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 desenvolveimento orientado a tipo para evitar e mitigar erro operacional

São Paulo
2022

JEAN PIERRE DE BRITO

Estudo sobre a eficacia do desenvolveimento orientado a tipo para evitar e mitigar erro operacional

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

Lista de tabelas

Sumário

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 responsável 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

O que realmente nos motivou para a confecção deste trabalho foi realmente o desejo de apresentar uma questão que é 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.

1.2 Justificativa

Com relação á justificativa de nossa proposta, poderíamos dizer que uma parcela de seu aspecto já foi delimitado nas linhas acima e em poucas palavras. Contudo, há algo mais profundo envolvido nisso. 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.

1.3 Objetivos

Sendo que o objetivo geral é avaliar uma linguagem de programação que seja altamente tipada de forma a oferecer e reduzir os tipos de erros em programação, bem como as possíveis falhas.

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 - Descrever projetos de autores renomados que incentivem ao mínimo a ocorrência de tipos de erros ocorridos nas linguagens de programação.

1.3.2 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 Gil (2008):

> 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 Scorpus.

O método empírico é algo que se apoia apenas em experiências vividas, por meio da obervação 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.

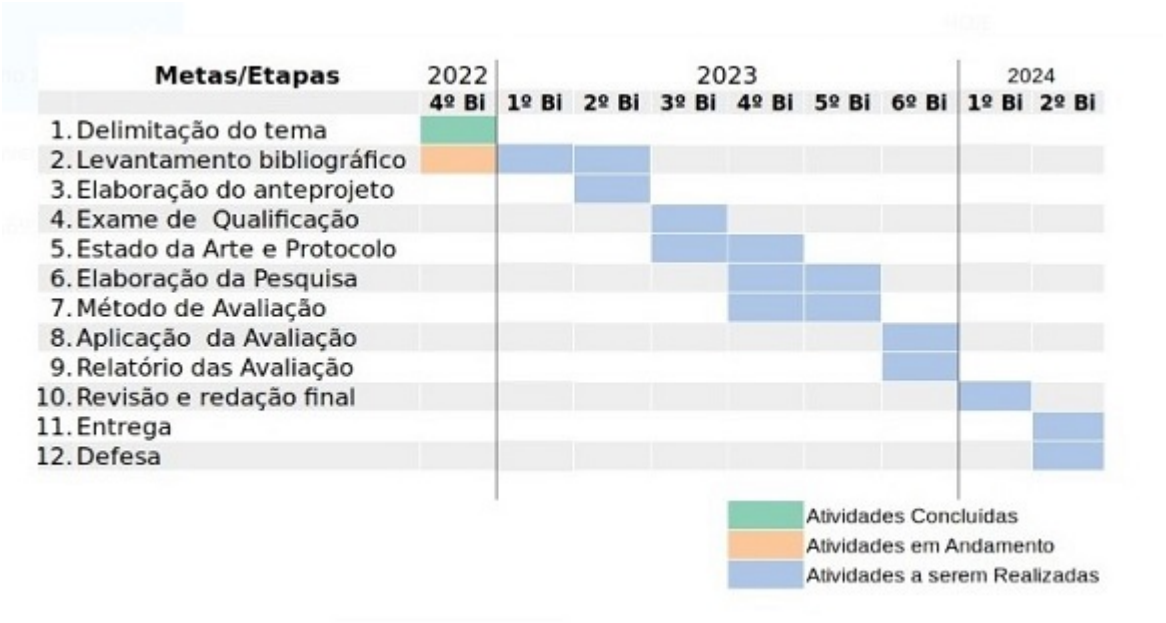
1.4 Estrutura da pesquisa

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.

1.5 Cronograma

A estimativa para elaboração do projeto é de janeiro de 2022 até junho de 2023 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. 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. 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 tabela 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

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 razão 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 é amplamente poderosa. É o que veremos em sua aplicação orientada a tipo descritas nas seções seguintes.

2.1 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 obedecessem 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 apena 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.

2.2 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.

2.2.1 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 pessoa 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 parameteres 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 * ⇒* (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 *⇒*⇒*, 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 *⇒*⇒*. Na linguagem Haskell, existem vários tipos de Kinds, contudo a princípio devemos ficar atentos que em tipos de Kind* e *⇒*, 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', iriamos 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*⇒*⇒*. 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 vário 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: $[3, 6, 7, 3 * 10^{89}, 0] + +[-1, 9]$ ela produzirá a lista $[3, 6, 7, 3 * 10^{89}, 0, -1, 9]$ sem que haja a necessidade de calcular a expressão $3 * 10^{89}$ 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.

2.3 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 funciona. 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.

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.

Figura 2 – 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

2.3.1 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 3 – 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

2.3.2 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 enlacem outros componentes de softwares que suportem tipos particulares de falhas.

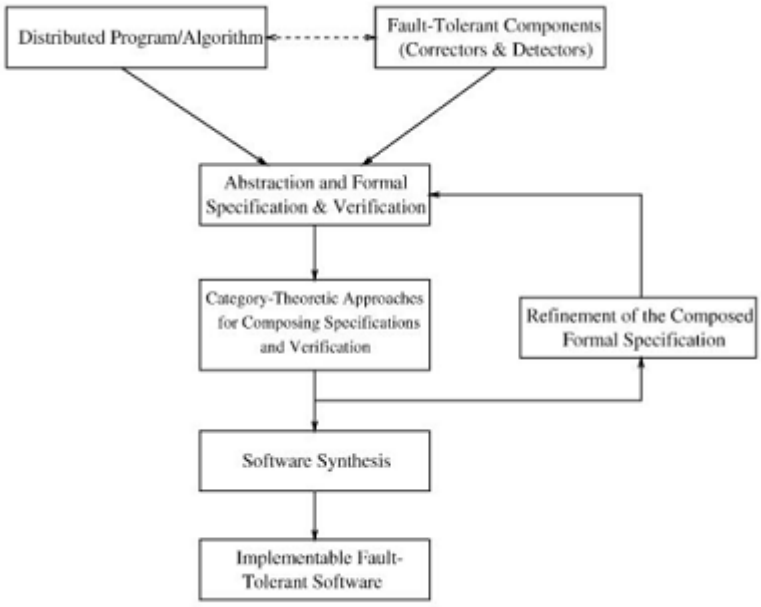
Assim sendo, Sinha e Hanumantharya (2005) se propuseram a desenvolver um programa composto tolerante a falhas, para isso utilizaram a ferramenta formal mecanizada q 2004 Elsevier Bv. Porém, antes de adentrarmos nesta análise do eles desenvolveram vamos analisar um pouca a categoria exibida que se apresenta 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.

3.0.1 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 4 – Estrutura da composição



Fonte: Sinhaa 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 convertes os componentes (elementos) em alguma linguagem de programação executável usando o gerador de código.

3.0.2 Cruzamento dos artigos e dissertações analisadas

Nesta seção, presentamos 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 abreviativas:

1 - Categoria de Tipos (CT);

2 - Tolerante a Falhas (TF);

3 - Eventos/Token (ET);

4 - Teoria dos Tipos (TT);

5 - Pura Matemática (PM);

6 - Indução (IND);

8 - Álgebra Data Type (ADT);

9 - Categoria Exibida (CE).

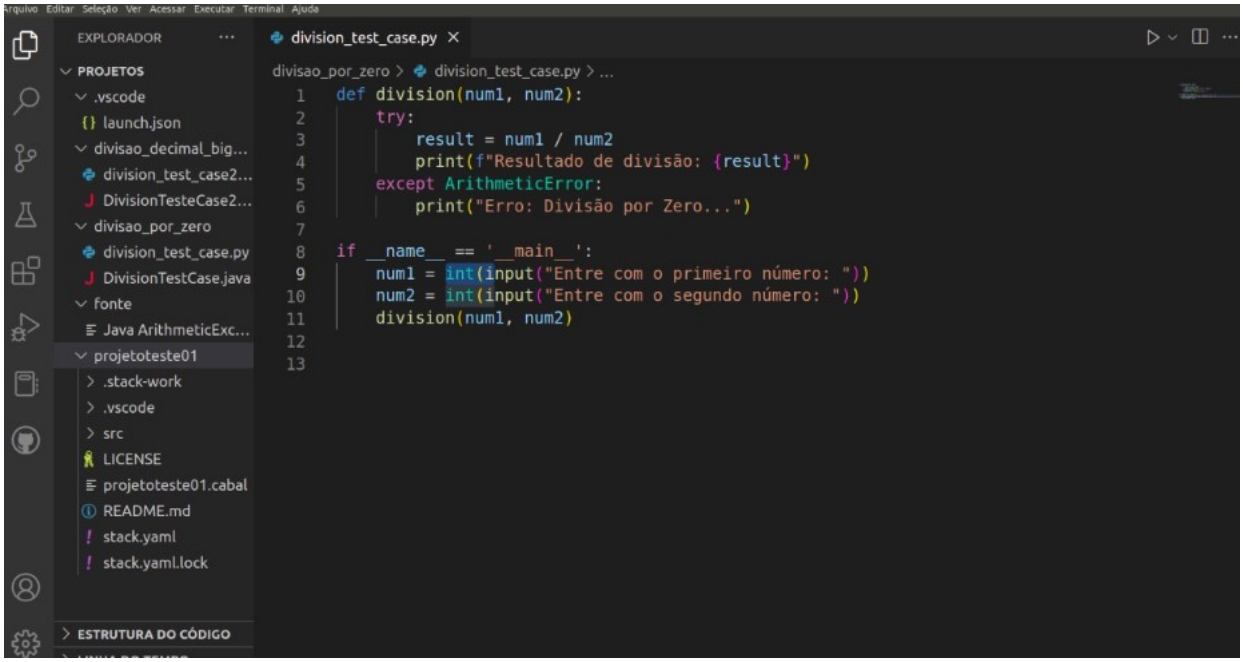
Tabela 1 – Cruzamento dos temas dos artigos

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

Fonte: Elaborada pelo autor (2022)

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.

Figura 5 – Programa erro divisão por zero em phyton



Fonte: Elaborado pelo autor (2022)

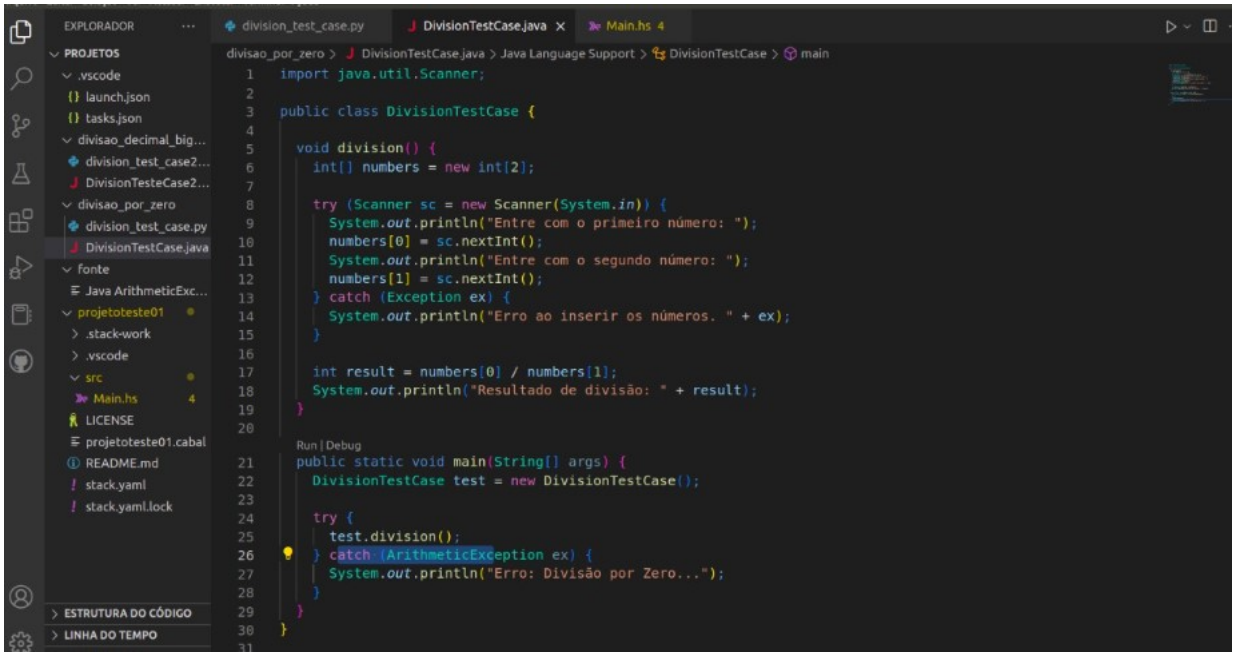
E desse modo, encontramos nesse artigo em particular algo que compensou a nossa análise e proporcionou um ganho significativo em termos de qualidade em nossa pesquisa e que assim pudesse contribuir para que a pesquisa não se tornasse em em si meramente apresentação de vagas teorias.

3.1 Algumas considerações sobre uma estrutura de aplicação de tipos de erros

Veja no caso de um exemplo de um simples programa que gera um erro ao efetuarmos uma divisão por zero, na linguagem phyton. 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:

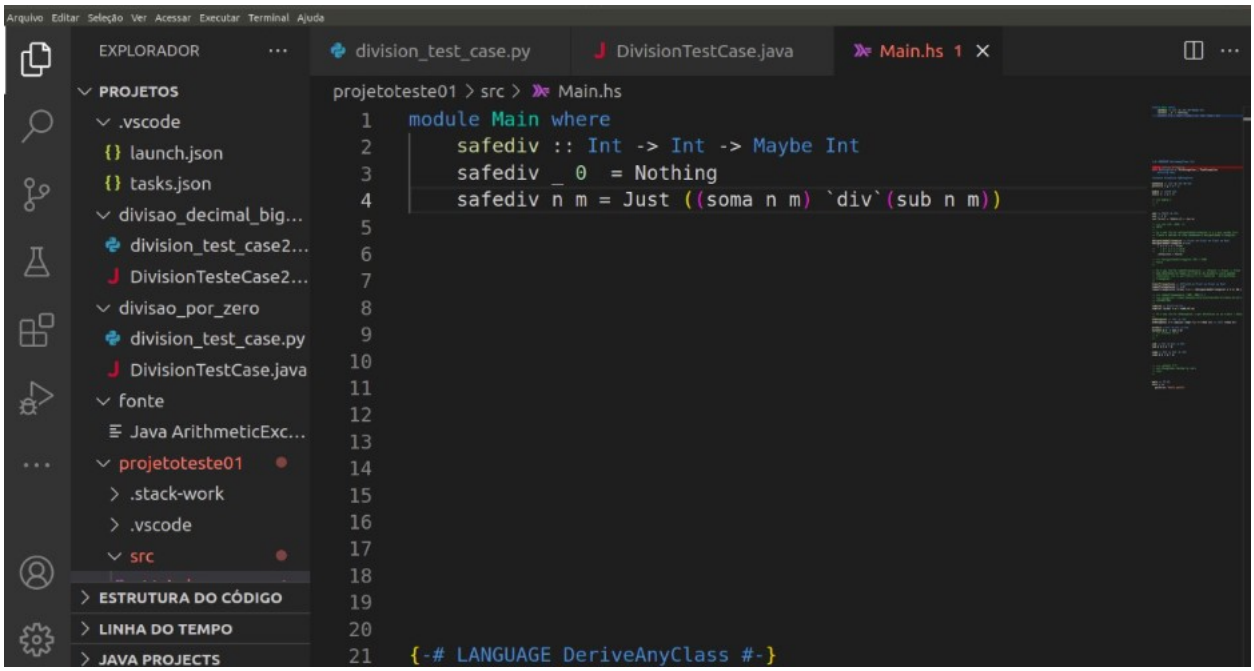
Figura 6 – Programa erro divisão por zero em Java



Fonte: Elaborado pelo autor (2022)

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

Figura 7 – Programa erro divisão por zero em Haskell



Fonte: Elaborado pelo autor (2022)

Vimos dessa forma, que o programa se torna simples, quado 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.

3.2 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 possuíssem 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, qeu 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 trabalh, 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 artigo 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.

APÊNDICE A - Protocolo De Revisão De Escopo

A.1 O tema da pesquisa e objetivos da pesquisa

Não temos dúvidas de que o tema de uma pesquisa é de suma importância ao pensarmos em uma revisão bibliográfica e foi escolhido a programação orientada a tipos para se evitar erros (ou bugs) em operações.

O objetivo geral de nossa pesquisa é analisar quais são as formas mais usuais nos métodos de programação para evitarmos tipos de erros em operações, tendo como objetivos específicos determinar os tipos de erros mais comuns dentro da programação de forma a evitá-los durante o processo de desenvolvimento de aplicações, analisar os tipos de erros comuns, mensurar erros lógicos, erros de tipagem e conduzir de forma coerente a pesquisa em seus aspectos teóricos e práticos.

A.2 Questões da pesquisa da revisão

1) Como detectar os tipos de erros ou bugs mais comuns na programação orientada a tipos?

2) Quais os fatores realmente influenciam no acometimento de erros em programação orientada a tipos?

3) Quais os tipos de metodologias poderiam serem usadas para a diminuição de erros em programação orientada a tipos.

A.2.A Da motivação para a realização das questões da pesquisa

Existem diversos fatores que nos motivam a falar sobre um tema ou assunto em uma pesquisa científica. A motivação para a escrita do tema surgiu da necessidade de termos um conjunto de ferramentas para que pudéssemos apresentar de uma forma prática e condizente tipos de erros em operações em programação Haskell que nos permita realizar um projeto de programação, cometendo o mínimo de erros possível de forma a tornar mais rápida e eficaz o trabalho do programador no seu cotidiano da programação e que facilitasse futuras adaptações ou outras correções caso fosse necessário.

O tema nos motivou por se tratar de um assunto que se encontra intimamente relacionado ao cotidiano dos programadores e desenvolvedores de softwares e aplicações em geral, em termos de back-end e front-end. Observamos que a maioria dos programadores encontram diversos erros em suas aplicações no momento da programação e também depois quando a aplicação já deve estar no poder do consumidor final e ainda se torna que nesta etapa ainda existam erros que não deveriam existir tornando-se necessária uma boa consideração e análise desses erros para que isso não ocorra.

A.2.B O que esperamos desta nossa pesquisa

Esperamos responder à nossa pergunta de pesquisa que é: como apresentar de uma forma prática e condizente tipos de erros em operações em programação Haskell que nos permita realizar um projeto de programação, cometendo o mínimo de erros possível de forma a tornar mais rápida e eficaz, visando contribuir para a comunidade acadêmica de forma a lançar luz sobre os problemas mais comuns em tipos de erros encontrados nas mais diversas linguagens de programação, de forma que se possa evitá-los e que eles sejam os mínimos erros possíveis contribuindo de maneira eficaz no cotidiano dos mais de 400 mil programadores existentes no país.

A.2.C Estratégias da para responder a pergunta de pesquisa

Neste caso, tivemos uma combinação de métodos descritos que em estudos qualitativos e ainda utilizados por meio de métodos quantitativos, que segundo Combinação de métodos. A estratégia utilizada para a solução do problema de pesquisa foi uma revisão da literatura que versa sobre assuntos semelhantes ao que é proposto nessa nossa pesquisa. Esses resultados corroboraram para que descobríssemos semelhanças ao que já havíamos esperado para que fosse o resultado de nosso trabalho.

Contudo, certos desses artigos não nos auxiliaram para que chegássemos à conclusão alguma e por esse motivo foram descartados pois se referiam a assuntos distintos do nosso tema, não fazendo parte do escopo que pretendíamos abordar. Os métodos acima descritos podem ser combinados em estudos qualitativos ou utilizados em conjunto com métodos quantitativos. Segundo Flick (2009) trata-se de uma combinação denominada de

triangulação, que foi pensada, de começo, como uma estratégia para fornecer uma validação de resultados peculiares, porém, atualmente é vista como uma maneira de complementar o saber e superar os eventuais limitados métodos limitados àqueles que eram totalmente individuais.

A.3 String de busca canônica e os detalhes da estratégia usada para sua criação e aplicação

TITLE-ABS-KEY (category AND type AND theory AND develop) AND (LIMIT-TO (DOCTYPE , "ar") OR LIMIT-TO (DOCTYPE , "cp") OR LIMIT-TO (DOCTYPE , "re")) AND (LIMIT-TO (SUBJAREA , "COMP") OR LIMIT-TO (SUBJAREA , "MATH") OR LIMIT-TO (SUBJAREA , "BUSI") OR LIMIT-TO (SUBJAREA , "ENGI") OR LIMIT-TO (SUBJAREA , "PHYS") OR LIMIT-TO (SUBJAREA , "CENG") OR LIMIT-TO (SUBJAREA , "CHEM")) AND (LIMIT-TO (LANGUAGE , "English")).

Essa string traduzida significa isso: TITLE-ABS-KEY (category AND type AND theory AND develop). Document type (Article, Conference Paper e review) Subject Area(Ciência da Computação, Matemática, Negócios, Gestão e Contabilidade,Engenharia, Física e Astronomia, Engenheiro químico, Química)

A.4 As principais bases de dados utilizadas

Bases de dados utilizadas: Utilizamos a base do Scopus pela a estrategia de ser a mais usada, pela sua facilidade de uso e ainda por ser a mais recomendada pelos mestres e doutores. Sendo ela a nossa única base de dados para a referenciação de nossa pesquisa, cuja url é esta: https://www.scopus.com/home.uri. Este site é a base para a construção do nosso referencial teórico bem como nossa fonte de dados para que a pesquisa seja realizada com referências atualizadas e confiáveis.

Todas essas palavras-chave foram as buscas realizadas na plataforma Scopus para que de certa forma chegássemos aos resultados esperados para a nossa pesquisa.

Com relação à estratégia utilizada em nossa pesquisa, procuramos de alguma forma fixar critérios que fossem plausíveis de confiança e digno de nota, como por exemplo: autores renomados, sites e plataformas confiáveis, além de outros requisitos que nos oferecessem confiança. Foram feitas pesquisas nas mai variadas fontes que versam sobre o assunto

e averiguamos que tudo tende a se resumir em palavras chaves de grande valor e teor conteudista de mérito para a nossa averiguação.Todos os conteúdos aqui observados tendem a nos direcionar para conclusões claras e acertadas que nos remetem a tendências que são plenamente visíveis e dignas de confiança, tendo indícios claros de que são verídicos e de confiança.

Não pretendemos aqui nos aproximar de algo que seja tangível ou que seja norteador de algo próximo de ser certeiro, contudo nos referimos ao estudo que desejamos defender e nos fixar em termos de que seja como certo e plausível de nota. Pois se desejamos escrever algo deve estar em consonância com a nossa realidade ser é claro verdadeiro e digno de confiança, observando os critérios de confiança da comunidade científica.

A.5 Estratégia de seleção dos artigos

Nas subseções seguintes apresentaremos os critérios de inclusão e exclusão dos artigos para a elaboração de nossa pesquisa.

A.5.A Critérios de inclusão

Somente iríamos incluir em nossa pesquisa artigos bases gratuitos, ou seja, que estivessem disponíveis para download.

A.5.B Critérios de exclusão

Foram excluídos de nossa pesquisa aqueles artigos que não estivessem disponíveis para download, ou seja, aqueles que fosse necessário pagar para obter o download e também artigos duplicados, que por algum motivo apareceram dois ou mais artigos iguais nas nossas Strings de busca.

A.8 Estratégia planejada para extração de dados e para síntese de resultados

Utilizei a estrategia de extração via o download do site Scopus e cópias da abas dos filtros, haja vistas que nelas havia uma limitação da quantidade de registros retornados

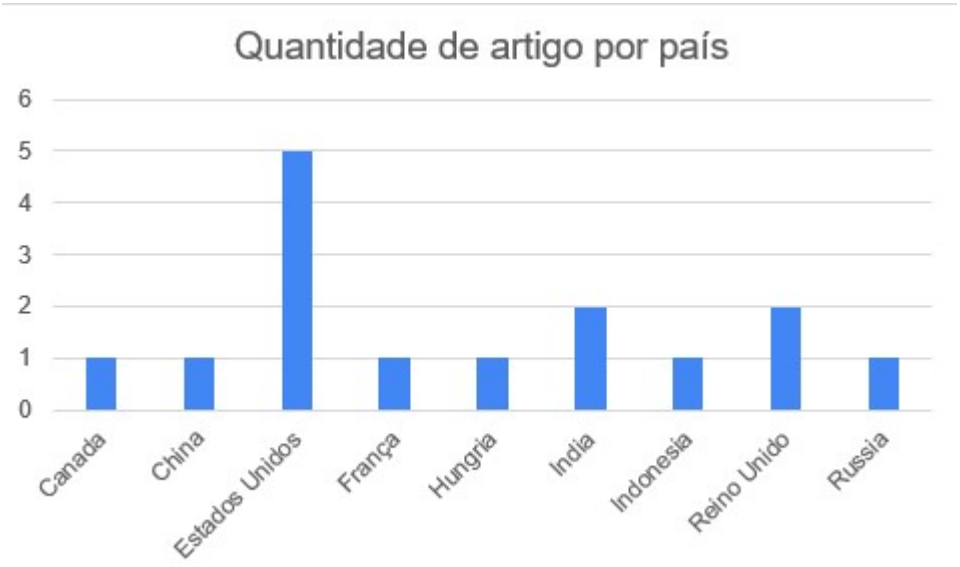
pelo filtro. E em alguns casos como o país de origem, não estavam sendo feitas com sucesso pelo site scopus via download csv.

APÊNDICE B - Em relação à condução da pesquisa

B.1 Relato da execução do procedimento

Para seleção dos estudos candidatos (incluindo informações referentes às datas de execução da strings, filtros aplicados nos mecanismos de busca, número de registros retornados); data de extração 14/10/2022. Foram aplicados os filtros mencionados no apêndice A DOCTYPE e SUBJAREA e foram encontrados 328 resultados.

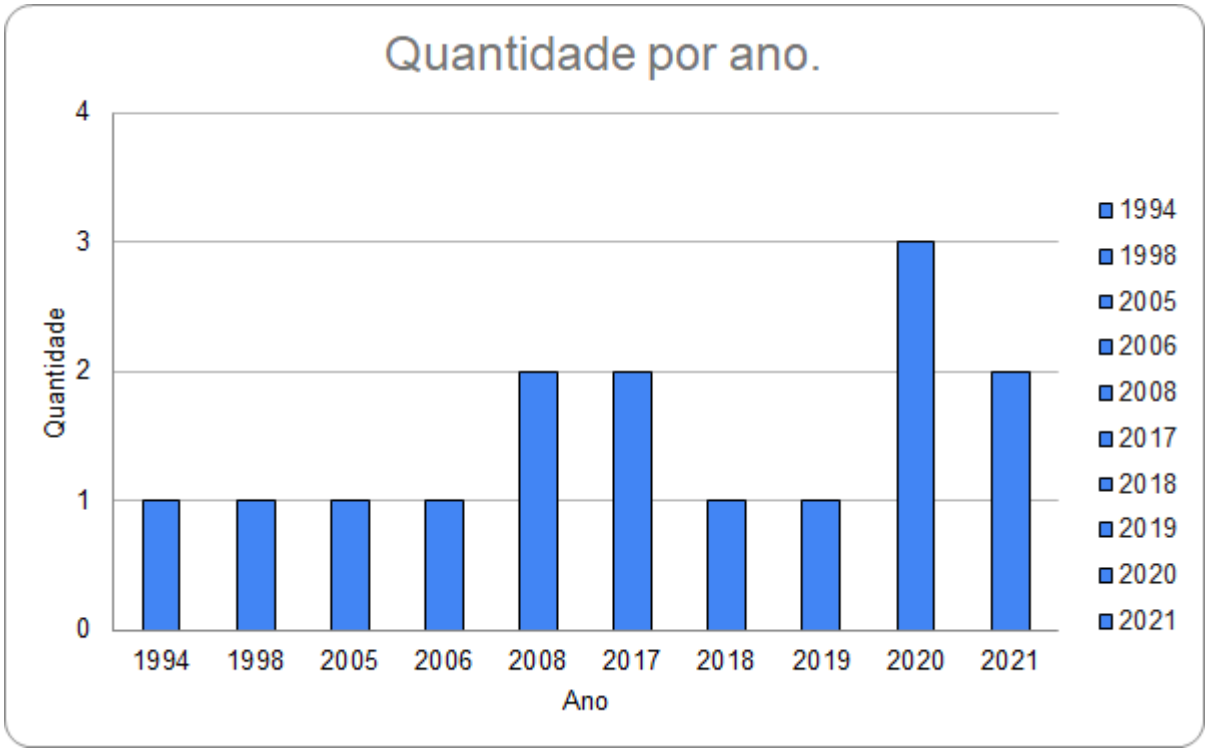
Figura 8 – Gráfico quantidade de artigos por país



Fonte:Elaborada pelo autor

Este gráfico apresentado na Figura 1, apresenta os países que mais investem em tecnologia e o que podemos observar é que eles fazem parte dos países do primeiro mundo, sendo o principal deles os Estado Unidos, ocupando a primeira posição, seguido da Índia e Reino Unido como podemos observar.

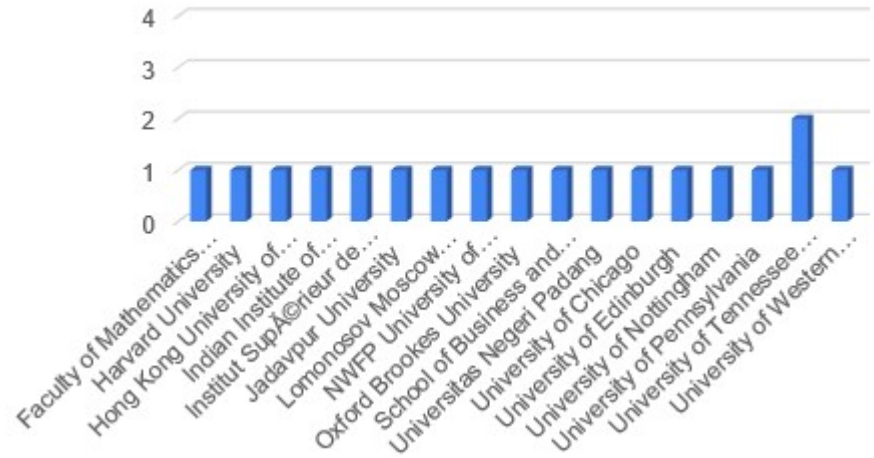
Figura 9 – Gráfico quantidade versus anos pesquisados



Fonte:Elaborada pelo autor

Nesta Figura 2 apresentamos o gráfico com a quantidade de artigos compartilhados no decorrer dos anos. Podemos notar que no ano de 2020 houve um aumento expressivo da quantidade de compartilhamentos e decresceu um pouco no ano de 2021, mas a tendência é de que esses compartilhamentos cresçam cada vez mais.

Figura 10 – Gráfico quantidade de artigos por filiação

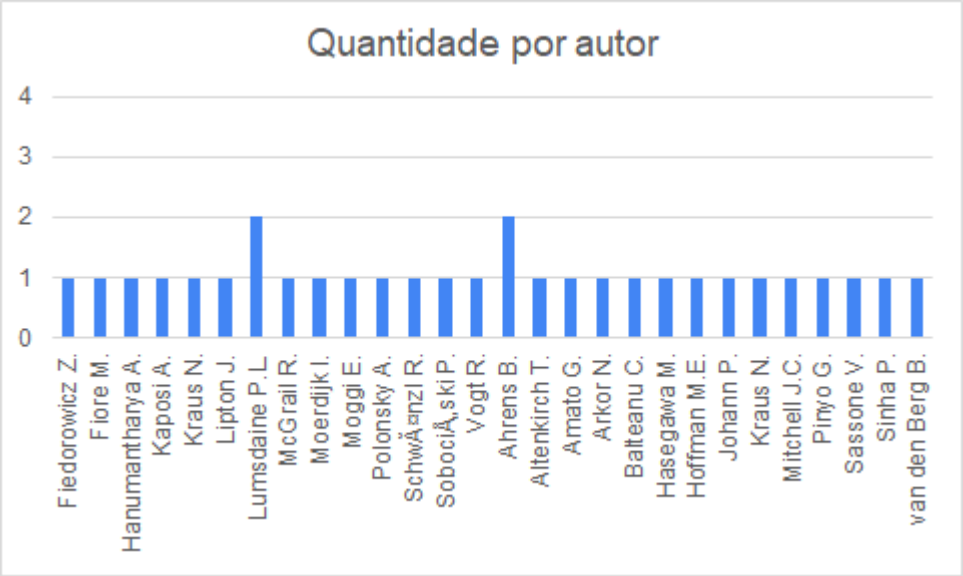


Fonte:Elaborada pelo autor

Na Figura 3 apresentamos o gráfico onde são mostrados a quantidade de artigos por filiação que incentivaram a produção acadêmica e sua publicação. O grande problema

encontrado ocorreu no momento da busca no site da Scopus, visto que ele não apresentava as universidades locais. Dessa forma foi realizada uma busca de alguns dados manualmente linha por linha copiando os dados das universidades.

Figura 11 – Gráfico quantidade por autor



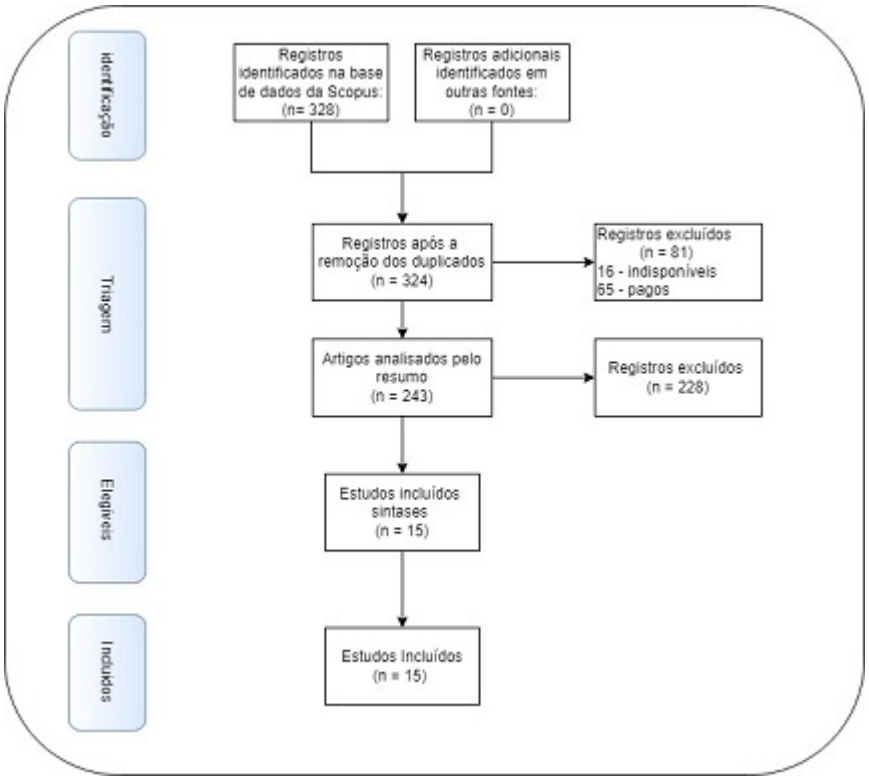
Fonte:Elaborada pelo autor

Notamos no gráfico da Figura 4 os autores que colaboraram para a confecção dos artigos, e as vezes um mesmo autor participou na confecção de mais de um artigo, haja vistas que a quantidade de autores apresentada na no gráfico da Figura 6 é bem menor que a quantidade de artigos encontrados (328). Observamos também que não está tendo mais demanda para a confecção desses artigos, pelo fato de não haver somente um autor específico para cada assunto, geralmente são mais de um autor para a confecção de um único artigo ou periódico.

B.2 Resumo dos dados em figura

No diagrama da Figura 5 podemos facilmente observar que a quantidade de artigos encontrados na base de dados da Scopus foi de 328 artigos, sendo que foram utilizados os critérios de exclusão para a seleção dos artigos a serem analisados. Inicialmente excluímos 4 artigos pelo fato de que estes se encontravam duplicados por algum motivos no momento da busca por meio das Strings. Posteriormente, segundo os critérios de exclusão, mais 81 artigos foram excluídos, sendo que 16 por não estarem disponíveis para download e 65 por serem artigos pagos. E finalmente, 228 artigos foram ainda excluídos pois analisamos esses artigos e percebemos que eles não abordavam critérios específicos da Engenharia de software que de alguma forma pudesse contribuir para a nossa pesquisa, restando portanto 15 artigos elegíeis.

Figura 12 – Diagrama exclusão



Fonte:Elaborada pelo autor

Referências

AHRENS, B.; LUMSDAINE, P. L. Displayed categories. *arXiv preprint arXiv:1705.04296*, 2017. Citado 2 vezes nas páginas 21 e 24.

ALTENKIRCH, T.; KAPOSI, 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 na página 21.

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 21.

AMATO, G.; LIPTON, J.; MCGRAIL, R. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, Elsevier, v. 410, n. 46, p. 4626–4671, 2009. Citado na página 21.

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 6 e 21.

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 5 e 7.

BALTEANU, C.; FIEDOROWICZ, Z.; SCHWÄNZL, R.; VOGT, R. Iterated monoidal categories. *Advances in Mathematics*, Elsevier, v. 176, n. 2, p. 277–349, 2003. Citado na página 21.

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 na página 21.

FLICK, U. *Qualidade na pesquisa qualitativa: coleção pesquisa qualitativa*. [S.l.]: Bookman editora, 2009. Citado na página 28.

GIL, A. C. *Métodos e técnicas de pesquisa social*. [S.l.]: 6. ed. Ediitora Atlas SA, 2008. Citado na página 8.

HOFFMAN, M. E. Updown categories: Generating functions and universal covers. *Discrete Mathematics*, Elsevier, v. 339, n. 2, p. 906–922, 2016. Citado na página 21.

HUGHES, J. Functional pearl global variables in haskell. 1990. Citado 2 vezes nas páginas 15 e 16.

JOHANN, P.; POLONSKY, A. Deep induction: Induction rules for (truly) nested types. In: *FoSSaCS*. [S.l.: s.n.], 2020. p. 339–358. Citado na página 21.

KRACHT, M. Partial algebras, meaning categories and algebraization. *Theoretical Computer Science*, Elsevier, v. 354, n. 1, p. 131–141, 2006. Citado na página 21.

KRAUS, N. Internal-categorical models of dependent type theory: Towards 2ltt eating hott. In: IEEE. *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. [S.l.], 2021. p. 1–14. Citado na página 21.

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 na página 21.

MORIYA, S. The de rham homotopy theory and differential graded category. *Mathematische Zeitschrift*, Springer, v. 271, n. 3, p. 961–1010, 2012. Citado na página 21.

PARK, S.; KIM, J.; IM, H. Proceedings of the 12th acm sigplan international conference on functional programming. 2007. Citado 2 vezes nas páginas 12 e 15.

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 18.

PEREIRA, V. F. Paralelismo na linguagem haskell. 2014. Citado 3 vezes nas páginas 12, 17 e 25.

SASSONE, V.; SOBOCIŃSKI, P. Locating reaction with 2-categories. *Theoretical Computer Science*, Elsevier, v. 333, n. 1-2, p. 297–327, 2005. Citado na página 21.

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 6 vezes nas páginas 6, 19, 20, 21, 24 e 25.

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 18.