito na carga de trabalho. O benchmarking de ponto de interrupção empurra um software para ver quanto estresse ele pode lidar antes de travar (LEITÃO, 2016).

Rocha et. al., (2017), cita que se um bug for encontrado no software, ele deverá ser depurado. A depuração envolve as três etapas a seguir: isolando o bug; determinando a causa raiz; corrigindo o problema.

A depuração é um processo de programação de computador para encontrar e resolver erros em software ou em um site, muitas vezes referidos como bugs. Muitas vezes, requer um procedimento abrangente para identificar a razão pela qual um bug ocorreu e desenvolver estratégias para garantir que um programa possa ser executado sem problemas para os usuários no futuro (COMIS, et. al., 2022).

Desenvolvedores e engenheiros de software geralmente depuram programas usando uma ferramenta digital para exibir e editar a linguagem de codificação, que contém instruções sobre como um programa funciona. A depuração permite que eles abordem

seções individuais de código para garantir que cada parte de um programa opere de maneira esperada e ideal (CENDRON; ANNES, 2014). Rocha et. al., (2017), cita que:

Ferramenta de depuração: Muitas vezes chamado de depurador, este software permite localizar a área em um código onde o erro ocorre e fazer os ajustes necessários. Muitas ferramentas de software de codificação vêm com ferramentas de depuração integradas.

Exceção: Esse é um evento que altera o fluxo esperado do código de um programa, como tentar abrir um arquivo que não existe em um programa. Em algumas circunstâncias, o programa pode manipular a exceção e continuar a ser executado, mas se a exceção interromper o programa, você normalmente iniciará um processo de depuração.

Interface de programação de aplicativos: Normalmente referida como API, essa ferramenta permite que diferentes linguagens de codificação operem em um programa. Por exemplo, uma API da Internet pode permitir que um usuário faça login em um site usando uma conta de outro.

Compilador: Este é um programa que traduz uma linguagem de codificação em um formato utilizável, dando-lhe a capacidade de depurar o programa. Ele pode identificar alguns erros no código e oferecer soluções, mas você pode precisar de uma ferramenta de depuração para resolvê-los. Variável: Uma variável refere-se a como você categoriza o armazenamento no código de um programa. Você pode organizar variáveis no código para executar ações diferentes, como usar um para um conjunto de números para incluí-lo em uma linha de código.

Valor: Isso representa o nome específico que você atribui a uma variável depois de criá-la. Por exemplo, o valor de um conjunto de números pode ser "informações da folha de pagamento".

Função: Uma função descreve como as variáveis se traduzem em ação no código de um programa. As funções permitem que um programa opere da maneira que os desenvolvedores e engenheiros de software pretendem.

Ponto de interrupção: As pessoas normalmente usam pontos de interrupção ao investigar uma exceção ou um evento semelhante durante o processo de depuração, pois você pode usá-los para impedir que o código seja executado em seções que contenham um erro identificado anteriormente. Isso pode economizar o tempo do programador, pois eles não precisam esperar que seções de código que não estão relacionadas ao erro sejam processadas.

Entrada: Isso se refere à linguagem de codificação que instrui um programa a concluir uma determinada ação ou função. Você pode inserir uma linguagem de codificação em um compilador para ver se um programa funciona.

Saída: Isso representa os dados codificados que um programa cria depois de executá-lo usando um compilador e normalmente parece diferente da entrada devido a como as funções no código interagem. As pessoas geralmente depuram um programa para fazer com que a saída reflita um arranjo esperado de dados.

Pode ser difícil para os programadores que escreveram um pedaço de código refazer seus passos e examinar linhas de código complexas e densas. Um programa de recompensa de bugs é uma maneira de crowdsourcing um esforço de depuração. Com o crowdsourcing, os pesquisadores de segurança de software e hackers éticos são recompensados por encontrar problemas e fornecer relatórios de bugs que reproduzem ou mitigam a vulnerabilidade (DO AMARAL; SIRQUEIRA, 2022).

As organizações que procuram minimizar bugs de software devem equilibrar o número de implementações e reversões de versões de software que fazem. Ao fazer isso, eles garantem que o processo de depuração não atrapalhe um cronograma de lançamento de software consistente. Isso geralmente é o que as organizações que trabalham em um ambiente de desenvolvimento ágil fazem (COMIS, et. al., 2022).

No entanto, alguns bugs chegam ao produto lançado. As equipes de desenvolvimento podem tratar uma versão como parte do processo de depuração, coletando feedback sobre ela, falhando rapidamente e fazendo melhorias (DE AGUIAR, 2016). Uma equipe ou um indivíduo da equipe pode agendar um horário fixo todos os dias para resolver bugs de software (JÚNIOR et. al., 2017). Dessa forma, a coleta de dados sobre bugs e o próprio processo de depuração passam a fazer parte da programação diária. Uma equipe pode usar dados sobre o processo de depuração para estimar quanto tempo uma determinada correção levará e organizar seus esforços de acordo (ROCHA, et. al., 2017).

É impossível corrigir todos os bugs de uma só vez, e leva tempo para coletar os dados necessários para criar estimativas precisas de bugs (SOUZA, 2018). Os programadores diferem em níveis de habilidade e capacidades. E as estimativas de correção de bugs também podem variar entre os programadores que trabalham em diferentes países. Com o tempo, uma equipe pode desenvolver estimativas de benchmark para quantos bugs pode corrigir em um mês (BARICHELO, 2018).

A depuração nunca é perfeita ou completa (GOMES et. al., 2008). Novos bugs sempre aparecem. As equipes de desenvolvimento devem ter como objetivo abordar os bugs de forma eficiente e fornecer valor líquido positivo para as partes interessadas com cada versão de software (DE AGUIAR, 2016). Depois deste capítulo, seria interessante vc entrar com o haskel que vc havia escrito.

## 2.7 reformular capitulos section

## 2.8 Fundamentação teórica

(( todo o item 2.x esta dentro desse capitulo terei que ver como )) .

Haskell é uma linguagem de programação do tipo funcional, de caráter geral, foi difundida em homenagem ao lógico Haskell Curry. Sendo assim, sua estrutura de controle primária é a função; baseada em observações de Haskell Curry e seus descendentes intelectuais. Tendo como último padrão semi-oficial o Haskell 98, com o objetivo de especificar uma versão mínima e portátil da linguagem para a educação com futuras extensões. É por meio da linguagem Haskell que mais se realizam pesquisas nos nossos dias, altamente utilizada no meio acadêmico e relativamente nova, oriunda de outras linguagens funcionais, ao qual podemos citar o Miranda e ML. Ela é baseada em um estilo de programação em que a ênfase deve ser feita em (what) em vez de como deve ser realizado (how). É uma linguagem que tem como propósito solucionar problemas matemáticos, com clareza, e de fácil manutenção em seus códigos, além disso possui uma ampla diversidade de aplicações, contudo apesar de ser simples e amplamente poderosa. É o que veremos em sua aplicação orientada a tipos descritas nas seções seguintes.

## 2.9 Introdução

As mais diversas linguagens que se encontram no mercado, apesar de modernas, evoluem para que possamos utilizar sistemas de tipos mais flexíveis e que forneçam aos programadores a facilidade de escreverem seus programas sem que haja a necessidade de ficarem restritos aos tipos de dados, de forma que eles se sintam à vontade, como se o seu modo de programar se comportasse como uma linguagem não tipada, ou que tenha uma tipagem dinâmica. Desse modo a utilização de sistemas que façam de forma automática

a inferência dos tipos de dados seriam algo interessante ao contrário dos sistemas por verificação de tipos. Se a maioria das linguagens obedecesse à essa perspectiva, sem sobra de dúvidas, teríamos

A utilização dos sistemas de inferência por tipo não acontece no processo de execução de programas. Usarmos os sistemas de inferências por tipos, em seu lugar, nos garante que nossos projetos ganhem uma característica relevante nesse sentido, que, apesar de que ainda traga certos desafios pelo fato de termos que manter a necessária forma de inferência de tipos divisíveis e eficientes. Em concordância a isso designamos o nome de polimorfismo universal, paramétrico, poliformíssimo via-let ou ainda de polimorfismo de Damas-Milner, para a conceituação de um mecanismo que nos forneça criar funções que se comportem de forma idêntica com relação a todos os valores de tipos que são instâncias de um tipo (PEREIRA, 2014).

Mas como veremos em um momento oportuno de nosso trabalho de linguagens que estendem o suporte a polimorfismo paramétrico para chegarmos uma sobrecarga como a que pretendemos aqui que é a linguagem Haskell, isso acarreta situações de incoerência com relação a definições da semântica da própria linguagem pelo fato da indução em derivações do sistema de tipos. Contudo, termos ou não uma política de sobrecarga independente de contexto pode simplificar e muito a resolução de sobrecarga e a detecção dessas ambiguidades de uma maneira mais restrita. Poderíamos citar como exemplo, certas constantes que não podem ser sobrecarregadas e não permitem a sobrecarga, ou seja, funções em que apenas o tipo do valor é retornado e que seja distinta das definições. Isso, em se tratando de uma função de leitura apenas por conversão de valores para strings, numa função `read` conceitua na biblioteca de padrão Haskell (PARK; KIM; IM, 2007). Ela possui tipos elementares e é sobrecarregada na biblioteca em Haskell para os mais variados tipos elementares da biblioteca padrão (`Int`, `Float`, `Bool`, e outros demais).

Como objetivo geral de nosso trabalho torna-se relevante com relação aos algoritmos é que iremos descrever como será realizada automaticamente esses tipos de dados por meio de inferência que possibilitam a definição de símbolos sem que haja a necessidade de termos que declarar uma classe de tipo de dados, levando em contas como poderíamos aplicar essa inferência em tipos de dados, voltado para o tipo de programação em Haskell voltado para operações matemáticas sem fugir aos conceitos da linguagem.

### 2.10 A linguagem de programação Haskell

O paradigma da linguagem de programação Haskell eleva-se a quem deseja aprender sobre ela e para quem vem adotando por meio de um mercado cada vez mais abrangente. Um pré-requisito que, contudo, possui uma lógica de programação de estruturas de dados que fere, e servem para que possamos entender um dos dois exemplos que citaremos. O que abordaremos aqui serve para que possamos entender a facilidade que a matemática oferece para quem possua paixão pelo tema, para seguir os exemplos abordados aqui. A construção da linguagem Haskell, iniciando pelo zero, a partir de sua instalação e pela sua distribuição nos leva ao conceito de Mônadas que é um relevante ao abordarmos essa linguagem.

Iremos avançar no poder dos tipos, utilizando o conceito de polimorfismo paramétrico, a partir das classes de tipos e veremos como exemplo a utilização dos monoides, e ainda abordaremos o problema da introdução informal a respeito da Teoria das categorias, como um conteúdo extra a ser abordado aqui nesse nosso trabalho.

Outra coisa relevante com relação aos algoritmos é que iremos mostrar que é realizada automaticamente esses tipos de dados por meio de inferência que possibilitam a definição de símbolos sem que haja a necessidade de termos que declarar uma classe de tipo de dados. Por outro lado, além dessa facilidade oferecida pela linguagem Haskell, ela possui ainda um front-end composto por um compilador que haja essa implementação e isso complementa o algoritmo pela inferência em relação a outras linguagens de programação.

#### 2.10.1 Tipos com parâmetros

De acordo com a linguagem de programação Haskell, os tipos com parâmetros, ou seja, em inglês: type parameters, são algo que equivale aos generics do Java presente no Haskell. De uma maneira geral é uma forma de implementarmos a definição de polimorfismo paramétrico, isto é, denominaremos de contêiner ou então de caixa um tipo que pode ter outro type parameter, devido a sua semelhança com uma caixa de tipos. Essa caixa poderá guardar um ou mais valores de diversos tipos de dados, sem que haja a necessidade de especificarmos qual é o seu tipo de dados. Podemos ter como exemplo real as listas, que são algo comum no nosso dia a dia como programadores. Estas listas podem ser vistas

como sendo caixas que contenham diversos valores e vários elementos de um mesmo tipo de dados. Por meio dessas definições, poderíamos ter em um mesmo programa lista de dados do tipo inteiros (`int`), de dados booleanos (`bool`), caracteres strings ou (`char`) e demais dados.

A ideia básica aqui é que podemos operar listas, ou demais contêineres sem que haja a necessidade de ter que nos preocuparmos com o que está dentro delas. E a partir do momento em que um certo tipo de dado é inserido ali dentro desse contêiner, nós só precisaríamos estar preocupados com o contêiner em si, e não com o dado que ali foi inserido. Dessa forma, poderemos transformar o que está dentro desta caixa, contudo, não importa quem ou o que está ali dentro. Devemos utilizar agora para mostrar isso, um exemplo bastante simples que ainda diz respeito as listas, de maneira que ao fazermos o ambiente com a definição que acabamos de ver.

De certa forma, representaremos algo que poderemos guardar nenhum, um ou dois elementos de um mesmo tipo de dados: `data Coisa a = umacoisa — duacoisas a — zerocoisas`. Esse tipo que demonstramos anteriormente possui um tipo de parâmetro que significa que qualquer tipo deverá ser passado a essa caixa ou contêiner, qualquer coisa, assim o `value constructor` `duascoisas` carregará dois campos do tipo `a`, ao passo que o `value constructor` `umacoisa` deverá carregar somente um `value`, por último `zerocoisa`, representa apenas um `value constructor` sem nenhum campo.

Como exemplo do que acabamos de ver (`duascoisas`: “Ola” “Mundo!”), ou ainda (`umacoisa`: “true”). Veja que podemos ter caixa de listas de diversos tipos de dados ao mesmo tempo, como por exemplo, algo do tipo “string”, algo do tipo “bool”, algo do tipo “Int”, algo do tipo `person` etc., dessa forma teremos com essas listas diversas coisas que desejarmos e poderemos ficar à vontade, sem que haja a necessidade de definirmos o tipo de dados que ali estarão. O que torna a aplicação relevante é o contêiner `Coisa` e não o que existe ali dentro.

Portanto ao trabalharmos com esses contêineres, o comando `:kind` do `GHCi` deverá identificar a quantidade `types parameters` que existem em seu tipo. Com relação a se existem mais dados, seria mais complexo para se trabalhar com ele. Dessa forma, o `Kind` vê os tipos como se fossem funções de tipos. Por exemplo o `Int` não tem um parâmetro de tipo, e dessa forma seu `Kind` é `*` (e dessa maneira, podemos pensar com um `Kind1`, sendo um abuso da linguagem) por outro lado que a coisa possui um parâmetro de tipo e seu `Kind` é `*  $\Rightarrow$  *` (ou seja, o `Kind 2`, abusaria mais um pouquinho da linguagem). Falando um



pouco mais, por exemplo se declarássemos um tipo: `data FOO a b = FOO a b` e ainda fossemos inspecionar seu `Kind`, chegaríamos ao resultado  $* \Rightarrow * \Rightarrow *$ , visto que `FOO` possui na verdade dois parâmetros, o parâmetro `a` e o parâmetro `b`. E ao realizarmos uma forma de comparação com a linguagem Java, a classe `HshMap` (que é uma estrutura parecida às demais listas que possuem índices de todos os outros tipos), teríamos um exemplo de um tipo `Kind`  $* \Rightarrow * \Rightarrow *$ . Na linguagem Haskell, existem vários tipos de `Kinds`, contudo a princípio devemos ficar atentos que em tipos de `Kind*` e  $* \Rightarrow *$ , ou seja, nas caixas somente podemos guardar elementos do mesmo tipo somente (e dessa forma, tipos distintos não seriam permitidos, como o `Foo`). Se fossemos tentar inspecionar o tipo de duascoisas `True 'A'`, iríamos obter erro de compilação, visto que `duascoisa` tem na verdade dois campos genéricos de mesmo tipo (`strings`). Por outro lado, veja: `prelude>:t (Foo true A) (Foo true 'A')`: `Foo Bool char` esse código acontece sem gerar erros, pelo fato de possuir um contêiner de `kind * \Rightarrow * \Rightarrow *`. E assim podemos usar o conceito de polimorfismo paramétrico para gerar tipos recursivos (ou seja, tipos que possuem certo campo que é próprio) tais como listas ligadas e árvores.

Ao usarmos a definição de polimorfismo paramétrico para que possamos criar os tipos recursivos (ou seja, tipos que tem algum campo sendo do tipo próprio) tais como listas relacionadas a árvores. Sendo que uma árvore binária poderá ser escrita fazendo uso de vários tipos polimórficos, e nesse caso podemos citar como exemplo `Árvore a = nulo — leaf a — Branch a (Arvore a) (Arvore a)` deriving `show`, e assim esse tipo `Arvore` possui três value constructors: `Nulo`, que não possui nenhum campo; `leaf a`, que tem um campo para representar o nó do filho esquerdo de tipo `Arvore a` (e este poderá ser novamente `Nulo`, `Leaf` ou `Branch`) e ainda um campo para representar o nó filho direito de tipo `Arvore a`. Esses tipos de campos representam o elemento a ser colocado na árvore, e posteriormente os campos de tipo `arvore a` tornam uma continuidade da estrutura, nos dois sentidos: esquerda e direita.

Esta linguagem teve início em 1987, segundo [Park, Kim e Im \(2007\)](#) em uma conferência de programação funcional. E nesse evento, um comitê de intelectuais se formou para que criassem um padrão de programação funcional. E que possuísse todas as características de programação de um paradigma funcional realizadas anteriormente, com outros elementos presentes que são: *laziness*, e do fato de ser uma linguagem de programação funcional pura e ainda estaticamente tipada ([PARK; KIM; IM, 2007](#)). Esse conceito que aqui expomos de *laziness* (ou seja, processamento preguiçoso) é o ato de que essa linguagem só calcular expressões que realmente forem necessárias ([HUGHES, 1990](#)). E

dessa forma, isso evita certos processamentos que não são necessários, como por exemplo, a função `++` em Haskell quer dizer concatenação de listas. E dessa forma se tivermos a expressão seguinte: `[3, 6, 7, 3*1089, 0] ++ [-1, 9]` ela produzirá a lista `[3, 6, 7, 3*1089, 0, -1, 9]` sem que haja a necessidade de calcular a expressão `3 * 1089` e isso economizaria tempo de processamento, como era de se esperar. E ainda o processamento neste caso, como em [Hughes \(1990\)](#) e [Park, Kim e Im \(2007\)](#), seria provável ver que, em casos antigos, o conceito de computação preguiçosa e seus efeitos externo (leitura e escrita de arquivos, por exemplo) de forma alguma poderiam coexistir.

”Descrevemos a avaliação preguiçosa no contexto da idiomas, mas certamente um recurso tão útil deve ser adicionado linguagens não funcionais - ou deveria? Pode avaliação preguiçosa e efeitos colaterais coexistem? Infelizmente, eles não podem: Adicionar preguiçoso avaliação para uma notação imperativa não é realmente impossível, mas a combinação tornaria a vida do programador mais difícil, ao invés do que mais fácil. Porque o poder da avaliação preguiçosa depende do programador abrindo mão de qualquer controle direto sobre a ordem em que o partes de um programa são executadas, faria a programação com efeitos colaterais bastante difícil, porque prever em que ordem - ou mesmo. ([HUGHES, 1990](#), p.114).

Como podemos observar seria muito comum em casos mais antigos, esse conceito de computação de preguiçosos e seus efeitos externos como, a leitura e escrita de arquivos, e assim eles poderiam coexistir de forma simultânea.

### 2.11 Descrição de tipos

Quando nos referimos a um módulo `Table`, toda definição é previamente antecipada por uma declaração de tipo. E seus módulos definidos no módulo `Table`, são na verdade polimórficos. Veremos que numa constante `empty` que possui um tipo `Table a` que é um símbolo de tipo `[(string a)]`. Indica que `empty` são usados em situações que necessitam que esses valores de tipos que na verdade são instâncias do tipo `8a [(String a)]`, ou como `[(String Bool)]`, `[(String Int)]` `8a [(String [a])]` etc.

Os tipos funcionais nos levam a outros tipos de parâmetros e implicam em um resultado de função que podem ser do tipo funcional. Assim, um símbolo `Search` que tem a seguinte conotação de tipo `string —Table a —`, que diz respeito a esta função que adquire um parâmetro um valor do tipo `String` e devolve uma função, que implica em uma lista de pares formados por um valor de tipo `String` e um elemento de outro tipo qualquer e devolve como resultado um elemento deste tipo.

Figura 3 – Tipos de erros

```
Prelude> not 'A'

<interactive>:6:5:
  Couldn't match expected type 'Bool' with actual type 'Char'
    In the first argument of 'not', namely 'A'
    In the expression: not 'A'
    In an equation for 'it': it = not 'A'
```

Fonte:PEREIRA, 2014

Podemos afirmar, de maneira não formal, que Search recebe dois parâmetros (sendo um de cada vez), sendo um do tipo String e uma lista de pares. Podemos ressaltar que em anotações de tipos se comportam, em um contexto geral, que não são obrigatórios em programas Haskell, pois o compilador pode interferir no tipo de cada expressão. Isso é denominado de inferência por tipo. Dessa forma, o programador pode fornecer uma anotação de tipo para uma dada expressão, e o compilador averigua se a definição dada pode ter o tipo anotado. Esse processo é denominado de verificação de tipo.

### 2.11.1 Checagem de erros de tipos

A maioria das expressões que possuem uma sintaxe correta possui o seu tipo calculado em tempo de compilação. Caso não seja possível especificar o tipo de uma expressão certamente ocorrerá um erro de tipo. Dessa forma, a aplicação de uma função a um ou mais argumentos de tipos de dados inapropriados ocasionará um erro (PEREIRA, 2014). Veja o exemplo abaixo:

#### EXEMPLO 01

A explicação para esse erro: Temos que a função not necessita de um valor booleano, contudo foi definido ao argumento 'A', que na verdade é um caracter Haskell uma linguagem fortemente tipada, e possui um sistema muito avançado. Assim todos os prováveis erros encontrados em tempo de compilação (tipagem estática). Isso ocasiona que os programas mais seguros e mais rápidos, elimina a necessidade de averiguações de tipo em tempo de execução (PEREIRA, 2014).

#### EXEMPLO 2

Figura 4 – Tipo de erro

```
7 :: Char      -- 7 não pode ser do tipo Char
'F' :: Bool    -- 'F' não pode ser do tipo Bool
not True :: Float -- (not True) não pode ser do tipo Float
min (4::Int) 5.0 -- (4::Int) e 5.0 tem tipos incompatíveis
```

Fonte:PEREIRA, 2014

### 2.11.2 Erros por bugs causados por código ausente

Outras falhas muitas vezes são resultadas de código ausente, os bugs em sua maioria podem ser oriundos da falta de uma expressão explícita no código fonte ao contrário do que se poderia pensar como sendo um erro em suas expressões. [Pearson \*et al.\* \(2016\)](#) dizem que para 30% das situações, uma correção de bug ocorre pelo fato de o programador adicionar um novo código ao invés de mudar o código ali existente. Por outro lado, [Wong \*et al.\* \(2016\)](#) afirma que, apesar de que o bug seja causado por uma parte ausente no código fonte original, as vezes por um canto que não foi tratado, existem técnicas para a localização de erros que podem ser bastante relevantes para chamar a atenção de partes prováveis de onde se encontra o bug, para verificarmos as anomalias de fluxo de controle. [Pearson \*et al.\* \(2016\)](#) averiguou os diferentes métodos de erros em relação aos casos de relacionados à código ausente, levando em conta que a diretriz para estas situações seriam realizar um reporte a declaração logo a seguir.

Segundo este autor, o ideal a ser feito pelo programador é inserir o código, e dessa forma, as técnicas de localização dos erros seriam capazes de inserir outro código que fosse o correto e que causou aquele bug no código fonte.

### 3 Estado da Arte

[Sinha e Hanumantharya \(2005\)](#), propuseram uma nova abordagem de construção de softwares tolerantes a falhas, por meio de construções modulares de categorias de tipos. Esses softwares deveriam ser baseados em componentes, haja vistas que essa forma de modulação está crescendo a cada dia, desenvolvimento de aplicativos de sistema com confiança utilizando para isso componentes pré-validados e testados. Sua utilização não deve ser realizada de uma forma direta, deve haver um método que preconize essa utilização de maneira que se possa construir um software composto com base em componentes que enlacen outros componentes de softwares que suportem tipos particulares de falhas.

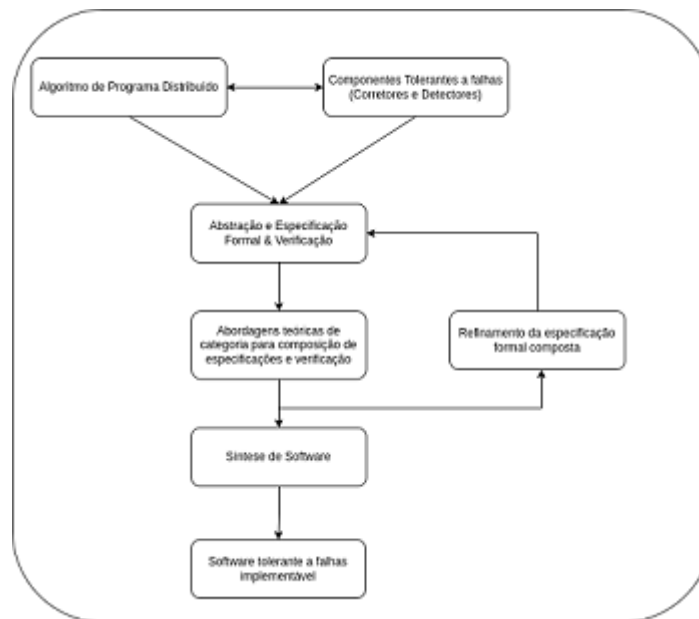
Assim sendo, [Sinha e Hanumantharya \(2005\)](#) se propuseram a desenvolver um programa composto tolerante a falhas, para isso utilizaram a ferramenta formal mecanizada q 2004 Elsevier Bv. Porém, antes de adentrarmos nesta análise do eles desenvolveram vamos analisar um pouca a categoria exibida que se apresenta como sendo uma ferramenta poderosa na formalização computacional, que nessa abordagem nos fornece ferramentas para que se possa ter um raciocínio modular sobre categorias de estruturas multicomponentes, que é justamente o que [Sinha e Hanumantharya \(2005\)](#) afirmam na composição de seu programa.

#### *3.1 Uma base que compõe os elementos de software*

[Sinha e Hanumantharya \(2005\)](#) afirmam que os princípios gerais de seu trabalho são: primeiro identificar os cenários de erros em programas/algoritmo de modo a realizar uma distribuição do seu estudo e construir elementos tolerantes a falhas de modo apropriado e seguindo os rumos de sua proposta. Assim, o programa, juntamente com os elementos são dessa maneira especificados em uma especificação formal e por abstraírem e depois averiguarem.

Após a especificação formal abstrata eles passam a verificar os componentes de softwares individualmente, fazendo uso de conceitos da teoria das categorias de tipos para formarem o programa e seus elementos tolerantes a falhas, implicando em um software único composto que possa lidar com tipos peculiares de erros. E dessa forma, a especificação

Figura 5 – Estrutura da composição



Fonte: Sinha e Hanumantharyab (2005)

composta que ora resulta desse programa tolerante a falhas é novamente verificada para poder garantir que o software satisfaça os requisitos de confiabilidade determinados.

Sinha e Hanumantharya (2005) ainda completam que a estrutura ainda permite que haja um refinamento posterior sobre a especificação de componentes compostos individuais. E isso pode ainda colaborar para que a estrutura seja enriquecida com maiores detalhes adicionais em termos de estrutura das operações. E a cada nova fase desse refinamento posterior essa verificação só tende a garantir cada vez mais que não haja inconsistências no programa. E assim, ao fazerem uso da síntese de softwares, para um dado grau de abstração, esses autores obtiveram um código implementável que seria tolerante a falhas em razão da abordagem correta atribuída na sua construção.

E essa abordagem correta foi proposta com a utilização da ferramenta Specware do Kestrel Institute. Sendo que fazendo uso de uma especificação que tenha sido refinada de maneira correta, o Specware pode suportar elementos de biblioteca padrão para poderem converter os componentes (elementos) em alguma linguagem de programação executável usando o gerador de código.

### 3.2 Cruzamento dos artigos e dissertações analisadas

Nesta seção, apresentamos o cruzamento dos temas abordados, comparando quais artigos versam sobre os temas encontrados. Separamos a tabulação em doze temas principais e veremos quais os artigos melhor se encaixaram para que pudéssemos atingir nosso problema de pesquisa e nossos objetivos. As doze assuntos principais encontrados nos artigos foram os seguintes, com suas respectivas siglas abreviadas:

- 1 - Categoria de Tipos (CT);
- 2 - Tolerante a Falhas (TF);
- 3 - Eventos/Token (ET);
- 4 - Teoria dos Tipos (TT);
- 5 - Indução (IND);
- 6 - Álgebra Data Type (ADT);
- 7 - Categoria Exibida (CE).

Tabela 1 – Cruzamento dos temas dos artigos

<b>Estudo</b>	<b>CT</b>	<b>TF</b>	<b>ET</b>	<b>TT</b>	<b>IND</b>	<b>ADT</b>	<b>CE</b>
<a href="#">Sinha e Hanumantharya (2005)</a>	X	X	X				
<a href="#">Altucher e Panangaden (1990)</a>				X	X		
<a href="#">Arkor e Fiore (2020)</a>					X	X	
<a href="#">Johann e Polonsky (2020)</a>	X				X	X	X
<a href="#">Ahrens e Lumsdaine (2017)</a>	X				X		
<a href="#">Berg e Moerdijk (2018)</a>	X				X	X	X
<a href="#">Kraus (2021)</a>	X						
<a href="#">Balteanu <i>et al.</i> (2003)</a>	X						
<a href="#">Mitchell e Moggi (1991)</a>					X		
<a href="#">Sassone e Sobociński (2005)</a>							X
<a href="#">Altenkirch e Kaposi (2016)</a>					X		
<a href="#">Amato, Lipton e McGrail (2009)</a>						X	X
<a href="#">Kracht (2006)</a>				X		X	
<a href="#">Moriya (2012)</a>				X			
<a href="#">Hoffman (2016)</a>					X		

Fonte: Elaborada pelo autor (2022)

Os artigos utilizados foram [Sinha e Hanumantharya \(2005\)](#): Uma nova abordagem para desenvolvimento de software tolerante a falhas baseado em componentes, [Altucher e Panangaden \(1990\)](#): Uma prova construtiva mecanicamente assistida na teoria das categorias, [Arkor e Fiore \(2020\)](#): Modelos algébricos de teorias de tipo simples: Uma abordagem

polinomial, [Johann e Polonsky \(2020\)](#): Indução profunda: regras de indução para tipos (verdadeiramente) aninhados, [Ahrens e Lumsdaine \(2017\)](#): Categorias exibidas, [Berg e Mordijk \(2018\)](#): Conclusão exata de categorias de caminho e conjunto algébrico teoria: Parte i: conclusão exata das categorias de caminho, [Kraus \(2021\)](#): Modelos categóricos internos da teoria do tipo dependente, [Balteanu \*et al.\* \(2003\)](#): Monoidal terado categorias: Avanços em Matemática, [Mitchell e Moggi \(1991\)](#): Modelos de estilo ripke para cálculo lambda digitado, [Sassone e Sobociński \(2005\)](#): Localizando a reação com 2 categorias, [Altenkirch e Kaposi \(2016\)](#): Normalização por avaliação para tipos dependentes, [Amato, Lipton e McGrail \(2009\)](#): Sobre a estrutura algébrica da declarativa linguagens de programação, [Kracht \(2006\)](#): álgebras parciais, significando categorias e algebrização, [Moriya \(2012\)](#): A teoria da homotopia de Ram e a categoria graduada diferencial, e ainda os autores: [Hoffman \(2016\)](#): Categorias updown: Gerando funções e coberturas universais.

Os aspectos referentes a essa proposta [Sinha e Hanumantharya \(2005\)](#) podem ser vistos como uma forma de realizar um refinamento do programa de modo que haja uma filtração do software até que não sejam encontradas mais falhas que causem erros de tipos de dados ou falhas no programa final. E desse modo, encontramos nesse artigo em particular algo que compensou a nossa análise e proporcionou um ganho significativo em termos de qualidade em nossa pesquisa e que assim pudesse contribuir para que a pesquisa não se tornasse em em si meramente apresentação de vagas teorias.

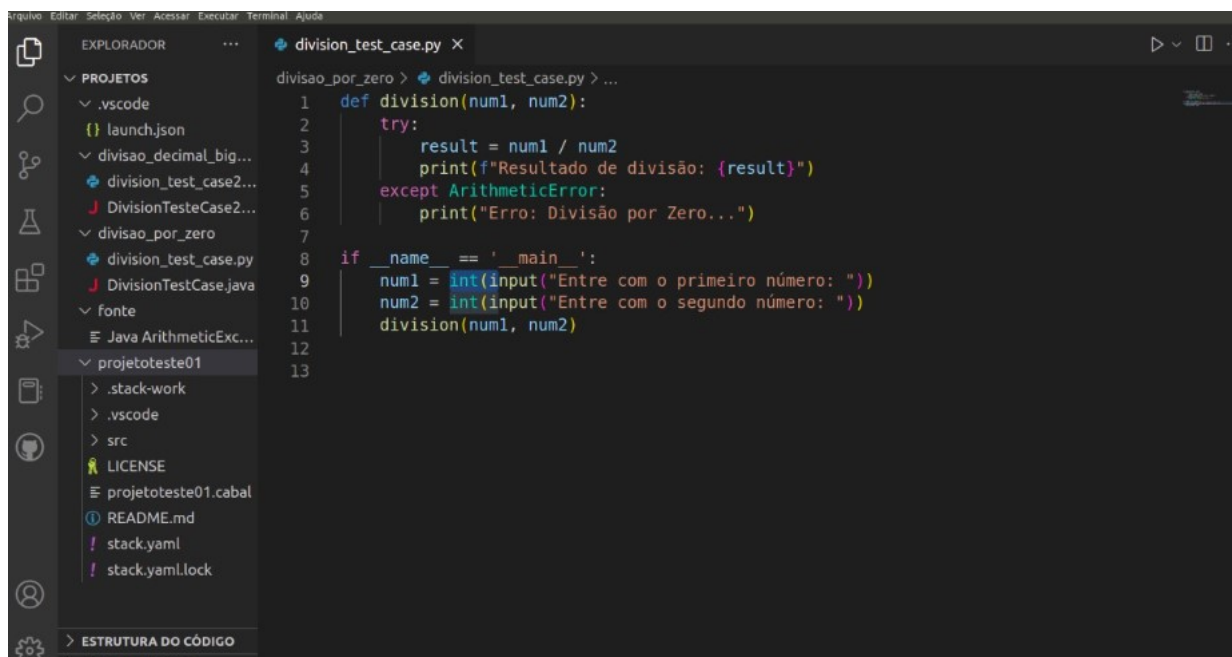
### 3.3 Algumas considerações sobre uma estrutura de aplicação de tipos de erros

Vejamos o caso típico de um simples programa que gera um erro ao efetuarmos uma divisão por zero, na linguagem python. Na realidade é apenas um teste para vermos as funcionalidades e podermos verificar em qual linguagem haveria um melhor tratamento do erro com pouca codificação, ele ficaria de acordo com a figura abaixo:

Na linguagem java o código ficaria mais ou menos dessa forma, veja que o tamanho já se estende um pouco mais do que seria necessário:

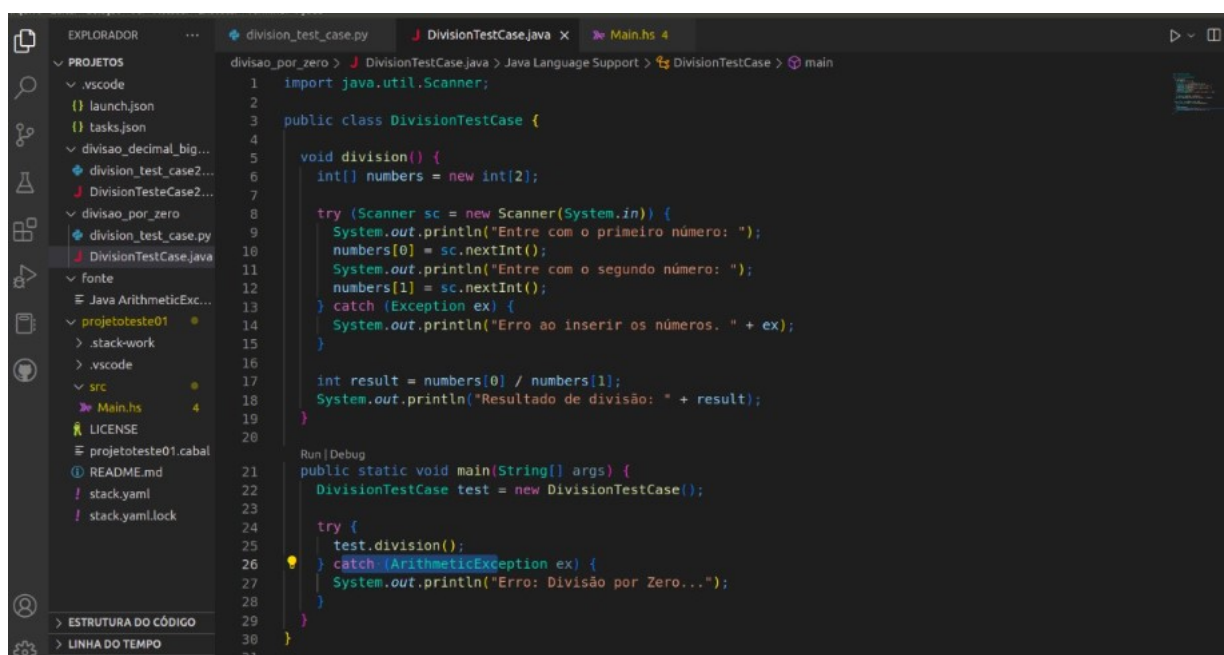


Figura 6 – Programa erro divisão por zero em python



Fonte: Elaborado pelo autor (2022)

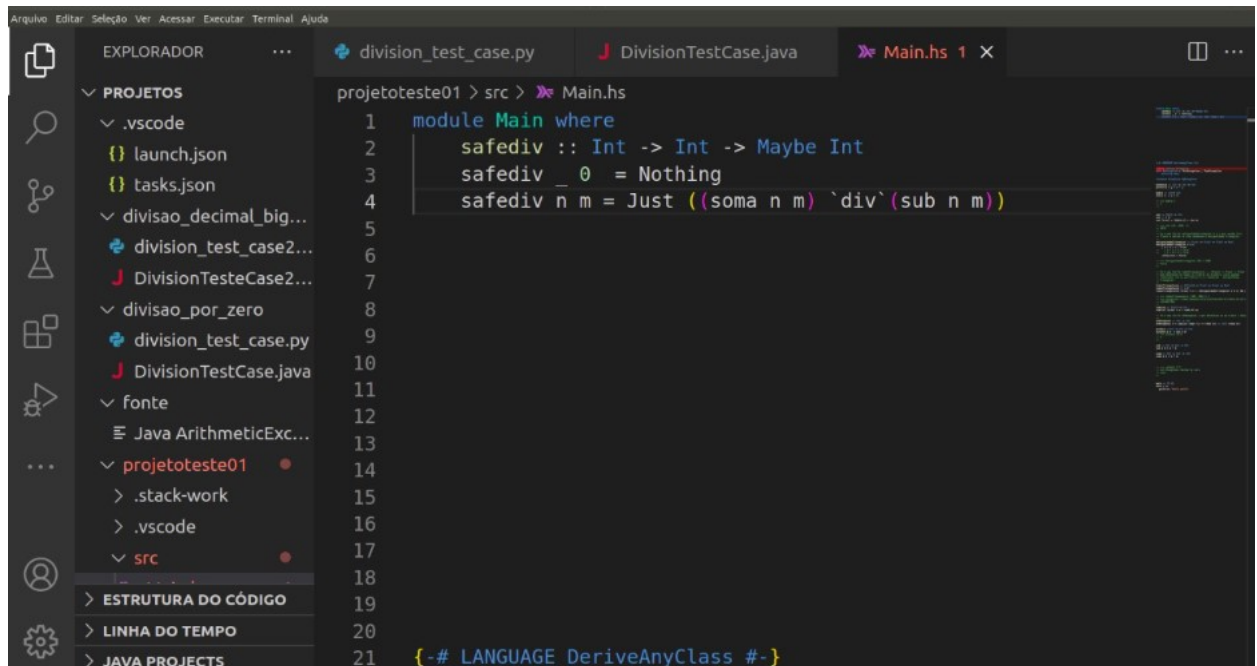
Figura 7 – Programa erro divisão por zero em Java



Fonte: Elaborado pelo autor (2022)

Já na linguagem de programação Haskell, o código ficaria bem reduzido, ocasionando menos trabalho e resultaria numa mesma solução viável para a mesma aplicação, vejamos:

Figura 8 – Programa erro divisão por zero em Haskell



```
projetoteste01 > src > Main.hs
1  module Main where
2      safediv :: Int -> Int -> Maybe Int
3      safediv _ 0 = Nothing
4      safediv n m = Just ((soma n m) `div` (sub n m))
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21  {-# LANGUAGE DeriveAnyClass #-}
```

Fonte: Elaborado pelo autor (2022)

Vimos dessa forma, que o programa se torna simples, quando utilizamos uma linguagem de programação fortemente tipada e que em poucas linhas de código podemos criar a nossa aplicação de forma que ela funcione corretamente e faça o mesmo que outras linguagens fariam, a diferença é que utilizamos menos linhas de código e isso facilita e muito o trabalho do desenvolvedor.

### 3.4 Conclusão

Foi surpreendente o resultado ao qual chegamos, e existe uma razão bastante relevante para essa situação, haja vista que tínhamos como objetivo geral avaliar uma linguagem de programação que fosse altamente tipada de forma a oferecer e reduzir os tipos de erros em programação, bem como as possíveis falhas, concluímos que por meio dos artigos analisados houve uma extrema lacuna que nos impediu de atingir este objetivo.

Por mais que esses 17 artigos versassem sobre o tema escolhido para nossa pesquisa, o que mais nos motivou e nos apresentou algo satisfatório para o que necessitávamos, foi o artigo de [Sinha e Hanumantharya \(2005\)](#), no qual nos apresentou um programa que pudesse criar outros softwares que seriam totalmente tolerante a falhas, e neste caso seria justamente do que necessitávamos. Isto se deu pelo fato de que esses autores que acabamos de citar por meio de reinserções e reescrita do software, chegaram a um programa que

fosse tolerante a falhas vindo a realocar nesse ínterim programas que possuísem falhas, e desta maneira conseguissem se ver livres delas.

Contudo, o que mais nos surpreendeu foi o fato de existir esse grande lacuna em nossa pesquisa, mas de tudo ela não foi necessariamente em vão e decerto conseguimos atingir alguns dos objetivos propostos. Tendo em vista a grande abstração que a maioria dos outros artigos impunham, como tipos de categorias, análise de teoremas algébricos, isso seria mais uma forma de teorizar do que trazer algo útil para nossa pesquisa e dessa forma, como os autores [Ahrens e Lumsdaine \(2017\)](#) que trata das categorias exibidas, que por meio de um conjunto  $C$  obtivemos um elemento  $c$  derivado deste conjunto que está baseado em raciocínio modular sobre as categorias de estruturas multicomponentes, e assim esses autores formalizaram em Coq baseado na biblioteca UniMath, e que deu origem a uma outra biblioteca pronta para ser utilizada posteriormente pelos desenvolvedores.

Mas algo certo e digno de nota é que a linguagem de programação Haskell, que é uma linguagem das quais mais analisadas e muito nova no meio acadêmico, advinda de outras linguagens funcionais, tornou-se uma linguagem baseada em estilo de programação em que se enfatiza em buscar uma razão de como deve ser feita uma tarefa (what) e de como deve ser realizada (how).

Por ser uma linguagem que possui a função de sanar, e tornar fácil a solução de problemas matemáticos, com clareza, e tornar simples a manutenção em seus códigos, possuindo uma gama enorme e diversa de aplicações, mesmo sendo simples é largamente poderosa.

Assim, chegamos à conclusão de que a utilização de sistemas de inferência por tipo, não ocorre no processo de execução de programas. E se assim, utilizássemos sistemas de inferências por tipo, em seu lugar, isso iria nos garantir que nossos projetos ganhassem uma característica relevante no sentido de que, mesmo que ainda não traga certos desafios pelo fato de termos que continuar com a forma necessária de inferência de tipos divisíveis e eficientes ([PEREIRA, 2014](#)).

Ao utilizarmos uma linguagem de programação fortemente tipada e que realize uma inferência de tipos de dados, fica evidente que as chances de ocorrência de falhas se torna cada vez mais rara de ocorrer. Uma linguagem que se aproxime de forma mais próxima da linguagem humana, como é o caso da linguagem de programação Haskell, contorna inúmeros problemas com relação à erros de digitação uma vez que o código é provavelmente menor e exige poucos termos técnicos e palavras reservadas que confundiriam o raciocínio

do desenvolvedor, pelo fato de não ter que indicar os tipos de dados de que se trata por exemplo as variáveis.

Assim, a melhor forma para detectarmos os tipos de erros ou bugs mais comuns na programação orientada a tipos, conforme exposto acima e que trata de um dos nossos objetivos propostos no Apêndice A, como sendo uma das melhores alternativas que encontramos, é a utilização de linguagens fortemente tipadas, que exigem menos códigos de programação e que possuem inferência de tipos de dados.

E com relação aos tipos de metodologias que poderiam ser utilizadas para a redução de erros de programação orientada a tipos, podemos citar a metodologia utilizada e propostas pelos autores [Sinha e Hanumantharya \(2005\)](#) por meio do refinamento de dados até que não se encontrassem mais erros no software produzido e que ele pudesse ser um programa tolerante à falhas apesar de que o sistema fosse tolerante à falhas.

Como forma de encerrarmos aqui o nosso trabalho, deixamos abaixo dessa pesquisa uma imensa lista de referências bibliográficas que poderão de alguma forma contribuir para futuros trabalhos de outros acadêmicos seguindo essa mesma temática que nos propusemos a discutir e mostrar em nossa pesquisa. Decerto, como já mencionado, houve uma lacuna em nossa pesquisa, mas por outro lado de uma certa forma ela foi imperativa aos objetivos propostos e culminou em lançar mão dos recursos que tínhamos a disposição. Outros acadêmicos poderão dessa forma encontrar razões satisfatórias para dar continuidade a pesquisas semelhantes, haja vistas que as dissertações e artigos sobre o tema pouco são discutidos.

Assim esperamos que esta seja um incentivo e não um empecilho para eles. Espera-se que nossa pesquisa possa ter contribuído e muito para a comunidade acadêmica e que estudos futuros sobre o mesmo tema possam ser iniciados.

## Referências

- AHRENS, B.; LUMSDAINE, P. L. Displayed categories. *arXiv preprint arXiv:1705.04296*, 2017. Citado 3 vezes nas páginas 38, 39 e 42.
- ALTENKIRCH, T.; KAPOSÍ, A. Normalisation by evaluation for dependent types. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. [S.l.], 2016. Citado 2 vezes nas páginas 38 e 39.
- ALTUCHER, J. A.; PANANGADEN, P. A mechanically assisted constructive proof in category theory. In: SPRINGER. *International Conference on Automated Deduction*. [S.l.], 1990. p. 500–513. Citado na página 38.
- AMATO, G.; LIPTON, J.; MCGRIL, R. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, Elsevier, v. 410, n. 46, p. 4626–4671, 2009. Citado 2 vezes nas páginas 38 e 39.
- ARKOR, N.; FIORE, M. Algebraic models of simple type theories: A polynomial approach. In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. [S.l.: s.n.], 2020. p. 88–101. Citado 2 vezes nas páginas 7 e 38.
- ARORA, S.; AGRAWAL, C.; SASIKALA, P.; SHARMA, A. Developmental approaches for agent oriented system—a critical review. In: IEEE. *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*. [S.l.], 2012. p. 1–5. Citado 2 vezes nas páginas 6 e 8.
- BALTEANU, C.; FIEDOROWICZ, Z.; SCHWÄNZL, R.; VOGT, R. Iterated monoidal categories. *Advances in Mathematics*, Elsevier, v. 176, n. 2, p. 277–349, 2003. Citado 2 vezes nas páginas 38 e 39.
- BERG, B. van den; MOERDIJK, I. Exact completion of path categories and algebraic set theory: Part i: Exact completion of path categories. *Journal of Pure and Applied Algebra*, Elsevier, v. 222, n. 10, p. 3137–3181, 2018. Citado 2 vezes nas páginas 38 e 39.
- GIL, A. C. *Métodos e técnicas de pesquisa social*. [S.l.]: 6. ed. Editora Atlas SA, 2008. Citado na página 9.
- HOFFMAN, M. E. Updown categories: Generating functions and universal covers. *Discrete Mathematics*, Elsevier, v. 339, n. 2, p. 906–922, 2016. Citado 2 vezes nas páginas 38 e 39.
- HUGHES, J. Functional pearl global variables in haskell. 1990. Citado 2 vezes nas páginas 32 e 33.
- JOHANN, P.; POLONSKY, A. Deep induction: Induction rules for (truly) nested types. In: *FoSSaCS*. [S.l.: s.n.], 2020. p. 339–358. Citado 2 vezes nas páginas 38 e 39.
- KRACHT, M. Partial algebras, meaning categories and algebraization. *Theoretical Computer Science*, Elsevier, v. 354, n. 1, p. 131–141, 2006. Citado 2 vezes nas páginas 38 e 39.

KRAUS, N. Internal-categorical models of dependent type theory: Towards 2l<sub>tt</sub> eating hott. In: IEEE. *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. [S.l.], 2021. p. 1–14. Citado 2 vezes nas páginas 38 e 39.

MITCHELL, J. C.; MOGGI, E. Kripke-style models for typed lambda calculus. *Annals of pure and applied logic*, Elsevier, v. 51, n. 1-2, p. 99–124, 1991. Citado 2 vezes nas páginas 38 e 39.

MORIYA, S. The de rham homotopy theory and differential graded category. *Mathematische Zeitschrift*, Springer, v. 271, n. 3, p. 961–1010, 2012. Citado 2 vezes nas páginas 38 e 39.

PARK, S.; KIM, J.; IM, H. Proceedings of the 12th acm sigplan international conference on functional programming. 2007. Citado 3 vezes nas páginas 29, 32 e 33.

PEARSON, S.; CAMPOS, J.; JUST, R.; FRASER, G.; ABREU, R.; ERNST, M. D.; PANG, D.; KELLER, B. *Evaluating & improving fault localization techniques*. University of Washington Department of Computer Science and Engineering, Seattle, WA. [S.l.], 2016. Citado na página 35.

PEREIRA, V. F. Paralelismo na linguagem haskell. 2014. Citado 3 vezes nas páginas 29, 34 e 42.

SASSONE, V.; SOBOCIŃSKI, P. Locating reaction with 2-categories. *Theoretical Computer Science*, Elsevier, v. 333, n. 1-2, p. 297–327, 2005. Citado 2 vezes nas páginas 38 e 39.

SINHA, P.; HANUMANTHARYA, A. A novel approach for component-based fault-tolerant software development. *Information and Software Technology*, Elsevier, v. 47, n. 6, p. 365–382, 2005. Citado 7 vezes nas páginas 7, 36, 37, 38, 39, 41 e 43.

WONG, W. E.; GAO, R.; LI, Y.; ABREU, R.; WOTAWA, F. A survey on software fault localization. *IEEE Transactions on Software Engineering*, IEEE, v. 42, n. 8, p. 707–740, 2016. Citado na página 35.