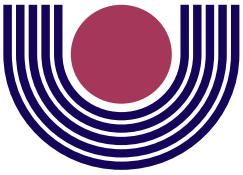


**Unioeste - Universidade Estadual do Oeste do Paraná**
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
***Curso de Bacharelado em Ciência da Computação***

# Variabilidade de Software em Linguagens Funcionais: Um Estudo Exploratório em Haskell

*Clayton Wilhelm da Rosa*

**CASCAVEL**
**2016**

**CLAYTON WILHELM DA ROSA**

# VARIABILIDADE DE SOFTWARE EM LINGUAGENS FUNCIONAIS: UM ESTUDO EXPLORATÓRIO EM HASKELL

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Ivonei Freitas da Silva

CASCAVEL
2016

**CLAYTON WILHELM DA ROSA**

# VARIABILIDADE DE SOFTWARE EM LINGUAGENS FUNCIONAIS: UM ESTUDO EXPLORATÓRIO EM HASKELL

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

---

Prof. Ivonei Freitas da Silva (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

---

Prof. Edmar André Bellorini
Colegiado de Ciência da Computação,
UNIOESTE

---

Prof. Elder Schemberger
Universidade Tecnológica Federal do Paraná,
UTFPR

Cascavel, 20 de dezembro de 2016

„ *Ihr seid verfluchte Hunde!*“

(*Gladiador, 2000*)

「人間渡航積むとツンデレになるですね。」

(*Itoshiki Nozomu*)

“*Tu serás feliz, Bentinho; tu vais ser feliz.*”

(*Machado de Assis*)

“*Only dead fish swim with the stream.*”

(*Anônimo*)

# Lista de Figuras

# Lista de Quadros

# Lista de Tabelas

# Lista de Abreviaturas e Siglas

| | |
|---|---|
| FOSD | *Feature-Oriented Software Development* |
| GoF | *Gang of Four* |
| IDE | *Integrated Development Environment* |
| LF | Linguagem Funcional |
| LP | Linguagem de Programação |
| LPS | Linha de Produtos de Software |
| OO | Orientado(a) a Objetos |

# Sumário

# Resumo

As linhas de produtos de software tem se destacado como um dos mais promissores paradigmas de desenvolvimento de software. Para se obter sucesso no uso das linhas de produtos é necessária uma boa gerência de suas variabilidades. Diversas técnicas que possibilitam a satisfação dessas variabilidades em nível de implementação estão relacionadas às teorias das linguagens de programação, compilação e interpretação de linguagens. Porém, tais técnicas tem sido aplicadas predominantemente por meio de linguagens imperativas, especialmente aquelas orientadas a objeto. Reconhecendo a carência de estudos sobre a implementação de variabilidades em outros paradigmas de programação, este trabalho apresenta um estudo exploratório dessa implementação por meio do paradigma funcional e de seus conceitos básicos, apresentando seus resultados e discussões na forma de *patterns* de software, onde verificamos a satisfação das variabilidades de software pela linguagem funcional Haskell e seus mecanismos.

**Palavras-chave:** Variabilidade de Software, Programação Funcional, Haskell.

# Capítulo 1

# Introdução

Escrever software de qualidade para diferentes domínios é uma tarefa difícil. Desde muito cedo, logo após a introdução das primeiras Linguagens de Programação (LPs) de alto nível, as dificuldades no desenvolvimento de software têm sido discutidas, tendo os primeiros esforços formais em busca de soluções para estas dificuldades iniciado a partir de 1968[1] [3].

Graças a evolução significativa de áreas como a engenharia de software e das LPs, hoje o tempo para projetar e desenvolver software foi consideravelmente reduzido, assim, diversificar e entregar software de qualidade de maneira ágil e eficiente é cada vez mais exigido, e torna-se um desafio para as organizações.

Na última década, as Linhas de Produtos de Software (LPSs) têm se destacado como um dos mais promissores paradigmas de desenvolvimento de software, trazendo flexibilidade e vantagens econômicas à longo prazo [4]. As LPSs baseiam-se no conceito de reúso de artefatos, código, documentação, e requisitos, possibilitando redução no tempo de desenvolvimento, e maior qualidade do software [4, 5].

Em uma LPS, realiza-se uma análise do domínio de aplicação, buscando determinar as características comuns e variáveis dos produtos nesse domínio [1]. Quando uma determinada característica pode ser desenvolvida de diferentes maneiras diz-se que esta define um ponto de variação, que indica a existência de diferenças entre os produtos de uma mesma linha [1, 2].

É fundamental para o sucesso no uso das LPSs uma boa gerência das variabilidades dos produtos em um determinado domínio. Existem diversas técnicas para a implementação de variabilidades, e uma grande parcela destas técnicas está diretamente relacionada às LPs, com-

[1]Neste ano ocorre a *NATO Conference on Software Engineering*, onde o termo engenharia de software é utilizado pela primeira vez.

piladores e interpretadores. Hoje as linguagens Orientadas a Objetos (OOs) são as principais ferramentas nos estudos e implementações de variabilidades de software [6].

Ao longo dos anos, as LPs introduziram novos mecanismos de abstração, novos paradigmas, e se especializaram em diferentes domínios. Ainda assim a adoção de outros paradigmas de programação na implementação de variabilidades é recente e muito pequena, mesmo para um paradigma como o orientado a aspectos[2], inicialmente proposto como complementar ao paradigma OO [7].

Um paradigma cada vez mais popular no desenvolvimento de aplicações científicas e comerciais [8, 9], porém não explorado no contexto da implementação de variabilidades, é o paradigma funcional. As Linguagens Funcionais (LFs) destacam-se por sua abordagem radical à forma de se escrever programas, vendo a computação como um processo de avaliar expressões.

Mesmo com sua maior adoção o paradigma funcional é visto como acadêmico. Em seu artigo "*Why no one uses functional languages*" [10], Philip Wadler elenca e observa diversos fatores que contribuíram para a significativa menor adoção das LFs. Porém, os fatores apresentados no artigo de Wadler, são de modo geral, limitações técnicas que hoje já não se aplicam às LFs e percepções subjetivas sobre o paradigma e suas linguagens.

Estudos como os de Sampson [11] e Ray [12], indicam que as LFs contribuem para a produção de programas mais concisos, mais claros, com menor número de defeitos, e consequentemente mais fáceis de manter. Outros dentre os principais argumentos apresentados pelos defensores das LFs são [13]:

- os programas podem ser escritos mais rapidamente;
- apresentam maior nível de abstração;
- são melhores de serem analisados formalmente; e
- executam mais facilmente em arquiteturas paralelas.

Tendo em vista as contribuições das LFs para o desenvolvimento de software de qualidade, bem como a carência de estudos sobre sua aplicação na implementação de variabilidades de software, e buscando responder questões do tipo, como um elemento de variabilidade de software é

[2]A orientação a aspectos não será abordada neste trabalho e é aqui utilizada para ilustrar a pequena adoção de outros paradigmas de programação.

satisfeito por uma LF? Quais os fatores e mecanismos estão envolvidos em sua satisfação? Propomos um estudo exploratório de sua aplicação neste domínio, através da linguagem Haskell e definimos ainda os objetivos específicos deste trabalho:

- realizar uma revisão sobre os conceitos e fundamentos da programação funcional, da linguagem Haskell, e dos elementos de variabilidade de software;
- realizar um estudo exploratório, por meio de provas de conceito, para verificar a satisfação ou não das variabilidades pelas LFs; e
- verificar, por meio deste mesmo estudo, alguns dos mecanismos oferecidos pelas LFs, mais especificamente pela linguagem Haskell, para implementação de variabilidades.

Este trabalho está organizado da seguinte forma:

- o **capítulo 2** apresenta de maneira breve o conceito de LPS. Posteriormente são introduzidos os conceitos de variabilidade de software que, juntamente com a programação funcional, formam o escopo deste trabalho;
- o **capítulo 3** é responsável por introduzir as principais características e conceitos inerentes às LFs, bem como um pequeno histórico sobre sua concepção e evolução.
- no **capítulo 4** descrevemos, de maneira mais abrangente, os materiais e métodos utilizados para realização deste trabalho;
- no **capítulo 5** tratamos efetivamente da implementação das variabilidades, bem como da análise das mesmas; e
- o **capítulo 6** apresenta as conclusões e considerações finais a respeito do estudo, bem como propostas de trabalhos futuros.

# Capítulo 2

# Variabilidade em Linha de Produtos de Software

Antes do advento da produção em massa de bens de consumo, cada produto era único em termos do seu processo de produção, feitos sob medida, de acordo com as necessidades e desejos do cliente. Mais tarde como consequência do processo de industrialização, a produção em massa baseada em linhas de montagem possibilitou um grande aumento de produtividade, reduzindo custos, aumentando a qualidade dos produtos e do processo de produção. Todos esses avanços vieram à custo da capacidade de incorporar requisitos individuais dos clientes [14].

Em busca de maior diversificação e reconhecendo que diferentes clientes têm necessidades distintas, fabricantes passaram a adotar a estratégia de **customização em massa**, onde se busca a produção em larga escala aproveitando-se de componentes reutilizáveis, ao mesmo tempo que se suporta a presença de variações no processo de produção, possibilitando a escolha de características particulares dos produtos [14].

Muito comum, especialmente na indústria automobilística, a customização em massa só é possível graças ao uso de plataformas. De modo geral, uma **plataforma** é qualquer base tecnológica sobre a qual outras tecnologias ou processos são construídos [1]. A princípio, na indústria automobilística, as plataformas eram constituídas apenas de assoalho, suspensão e painéis, até que posteriormente mais componentes fossem adicionados, e esta passou a representar o subsistema mais caro no projeto e preparação para fabricação [1].

## 2.1 Linhas de Produtos de Software

Não tão diferente dos bens físicos, os primeiros softwares também eram feitos sob medida para hardwares específicos e vendidos em conjunto com esses hardwares. Esse modelo de desenvolvimento tornou-se muito caro e complexo, logo as desenvolvedoras de software adotaram como método análogo à produção em massa utilizada pela indústria, a padronização de software. Ao invés de desenvolver um "mesmo software" repetidas vezes para cada hardware, o software seria desenvolvido para uma "plataforma padrão", possibilitando sua implantação em larga escala [14].

Assim como a produção em massa, a estratégia de padronização não suporta individualismo nas aplicações, e é orientada pelo princípio *one-size-fits-all* [14]. O software produzido satisfaz a maior parte dos requisitos de "todos" os possíveis clientes, porém os clientes deixam de ter certos requisitos contemplados, e/ou são sobrecarregados por *features* indesejadas. Além disso, esse generalismo torna os softwares muito complexos, lentos e defeituosos [14].

Como na produção industrial, a ausência de diversificação é um problema, onde softwares deixam de oferecer possíveis vantagens competitivas para certos clientes. Novamente os princípios de customização em massa fazem-se presentes, introduzindo no desenvolvimento de software a ideia de que estes deveriam ser criados a partir de artefatos reutilizáveis, e moldados aos requisitos de cada cliente. Chama-se de **engenharia de linha de produtos de software**, a combinação sistemática de customização em massa e plataforma de software [1], que será responsável por estabelecer uma LPS.

## 2.2 Engenharia de Linha de Produtos de Software

O paradigma das LPSs divide-se em dois processos, como ilustra a figura 2.1. O primeiro, a engenharia de domínio, será abordado por último e de maneira mais abrangente na seção 2.2.2. O segundo, a engenharia de aplicação, não é o foco deste trabalho e será abordado brevemente na seção 2.2.1.

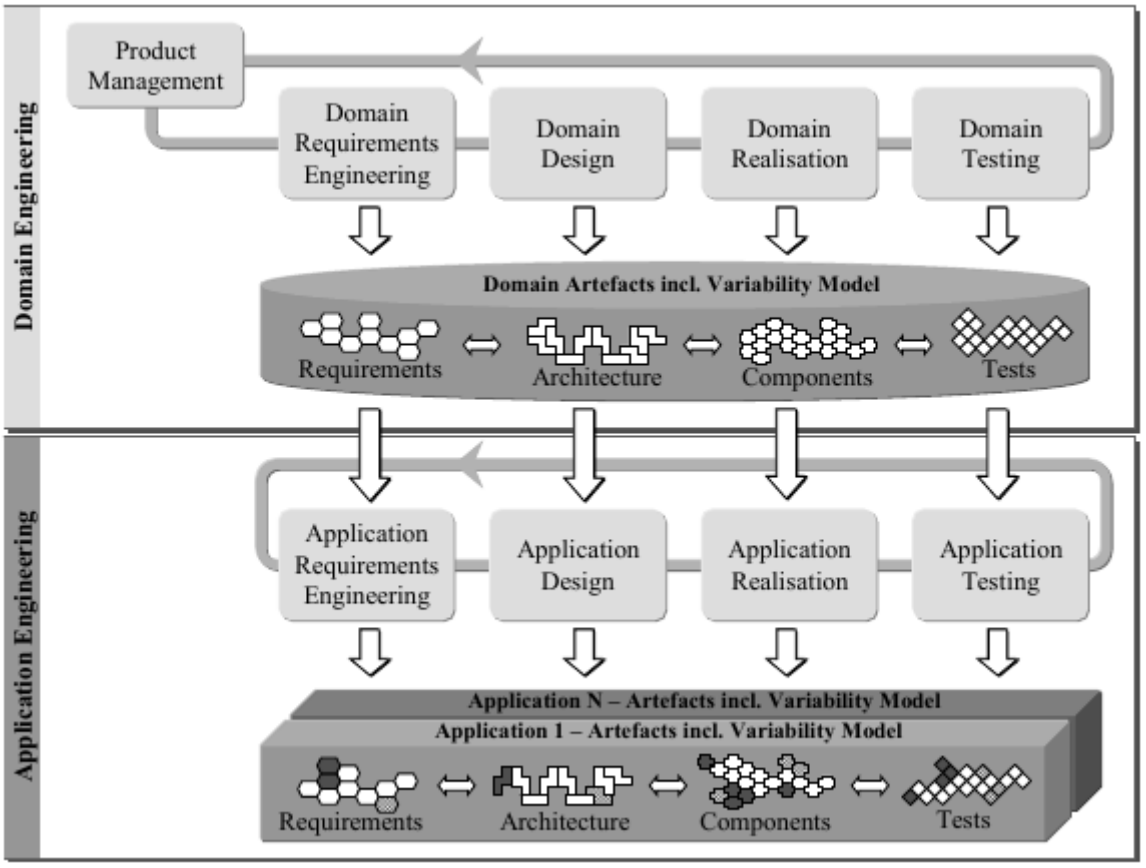



Figura 2.1: O *framework* de linha de produtos de software [1].

### 2.2.1 Engenharia de Aplicação

A engenharia de aplicação é o processo responsável por derivar aplicações com base na plataforma estabelecida pela engenharia de domínio, explorando as variabilidades presentes em uma LPS. A engenharia de aplicação corresponde ao processo de desenvolvimento de uma única aplicação na engenharia de software tradicional, e é repetida para cada produto que se deseja derivar [14].

### 2.2.2 Engenharia de Domínio

Este trabalho utiliza por muitas vezes o termo plataforma, neste capítulo referindo-se especificamente as plataformas de software, tentaremos portanto de defini-lo, antes de discutirmos a engenharia de domínio propriamente dita.

No início deste capítulo, é definida de maneira geral a ideia de plataforma. No domínio do

software esse termo é bastante utilizado para referir-se ao sistema sobre o qual uma aplicação é executada, sendo normalmente a combinação de hardware e sistema operacional. Para a engenharia de linhas de produtos de software porém, uma **plataforma de software** é um conjunto de subsistemas (código, requisitos, arquitetura) e interfaces que formam uma estrutura comum que permita que uma LPS possa ser desenvolvida de maneira eficiente [1].

A engenharia de domínio é o processo responsável por estabelecer uma plataforma, neste caso uma plataforma de software, que possibilitará o reúso de artefatos, definindo comunalidades e variabilidades de uma LPS.

O processo de engenharia de domínio ilustrado na figura 2.1 é dividido em quatro subprocessos, engenharia de requisitos de domínio, projeto de domínio, realização de domínio e testes de domínio. Dentre os subprocessos discutiremos apenas a realização de domínio, visto que o objetivo deste trabalho é parte do mesmo[1].

O subprocesso de **realização de domínio** lida com os detalhes de projeto e implementação de artefatos reutilizáveis. A realização de domínio proverá uma série de artefatos configuráveis, não uma aplicação propriamente dita, cada componente é planejado e implementado pensando em seu reúso em diferentes contextos (produtos de uma linha) [1].

Algumas propriedades importantes da realização de domínio ainda serão discutidas ao longo deste capítulo, por hora encerramos as explanações sobre LPS apresentando os artefatos de domínio.

### 2.2.3 Artefatos de Domínio

Os artefatos de domínio são elementos reutilizáveis que compõem a plataforma de software e garantem a definição consistente de comunalidades e variabilidades [1], são eles.

#### 2.2.3.1 Plano de Produto

O plano de produto descreve as funcionalidades das aplicações da linha de produtos enquanto as classifica como **comum**, aquela que pertence a todas as aplicações da linha, ou **variável**, particular a algumas aplicações. O plano de produto não se trata de um artefato de desenvolvimento, mas de um guia para o futuro do desenvolvimento da linha de produtos, guiando tanto a engenharia de domínio, quanto a engenharia de aplicação [1].

[1] Para detalhes sobre os subprocessos não abordados, vide [1]

#### 2.2.3.2 Modelo de Variabilidade de Domínio

O modelo define o que pode variar em um produto de software, apresentando pontos de variação e as possíveis variantes para um determinado ponto de variação. O modelo de variabilidade também define as dependências e restrições das variabilidades. Além disso, o modelo também relaciona as variabilidades presentes nos vários artefatos de desenvolvimento, suportando a definição de variabilidade em todos os artefatos de domínio [1].

#### 2.2.3.3 Requisitos de Domínio

Os requisitos de domínio englobam requisitos comuns a todas as aplicações da linha de produtos, assim como requisitos particulares a alguma aplicação [1].

#### 2.2.3.4 Arquitetura de Domínio

A Arquitetura de domínio define a estrutura e a disposição das aplicações de uma LPS. Onde a estrutura define a decomposição das aplicações, enquanto a disposição é o conjunto de regras comuns que guiam o projeto e realização de partes, e como essas são combinadas para formar uma aplicação. A disposição arquitetural atua tanto na engenharia de domínio, quanto na engenharia de aplicação [1].

#### 2.2.3.5 Artefatos de Realização de Domínio

Os artefatos de realização de domínio correspondem aos artefatos de projeto e implementação de componentes reutilizáveis e interfaces. Os artefatos de projeto são diferentes tipos de modelos que capturam as estruturas de cada componente. Dentre os artefatos de implementação estão código fonte, arquivos de configuração, e arquivos de compilação (*makefiles*) [1].

#### 2.2.3.6 Artefatos de Teste de Domínio

São os planos, casos, e cenários de teste, que definem respectivamente a estratégia de testes (artefatos, casos, além de cronograma e recursos para realização dos mesmos), e instruções detalhadas para realização dos testes [1].

## 2.3 Variabilidade

A variabilidade é uma propriedade essencial dos artefatos de domínio e uma boa gerência da mesma é fundamental para o sucesso de uma LPS. Ao processo de definir, representar, explorar, implementar, e evoluir variabilidades, dá-se o nome de **gerência de variabilidade** [2].

Antes de introduzir os principais conceitos relativos à variabilidade em LPS, cabe apresentar os três tipos de variabilidades distinguíveis em sua gerência. As características comuns a todos os produtos de uma LPS dá-se o nome de **comunalidade**. Por sua vez aquelas características comuns apenas a algumas aplicações, são chamadas de **variabilidade**. Existe ainda um terceiro tipo de variabilidade chamada de **específica de produto**, particular à engenharia de aplicação e que normalmente não fará parte da plataforma, mas que a plataforma deve suportar [2].

### 2.3.1 Ponto de Variação vs. Variante

Os **pontos de variação** definem onde existem diferenças entre os produtos de uma LPS, sendo um conjunto das propriedades variáveis de um item real abstraído para a LPS [1, 2]. O conceito de ponto de variação pode ser aplicado a todos os artefatos de desenvolvimento, requisitos, código, testes, e assim por diante.

Consideremos, por exemplo, no domínio automobilístico, que uma empresa queira produzir carros de variadas cores, define-se assim um ponto de variação “cor de um carro”. Dado que esta mesma empresa venha inicialmente a produzir carros apenas nas cores verde e amarelo, apenas duas variantes são definidas para o ponto de variação “cor de um carro”. Chama-se de **variante** as diferentes possibilidades de se satisfazer um ponto de variação [2].

### 2.3.2 Variabilidade no Espaço vs. Variabilidade no Tempo

Todo software, e logo os artefatos de seu desenvolvimento evoluem com o tempo. Esta evolução e consequente existência de diferentes artefatos válidos em diferentes momentos é chamada de **variabilidade no tempo**, e aplica-se não somente ao paradigma das LPSs, mas também ao desenvolvimento tradicional [1]. A variabilidade no tempo é um conceito correlato à área de evolução de software, por sua vez a variabilidade em LPS normalmente refere-se à variabilidade espacial [1].

A **variabilidade espacial** trata da existência de um artefato sob diferentes formas, no mesmo

momento no tempo, ou ainda do uso por diferentes produtos de um artefato variável [1], como no exemplo apresentado na seção 2.3.1, que trata das cores de automóveis.

Para as LPSs, onde o objetivo é produzir produtos similares, providos de diversificações específicas, que ocorrem "ao mesmo tempo", a gerência de variabilidade espacial é um fator crucial. Sendo este tipo de variabilidade o principal alvo deste trabalho.

### 2.3.3 Representação de Variabilidade

Diversas estratégias para representar variabilidade foram discutidas ao longo dos anos. Atualmente a maioria das representações utilizam *features* como base para representar a variabilidade. Diferentes interpretações do termo *feature* existem, o que induz a um espectro bastante amplo dessas representações [2]. Fundamentalmente, representar características que diferenciam uma aplicação de outra é chave para qualquer estratégia de representação [2].

A variabilidade pode ser representada por meio de modelos de decisão, descrevendo as decisões que devem ser tomadas para derivar um certo produto em uma LPS. É possível também, utilizar uma descrição textual baseada em lógica proposicional como em [14]. Este trabalho adota como base a representação gráfica de um modelo ortogonal de variabilidade, assim como [1, 2, 14], cujo os elementos estão sintetizados na figura 2.2, e sua utilização é ilustrada pela figura 2.3.

Os elementos de variabilidade ortogonal[2] são descritos a seguir:

- *Variation Point* (Ponto de Variação): este elemento representa o conceito de ponto de variação introduzido na seção 2.3.1, e é ilustrado por um triângulo.
- *Variant* (Variação): representa o conceito de variante (vide 2.3.1) e é ilustrado por um retângulo.
- *Variability Dependencies* (Dependências de Variabilidade): elementos que indicam as possíveis opções para se satisfazer um ponto de variação [2]:
  - *mandatory* (obrigatória): indica que uma variante deve necessariamente ser parte da aplicação. A dependência obrigatória é ilustrada por uma linha contínua;

[2]Termo utilizado em [1] para se referir aos elementos que representam variabilidade de uma LPS, segundo o modelo ortogonal.

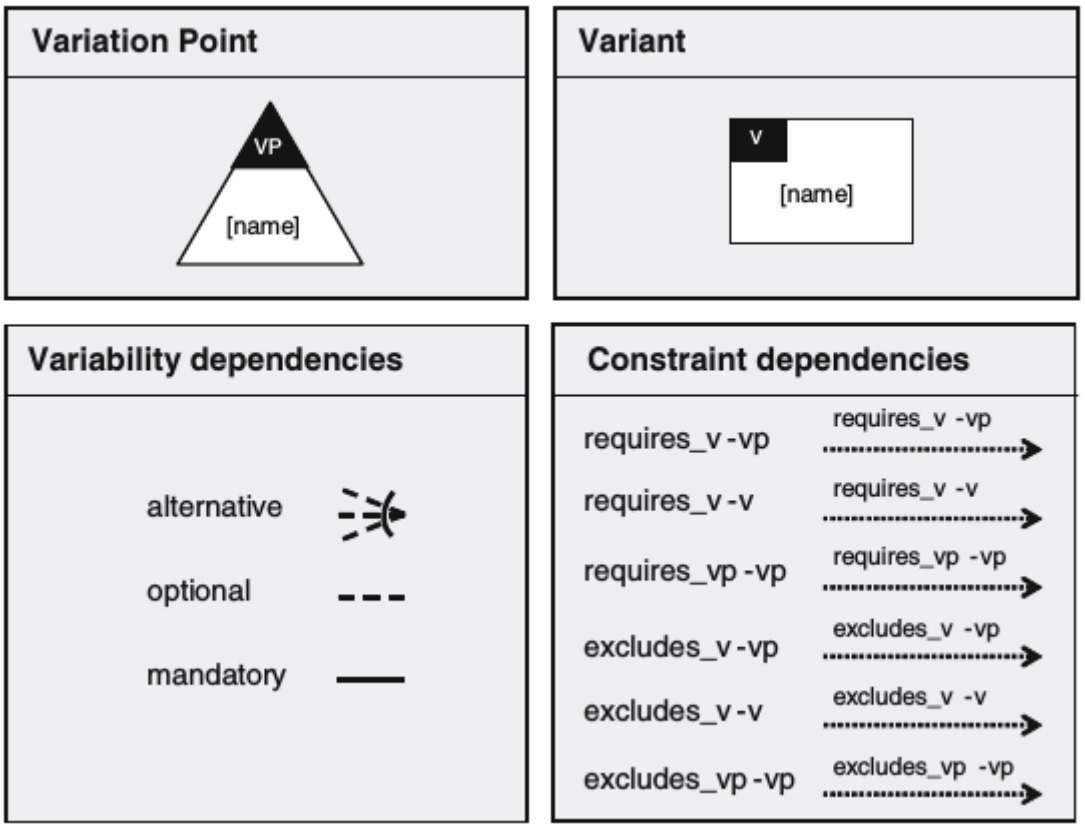



Figura 2.2: Elementos gráficos do modelo ortogonal de variabilidade [2].

  - *optional* (opcional): representam a possibilidade de uma variante fazer parte de um produto, mas que não há obrigatoriedade em sua seleção [1]. O elemento é ilustrado por uma linha tracejada;
  - alternative (alternativa): ilustrada por duas ou mais dependências de variabilidade opcionais interceptadas por um arco. Indica a possibilidade de escolha sobre um conjunto de alternativas (*selecet*). O elemento pode ser acompanhado de uma cardinalidade $M : N$, que indica o número mínimo e máximo de variantes selecionáveis do conjunto. A ausência de cardinalidade representa uma relação mutuamente exclusiva entre as variantes (*xor*) [1, 2].

- *Constraint Dependencies* (Restrições): descrevem restrições/dependências entre a seleção de certas variantes [2]. Ilustradas por meio de setas pontilhadas acompanhadas de um rótulo, são dois os tipos de restrições:

  - *requires* (requer): a seleção de uma variante pode implicar na necessidade (obrigato-

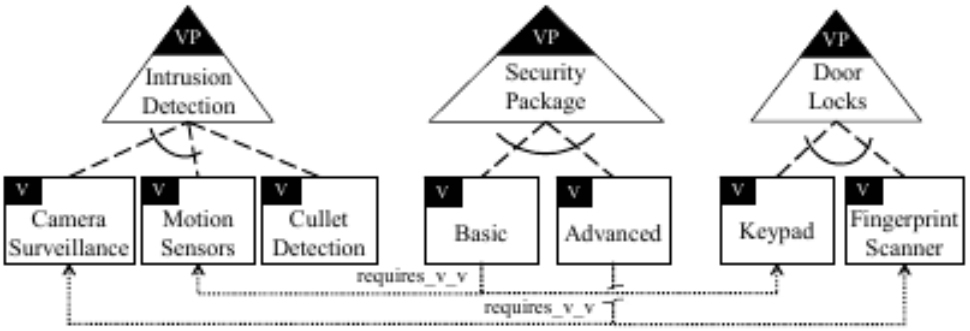



Figura 2.3: Modelo de variabilidade de um pacote de soluções de segurança [1].

riedade) de se selecionar uma outra variante (*requires_v-v*), ou ainda na necessidade de se selecionar um ponto de variação (*requires_v-vp*), e um ponto de variação pode requerer a consideração de um outro ponto de variação (*requires_vp-vp*);

- *excludes* (exclui): a seleção de uma variante pode implicar na necessidade de se excluir uma outra variante (*excludes_v-v*), ou ainda na necessidade de se excluir um ponto de variação (*excludes_v-vp*), e um ponto de variação pode remover a consideração de um outro ponto de variação (*excludes_vp-vp*).

São estes elementos de variabilidade que servirão como referência para elaboração e implementação das provas de conceito presentes neste trabalho.

# Capítulo 3

# Programação Funcional

As primeiras LPs tinham um objetivo bastante simples, prover um intermediário pelo qual se pudesse controlar o comportamento de um computador, representando a estrutura da máquina sobre a qual trabalhavam de maneira bastante fiel [13].

A partir da primitiva classe *assembly* de linguagens, evoluíram/surgiram diversas linguagens de alto nível, nos anos 80 eram tantas as linguagens que era mais fácil classificá-las em famílias que representavam um mesmo estilo de programação e/ou refletissem um modelo computacional comum [13].

Uma dessas classes de LPs (paradigma) é a classe das LFs. Este capítulo apresenta os principais conceitos do paradigma funcional de programação.

Antes de iniciar as discussões sobre tais conceitos, observemos que este trabalho não tem como objetivo introduzir uma linguagem de programação, apresentar comandos, aspectos sintáticos, ou funcionalidades providas por uma biblioteca. Consideramos que a compreensão dos conceitos sendo apresentados é também suficiente para se ter uma visão geral da linguagem Haskell. Particularidades da linguagem serão eventualmente discutidas, quando necessário.

## 3.1 Computação sem Estado

Linguagens imperativas caracterizam-se por possuírem um estado implícito, que é modificado por comandos da linguagem fonte, o que permite o controle preciso e determinístico do estado de um programa, normalmente dando a impressão de sequenciamento das ações [13].

Em contrapartida, as LFs não apresentam estado implícito, e a computação ocorre inteiramente por meio da avaliação de expressões. As linguagens funcionais são uma classe de

linguagens declarativas, cujo modelo computacional base é a função[1] [13].

O conceito de estado implícito, ou efeito colateral, pode ser descrito por um simples comando de atribuição. O exemplo do quadro 3.1 computa o fatorial de um inteiro positivo $n$:

Quadro 3.1: Fatorial imperativo.

```
1 x = n
2 a = 1
3 while n > 0 do
4       a = a*n
5       n = n-1
6 end
```

Ao final da execução desse trecho de código a variável `a` conterá o valor correspondente ao fatorial de `n`, para tal, o estado do programa foi alterado diversas vezes, e `a` já não designa o mesmo valor de quando foi definida.

Nas LFs as computações são realizadas transportando explicitamente seu estado, e iterações são possíveis por meio de recursão, e não sequenciamento [13]. Utilizamos novamente o cálculo do fatorial (quadro 3.2) para exemplificar[2]:

Quadro 3.2: Fatorial funcional.

```
1 fact x 1
2       where fact n a =
3             if n > 0 then fact (n-1) (a*n)
4             else a
```

Os parâmetros `n` e `a` de `fact`, "carregam" explicitamente o estado da computação, ao invés de armazenar seu valor de maneira implícita, assim o programa retorna de fato o fatorial desejado.

Essa distinção entre paradigma funcional e imperativo é fundamental. Hoje muitas LPs implementam mecanismos oriundos do paradigma funcional, porém não se caracterizam como LFs. É um erro pensar que um subconjunto puramente funcional de uma linguagem "convencional" seja satisfatório em termos de sua funcionalidade/aplicabilidade[3] [13].

[1] O termo função deve ser visto de uma perspectiva matemática, diferente das noções de função, procedimento, ou ainda método, presentes em linguagens imperativas.

[2] A recursão foi arranjada para que fosse similar a sua contraparte imperativa. Existem outras formas mais "abstratas" de representá-la.

[3] Um subconjunto puramente funcional de uma linguagem convencional normalmente apresentará sérias limitações técnicas, embora exceções existam, como por exemplo, a linguagem Scheme [13].

As próximas seções complementarão o que foi aqui iniciado, apresentando conceitos importantes para linguagens funcionais modernas como: funções de alta ordem, "avaliação preguiçosa", abstrações de tipos de dados, equações e "casamento de padrões".

## 3.2 Lambda Calculus

De maneira informal *calculus* é uma sintaxe para representar um conjunto de termos, e regras para alterar estes termos [13]. O *lambda calculus* foi criado com o objetivo de representar de forma intuitiva o comportamento de funções [13], formalizar processos por meio de expressões parametrizadas, onde cada ocorrência de um desses parâmetros é indicada pela letra grega $\lambda$, por isso de seu nome [8].

O *lambda calculus* é um sistema para tratamento de expressões $\lambda$, onde estas podem ser um **identificador** ($id$), **aplicação** ($aplic$), ou ainda uma **função** ($func$):

$$\begin{array}{lcl} expr & \rightarrow & id \mid aplic \mid func \\ aplic & \rightarrow & expr_1\ expr_2 \\ func & \rightarrow & \lambda id.\ expr \end{array}$$

A expressão $func$ representa uma função, onde o terminal *id* é parâmetro formal da mesma, e $aplic$ representa a aplicação de uma função. A aplicação é por convenção associativa a esquerda, logo $(expr_1\ expr_2\ expr_3)$ é equivalente a $((expr_1\ expr_2)\ expr_3)$ [13, 15].

As regras para modificação/reescrita de uma expressão $\lambda$ baseiam-se na ideia de redução, substituição de uma expressão $expr_1$ por todas as ocorrências livres de um identificador *id* em uma segunda expressão $expr_2$[4]. Diz-se que um expressão $\lambda$ está em sua **forma normal** se esta não puder mais ser reduzida por qualquer método [13].

Assim podemos inferir que o resultado de uma computação em *lambda calculus* é dado pela avaliação de uma expressão reduzida a sua forma normal, para tal apresentamos duas estratégias de redução. É importante salientar que nem toda expressão possui forma normal, como no caso da expressão $(\lambda x.(x\ x))(\lambda x.(x\ x))$.

A primeira forma de redução chamada de **redução em ordem aplicativa**, substitui para cada ocorrência de *id* no corpo da expressão $expr_1$, o valor da expressão $expr_2$ [15]. É uma redução sequencial em que a expressão redutível (*redex*) mais à esquerda e mais interna é avaliada [13]:

[4]Quando se fala de substituições de expressões em *lambda calculus*, faz-se necessário considerar alguns aspectos formais de suas definições, que para fins de simplicidade não serão abordados neste trabalho. O leitor pode encontrá-los em [13].

$$
\begin{array}{ll}
 & (\lambda x.\ (+\ x\ x))\ (*\ 5\ 4) \\
\Rightarrow & (\lambda x.\ (+\ x\ x))\ 20 \\
\Rightarrow & (+\ 20\ 20) \\
\Rightarrow & 40
\end{array}
$$

A segunda forma de redução chamada **redução em ordem normal**, substitui, sempre que houver mais de uma $redex$, àquela que estiver mais à esquerda [13]. Cada ocorrência de *id* no corpo de $expr_1$ é substituída por $expr_2$ não avaliada:

$$
\begin{array}{ll}
 & (\lambda x.\ (+\ x\ x))\ (*\ 5\ 4) \\
\Rightarrow & (+\ (*\ 5\ 4)\ (*\ 5\ 4)) \\
\Rightarrow & (+\ 20\ (*\ 5\ 4)) \\
\Rightarrow & (+\ 20\ 20) \\
\Rightarrow & 40
\end{array}
$$

Ambos os métodos de avaliação são base para implementações de linguagens de programação, não se restringindo às linguagens funcionais. O que aqui introduzimos sobre *lambda calculus* é apenas o necessário para compreensão de conceitos ainda por apresentar, outros resultados importantes a respeito do *lambda calculus* (incluindo vantagens e desvantagens de cada método de redução) podem ser encontrados em [13, 15].

## 3.3 Funções de Alta Ordem

Uma função é dita de **alta ordem** se recebe como parâmetro, ou retorna como resultado uma função. Uma LP apresenta funções de alta ordem se permite que funções sejam armazenadas em estruturas de dados, passadas como argumentos ou retornadas como valores [13].

Nas LFs funções são o principal mecanismo de abstração sobre valores, logo tratar funções como qualquer outro valor é um importante facilitador para a utilização desse tipo de abstração.

Quadro 3.3: Funções de alta ordem.

```
1 applyTwice f x = f (f x)
2
3 sum2 = applyTwice succ'
4        where succ' = (+ 1)
```

O exemplo apresentado no quadro 3.3 faz uso de funções de alta ordem de duas formas distintas. Para a função `applyTwice` é possível verificar a aplicação de funções de alta ordem em seu parâmetro `f`, que representa uma função qualquer a ser aplicada duas vezes a algum valor `x`. Por sua vez, a função `sum2` retorna como resultado a função `(+ 1)`, que espera por

um argumento numérico ao qual somará `1`. Este último resultado ilustra o princípio chamado de *currying*, uma função quando aplicada a um argumento retorna uma função que espera por mais um argumento. A aplicação de *currying* é possível graças a agregação de funções de alta ordem e "avaliação preguiçosa".

## 3.4 Avaliação Preguiçosa

Um resultado sobre *lambda calculus* não apresentado na seção 3.2, é que dentre os mecanismos de redução, a avaliação em ordem normal é mais geral do que em ordem aplicativa, sua computação termina para mais expressões. Este resultado torna a redução em ordem normal uma funcionalidade bastante desejável.

Quadro 3.4: Teste para o mecanismo de redução.

```
1  (define (p) (p))
2
3  (define (test x y)
4       (if (= x 0)
5            0
6            y ))
```

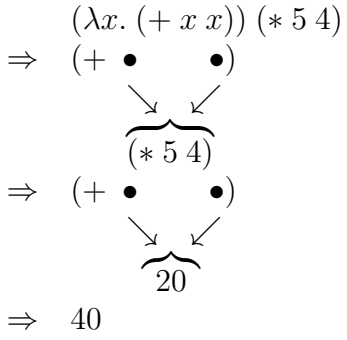
O trecho de código Scheme no quadro 3.4 é utilizado para testar se a implementação de um certo interpretador aplica redução em ordem normal, ou em ordem aplicativa, e demonstra o maior generalismo da redução em ordem normal. O procedimento `p` é definido como uma recurção infinita, enquanto `test` avalia se seu primeiro parâmetro é igual a $0$, retornando $0$ em caso positivo, ou o valor de seu segundo parâmetro em caso contrário.

Ao aplicarmos o teste `(test 0 p)` em um interpretador com avaliação aplicativa, este entrará em *loop* infinito, pois a primeira expressão a ser substituída é `y`, cujo valor é o procedimento `p`, que não pode ser reduzido. Em uma avaliação em ordem normal a computação encerraria normalmente, retornando $0$ como resultado, já que o primeiro valor substituído é `x`, que por sua vez é suficiente para satisfazer a condicional `if (= x 0)`.

Embora muito atraente, se mal implementada, a avaliação em ordem normal pode ser muito ineficiente, um exemplo deste fato é que ao se examinar a redução em ordem normal realizada na seção 3.2, observamos que a computação do termo $(* \ 5 \ 4)$ é realizada duas vezes.

Buscando tornar o processo de reduções em *lambda calculus* mais eficiente, Chris Wadsworth sugeriu em sua tese de doutorado [16], que estas fossem realizadas por meio de

reduções em grafos, o que possibilitaria o compartilhamento de valores entre os passos de uma redução, como ilustrado a seguir.




A uma implementação de redução em ordem normal em que computações redundantes são evitadas, dá-se o nome de **avaliação preguiçosa**, ou chamada por demanda/necessidade [13].

A partir de meados dos anos 70 uma série de pesquisas na área das LFs introduziram a ideia de avaliação preguiçosa, que seria mais tarde um dos principais fatores na criação da linguagem Haskell [17]. Segundo Huges, a avaliação preguiçosa é a principal ferramenta de modularização no repertório de um programador funcional [18].

## 3.5 Abstração de Dados

Seja no desenvolvimento de LFs, ou de linguagens imperativas, esforços consideráveis são realizados na área de abstração de dados e tipos [13]. Tipos podem ser considerados importantes pelo simples fato de auxiliarem na compreensão das LPs [13].

No paradigma das LFs em especial, existe grande apelo pelo uso de abstrações de dados, e em particular pela aplicação de tipagem forte. Argumenta-se que as abstrações de dados beneficiam a modularidade, segurança, e clareza dos programas [13]. A melhoria da modularidade ocorre porque abstrações de dados permitem que nos distanciemos dos detalhes da implementação; torna violações de interface impossíveis, melhorando assim a segurança; e por fim provê um aspecto de alto documentação aos programas [13].

Por sua vez a tipagem forte melhora o processo de depuração dos programas, pois se um programa compila sem erros, nenhum erro em tempo de execução pode ocorrer em virtude de erros de tipo. Melhorias de desempenho também são possíveis, removendo fatores como checagem de tipo em tempo de execução, o que também reduzirá os custos da implementação das abstrações de dados [13].

### 3.5.1 Tipos de Dados Concretos

Esta seção tem por objetivo apresentar o conceito de tipos de dados concretos, ou algébricos, bem como a ideia de sinônimos de tipos. Consideremos a seguinte implementação do tipo `Tree a` e do sinônimo `IntTree` na linguagem Haskell.

Quadro 3.5: Tipos de dados algébricos e sinônimos.

```
1 data Tree a = Empty
2             | Node a (Tree a) (Tree a)
3
4 type IntTree = Tree Int
```

No trecho de código no quadro 3.5, `Tree` é chamado de **construtor de tipo** e o identificador `a` é chamado de **variável de tipo**, que é quantificada em todo o escopo de `Tree`. Os identificadores `Empty` e `Node`, são chamados **construtores de dados**, ou simplesmente **construtores**.

`Tree` é na verdade uma função que toma um tipo como parametro para produzir (construir) novos tipos, ao substituirmos a variável de tipo `a` por `Int`, obtemos o tipo `Tree Int`, chamamos assim `Tree` de construtor de tipos[5].

Do lado direito da definição do tipo `Tree a`, temos os construtores de dados, estes elementos não representam tipos, mas possíveis valores de um tipo. `Tree a` é um tipo cujos valores podem ser, uma árvore vazia, `Empty`, ou uma árvore como raiz do tipo `a` e duas subárvores, cujos nós são do mesmo tipo `a`, `Node a (Tree a) (Tree a)`.

Por último, na linha 4 do quadro 3.5 implementamos o **sinônimo de tipo** `IntTree`, que permite a criação de novos nomes para tipos. `IntTree` define uma especialização de `Tree a`, onde a variável de tipo `a` é explicitamente definida como `Int`, fazendo de `IntTree` uma árvore cujo os elementos são inteiros. Sinônimos não definem novos tipos de dados, apenas criam um *alias* (sinônimo), afim de tornar a utilização dos tipos mais clara.

### 3.5.2 Tipos de Dados Abstratos

Provenientes das linguagens imperativas, os tipos de dados abstratos, são aqueles onde detalhes de sua implementação estão escondidos do usuário, melhorando assim a modularidade e segurança [13]. Diferentes linguagens funcionais implementam tais abstrações de maneiras distintas, logo não serão apresentadas discussões mais amplas a respeito.

[5]Não confundir com os métodos construtores do paradigma OO.

É importante e suficiente assimilar a ideia de que tipos abstratos são tipos aos quais estão associados operações, exemplos de construções que possibilitam tipos abstratos são módulos e, em especial dentro do modelo imperativo OO, as classes [8].

## 3.6 Equações e Casamento de Padrões

Uma das práticas mais incentivadas pela comunidade das LFs modernas é a aplicação do conceito de *equational reasoning*, termo que optamos por não traduzir[6].

A aplicação de *equational reasoning* baseia-se no fato de que em uma linguagem puramente funcional (vide seção 4.3), qualquer igualdade entre expressões definida a qualquer momento será válida em qualquer outro momento do tempo [8]. O que só é possível graças ao conceito de transparência referencial, onde a ausência de efeitos colaterais torna as expressões independentes da ordem de avaliação [8].

Ilustramos o princípio da transparência referencial com o seguinte trecho de código:

```
...x + x...
where x = f a
```

Consideremos agora a expressão "iguais podem ser substituídos por seus iguais". O que tal expressão, bem como o exemplo anterior representam é o fato de que, dado que nenhum comando pode produzir efeitos colaterais, pode-se substituir qualquer ocorrência livre de `x` por `f a`, e vice-versa [13, 18].

Um dos principais meios para se aplicar *equational reasoning* é o uso de casamento de padrões, onde se pode construir uma sequência de equações para definir a mesma função, das quais apenas uma será aplicável para uma determinada situação.

O **casamento de padrões** pode ser visto como uma expressão *case*, na qual se a estrutura de uma expressão $e$ corresponde a um padrão $p$, o resultado da expressão como um todo é a expressão $e_i$ correspondente.

Para a definição recursiva do enésimo número de Fibonnaci no quadro 3.6, a expressão sendo testada é qualquer valor de $n$ passado como parâmetro à função `fibonnaci`, os padrões contra os quais a expressão será testada são os valores predefinidos dos parâmetros de `fibonnaci`.

[6]Possíveis traduções seriam "discurso por meio de equações", "raciocínio via equações", ou ainda "raciocínio equacional", os quais não pareceram razoáveis do ponto de vista do autor.

Quadro 3.6: Equações e casamento de padrões.

```
1 fibonacci 0 = 0
2 fibonacci 1 = 1
3 fibonacci n =
4      fibonacci (n-1) + fibonacci (n-2)
```

Os dois primeiros padrões representam os casos base da recursão, e retornam respectivamente as expressões $0$ e $1$ como resultados. O último padrão é o que se chama de *catch-all*, que será responsável por tratar qualquer caso aos quais nenhuma das definições anteriores for aplicável, retornando como resultado a soma das computações recursivas de `fibonnaci`.

Encerramos assim as discuções sobre os conceitos básicos das LFs. Mais resultados sobre LFs, e tópicos avançados sobre as mesmas, podem ser encontrados em [13, 18].

# Capítulo 4

# Metodologia

Este trabalho apresenta algumas particularidades importantes em sua metodologia. Este capítulo discorre sobre estas particularidades e estratégias, apresentando considerações e decisões de pesquisa, uma visão geral de como o trabalho foi idealizado, além de uma pequena contextualização do domínio de pesquisa e motivações.

## 4.1 Trabalhos "Correlatos"

Afirmamos na introdução deste trabalho que, embora haja uma maior popularização do paradigma funcional de programação, este mesmo paradigma não tem sido explorado na implementação de variabilidades de software. Buscando retratar melhor essa realidade, e uma melhor compreensão da motivação deste trabalho, apresentamos dois trabalhos avaliados durante os estudos preliminares.

O primeiro, um livro chamado "*Haskell Design Patterns*" [19], introduz conceitos de desenvolvimento na linguagem Haskell. Embora apresente a implementação de alguns *patterns* presentes em [20], dos quais, alguns são reconhecidos como mecanismos apropriados para implementação de variabilidades, o livro não trata de maneira explícita da implementação das mesmas, e não tem a intenção de ser uma adaptação de [20] para a linguagem Haskell.

O segundo, um artigo intitulado "*Feature (De)composition in Functional Programming*" [21], trata de *crosscutting concerns* em programas Haskell. *Crosscutting concerns* são elementos de um domínio, que se espalham por diferentes camadas de uma aplicação, podendo afetar grande parte de sua base de código [21].

O artigo identifica a ocorrência dos *concerns* em duas aplicações Haskell, e propõe solu-

ções baseadas em *Feature-Oriented Software Development* (FOSD), utilizando-se de ferramentas para decomposição das aplicações em *features*. Embora trate de FOSD, paradigma que se utiliza dos conceitos de variabilidade, o trabalho não trata da implementação de variabilidades. O artigo ainda chama a atenção para o também pequeno número de pesquisas tratando de *crosscutting concerns* em programação funcional.

## 4.2 Haskell

Haskell é uma linguagem puramente funcional, de propósito geral, que implementa algumas das principais inovações produzidas nas últimas três décadas de pesquisa em LPs. De maneira "informal", e para fins de simplicidade, uma LF é dita **puramente funcional**[1] se apresenta transparência referencial [13, 18].

Em meados dos anos 80 uma série de pesquisas que diziam respeito à implementação de linguagens puramente funcionais e "preguiçosas" estavam em andamento, e uma série de linguagens haviam sido implementadas como frutos dessas pesquisas [17].

O cenário descrito anteriormente é chamado de "torre de Babel" das linguagens preguiçosas, haviam muitas linguagens com propriedades interessantes, porém nenhuma delas parecia se destacar, ou ser completa em termos das necessidades apresentadas pelos usuários das mesmas [17]. Apresentava-se aí a demanda por uma linguagem da qual toda uma comunidade pudesse se beneficiar.

Durante o período da *Conference on Functional Programming Languages and Computer Architecture* realizada em Portland, Oregon, no ano de 1987 [17], foi criado um comitê para o desenvolvimento de uma linguagem que satisfizesse as necessidades da comunidade de programadores funcionais da época, três anos mais tarde era lançada a Haskell versão 1.0 [17]. Os objetivos iniciais da linguagem eram de satisfazer os seguintes requisitos [23]:

- ser apropriada ao ensino, pesquisa, e desenvolvimento de aplicações, incluindo a construção de grandes sistemas;
- deveria estar disponível a qualquer um que quisesse implementar e distribuir a linguagem;

[1] Não há acordo a respeito do significado do termo, embora trabalhos como o de Sabry [22] tentem estabelecer critérios formais para defini-lo.

- deveria ser baseada em ideias compartilhadas em consenso; e
- deveria reduzir a diversidade desnecessária de linguagens de programação funcionais.

Outras LFs além da Haskell poderiam ser consideradas na elaboração de um trabalho tratando do paradigma. Destacam-se dentre elas, a família LISP, a família ML, Miranda, Scala, F#, no domínio das aplicações de tempo real e de alta disponibilidade Erlang[2].

Dentre as linguagens citadas anteriormente, Miranda é uma linguagem proprietária. As linguagens Scala e F#, são linguagens híbridas e incorporam diversos aspectos do paradigma OO, gostaríamos que tais aspectos não influenciassem na implementação das variabilidades. A natureza dinâmica das linguagens da família LISP torna mais complexa a representação de certos conceitos, como a parametrização. Para alguns membros da família ML valem as mesmas observações feitas sobre Scala e F#, outros membros são hoje "pouco ativos". A partir destas considerações adotamos Haskell como linguagem para elaboração deste trabalho.

## 4.3 Visão Geral

Este trabalho tem como objetivo explorar a implementação de variabilidades com base nos conceitos da programação funcional, através da linguagem Haskell. Para tal adotamos uma abordagem bastante direta.

Seja uma calculadora um produto de software, onde o ponto de variação "Base Numérica" apresenta duas possíveis variações, "Decimal" e "Binária". Assumamos agora que estas variações apresentam um relacionamento mutuamente exclusivo (*xor*), como satisfazer este elemento de variabilidade? Quais os fatores e mecanismos envolvidos em sua satisfação? Para responder questões como esta, este trabalho se utiliza de provas de conceito.

Para cada elemento de variabilidade presente na representação visual do modelo ortogonal de variabilidade (vide figura 2.3), um pequeno cenário fictício, como o apresentado anteriormente, é descrito e implementado para servir de prova conceitual da satisfação das variabilidades pela linguagem Haskell[3].

[2] A ordem em que as linguagens estão apresentadas não deve de forma alguma ser considerado como indicador de "relevância" ou popularidade, este trabalho não está preocupado com estes fatores.

[3] O número de provas de conceito não foi estipulado devido às particularidades do trabalho, e a falta de referência em relação a qual seria a um número satisfatório.

As provas de conceito estão contidas, estruturadas, e serão analisadas, na forma de *patterns* de software. Maiores detalhes sobre o formato do *pattern* utilizado, e das informações que são capturadas pelo mesmo são discutidos na seção 4.4.

## 4.4 *Patterns*

Alexander [24] define *pattern* como uma entidade que descreve um problema recorrente em nosso ambiente, e os principais elementos da solução desse problema, de forma que se possa utilizar tal solução inúmeras vezes, sem fazê-lo da mesma forma uma única vez.

Embora Alexander estivesse se referindo aos *patterns* no domínio da arquitetura e planejamento de construções, seu trabalho serviu de inspiração para elaboração dos *patterns* de software [25], e sua definição também é valida neste domínio [20]. Informalmente um *pattern* também pode ser visto como um caminho que leva a um objetivo específico, em uma determinada área [26].

Um *pattern* é composto de cinco elementos essenciais[4] [26]:

- O **contexto** ajuda a ter uma visão geral do problema, onde ele ocorre, para qual LP, soluções prévias, ou qualquer outro fator que possa "invalidar" o *pattern* [25, 27].
- O **problema** é uma sentença simples que apresenta um objetivo, e o que interfere/impede que este seja atingido [26, 27].
- As **forças** são considerações, fatores, restrições, e requisitos, que estão relacionados ao contexto, e que devem ser balanceados para se elaborar uma solução satisfatória [26, 27].
- Uma **solução** é um caminho conhecido para se atingir um objetivo, que deve levar em consideração todas as forças, enquanto apresenta de maneira clara o que é necessário para atingir tal objetivo [26, 27].
- Um *pattern* não é absoluto, diferentes soluções para um mesmo problema podem existir, trazendo diferentes vantagens, introduzindo novos problemas. As **consequências** resumem os pros e os contras de um *pattern* [25, 26].

[4] O número de elementos, bem como a nomenclatura dos mesmos, varia dependendo do(s) autor(es).

Diferentes formatos de *pattern* capturam estes elementos em estruturas distintas. Coplien apresenta os quatro formatos mais populares de *patterns* de software, o formato de Alexander, Portland, Coplien, e *Gang of Four* (GoF) [25]. Os dois primeiros são “menos rigorosos” em termos de sua estrutura, sem seções fortemente delimitadas, apresentando elementos de forma "narrativa". Os dois últimos se apresentam em forma de tópicos, seções, palavras chave, delimitando elementos de maneira explícita.

De modo geral a elaboração de *patterns* está condicionada à experiência de indivíduos em um determinado domínio, e ao conhecimento prévio de alguma solução que já se provou satisfatória em algum momento [26]. Porém, este trabalho trata de um estudo exploratório com uma série de aspectos particulares, como a pouca experiência no uso das LFs, a falta de referências sobre a aplicação das mesmas na implementação de variabilidades, além de utilizar os *patterns* de maneira distinta, para estruturar e documentar a análise das implementações.

Devido às particularidades deste trabalho, não é possível utilizar um *pattern* em formato de seções pré-estabelecido, pois algumas das seções destes *patterns* não se aplicam a este estudo. Propomos então uma abordagem que combine os formatos narrativos e de seções. A fim de ilustrar o processo que levou a elaboração do formato utilizado neste trabalho, e tendo como referência o formato GoF e seus elementos [20], fazemos as seguintes observações:

- Sobre a seção **Nome**: Nomes são elementos utilizados para resumir/capturar, em poucas palavras, a “essência” de um problema, de suas soluções, consequências, e que estabelecerão um vocabulário (conjunto de estratégias) em um determinados domínio. Os *patterns* neste trabalho não têm tal objetivo, mas sim servir como ferramenta de documentação e análise, assim consideramos que não há real necessidade de dar nomes aos *patterns*.

- **Classificação**: Para classificar um *pattern* seria necessário estabelecer novas categorias. No formato GoF os *patterns* são classificados em termos de seu propósito e escopo, estabelecer critérios como estes está além do escopo deste trabalho.

- **Objetivo**: O objetivo de todos os *patterns* deste trabalho é essencialmente o mesmo, implementar um certo tipo de variabilidade, utilizando os mecanismos oferecidos pelas LFs. Não há um problema específico (objetivo) a ser resolvido.

- **Sinônimos**: Não há sinônimos se não existem nomes, além de não existirem outros estudos nos quais possíveis sinônimos poderiam ser identificados.

- **Motivação**: Não há cenários a serem descritos, dos quais se tenha conhecimento prévio de problemas de projeto, assim não há como apresentar uma motivação para a existência do *pattern*.

- **Aplicabilidade**: Assim como no caso da motivação, não há conhecimento prévio de situações em que o *pattern* pode ser aplicado, ou de como reconhecer tais situações.

- **Estrutura**: Este é um elemento particular ao formato GoF devido a sua relação com o paradigma OO, nenhum outro dos formatos citados anteriormente utiliza representações visuais (diagramas).

- **Uso comum**: Mesmo que fosse possível identificar em aplicações reais, a ocorrência de trechos de código utilizando os mesmos conceitos, e com estrutura similar a apresentada no *pattern*, está além do escopo deste trabalho fazê-lo.

O formato de Coplien apresenta seções mais gerais, e é significativamente mais flexível, porém para alguma de suas seções cabem as mesmas considerações feitas ao formato GoF. Consideramos ainda que, no contexto deste trabalho, o maior generalismo das seções de Coplien podem introduzir redundâncias desnecessárias. Assim optamos por capturar os elementos essenciais das "soluções" apresentadas neste trabalho, em um formato particular de *pattern*, que se apresenta sob a seguinte estrutura:

**Enunciado** Uma breve descrição textual da prova de conceito. Similar, até certo ponto, ao paragrafo introdutório de Alexander, ou às seções, contexto e motivação, de Coplien e GoF. Porém, sem o mesmo peso ou conotação, trata-se de uma mera "ilustração" do cenário considerado para implementação da variabilidade.

**Fonte** Trechos de código, na linguagem Haskell, ilustrando uma possível maneira de se satisfazer uma certa variabilidade.

**Variabilidades** Um conjunto de "palavras-chave" que apresentam explicitamente os elementos (tipos) de variabilidades que o *pattern* busca satisfazer/implementar.

**Mecanismos** Um conjunto de “palavras-chave” que apresentam explicitamente os principais mecanismos (participantes) utilizados para satisfazer uma variabilidade.

**Discussão** A análise da implementação. Onde se discorre sobre como a variabilidade é satisfeita, como os mecanismos interagem, possíveis forças e consequências identificadas durante a implementação, e possíveis alternativas para a satisfação da(s) variabilidade(s).

# Capítulo 5

# Resultados

Variabilidades e seus modelos dependem dos requisitos de diferentes indivíduos/atores, logo não são absolutas. Os cenários aqui apresentados podem não representar a realidade dos requisitos de certos domínios de software, nem como softwares são distribuídos nesses domínios, e buscam meramente ilustrar a presença de variabilidades nos mesmos.

Neste capítulo apresentamos as implementações e análises destes cenários, na forma de *patterns*, conforme estrutura proposta na seção 4.4.

Como nem todos os elementos dos *patterns* são demarcados por seções ou itens, sua apresentação se dá conforme a ordem dos elementos definida na seção 4.4, primeiro o enunciado, depois código, e assim sucessivamente. Além disso, por não apresentarem nomes, os *patterns* são identificados e referenciados por meio de um numeral romano, que se localiza à direita do separador ❖ ❖ ❖, que indica o início de um novo *pattern*.

A maioria dos *patterns* aborda mais de uma estratégia para a satisfação de um certo tipo de variabilidade, porém não são todos dentre estes a apresentar implementações (código), que chamaremos de alternativas. Para os casos onde isso ocorrer, considerarmos que os conceitos abordados são claros e intuitivos, não havendo a necessidade de exemplificar por meio de um trecho, possivelmente extenso, de código.

Separamos os *patterns* em duas categorias, variabilidades e restrições, por considerarmos que estas representam conceitos distintos. Trataremos primeiro das variabilidades, e posteriormente, na seção 5.2, das restrições.

## 5.1 Variabilidades

❖ ❖ ❖ **I**

Seja o ponto de variação “Conjunto de Instruções”, em uma família[1] de compiladores de uma linguagem qualquer. Cada produto dessa família compila para apenas um, e somente um conjunto de instruções, dentre diversos possíveis.

Quadro 5.1: Síntese de código para um conjunto de instruções alvo.

```
1 data Arch = Amd64 | Armv8 | Mips64
2
3 type IntSrc     = String
4 type TargetCode = String
5
6 toTarget :: IntSrc -> Arch -> TargetCode
7 src `toTarget` Amd64  = toAmd64 src
8 src `toTarget` Armv8  = toArmv8 src
9 src `toTarget` Mips64 = toMips64 src
```

**Variabilidades** Alternativa (*xor*).

**Mecanismos** Tipos algébricos, casamento de padrões, *equational reasoning*.

Este *pattern* utiliza *equational reasoning* para prover variabilidade do tipo *xor*. A função `toTarget` é polimórfica e atua como interface para as diferentes variantes de “Conjunto de Instruções”. Cada instância (equação) da função `toTarget` realiza casamento de padrão contra seu segundo parâmetro, e em caso de “sucesso”, delega a responsabilidade de sintetizar código para uma função "auxiliar".

Cada uma das funções `toAmd64`, `toArmv8`, `toMips64` é responsável por implementar uma das possíveis variantes. Como as equações que compõe a definição de `toTarget` são disjuntas, apenas uma das instâncias será avaliada, logo apenas uma das variantes será possível.

O casamento de padrão neste *pattern* só é possível devida à definição do tipo `Arch`, linha 1 do quadro 5.1. `Arch` é um enumerável, que permite a expansão do número de variantes de maneira bastante simples, para cada novo conjunto de instruções basta definir uma nova instância de `Arch`, e uma nova equação para `toTarget`.

[1] O termo família será utilizado como sinônimo de LPS.

Embora, da forma como está apresentada, esta solução seja bastante simples e intuitiva, caso queiramos, por uma melhor estruturação do projeto, maior nível de separação das funcionalidades, dentre outras justificativas, representar o ponto de variação "Conjunto de Instruções" e suas variantes como módulos, algumas considerações são necessárias.

Suponhamos, para fins de simplicidade, que apenas um módulo[2] atuará como interface para o ponto de variação e suas variantes. Neste caso existem diferentes estratégias para satisfazer a variabilidade do tipo *xor*. A primeira sendo, "em time que está ganhando não se mexe", o módulo exportará a função `toTarget` e a variabilidade será satisfeita da mesma forma.

Uma segunda possibilidade é abandonar a ideia de utilizar uma função polimórfica como interface. O módulo exportará todas as funções de síntese (variantes), e apenas aquela particular a um determinado produto será importada.

```
import Module.Name (function)
```

Ambas as estratégias que utilizam módulos são pouco flexíveis. A primeira se utiliza de parametrização, por meio do tipo `Arch`, e estando a definição de `Arch` encapsulada em um módulo, não será possível estendê-la, ou mesmo estender às equações de `toTarget`. A segunda estratégia não necessita da definição de `Arch`, porém as funções de síntese estarão isoladas no módulo e novas funções não poderão ser facilmente incluídas.

❖ ❖ ❖

## II

Um editor de texto deve exibir o conteúdo proveniente de alguma *stream* de entrada em algum dispositivo de saída. De modo particular, é interessante, em algumas situações, que certos elementos do texto sejam exibidos de maneira especial, palavras-chave sejam destacadas, ou que caracteres especiais sejam exibidos.

Quadro 5.2: Funções para transformação de uma *stream* de caracteres.

```
1 type Displayable = String
2
3 display :: (String -> Displayable) -> String -> Displayable
4 display f = f
5
6 plain :: String -> Displayable
```

[2]Uma maior separação das funcionalidades (granularidade) é possível, porém não é relevante nesta argumentação.

```
7  plain = id
8
9  whiteSpaces :: String -> Displayable
10 whiteSpaces = foldr printBlank []
```

**Variabilidades** Obrigatória, opcional.

**Mecanismos** Funções de alta ordem.

Este *pattern* descreve uma configuração, onde um ponto de variação apresenta um comportamento padrão (obrigatório), ao mesmo tempo que é possível estender este comportamento (opcional), ou incorporar comportamentos independentes (opcional).

Na implementação ilustrada no quadro 5.2, a função `display` é responsável por "transformar" uma *stream* de texto em um "valor" que possa ser exibido por algum componente da interface com o usuário. A função `display` nada mais é que um intermediário responsável por aplicar uma segunda função, que de fato implementa o comportamento desejado.

As demais funções, `plain` e `whiteSpaces`, implementam efetivamente os comportamentos possíveis. Sendo que `plain` representa o comportamento padrão, manter o conteúdo da *stream* de entrada inalterado, qualquer outra função passada como parâmetro para `display` representará um comportamento opcional. A função `whiteSpaces` é responsável por substituir toda ocorrência de '␣'[3] por um caractere "visível".

Caso as variantes opcionais representem extensões do comportamento obrigatório, ao invés de adicionar funcionalidades independentes, o uso de funções de alta ordem como apresentado neste *pattern*, pode não ser o ideal. Nesse caso o comportamento padrão teria de ser incorporado de alguma forma, a cada uma das funções que implementam as extensões. Poderíamos incorporar este comportamento por meio de composição de funções, de forma similar à ilustrada ao final das discussões do *pattern* III.

Novamente podemos "incrementar" nossa implementação com o uso de módulos. O comportamento obrigatório pode ser representado isolando as funções `display` e `plain`, que serão exportadas por um módulo, e todos os produtos da linha deverão incluir este comportamento (módulo). Por sua vez os comportamentos opcionais podem ser implementados em módulos distintos, e serem incluídos apenas quando fizerem parte de um produto.

[3] O símbolo ␣ representa um espaço (ASCII 32).

Quadro 5.3: Implementação alternativa para variantes opcionais.

```
1 import Module.Default
2
3 import Module.WhiteSpaces
4 import Module.Colored
```

A primeira linha do quadro 5.3 implementa o comportamento padrão, e as linhas seguintes duas variantes opcionais.

Considerando que o termo "comportamento" pode parecer inapropriado, encerramos as discussões deste *pattern* com um "exercício" para compreensão das variantes opcionais. Utilizamos o termo comportamento para ilustrar o fato de que o software normalmente "opera" de uma certa maneira, mas pode ter sua operação ligeiramente modificada, ou incorporar elementos a sua operação. Uma pessoa age (se comporta) de uma certa forma, porém pode modificar seu comportamento, ou incorporar maneirismos a sua forma de agir.

Em resumo, o termo comportamento poderia ser substituído por algoritmo, estratégia, *feature*, entre outros "sinônimos". No domínio automobilístico, todo carro apresenta variantes e pontos de variação obrigatórios como "Motor", "Transmissão", "Carroceria", e opcionalmente pode apresentar componentes opcionais como "Engate p/ Reboque", onde componente pode ser considerado "sinônimo" de comportamento.

❖ ❖ ❖

**III**

O ponto de variação "Filtro de Ruído", em uma família de editores de imagens, deve ser satisfeito pom um mínimo de $n$ variantes. Não há um limite para o número máximo de variantes, porém qualquer subconjunto de variantes, de tamanho maior ou igual a $n$, é possível.

Quadro 5.4: Módulo que implementa "Filtro de Ruído" e suas variantes.

```
1  module Filter (aImage, avg, barlett, med, sobel) where
2
3  data Image a = Img [a] deriving Show
4
5  avg :: Integral a => Image a -> Image a
6  avg (Img i) = Img [sum' `div` length' | _ <- i]
7
8  barlett :: Integral a => Image a -> Image a
9  barlett (Img i) = Img [1 | _ <- i]
10
11 med :: Integral a => Image a -> Image a
12 med (Img i)
13      | odd $ length i = Img [i !! center | _ <- i]
```

```
14      | otherwise      = Img [centerAvg | _ <- i]
15
16 sobel :: Integral a => Image a -> Image a
17 sobel (Img i) = Img [0 | _ <- i]
```

Quadro 5.5: Um produto que seleciona os filtros de Barlett e da mediana.

```
1 import Filter (aImage, barlett, med)
2
3 -- some extra code
```

**Variabilidades** Alternativa (*select*).

**Mecanismos** Módulos.

Neste *pattern* um conjunto mínimo de $n$ variantes opcionais é obrigatório, porém qualquer configuração destas variantes é possível. Suponhamos para nossa discussão, que um mínimo de dois filtros deve estar presente em cada produto da família.

Dividimos a satisfação da variabilidade em duas "etapas". No quadro 5.4 está representada a implementação do módulo `Filter`, nele estão codificados cada um dos filtros (variantes) que podem satisfazer o ponto de variação "Filtro de Ruído". O módulo também exporta, na linha 1, todo o conjunto de possíveis variantes que será "configurado" em cada produto da linha.

Por sua vez, o quadro 5.5, ilustra a implementação de um dos possíveis produtos da família de editores de imagens. Na primeira linha do quadro são importadas as variantes que integrarão o produto, neste exemplo, os filtros de Barlett e da mediana, satisfazendo os requisitos mínimos para um produto da linha. Outros produtos poderiam ser obtidos da mesma forma, bastando apenas modificar as funções que compõem o `import`, ou importar um número maior de filtros.

Escolhemos para a implementação do *pattern*, concentrar todas as variantes em um único módulo. Uma maior granularidade seria possível de duas maneiras. A primeira, implementar cada um dos filtros em um módulo distinto. Assim para se satisfazer a variabilidade seria necessário, em cada produto, importar individualmente cada um dos módulos que o comporiam.

A segunda alternativa, implementaria as variantes em módulos distintos, porém utilizaria um módulo auxiliar, que atuaria como facilitador. Neste caso a configuração de cada produto seria a mesma apresentada no quadro 5.5.

Quadro 5.6: Implementação alternativa para *select*.

```
1 module Filter (aImage, avg, barlett, med, sobel) where
2
3 import Image
4
5 import Avg
6 import Barlett
7 import Med
8 import Sobel
```

Ambas as implementações alternativas resultam na introdução de "complexidades" relacionadas ao tipo `Image a`. Seria necessário definir `Image a` separadamente e importá-lo onde fosse necessário, em cada módulo que define uma função sobre `Image a`. É importante também salientar, que não há mecanismo em Haskell, que possibilite garantir a cardinalidade atrelada a variabilidade do tipo *select*.

Este é o primeiro *pattern* que aborda a satisfação da variabilidade por meio do uso de módulos, embora estratégias similares tenham sido brevemente discutidas em *patterns* anteriores. O motivo desta abordagem aqui, é a dificuldade para representar a seleção de variantes por meio de conceitos mais "básicos", como funções.

Consideremos, por exemplo, tratar da variabilidade do tipo *select*, em termos de funções de alta ordem. Para tal, imaginemos que o *pattern* II fosse expresso em termos de um conjunto de alternativas das quais, apenas uma é obrigatória. Assumindo que todas as variantes podem ser compostas, podemos satisfazer a variabilidade por meio da composição de um conjunto de funções, cada uma representando uma variante.

```
display (whiteSpaces . plain)
```

Assim cada produto comporia um conjunto distinto de funções, neste caso, `whiteSpaces` e `plain`, onde `(.)` representa a composição. A dificuldade em representar a satisfação de variabilidades não está na complexidade da codificação, mas no conjunto limitado de cenários em que uma estratégia pode ser aplicada de maneira eficiente, neste caso, apenas quando um dos conjuntos de variantes puder ser composto.

❖ ❖ ❖ **IV**

Seja a biblioteca que implementa uma estrutura de dados do tipo árvore binária. Um conjunto mínimo, obrigatório, de operações sobre a estrutura deve ser incluído.

Quadro 5.7: Implementação de uma árvore binária.

```
1  module Tree (singleton, treeInsert, treeSearch) where
2
3  data Tree a = Empty | Node a (Tree a) (Tree a) deriving Show
4
5  singleton :: a -> Tree a
6  singleton n = Node n Empty Empty
7
8  treeInsert :: Ord a => a -> Tree a -> Tree a
9  treeInsert n Empty = singleton n
10 treeInsert n (Node a left right)
11       | n == a = Node n left right
12       | n < a  = Node a (treeInsert n left) right
13       | n > a  = Node a left (treeInsert n right)
14
15 treeSearch :: Ord a => a -> Tree a -> Maybe a
16 treeSearch _ Empty = Nothing
17 treeSearch e (Node a left right)
18       | e == a = Just a
19       | e < a  = treeSearch e left
20       | e > a  = treeSearch e right
```

**Variabilidades** Obrigatória.

**Mecanismos** Tipos algébricos, tipos abstratos.

Antes de discutirmos o *pattern*, façamos algumas considerações em relação ao tipo de variabilidade que buscamos satisfazer aqui. Embora tenhamos abordado variabilidade do tipo obrigatória anteriormente (*pattern* II), tratávamos então de um produto com comportamento padrão, estendido, ou complementado, por comportamento(s) opcional(is).

A fim de obtermos um "fecho" dos tipos de variabilidades possíveis em uma LPS (trataremos de restrições/interações na seção 5.2), queremos ilustrar um cenário, onde dada a seleção de um ponto de variação $PV$, todo um conjunto de variantes $V$ de $PV$ deve compor um produto. Tratamos assim esse conjunto $V$ como obrigatório.

O "problema" que queremos discutir é, representar o comportamento padrão de um software não introduz nenhum mecanismo ou técnica, conceitualmente interessantes, para implementação de variabilidades. Todos os mecanismos apresentados até o momento poderiam ser utilizados para satisfazer variabilidades obrigatórias, o único requisito para tal é que eles estejam codificados no software, afinal eles representam uma funcionalidade obrigatória.

Reconhecendo o caráter confuso dessa afirmação, observemos a satisfação dos demais tipos de variabilidade, discutidos até o momento. Em ordem, no *pattern* I está claro que o uso de

um conjunto de mecanismos possibilita a satisfação de variabilidade do tipo *xor*. No *pattern* II, ilustramos o uso de funções de alta ordem na satisfação de variabilidades opcionais. Por último, implementamos variabilidade de tipo *select* por meio de módulos e *import* seletivo de funcionalidades.

Em todos os *patterns* foi possível verificar, claramente, que o uso de um ou mais mecanismos possibilitam/contribuem para implementação de um ou mais tipos de variabilidades, sendo possível identificar uma "relação" entre a variabilidade e um ou mais mecanismos. O mesmo não é possível para as variabilidades obrigatórias, consideremos agora a implementação deste *pattern*.

O módulo `Tree` define um novo tipo de mesmo nome, e em seguida são definidas um conjunto de funções, que representam as operações (variantes) obrigatórias sobre a estrutura de dados árvore, criação, inserção, e busca, respectivamente. As funções são exportadas pelo módulo, bastando para satisfação da variabilidade, incluir o módulo nos produtos.

Diferente do paradigma OO, onde mecanismos como os padrões de projeto podem ser classificados em termos do tipo de variabilidade que satisfazem, e alguns destes padrões são considerados inapropriados para implementação de variabilidades obrigatórias/mandatórias, como o padrão Decorator [14], consideramos neste estudo que qualquer mecanismo funcional é apropriado[4] para implementação de variabilidades obrigatórias.

O que queremos ilustrar é que as variabilidades obrigatórias são satisfeitas devido a sua natureza, e não pela natureza de sua implementação. Implementamos variabilidade obrigatória no *pattern* II por meio de funções de alta ordem. Se alterarmos ligeiramente o cenário do *pattern* III, para que todo o conjunto de filtros apresentados seja obrigatório, tanto a estratégia que utiliza módulos, quanto a alternativa, que utiliza composição de funções, poderiam ser utilizadas para implementar o cenário.

## 5.2 Restrições

Restrições não representam variabilidades propriamente ditas, mas as interações entre elas. Ainda assim, a presença de restrições pode dar origem a variabilidades que não são ilustradas

[4]Nenhum estudo foi realizado para determinar um limiar para uma definição mais concreta do termo. Complexidade da implementação, clareza, entre outros fatores poderiam ser considerados para uma melhor definição, fazê-lo está além do escopo deste trabalho.

pelo modelo de variabilidades, como no cenário descrito pelo *pattern* VII.

Restrições são facilmente visualizadas quando tratadas em termos de configuração, quando por exemplo, um produto é derivado pela aplicação de um pré-processador (`#ifdef funcionalidade...#else...`), ou por um *script*. Pode não ser possível expressar o mesmo tão claramente quando desconsideramos o uso de ferramentas como pré-processadores, *scripts*, e/ou *Integrated Development Environments* (IDEs).

Esta seção busca ilustrar a satisfação de restrições, pela linguagem Haskell, sem depender de qualquer mecanismo de configuração como os apresentados[5].

❖ ❖ ❖ **V**

Em uma linha de calculadoras onde a representação de diferentes conjuntos numéricos, como o dos números racionais ou complexos é possível, a seleção do ponto de variação "Operações sobre $\mathbb{C}$", requer a inclusão da variante "Complexos", que possibilidade a representação do conjunto numérico homônimo.

Quadro 5.8: Definição da variante "Complexos".

```
1 module Complex (Complex(..)) where
2
3 data Complex a = a :+ a deriving Show
```

Quadro 5.9: Operações sobre números complexos.

```
1  import Complex
2
3  add :: Num a => Complex a -> Complex a -> Complex a
4  (a:+b) `add` (c:+d) = (a + c) :+ (b + d)
5
6  mul :: Num a => Complex a -> Complex a -> Complex a
7  (a:+b) `mul` (c:+d) =
8       let real x y = x * y
9           imag x y = x * y
10      in (real a c - imag b d) :+ (imag a d + imag b c)
```

**Variabilidades** Restrição *requer*.

**Mecanismos** Tipos algébricos.

[5]É possível argumentar, que *imports* são equivalentes as diretivas `#include` de linguagens como C, onde a mesma é uma diretiva do pré-processador.

Neste *pattern* separamos a implementação da variante "Complexos", e do ponto de variação "Operações sobre $\mathbb{C}$". No quadro 5.8 é definido o tipo `Complex a`, que será responsável por garantir a restrição *requer*. Já no quadro 5.9 são definidas algumas funções que compõem o ponto de variação "Operações sobre $\mathbb{C}$".

A satisfação da restrição ocorre na assinatura das funções `add` e `mul`, onde o tipo `Complex a` é responsável por garantir que as mesmas operem apenas sobre o tipo de dados apropriado. Por este motivo a separação da implementação da variante, e do ponto de variação, não é necessária para a satisfação da restrição, porém esta separação é conceitualmente interessante.

A separação da implementação torna-se interessante, quando consideramos a existência de uma segunda camada capaz de satisfazer a restrição, onde além da assinatura das funções, a inclusão do módulo `Complex` também tem papel significativo. Em verdade, essa segunda camada está presente na implementação do *pattern*, sem a inclusão do módulo ocorrerão erros de compilação, dado que elementos como o construtor de tipo `Complex`, ou o construtor de dados '`:+`', não serão reconhecidos como símbolos válidos.

Essa separação possibilita identificar de maneira mais clara a satisfação da restrição. Sabendo que a implementação do ponto de variação (quadro 5.9) pode ocorrer na forma de um módulo, se ao implementar um produto qualquer da família de calculadoras incluirmos este módulo, teremos também que incluir o módulo `Complex` para que possamos obter um produto funcional (*pv_requires_v*). Embora a separação de elementos seja interessante, o grau de separação das componentes do software deve ser considerado com cautela, para não se introduzir complexidades desnecessárias.

Algo que pode estar passando desapercebido na implementação do *pattern*, é que o tipo parametrizado `Complex a` também está sobre efeito de uma restrição. As definições das funções `add` e `mul` (quadro 5.9) apresentam o seguinte tipo.

```
Num a => Complex a -> Complex a -> Complex a
```

Uma função qualquer toma dois parâmetros do tipo `Complex a` e retorna um valor de mesmo tipo, onde os possíveis "valores" das variáveis de tipo `a`, devem ser instâncias da classe de tipos `Num`, devem ser valores numéricos. Não faz sentido a existência de um tipo `Complex String`, onde as componentes de um número complexo sejam listas de caracteres, e não um número real e um imaginário.

Esse mecanismo chamado de restrição de classe também está presente em outros *patterns*. No *pattern* IV, `Tree a` é um tipo parametrizado, e para que as funções de inserção e busca executem corretamente, é necessário que os nós da árvore sejam de elementos que apresentem algum critério de ordenação, representandos pela classe de tipos `Ord`.

Classes de tipos[6] são interfaces para definir algum comportamento, quando um tipo se "enquadra" neste comportamento ele pode ser feito instância de uma certa classe de tipos [28]. Uma classe de tipos define algumas funções, e quando fazemos um tipo instância dessa classe, definimos o que essas funções significam para esse tipo [28]. Uma restrição de classe é um mecanismo que permite introduzir uma limitação/condição em relação a uma variável de tipo, um tipo `a` deve ser instância de uma classe de tipos `C`.

Embora tipos algébricos sejam um mecanismo bastante poderoso, acreditamos que observar a satisfação de restrições *requer*, do ponto de vista dos módulos é importante. De fato, a simples inclusão de um módulo, dada a seleção de uma variante ou ponto de variação, poderia satisfazer uma restrição do tipo *requer* (uma vez que este módulo implemente o ponto de variação ou variante requeridos). Discutiremos no *pattern* VII, a implementação de restrições *requer* no contexto da interação entre *features*, onde os módulos estarão "mais" presentes.

❖ ❖ ❖

## VI

Suponha uma aplicação *web* qualquer, onde um conjunto de funcionalidades/serviços básicos é oferecido, o acesso a essas funcionalidades é possível por meio da vinculação de uma conta provida por serviço de terceiro, provedores de e-mail ou redes sociais. A aplicação provê ainda um conjunto extra de funcionalidades/serviços, porém o acesso a estas funcionalidades está restrito apenas aos usuários cadastrados por meio de serviço próprio, onde dados particulares são coletados. A seleção da variante "Login Terceiro" excluirá o ponto de variação que provê o conjunto de funcionalidades extra, que chamaremos de "Pro".

Quadro 5.10: Conjunto de todas as funcionalidades do produto.

```
1 module FullFeatureSet ( basicFeature1
2                       , basicFeature2
3                       , proFeature1
4                       , proFeature2 ) where
```

[6] Não confundir com o conceito de classe das linguagens OO.

Quadro 5.11: Exclusão das funcionalidades “Pro”.

```
1 import FullFeatureSet hiding (proFeature1, proFeature2)
2
3 -- some extra code
```

**Variabilidades** Restrição *exclui*.

**Mecanismos** Módulos.

Para esta implementação escolhemos agrupar todas as funcionalidades em um único módulo, e então exportá-las todas. Assim o módulo que chamamos de `FullFeatureSet` é responsável por implementar mais de um ponto de variação/variantes. A restrição de tipo *exclui*, é satisfeita pela cláusula `hiding` do *import* realizado na linha 1 do quadro 5.11.

O papel da cláusula `hiding` é exatamente o indicado pelo seu nome, “esconder” um conjunto de funcionalidades (variantes/pontos de variação), no caso da implementação deste *pattern*, o trecho de código no quadro 5.11 representa um produto que inclui a variante “Login Terceiro”, portanto este deve excluir (esconder) as funcionalidades “Pro”, implementadas pelas funções `proFeature1` e `proFeature2`.

O leitor mais atento pode estar se perguntando, “O princípio utilizado para implementar este *pattern* não é o mesmo utilizado para implementar o *pattern* III? Qual a diferença? Por que essa nova abordagem?". Em termos objetivos, onde os fins justificam os meios, as duas soluções são iguais. O *import* seletivo utilizado no *pattern* III poderia ser utilizado para implementação da restrição *exclui*, importaríamos apenas as funções `basicFeature1` e `basicFeature2` ao invés de excluirmos as funcionalidades “Pro”.

De um ponto de vista mais crítico o uso da cláusula `hiding` é mais interessante, por três razões. Primeiro, utilizar `hiding` ilustra de maneira mais clara, que tipo de restrição está sendo satisfeita. Segundo, quando o conjunto de funcionalidades a ser excluído é muito menor do que o conjunto a ser incluído, `hiding` é mais conveniente, e é bom que tenhamos essa opção. E por último, no caso de termos informação apenas a respeito do que deve ser excluído, ou de não querermos considerar as complexidades a respeito do que deve ser incluído no produto, temos em `hiding` um elemento de abstração interessante.

Poderíamos separar a implementação das funcionalidades básicas, e adicionais, em módulos distintos, assim o módulo `FullFeatureSet` atuaria apenas como interface.

Quadro 5.12: Implementação alternativa para restrição *exclui*.

```
1 module FullFeatureSet (module Basic, module Pro) where
2
3 import Basic
4 import Pro
```

Aqui `FullFeatureSet` exporta outros dois módulos, e não cada uma das funcionalidades, como no quadro 5.10. Desta forma, na implementação de um produto que seleciona a variante "Login Terceiro", excluiríamos todo o módulo que implementa o ponto de variação "Pro", sem precisarmos saber, especificamente, quais funcionalidades devem ser excluídas.

```
import FullFeatureSet hiding (Pro)
```

Além das estratégias apresentadas até o momento, o uso de tipos também poderia ser considerado na implementação das restrições *exclui*. O uso de tipos seria bastante simples, na verdade, já vimos no *pattern* V como implementar restrições *requer* por meio de tipos, para fazermos algo similar aqui, bastaria uma mudança de ponto de vista. Ao invés de garantir que uma função opere apenas sobre o tipo apropriado, uma função requer um certo tipo, podemos imaginar que um tipo "exclui" a possibilidade de uma função operar sobre tipos inapropriados.

❖ ❖ ❖ **VII**

Em um sistema de multimídia para dispositivos móveis, o ponto de variação "Mídia" é obrigatório, e ao menos uma de suas variantes deve compor um produto. O ponto de variação "Mídia" requer a inclusão da variante "Cópia", que provê a capacidade de copiar/transferir arquivos dos diferentes tipos de mídia (variantes), presentes no sistema.

Quadro 5.13: Implementção da variante "Cópia".

```
1 module Copy (Path, CopyAble(..)) where
2
3 type Path = String
4
5 class CopyAble a where
6     copy :: a -> Path -> Path -> (a, Path)
```

Quadro 5.14: Módulo que implementa a interação entre *features*.

```
1  module CopyMedia (copyMedia) where
2
3  import Copy
4  import Media.Photo
5  import Media.Music
6  import Media.Video
7
8  instance CopyAble Photo where
9       copy _   _ "" = error "Could not copy photo!"
10      copy mma _ p  = (mma, p)
11
12 instance CopyAble Music where
13      copy _   _ "" = error "Could not copy music!"
14      copy mma _ p = (mma, p)
15
16 instance CopyAble Video where
17      copy _   _ "" = error "Could not copy video!"
18      copy mma _ p = (mma, p)
19
20 copyMedia :: CopyAble a => a -> Path -> Path -> (a, Path)
21 copyMedia = copy
```

Quadro 5.15: Produto que inclui a mídia "Foto" e sua interação.

```
1  import Media.Photo
2  import CopyMedia
3
4  -- some extra code
```

**Variabilidades** Restrição *requer*, alternativa (*select*).

**Mecanismos** Tipos algébricos, tipos abstratos, classes de tipos.

Este *pattern* adapta o cenário e o problema descritos no artigo [29]. O artigo trata de FOSD, especificamente da implementação de *features* que surgem da interação entre outras *features*. Consideramos que o problema apresentado no artigo pode ser descrito em termos de restrições do tipo *requer*, como fizemos no "enunciado" deste *pattern*[7].

O artigo argumenta que ao realizar uma implementação orientada a *features*, o ideal é representar cada uma das *features* como um módulo independente, porém ao se considerar a implementação da interação entre duas *features* $F1$ e $F2$, tanto o módulo que representa $F1$, quanto o módulo que representa $F2$, não devem implementar sua interação. A solução então

[7]Caso necessário, para melhor compreensão do problema em questão, recomendamos a leitura do artigo.

seria implementar a interação entre $F1$ e $F2$ em um terceiro módulo independente, e essa é a abordagem que adotamos em nossa implementação.

Dividimos nossa implementação entre a variante "Cópia" e as três possíveis variantes de "Mídia", que para nosso cenário são "Foto", "Música", e "Vídeo". Além do módulo que representa a interação entre "Cópia" e as demais.

O quadro 5.13 ilustra a implementação do módulo `Copy`, que representa a *feature* "Cópia". No módulo é definida a classe de tipos `CopyAble`, que "classifica" os elementos que podem ser copiados e define a interface do comportamento `copy`, inerente as instâncias da classe.

Em nossa implementação o módulo `CopyMedia` representará a interação entre *features*, e está representado no quadro 5.14. No módulo fazemos cada um dos tipos de mídia instâncias de `CopyAble`, e definimos a função `copy` para cada uma delas. Por fim definimos a função polimórfica `copyMedia`, que será a interface para cópias de dados multimídia.

Por último, temos no quadro 5.15 nossa implementação de um produto, em que apenas uma variante/*feature*, "Foto", é selecionada. Embora os módulos `Media.*` sejam componentes integrantes da implementação, optamos por não apresentá-los, neles estão tão somente implementados os tipos `Photo`, `Music`, e `Video`; incluí-los ocuparia um espaço considerável e não traria nenhum benefício real para compreensão da solução.

Neste *pattern* são satisfeitos pelo menos dois tipos de variabilidade. Primeiro a seleção, onde no mínimo uma variante de "Mídia" deve compor o produto é satisfeita pelo *import* do módulo `Media.Photo`, na linha 1 do quadro 5.15. E por último a interação entre *features*, que representamos em termos de uma restrição *requer*.

A interação entre as *features* é possível pela colaboração dos tipos definidos nos módulos `Media.*` e a classe de tipos definida no módulo `Copy`. Essa colaboração é implementada no módulo `CopyMedia`, ao fazermos os tipos `Photo`, `Music`, e `Video` instâncias de `CopyAble`, isso permitirá a definição da função `copyMedia` como interface comum para a cópia de diferentes tipos de mídia, da forma como está ilustrada no quadro 5.14.

Poderíamos, de forma alternativa, utilizar equações e casamento de padrões para obter resultado "similar", porém teríamos que trabalhar com um único tipo `Media`, e definirmos uma função `copyMedia` composta por diversas equações, neste caso nove, pois este é número total de possíveis valores dentre os três tipos definidos, `Photo`, `Music`, `Video`. Trabalhar com

um único tipo, e um grande número de equações, pode prejudicar a modularidade do sistema, especialmente se novos tipos de mídia forem introduzidos ao sistema.

O que queremos apresentar por meio deste *pattern* é a ideia de que interações entre *features* podem ser expressas como consequência de restrições do tipo *requer*. Em nosso cenário, a existência da *feature* `CopyMedia` é consequência da seleção de um tipo de mídia para compor nosso produto, o que requer que a cópia dos arquivos dessa mídia seja possível no sistema.

# Capítulo 6

# Conclusões e Considerações Finais

Apresentaremos nossas conclusões conforme as questões e objetivos específicos deste trabalho, definidos ao final do capítulo 1.

Primeiro trataremos da verificação da satisfação das variabilidades pela linguagem Haskell. Consideramos, conforme resultados apresentados no capítulo 5, e amparados pelos argumentos apresentados nas discussões dos *patterns*, que a satisfação de variabilidades pelos mecanismos oferecidos pelas LFs é possível, e foi atingida neste trabalho.

Em relação a verificação de alguns dos mecanismos oferecidos pela Haskell para satisfação de variabilidades, apresentamos a tabela 6.1, onde estão resumidos quais foram os mecanismos utilizados nas implementações dos *patterns*, e para qual tipo de variabilidade foram aplicados.

| **Mecanismo** | **Variabilidade** |
|---|---|
| Tipos algébricos | Obrigatória<br>Alternativa (*xor*)<br>Restrição *exclui*<br>Restrição *requer* |
| * Casamento de padrões e *equational reasoning* | Alternativa (*xor*)<br>Restrição *requer* |
| Funções de alta ordem | Obrigatória<br>Opcional<br>Alternativa (*select*) |
| Classes de tipos | Restrição *requer* |
| ** Tipos abstratos e módulos | Alternativa (*select*)<br>Obrigatória<br>Restrição *exclui*<br>Restrição *requer* |

Tabela 6.1: Resumo dos mecanismos aplicados nos *patterns*.

É importante salientar que a tabela 6.1 não ilustra todo o potencial, nem mesmo todos os mecanismos funcionais disponíveis para implementação de variabilidades, mas apenas aqueles utilizados durante a elaboração dos *patterns*. Um exemplo da relação mecanismo/variabilidade, que não esta representada na tabela 6.1, é a possível aplicação, em determinadas circunstâncias, das classes de tipo como mecanismo para implementar variabilidade opcional[1].

Não foi possível, dado o escopo deste trabalho, verificar todo mecanismo, em relação a implementação de um certo tipo de variabilidade. As seções a seguir apresentam considerações gerais sobre o trabalho, além de sugestões de estudos complementares.

## 6.1 Sobre Alguns Mecanismos

Seguem algumas observações sobre os mecanismos destacados, por meio de asteriscos, na tabela 6.1.

* Embora representem conceitos distintos, o casamento de padrões, e o *equational reasoning*, estão fortemente relacionados. O uso de casamento de padrões ocorre quando da aplicação de algum tipo de *equational reasoning*, apresentamos assim, ambos, na forma de um único mecanismo.

** Antes de mais nada, observemos, que tanto tipos abstratos, quanto módulos, não são mecanismos exclusivos das LFs. Porém, os módulos da linguagem Haskell introduzem ferramentas bastante interessantes para implementação de variabilidades, além de ser o principal habilitador dos tipos abstratos na linguagem, quando definimos um tipo algébrico e associamos ao mesmo um conjunto de operações, como ocorre em diversos *patterns*. Por esse motivo, ambos estão categorizados como um único mecanismo.

## 6.2 Sobre a Generalidade dos *Patterns*

Argumentamos aqui que, embora o conjunto de domínios representado pelos *patterns* que compõem este trabalho seja limitado, da forma como foram elaborados, os *patterns* são abstratos o suficiente para que sejam "transferíveis", aplicáveis a outros domínios.

[1] Não apresentaremos implementações ou argumentos sobre o uso do mecanismo, como descrito no exemplo, a afirmação pode ser verificada, ou contestada, pelo leitor.

Consideremos o enunciado do *pattern* II, que trata de um cenário onde um software apresenta um comportamento padrão, e estende este comportamento por meio de funcionalidades opcionais. O domínio representado no *pattern* II é o dos editores de texto, poderíamos facilmente transferir o conceito capturado pelo mesmo para o domínio dos gerenciadores de aquivos, como segue.

"Um gerenciador de arquivos deve listar o conteúdo de um diretório, uma listagem simples é obrigatória, e listagens que apresentam estatísticas sobre os arquivos, ou exibem os arquivos em estrutura hierárquica, são opcionais".

Tanto o enunciado captura o mesmo princípio, quanto os mecanismos utilizados no *pattern* II poderiam ser aplicados na implementação do gerenciador de arquivos. Argumentamos ainda, que a generalidade descrita também se aplica aos demais *patterns*.

## 6.3 Sobre a Estratégia Adotada

A estratégia que adotamos neste trabalho é similar a adotada em [20], onde consideramos um cenário, apresentamos implementações conceituais do mesmo, e então discutimos suas contribuições, e possíveis pontos negativos. Chamaremos esta estratégia de *top-down*, pois partindo de um contexto com maior nível de abstração, a camada da aplicação, por meio das provas de conceito atingimos um nível mais baixo, a implementação, onde identificamos quais o mecanismos habilitam a implementação das variabilidades.

No decorrer da fase final deste trabalho, poderíamos ter considerado que uma abordagem *bottom-up* seria "mais apropriada" para a realização do estudo, apresentando de forma mais clara um conjunto de mecanismos, e quais variabilidades seriam resolvidas pelos mesmos.

Partindo de um conjunto de mecanismos "característicos" das LFs, seriam consideradas/avaliadas quais variabilidades poderiam ser satisfeitas por um determinado mecanismo, para então se elaborar/obter um cenário onde tal mecanismo poderia ser aplicado. Porém, essa estratégia pode requerer uma maior experiência no uso das LFs, ou de um maior aprofundamento no estudo das mesmas, além de ser necessário delimitar um conjunto de mecanismos.

Outra observação que se pode fazer sobre a estratégia adotada neste trabalho, é a de que esta pode ser tendenciosa, implementar provas conceituais em diferentes domínios, pode ser um indicativo de que a satisfação de variabilidades só é possível nos cenários "escolhidos", e

não é possível em outros.

De certa forma, toda pesquisa é tendenciosa, um dos fatores que torna a estratégia adotada interessante, é justamente ser capaz de ilustrar a implementação de variabilidades em diferentes domínios, e como discutimos na seção 6.2 o generalismo apresentado pelos *patterns* permite sua adaptação para um conjunto muito maior de cenários/domínios do que foi apresentado.

Poderíamos argumentar que realizar um trabalho com base em um domínio específico, também produziria resultados tendenciosos, pois o trabalho seria muito restrito e o domínio escolhido favoreceria a obtenção de resultados "favoráveis". O que ocorre é que as abordagens distintas são na verdade complementares, e não excludentes, vide seção 6.4.

## 6.4 Trabalhos Futuros

Diversos trabalhos podem ser considerados para o futuro desta pesquisa. Primeiro, a complementação deste trabalho com o estudo de um ou mais mecanismos, como os *monads*, e/ou a incorporação de ferramentas, como o Cabal, um sistema para construção e empacotamento de bibliotecas e programas, para Haskell, na satisfação de variabilidades e derivação de produtos em uma LPS.

Outra alternativa é uma avaliação do desempenho da linguagem Haskell na implementação de uma aplicação "real", que apresente um conjunto razoável de funcionalidades, modelada com base nos conceitos das LPSs ou FOSD, onde a satisfação de variabilidades pode ser explorada, para então obter um estudo similar a [11].

Também é possível um estudo que aplica a estratégia *bottom-up* discutida na seção 6.3, e por meio dessa abordagem verificar quais os mecanismos das LFs podem ser utilizados para implementar variabilidades, além de quais variabilidades podem ser satisfeitas por estes mecanismos, e talvez realizar um estudo comparativo em relação aos resultados deste trabalho.

Uma outra possibilidade, é realizar uma coleta em bases de código, similar a aplicada em [12], de aplicações desenvolvidas principalmente em Haskell, para então efetuar uma análise dos códigos, buscando identificar variabilidades e verificar quais os mecanismos estão sendo utilizados para sua implementação. É possível uma ampliação do escopo do estudo, onde mais de uma linguagem funcional são consideradas para coleta e analise, e um estudo comparativo com os resultados deste trabalho, ou dos trabalhos sugeridos anteriormente.

# Referências Bibliográficas


[1] PHOL, K.; BöCKLE, G.; LINDEN, F. van der. *Software Product Lines Engineering: Foundations, Principles, and Techniques*. Berlin, DE: Springer, 2005.

[2] LINDEN, F. van der; SCHMID, K.; ROMMES, E. *Software Product Lines in Action*. Berlin, DE: Springer, 2007.

[3] WIRTH, N. A brief history of software engineering. *IEEE Anals of the History of Computing*, p. 32–39, Julho/Setembro 2008.

[4] CLEMENTS, P.; NORTHTROP, L. *Software Product Lines: Practice and Patterns*. 3. ed. US: Addison-Wesley, 2001.

[5] KANG, K. C.; SUGUMARAN, V.; PARK, S. *Applied Software Product Line Engineering*. Boca Raton, FL, US: CRC Press, 2010.

[6] ANASTASOPOULOS, M. Implementing product line variabilities. *ACM Sigsoft Software Engineering Notes*, New York, NY, US, p. 109–117, Março 2001.

[7] KURDI, H. A. Review on aspect oriented programming. *International Journal of Advanced Computer Science and Applications*, p. 22–27, 2013.

[8] SCOTT, M. L. *Programming Languages Pragmatics*. 3. ed. San Francisco, CA, US: Morgan Kaufman, 2006.

[9] O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. *Real World Haskell*. 1. ed. Sabastopol, CA, US: O'Reilly, 2008.

[10] WADLER, P. Why no one uses functional programming languages. *ACM Sigplan Notices*, New York, NY, US, p. 23–27, Agosto 1998.

[11] SAMPSON, C. J. Experience report: Haskell in the real world. *Proceddings of the 14th ACM SIGPLAN International Conference on Funtional Programming*, p. 185–190, Setembro 2009.

[12] RAY, B. et al. A large scale study of programming languages and code quality in github. *Proceddings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, p. 155–165, Novembro 2014.

[13] HUDAK, P. Conception, evolution and application of functional programming languages. *ACM Computing Surveys (CSUR)*, New York, NY, US, p. 359–411, Setembro 1989.

[14] APEL, S. et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, DE: Springer, 2013.

[15] MICHAELSON, G. *An Introduction to Functional Programming Through Lambda Calculus*. 1. ed. Cheltenham, UK: Addison-Wesley, 1989.

[16] WADSWORTH, C. *Semantics and Pragmatics of the Lambda Calculus*. Tese (Doutorado) — University of Oxford, 1991.

[17] HUGES, J. et al. A history of haskell being lazy with class. *Proceddings of the Third ACM SIGPLAN on History of Programming Languages*, p. 1–55, Abril 2007.

[18] HUGES, J. Why functional programming matters. *The Computer Journal*, v. 32, n. 2, p. 98–107, 1989.

[19] LEMMER, R. *Haskell Design Patterns*. 1. ed. US: Packt, 2015.

[20] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. Westoford, MA, US: Addison-Wesley, 1995.

[21] APEL, S. et al. Feature (de)composition in functional programming. *Proceedings of the 8th International Conference on Software Composition*, p. 9–26, Julho 2009.

[22] SABRY, A. What is a pure functional programming language. *Journal of Functional Programming*, p. 1–22, Janeiro 1993.

[23] MARLOW, S. *Haskell 2010 Language Report*. [S.l.], 2010.

[24] ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. *A Pattern Language*. 1. ed. New York, NY, US: Oxford University Press, 1977.

[25] COPLIEN, J. *Software Patterns*. 1. ed. New York, NY, US: SIGS Books & Multimedia, 2000.

[26] KOHLS, C. The structure of patterns. *Procedings of the 17th Conference on Pattern Languages of Programming*, n. 12, 2010.

[27] ANDRADE, R. M. de C. *Capture, Reuse, and Validation of Requirements and Analysis Patterns for Mobile Systems*. Tese (Doutorado) — University of Ottawa, 2001.

[28] LIPOVAČA, M. *Learn You a Haskell for a Great Good!* 1. ed. San Francisco, CA, US: No Starch Press, 2011.

[29] TAKEYAMA, F.; CHIBA, S. Implementing feature interactions with with generic feature modules. *Software Composition: 12th International Conference*, p. 81–96, Junho 2013.