UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

APLICANDO REGRAS DE PROGRAMAÇÃO PARA REFATORAÇÃO DE PROGRAMAS EM DAFNY: UMA LINGUAGEM IMPERATIVA COM ESPECIFICAÇÃO NATIVA

Larissa Oliveira Ribeiro da Paz

Trabalho de Graduação

Recife

FEVEREIRO DE 2015

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Larissa Oliveira Ribeiro da Paz

APLICANDO REGRAS DE PROGRAMAÇÃO PARA REFATORAÇÃO DE PROGRAMAS EM DAFNY: UMA LINGUAGEM IMPERATIVA COM ESPECIFICAÇÃO NATIVA

Trabalho apresentado ao Programa de GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO do CENTRO DE INFORMÁTICA da UNIVERSIDADE FEDERAL DE PERNAMBUCO como requisito parcial para obtenção do grau de Bacharel em CIÊNCIA DA COMPUTAÇÃO.

Orientador(a): *Prof Dr Márcio Lopes Cornélio*

Recife

FEVEREIRO DE 2015

À minha família, em especial aos meus avós.

Agradecimentos

Primeiramente agradeço a Deus por tudo que me foi proporcionado.

Gostaria de agradecer a minha família pelo apoio incondicional. Em especial aos meus pais e irmã, pela confiança quando nem mesmo eu acreditava que era possível, e por serem inspiração na vida. Meus tios e primos por terem me acolhido.

Aos professores que fizeram parte dessa jornada, em particular ao meu orientador Márcio Cornélio, pela extraordinária paciência e disposição para que esse trabalho fosse possível.

Aos colegas e amigos conquistados durante a graduação, por todos os desafios enfrentados juntos.

Aos amigos, que me apoiaram com tanto carinho e frequentemente foram um refúgio. Pelos momentos de alegria que dividimos.

A todos que direta ou indiretamente fizeram parte da minha formação.

Resumo

A refatoração é uma operação comum durante o desenvolvimento e manutenção de sistemas e consiste da modificação da estrutura do código sem que haja a alteração do comportamento do mesmo. Uma forma de garantir que um programa funciona de acordo com o determinado em especificação é por meio do *Design by Contract*, que através de verificações de condições garante a corretude do código. Portanto, se uma alteração no código não interferiu na verificação do sistema, o comportamento não foi alterado.

Para garantir que o estado do sistema seja consistente depois de uma modificação, leis de programação podem fazer a formalização desse processo. Essas leis descrevem o programa antes e depois de uma modificação. Para aplicação de cada uma delas, uma série de condições devem ser satisfeitas.

Neste trabalho foi investigado como mudanças em código fonte escrito na linguagem Dafny podem ser realizadas utilizando leis de programação. Para isso, foram analisadas construções presentes em Dafny para adaptar leis propostas para Java-JML e regras descritas para Haskell e Erlang. Como resultado, foi definido um conjunto de leis e regras de refatoração para Dafny. Para cada uma dessas leis e regras, foram escritos códigos de exemplo para simular situações em que elas se aplicariam.

Palavras-chave: Dafny, refatoração, leis de programação

Sumário

1. Introdução

Este capítulo apresenta os aspectos introdutórios desta monografia e está organizado em quatro seções. Na primeira, o contexto em que este trabalho foi escrito é retratado. Na segunda Seção, a motivação e os objetivos deste trabalho são apresentados. Posteriormente, a metodologia para realizar o trabalho é mostrada. E, por fim, a estrutura do documento é apresentada.

1.1. Contexto

Durante a vida útil de um software, frequentemente ocorre alterações de requisitos, que comumente implicam em mudanças internas no código. Com o amadurecimento desses sistemas há também a demanda de alterações para melhoramento da qualidade do código e de desempenho do sistema. Independentemente da mudança a ser realizada, é preciso bastante cautela para que essas alterações não sejam danosas para o comportamento de outras funcionalidades.

Uma metodologia utilizada para garantir que um sistema funcione de acordo com a especificação é o Projeto por Contrato (Design by Contract - DbC). Trata-se de uma abordagem de desenvolvimento de software que utiliza métodos formais, por meio da definição pré-condições, pós-condições e invariantes, a fim de facilitar a verificação e garantir a corretude dos sistemas.

Para tentar minimizar os danos decorrentes de alterações no código uma série de procedimentos devem ser empregados. Uma maneira de conduzir este processo é por meio da utilização de leis de programação, que estabelece um suporte para o desenvolvimento formal e rigoroso de sistemas [1]. As leis de programação já foram definidas e adaptadas para os diversos paradigmas de linguagens computacionais [2] [1] [3]. As leis de programação podem ser aplicadas não apenas no código do sistema, mas nos contratos também, para garantir o funcionamento adequado do sistema como todo.

Muitas linguagens vem sendo desenvolvidas com suporte à programação por contrato. Uma delas é Dafny [4,5], uma linguagem de programação que oferece suporte à verificação. Neste sentido, o principal objetivo de Dafny é permitir que programas não tenham erros em tempo de execução e também que funcionem de acordo com a especificação expressa por meio de anotações (contratos). Do ponto de vista de tipos, Dafny é considerada uma linguagem segura. Possui recursos de linguagem de programação imperativa como métodos (procedimentos), comandos de repetição e alocação dinâmica. Além destes, também possui recursos de linguagens de programação funcional (funções recursivas e tipos de dados indutivos e co-indutivos) [4] [5].

1.2. Motivação

Produzir sistemas com maior qualidade, em menor custo e tempo é um objetivo constante da Engenharia de Software. Porém, os sistemas de software não são estáticos, no sentido de que ao longo do tempo mudanças são requeridas e não podem ser realizadas de forma descuidada, para que sejam diminuídos os possíveis efeitos colaterais decorrentes delas.

Refatoração [6] define os passos a serem utilizados para a realização de mudanças em código, permitindo melhoria de fatores internos de qualidade de software como legibilidade e reuso, sem que as mudanças alterem o comportamento do software percebido pelo usuário. Apesar de algumas ferramentas de desenvolvimento oferecerem um certo suporte a esse processo, o que atualmente está comercialmente implementado é limitado e não é comumente utilizado em situações práticas, ou apresenta comportamento inadequado em certos cenários. O motivo desse comportamento está relacionado com o processo de implementação sem um processo rigoroso ou sistemático de formalização [3]. Esse é um dos primeiros passos para construção de ferramentas de automação das mudanças. O que pode gerar benefícios quanto a qualidade de código, como também referentes a tempo gasto por mudança, o que vem a diminuir o custo do projeto, ou seja, pode influenciar diretamente na eficiência da operação.

Diante dessas perspectivas definimos os objetivos desta pesquisa, que serão apresentados na próxima seção.

1.3. Objetivo do Trabalho

O objetivo deste Trabalho de Graduação é investigar como mudanças em código fonte escrito na linguagem Dafny podem ser realizadas de maneira sistemática utilizando leis de programação. Nesta direção, é necessário que, levando em consideração construções presentes em Dafny, as leis de programação sejam adaptadas para Dafny [4,5]. Por exemplo, leis que lidam com contratos de software escritas para a linguagem JML [3] serão avaliadas para adaptação para Dafny. Transformações de programas escritas para linguagens funcionais como Haskell e Erlang [7] [8] serão também avaliadas a fim de determinar quais podem ser adaptadas para Dafny.

1.4. Organização

Além desse capítulo introdutório, esse documento é organizado em mais três capítulos. No Capítulo 2 é apresentada a fundamentação teórica necessária para o embasamento dos tópicos abrangidos nos capítulos subsequentes. Nele são apresentados os conceitos referentes às Leis de Programação. Além disso, apresenta

uma introdução às características e sintaxe da linguagem Dafny [4]. Por fim, é feita uma síntese dos trabalhos relacionados.

No Capítulo 3, são apresentados os conjuntos de leis adaptadas para Dafny. É então abordada uma breve justificativa de quais aspectos levaram às alterações.

Finalmente, no Capítulo 4, serão apresentadas as conclusões deste trabalho, ressaltando as contribuições e limitações encontradas, tanto referentes a aspectos da linguagem quanto do trabalho desenvolvido. Além disso, são mostradas as perspectivas futuras. Finalmente, o capítulo descreve algumas considerações finais advindas da elaboração desta monografia.

2. Visão Geral

Esse capítulo apresenta uma visão geral nos aspectos teóricos envolvidos para o entendimento deste trabalho. Inicialmente iremos apresentar leis de programação, com sua importância no processo de desenvolvimento e manutenção de software. Em seguida será apresentada a linguagem Dafny, introduzindo as suas características e particularidades para justificar a sua escolha como estudo de caso. Posteriormente, iremos sumarizar os trabalhos relacionados desenvolvidos, tanto quanto com as Leis de Programação, como no processo de refatoração de programas, juntamente com uma análise de suas conclusões.

2.1. Leis de Programação

Refatoração [6] é o processo que consiste na reestruturação do código do sistema a fim de prover uma organização interna mais adequada, seja para futuras adições de novas funcionalidades ou apenas para aprimorar a qualidade do código existente, sem a alteração do comportamento atual externo do sistema.

A refatoração é uma atividade comum da rotina de um desenvolvedor. Contudo, apesar de algumas ferramentas de desenvolvimento oferecerem um certo suporte a esse processo, o que atualmente está comercialmente implementado nas principais IDEs (Ambiente de Desenvolvimento Integrado) é bem limitado. Frequentemente, da porção de possibilidades de refatoração existente, apenas a de renomear é na prática a mais utilizada de forma automática. O que leva a essa não aplicação das funcionalidades de refatoração das IDE's é, entre outros, o fato das refatorações estarem estritamente ligadas à estrutura delas. Não há atualmente definido um conjunto formal de operações básicas, que representam transformações mais simples e atômicas. Transformações essas que combinadas poderiam vir a compor transformações mais complexas do porte de uma refatoração.

Uma alternativa para a formalização do processo de refatoração é por meio do uso de leis de programação. Cada mudança no código de um programa pode ser realizada com a aplicação de uma lei. Leis de programação descrevem o programa antes e depois da transformação. Para aplicação delas, uma série de condições devem ser satisfeitas para assegurar a aplicação em um cenário adequado, a fim de garantir a integridade do sistema posterior a sua aplicação [1].

Há leis de programação considerando aspectos de vários paradigmas de linguagens: funcional, imperativa, orientado a objetos [2] [1] [3]. Assim como há também trabalhos que descrevem regras de refatoração para linguagem funcional como Haskell [7] e Erlang [8]. Partes de diversos desses trabalhos são diretamente relevantes para definição de leis e regras para Dafny.

2.2. Dafny

Comprovar a corretude de um código de acordo com a especificação e outros aspectos eventualmente apenas acontece como uma etapa do processo de desenvolvimento, e não como algo intrínseco da própria codificação. Pensando em tornar isso mais natural, confiável e eficiente, uma metodologia foi desenvolvida, o projeto por contrato (*Design by Contract* - DbC). A principal ideia por trás de DbC é que uma classe e seus clientes têm um "contrato" um com o outro. O cliente deve garantir certas condições antes de chamar um procedimento definido pela classe, e em troca a classe garante certas propriedades que irá concretizar após a chamada [9]. Na prática trata-se de uma abordagem de desenvolvimento de software que utiliza métodos formais, por meio da definição de pré-condições, pós-condições e invariantes, a fim de facilitar a verificação e garantir a corretude dos sistemas em relação à especificação.

Entretanto, boa parte das linguagens mais utilizadas atualmente não contam com suporte a DbC nativamente, um exemplo disso é Java [9] e C# [10], que contam com bibliotecas, pré-processadores e outras ferramentas externas para suporte de DbC. Pensando em facilitar o desenvolvimento, ao unificar ambiente e unir o código de verificação (código do DbC) com código do programa, algumas linguagens foram desenvolvidas com suporte a DbC de forma nativa, como parte integrante da sintaxe da linguagem, sem uso de bibliotecas para este fim.

Uma das linguagens com DbC nativo é Dafny [4] que foi projetada para tornar mais fácil a escrita de código correto, ou seja, sem erros de execução e de acordo com o determinado pela especificação. Para conseguir isso, Dafny conta com anotações de alto nível para verificar e provar a corretude do código. A verificação de um programa em Dafny é executada automaticamente em segundo plano no ambiente de desenvolvimento integrado para Visual Studio, e funciona com um clique de um botão na versão Web, juntamente com o compilador da linguagem [5]. Dafny tem sido usada para especificar e verificar alguns algoritmos difíceis e geralmente apresenta tempos de verificação baixos.

O comportamento esperado de um fragmento de código pode ser dado de forma abstrata usando expressões de alto nível, algo fácil de escrever e menos predisposto a erros. Por meio da verificação dessas expressões, um verificador prova que o código corresponde às anotações. Por outro lado, anotações ajudam não apenas na verificação do código, auxiliam também na compreensão do mesmo. O desafio passa então a ser escrever anotações sem erros.

Dafny também prova que não há erros de tempo de execução, tais como o índice fora dos limites, referência a ponteiro nulo, divisão por zero. Dafny também

comprova o termino da execução de um código, exceto em loops em que deseja-se uma execução ininterrupta até que a execução seja suspensa manualmente.

Em termos gerais Dafny é uma linguagem imperativa, sequencial, que suporta tipos indutivos e alocação dinâmica, e apresenta construções de especificação [11]. As especificações incluem pré-condições e pós-condições, construções de frame, e métricas de término da execução. O estilo de especificação é flexível, baseado em *dynamic frames*. Além de tipos de classe (*class*) a linguagem suporta conjuntos (*set*), sequências (*seq*) e tipos de dados algébricos (*datatypes*) [11].

Para auxiliar nas especificações, Dafny inclui construções *ghost*. A linguagem faz distinção entre dois conjuntos de construções [5]: as *ghost*, que representam as construções utilizadas em códigos de especificação; e o conjunto de construções físicas, ou compiladas, que formam a implementação propriamente da classe. O programa atualiza os dois conjuntos, mas ao ser compilado apenas as construções físicas constam no código executável. Dafny apresenta construções *ghost*, não apenas para variáveis, mas também para outras declarações como métodos e funções.

No que se segue, apresentamos uma introdução rápida das características de Dafny focando nos conceitos utilizados nesse trabalho, com base no guia apresentado por Leino [4] [12].

2.2.1. Variáveis

A declaração de um atributo *x* do tipo *T* é feita da seguinte forma:

```
var x: T;
```

A especificação do tipo de uma variável não é obrigatória em variáveis locais, isso, caso necessário, pode ser implicitamente inferido, contudo em variáveis no contexto de uma classe ou módulo deve ser explicitamente declarado. Para definir uma variável no contexto *ghost* basta preceder a declaração com a palavra *ghost*. O tipo de uma variável pode ser *bool* para booleanos, *int* para inteiros, ou *nat*, que representa o subintervalo de inteiros positivos de *int*. Outros tipos são as classes (*class*), e tipos de dados indutivos (*datatypes*) definidos pelo usuário, *set*<*T*> para um conjunto finito (imutável) de valores de *T* (onde *T* é qualquer tipo), e *seq* <*T*> para uma sequência (imutável) de valores de *T*. Além disso, existem tipos de matriz (que são como uma classe predefinida) de uma ou mais dimensões. O tipo *object* é um supertipo de todos os tipos de classe, isto é, um objeto denota qualquer referência, incluindo nulo.

2.2.2. Método

A declaração de um método tem o formato:

```
method M(a: A, b: B, c: C) returns (x: X, y: Y, z: Y)
   requires Pre;
   modifies Frame;
   ensures Post;
   decreases Rank;
{
 corpo
}
```

onde `a`, `b`, `c` são parâmetros de entrada do método `M` e `x`, `y`, `z` são parâmetros de saída.

Seguindo os parâmetros de entrada e saída são as declarações de especificação. A expressão booleana `Pre` denota a condição de execução do método, enquanto que `Post` denota a pós-condição do método, ou seja, condição a ser obedecida posterior a execução do mesmo, `Frame` denota o conjunto de objetos que podem ser modificados pelo método, `Rank` é uma função variante do método que deve reduzir a cada chamada recursiva ao método, isso garante que a execução do mesmo não decorra infinitamente. Já o corpo é a declaração que implementa o método. `Frame` pode ser uma lista de expressões, cada um das quais é um conjunto de objetos ou um único objeto. O *frame* do método é a união desses conjuntos, além do conjunto de objetos alocados pelo corpo do método. Se omitido, por padrão, pré e pós-condições são verdadeiros e o *frame* é o conjunto vazio.

A função variante é uma lista de expressões, que denota uma tupla das expressões dadas, ou seja, a tupla que deve ser modificada a cada execução. Se omitido, Dafny vai prever uma função variante para o método, ou seja, um tupla com a lista dos parâmetros de entrada do método.

Caso queira definir um método no contexto ghost, basta preceder a declaração com a palavra-chave *ghost*. Por padrão, um método está relacionado com um objeto do tipo da classe em que está contido, caso queira remover esse vínculo basta preceder a declaração com a palavra-chave *static*. Com isso não é preciso a instanciação de um objeto da classe para chamar o método.

Para declarar um construtor de uma classe basta substituir a palavra-chave *method* por *constructor*. Em uma classe com construtor, o mesmo deve ser declarado explicitamente na instanciação do objeto.

2.2.3. Função

Uma função tem o seguinte formato:

```
function F(a: A, b: B, c: C): T
  requires Pre;
  reads Frame;
  ensures Post;
  decreases Rank;
{
  corpo
}
```

De forma semelhante a declaração de um método, *a*, *b*, *c* são os parâmetros da função, *T* é o tipo de retorno da função. `Pre` é uma expressão booleana que representa a condição de execução da função, enquanto que `Post` denota a pós-condição da função, ou seja, condição a ser obedecida posterior a execução da mesma, `Frame` o conjunto de objetos cujos campos o corpo da função pode depender, `Rank` é função variante da função que deve ter seu valor reduzido a cada chamada recursiva à função, isso garante que a execução não decorra infinitamente., e o corpo é uma expressão que define a função.

A pós-condição não é necessária, contudo pode ser utilizada pera definir propriedades dos resultados retornados por tal função. Por exemplo, a função *Factorial* a seguir, pela cláusula de pós-condição diz que o resultado da função é sempre positivo. Dafny de forma indutiva verifica a partir do corpo da função. Para se referir ao resultado da função na pós-condição, usa-se a chamada à própria função.

```
function Factorial(n: int): int
  requires 0 <= n;
  ensures 1 <= Factorial(n);
{
  if n == 0 then 1 else Factorial(n-1) * n
}
```

Por padrão, uma função é *ghost*, e não pode ser chamado por código que não esteja no contexto *ghost*, ou seja, só é utilizada pelo verificador de Dafny. Para torná-la acessível pelo contexto físico, basta substituir a palavra-chave *function* por *method function*. Um outro detalhe, é que assim como métodos, por padrão, uma função está relacionada com um objeto do tipo da classe em que está declarada, logo depende da instanciação de um objeto. Para remover essa dependência, e tornar a função acessível sem a necessidade de um objeto, é preciso preceder a declaração da função com a palavra-chave *static*.

2.2.4. Classe

Uma classe é definida como a seguir, onde os membros da classe são atributos, métodos e funções, definidos anteriormente.

```
class C
{
  // declaração dos membros da classe
}
```

2.2.5. Tipos de dados indutivos

Um tipo de dados indutivo (*datatype*) é um tipo cujos valores são criados usando um conjunto fixo de construtores. Esses construtores são declarados separados por barras verticais (|). Um *datatype* de nome *Tree* com construtores *Empty* (sem parâmetros, logo não precisa de parênteses na declaração) e *Node* é declarado da seguinte forma:

```
datatype Tree = Empty | Node(Tree, int, Tree)
var t0 := Empty;
var t1 := Node(t0, 5, t0);
```

Para cada construtor de nome *Ct*, o tipo de dados implicitamente declara um membro booleano *Ct*?, que retorna verdadeiro para aqueles valores que foram construídos usando o construtor *Ct*. Por exemplo, a expressão *t1.Node*? é avaliada como verdadeira e *t0.Node*? é avaliada como falsa, já que *t1* foi criada a partir do construtor *Node*, mas *t0* não. Dois *datatype* são iguais se eles foram criados usando o mesmo construtor e valores de parâmetros. Portanto, para os construtores sem parâmetros, como *Empty*, t.Empty? e t == *Empty* retornam o mesmo valor.

Um construtor pode ser declarado com os nomes dos parâmetros de forma explicita, com isso é possível fazer comparação entre parâmetros específicos. Por exemplo, se *Tree* fosse declarada da forma a seguir seria possível referenciar a parâmetros como *t1.data* ou *t1.left*:

```
datatype Tree = Empty | Node(left: Tree, data: int,
right: Tree)
```

2.2.6. Tipos Genéricos

Dafny suporta tipos genéricos, ou seja, qualquer classe, tipo de dados indutivo, método e função podem ter parâmetros de tipo. Estes são declarados entre <> após o nome do que está sendo declarado. Por exemplo:

```
class Multiset<T> { /*…*/ }
datatype Tree<T> = Empty | Node(Tree<T>, T, Tree<T>)
```

```
method Find<T>(key: T, collection: Tree<T>) { /*…*/ }
function IfThenElse<T>(b: bool, x: T, y: T): T {/*…*/}
```

2.2.7. Instruções

A instrução de atribuição em uma variável tem o seguinte formato:

```
Lvalues := ExprList;
```

O lado direito *ExprList* é atribuído ao lado esquerdo correspondente *L-values*. Estas atribuições são realizadas em paralelo, de modo que o lado esquerdo deve denotar *L-values* distintos. Vale notar que atribuições são feitar utilizando ":=". Cada lado direito pode ser uma expressão ou uma criação do objeto de acordo com uma das seguintes formas:

```
new T
new T.Init(ExprList)
new T[SizeExpr]
new T[SizeExpr0, SizeExpr1]
```

A primeira forma aloca um objeto do tipo *T*. A seguinte aloca um objeto por meio da chamada do método construtor *Init*. As outras duas são exemplos de alocações de matrizes, em particular um *array* uni e bidimensional, respectivamente, de valores de *T*.

Todo o lado direito de uma atribuição também pode ser uma chamada de método, neste caso, o lado esquerdo da expressão são todos os parâmetros de saída do método (omitindo ":=" se não há parâmetros de saída).

A declaração *assert*, que pode ser declarada em qualquer parte do corpo de um procedimento, certifica que a expressão dada é avaliada como verdadeira pelo verificador em determinado ponto de código.

```
assert BoolExpr;
```

O comando *if* é o usual. Que pode apresentar alternativas de condição, *else if,* e o bloco a ser executado caso nenhuma condição seja satisfeita, o *else*.

```
if BoolExpr0 {
  Stmts0
} else if BoolExpr1 {
  Stmts1
} else {
  Stmts2
}
```

O comando *while* é o loop habitual, onde a declaração *invariant* define uma invariante de laço, ou seja, uma expressão que deve ser avaliada como verdadeira ao entrar no loop e após cada execução do mesmo. A cláusula *modifies* restringe o *frame* de modificação do loop, ou seja, o escopo de objetos que podem ser modificados, e a cláusula *decreases* introduz uma função variante para o loop, da mesma forma como em métodos e funções, para garantir que o loop não executa infinitamente. Por padrão, o invariante de laço é verdadeiro, o *frame* de modificação é o mesmo que o do contexto em que está inserido (normalmente a cláusula *modifies* do método), e a função variante é sugerida a partir da expressão de guarda do loop.

```
while BoolExpr
      invariant Inv;
      modifies Frame;
      decreases Rank;
{
  Stmts
}
```

A declaração *match* avalia a fonte *Expr*, uma expressão cujo tipo é um tipo de dados indutivo (*datatype*), e em seguida, executa o caso correspondente ao construtor que foi usado para criar *Expr*, e aos parâmetros do construtor são dados nomes, que podem ser utilizados nas declarações do corpo do caso.

```
match Expr {
  case Empty => Stmts0
  case Node(l, d, r) => Stmts1
}
```

A instrução *break* pode ser usada para sair de loops, e a instrução *return* pode ser usada para retornar de um método, com ou sem retornos.

```
break;
return;
```

2.3. Trabalhos Relacionados

Muitos trabalhos foram desenvolvidos no campo da refatoração para os diversos paradigmas de linguagem de programação e considerando aspectos das mais diversas linguagens. Alguns relacionados com o desenvolvimento de ferramentas de suporte ao processo, ou da formalização das mudanças por meio de regras ou leis. Alguns trabalhos propõe as regras para então aplicá-las por meio de ferramentas de automação ou verificação do processo.

Borba et al. [2] apresentam leis algébricas para uma linguagem semelhante a um subconjunto de Java com o escopo limitado adequado para o foco do estudo,

ROOL (*Refinement Object-Oriented Language*) [13]. Essas leis foram propostas quando não havia trabalhos bem consolidados sobre as leis para programação orientada a objetos. São apresentadas provas de que as leis são sólidas e suficientes para reduzir um programa para uma forma normal, semelhante a um programa imperativo, porém com testes de tipo e *casts*. Uma das principais aplicações dessas leis é no auxílio na preservação de comportamento de sistemas mais elaborados após transformações, seja para otimização ou reestruturação. Em especial, é mostrado como elas podem ser usadas para derivar refatorações comprovadamente corretas. Embora as leis sejam apresentadas para uma linguagem específica, podem ser empregadas para linguagens com aspectos semelhantes.

Seguindo esse trabalho, Cornélio [13] provou as leis de comandos de ROOL e recursos da orientação-objeto. Formalizou algumas refatorações já conhecidas na literatura e provou regras de refatoração pela aplicação de regras de programação. Uma preocupação extra exposta é quanto a utilização de padrões de design na estratégia de desenvolvimento.

Freitas [3] apresenta um conjunto de leis de programação para linguagens orientadas a objetos como Java, combinada com o Java Modelling Language (JML) [9]. O conjunto de leis lida com características da orientação a objetos, tendo em conta as suas especificações. Até então, o conjunto de leis de programação para linguagens orientada a objeto era projetado para a transformação do programa, sem relação com linguagens projetados para especificações com DbC. Também foi averiguado o impacto da redução de programas em Java-JML para uma forma normal.

Li [7] estudou as possibilidades de refatoração em Haskell para o estudo de refatoração em linguagens funcionais. Refatorações foram especificadas e verificadas. O intuito do estudo era o desenvolvimento de uma ferramenta, HaRe (*Haskell Refactorer*). Além das refatorações suportadas pela ferramenta, HaRe também é composto de uma API que permite aos usuários implementarem suas próprias refatorações de uma forma compacta e de alto nível. Além desse, Li et al. [8] descreveram refatorações para programas escritos em Erlang, e trabalharam na construção de duas ferramentas para dar suporte à refatoração de sistemas escritos em Erlang, cada uma com uma abordagem distinta quanto à representação interna do sistema, de como a informação sintática e semântica é armazenada e manipulada. Li et al. [14] também compararam os dois estudos e o desenvolvimento de ferramentas para suporte a refatoração. Essa comparação é feita levando em consideração as refatorações aplicáveis para cada linguagem, e as análises de programa exigidas pelas refatorações comuns, e pelas ferramentas de suporte a refatoração de programas em Haskell e Erlang.

3. Dafny e as Leis

Leis de programação descrevem programas antes e depois da transformação, então cada mudança no código de um programa pode ser realizada com a aplicação de uma lei especifica. Uma lei deve ser aplicada em um cenário adequando, para isso uma série de condições devem ser satisfeitas para assegurar a integridade do sistema posterior a sua aplicação [1].

Há leis de programação considerando aspectos de vários paradigmas de linguagens: funcional, imperativa e orientado a objetos [2] [1] [3]. Assim como há também trabalhos que descrevem regras de refatoração para linguagens funcionais como Haskell [7] e Erlang [8]. Diversos desses trabalhos são diretamente relevantes para definição de leis e regras para Dafny. A fim de definir regras e leis pensando em Dafny, nesse capitulo serão apresentadas as adaptações do que foi proposto para outras linguagens, aproveitando o que se adequa para a linguagem Dafny e fazendo adaptações quando necessário.

A linguagem Java principalmente quando combinada com a linguagem de especificação Java Modelling Language (JML) apresenta vários aspectos em comum com Dafny. Aproveitando isso, primeiramente adaptaremos as leis propostas para Java-JML fazendo os ajustes necessários. Em seguida serão apresentadas as adaptações das regras de refatoração existentes para linguagens funcionais, baseado nos trabalhos feitos para Haskell e Erlang.

3.1. Adaptação das leis de Programação imperativas e de contratos de software (JML)

Aqui iremos descrever o processo de adaptação para Dafny [4] das leis de programação definidas por Freitas [3] para Java-JML. Alguns dos aspectos distintos entre as linguagens que mais impactaram na adaptação das leis foi a não existência de modificadores de acesso em Dafny, bem como o conceito de herança.

Primeiramente definiremos a convenção para representação dessas leis e depois as leis propriamente. Posteriormente um sumario das leis será apresentado.

3.1.1. Convenções

As leis são escritas como equações onde cada lado da equação corresponde a um programa bem formado. Essas leis indicam quais modificações podem ser feitas em um programa de forma que não haja modificação no seu comportamento. Do lado esquerdo da equação o programa antes da modificação e do direito, posterior à modificação, juntamente com as condições que devem ser obedecidas. A aplicação

dessas leis deve seguir fielmente o que é descrito na lei, a fim de preservar o comportamento assim como conservar a boa formação [3].

Usualmente um programa em Dafny é composto de vários módulos, mas para os fins dos nossos estudos iremos simplificar e considerar a existência de um módulo único, que será representado por *mod*. Um módulo pode ser formado por classes, datatypes, tipos, métodos, funções, entre outros, representados respectivamente por *clas*, *dty*, *typ*, *met*, *func*.

Dafny não suporta sistemas concorrentes, logo só apresenta programas sequenciais. Um programa mod_1 será equivalente a um mod_2 ($mod_1 = mod_2$) quando apresentam o mesmo comportamento, de acordo com a especificação, quando executados de forma não concorrente. Em algumas leis será utilizada a representação $mod_1 \sqsubseteq mod_2$, que diz que mod_1 é refinado por mod_2.

As classes são formadas por atributos, funções, métodos e predicados, *atbs*, *funcs*, *mets* e *preds*, respectivamente.

Para representar a declaração de um método, nós usaremos a notação *m* (*prs*) (*rts*) { *mbody* }, onde *m* é nome do método, *prs* é a lista de parâmetros, *rts* é a lista de itens retornados e *mbody* é o corpo do método. Notação semelhante é utilizada para a representação de funções método *f* (*prs*) : *rt* { *fbody* }, onde *f* é o nome da função método; *prs,* a lista de parâmetros; *rt,* o tipo de retorno da função método; e *fbody,* o corpo da função método. O mesmo padrão é utilizado para representar predicados *p (prs) { pbody }*, onde *p* é o nome do predicado, *prs* a lista de parâmetros e *pbody* o corpo do predicado.

Usamos *B.a* quando queremos referir o acesso de um atributo chamado *a* pertencente ao tipo *B*. A notação *B.m* refere-se a uma chamada para um método chamado *m* pertencente ao tipo *B,* assim como *B.f* refere-se a chamada de uma função *f* de *B* e *B.p* a chamada de um predicado *p* de *B*. O símbolo *T* é usado para representar um tipo de atributo.

Predicados são descritos pela letra grega Ψ. Axiomas são descritos pela letra grega ω.

A notação *@spec_cases* é usada para representar um conjunto de especificações de um procedimento, que pode ser uma especificação, várias ou nenhuma. Na descrição das leis são representadas apenas expressões utilizadas nas condições das leis, isso não significa, por exemplo, que se não está descrito elas não devem existir, é apenas que não são relevantes para determinada lei.

Para auxiliar na descrição das leis, algumas funções são utilizadas. As funções *fspec*, *fpre* e *fpos*, quando aplicadas a um método, retornam sua especificação, pré-condição e pós-condição, respectivamente. A convenção *C.m[prs]* refere-se a um método *m*, com os parâmetros formais *prs* de uma classe *C*. Além disso, *fpre(C.m[prs])* retorna as precondições de um método *m* da classe *C*. Uma outra função é *vardecs(prs, e)* que introduz uma lista de variáveis que tem os mesmos nomes e tipos dos parâmetros *prs* e são inicializados com os valores dos argumentos *e*, usados na chamada de um procedimento.

Apesar de apresentar a especificação de *Frame*, para os fins das leis aqui descritas isso não será considerado, principalmente pelas particularidades que teriam que ser levadas em conta para isso. Logo, iremos assumir que todas as declarações podem modificar todas as variáveis da classe ou módulo em que estão inseridas.

Em Dafny, os atributos, métodos e funções declaradas fora de uma classe entram em uma classe implícita chamada *_default*, dando a aparência do programa ter variáveis, procedimentos e funções globais [15].

Nós escrevemos '→' para indicar as condições que devem ser satisfeitas para aplicar uma lei da esquerda para a direita. Da mesma forma, usamos '←' para indicar o que tem que ser satisfeitas para permitir a aplicação de uma lei no sentido oposto. Condições que devem ser satisfeitas em ambas as direções são indicados por '↔'.

Aqui não é fornecida uma prova formal das leis adaptadas. A fim de validar cada uma delas e comprovar sua solidez, foram escritos códigos de exemplo para simular situações em que as leis se aplicariam. Utilizamos então a ferramenta de verificação de código em Dafny, rise4fun para Dafny, antes e depois da aplicação das leis em cada código, para confirmar que o código estava e permaneceu de acordo com a especificação.

As leis aplicadas estão agrupadas em três categorias de acordo com a funcionalidade da linguagem: Classes, Procedimentos, Especificações.

3.1.2. Classes

Classes que não são mais utilizadas no programa podem ser eliminadas, como na Law 38 [3] adaptada para **Lei 1** de Dafny. Ao aplicar a lei na direção inversa ela permite a “Criação de Classe”.

Lei 1 - Eliminação de Classe

mod clas = *mod*

desde que:

(→) A classe declarada por *clas* não é referenciada em *mod*.
(←) Não há classe em *mod* com o mesmo nome de *clas*.

3.1.3. Procedimentos

Para eliminação de um procedimento, seja ele um método, um função método ou um predicado é preciso realizar inicialmente a remoção de qualquer chamada a esse procedimento (**Lei 2**, **Lei 3**, **Lei 5**, **Lei 7**) e então realizar a eliminação do procedimento propriamente (**Lei 4**, **Lei 6**, **Lei 8**). A remoção da chamada a um procedimento é uma das regras de transformação utilizada pelo *fold*/*unfold system* [7] para derivação de sistemas.

Ao aplicar a **Lei 2**, uma adaptação da *Law 83* [3], é possível eliminar chamadas à um método *void*, ou seja, um método que não apresenta parâmetros de retorno. Contudo, para que essa lei seja aplicada todos os atributos referenciados pelo método devem ser acessíveis e não deve haver chamada recursiva nesse método. De forma semelhante a **Lei 3** auxilia na eliminação de chamada a métodos com retorno, porém diferentemente do que é feito na *Law 84* [3], em Dafny há a possibilidade de múltiplos parâmetros de retorno, logo essa particularidade foi considerada na lei. Um outro detalhe é que em Dafny não há modificadores de acesso, logo as condições de aplicação referentes a isso, da lei original, não foram consideradas para a **Lei 2** e **Lei 3**, bem como a referente à redefinição de método, já que a linguagem não suporta vários procedimentos com o mesmo nome, ou seja, não suporta redefinição de procedimento.

No caso de Dafny, diferentemente do que é feito para Java-JML, que há duas leis referentes à eliminação de métodos (*Law 75* e *Law 76*), uma com declaração de especificação e outra sem. Tratando-se de Dafny, como não há a noção de herança, não há preocupações distintas referentes à declaração ou não da especificação [3]. Então, para a eliminação do método, posterior a eliminação de todas as suas chamadas, é aplicada a **Lei 4** que checa se não há mais chamadas ao método e então efetua a eliminação do código do mesmo. Porém, para aplicação inversa da lei uma peculiaridade de Dafny deve ser considerada: não pode existir nenhum outro membro no mesmo escopo com o mesmo nome, mesmo que de tipo diferente, como por exemplo uma variável e um procedimento. Isso em decorrência das regras de resolução de nomes utilizado pela linguagem.

A **Lei 5** e a **Lei 6** realizam, respectivamente, a operação de eliminação de chamada à função método e eliminação da função, de forma semelhante do que é feito para métodos. E a **Lei 7** e a **Lei 8**, de eliminação de chamada à predicado método e eliminação do predicado método, respectivamente. É importante citar que as definições de funções e predicados não são expostas fora do módulo onde são definidas, logo, essas leis só fazem sentido segundo a restrição de existência de um módulo único como foi definido.

Lei 2 - Eliminação de chamada de método void

Considere a seguinte declaração de classe com um método void

class C {
atbs
funcs
method *m* (*prs*) *@spec_cases* { *mbody* }
mets
preds
}

está incluída em *mod* e que *mod, A ▷ le : C,* onde *le* é do tipo *C* na classe *A*. Onde

mod, A ▷ le.m(e) = **assert** *le* != **null**;
vardecs(*prs*, *e*);
assert *fpre*(*C*[*m*(*prs*)])[*le*/**this**];
mbody[*le*/**this**]
assert *fpos*(*C*[*m*(*prs*)])[*le*/**this**];

desde que:

(→) (1) *mbody* não contém chamadas recursivas; (2) *prs* não ocorre em *e*; (3) *mbody* não tem cláusula de retorno; (4) todos os atributos acessados, assim como chamadas a procedimentos que ocorrem em *mbody*, são do tipo **this**.*a* e **this**.*p*(*e*), respectivamente.

Lei 3 - Eliminação de chamada de método não void

Considere a seguinte declaração de classe com um método void

class C {
atbs
funcs
method *m* (*prs*) (rts) *@spec_cases* { *mbody* }
mets
preds
}

está incluída em *mod* e que *mod, A ▷ le : C,* onde *le* é do tipo *C* na classe *A*. Assumindo que *res* representa a lista de variáveis de retorno do método. Onde

mod, A ▷ ***var*** *rts* := *le.m(e)* =

var *rts*;
assert *le* != **null**;
vardecs(*prs*, *e*);
assert *fpre*(*C*[*m*(*prs*)])[*le*/**this**];
mbody[*le*/**this**] [*res := result*/**return** *result*]
assert *fpos*(*C*[*m*(*prs*)])[*le*/**this**];

desde que:

(→) (1) *mbody* não contém chamadas recursivas; (2) *prs* não ocorre em e; (3) *mbody* não tem múltiplos pontos de retorno; (4) todos os atributos acessados assim como chamadas a procedimentos que ocorrem em *mbody* são do tipo **this**.*a* e **this**.*p*(*e*), respectivamente.

Lei 4 - Eliminação de método

class C {
atbs
funcs
method *m* (*prs*)
(*rts*) { *mbody* }
mets
preds
}

$=_{mod}$

class C {
atbs
funcs
mets
preds
}

desde que:

(→) *V.m(e)* não ocorre em *mod*, *funcs*, *mets* ou *preds*, onde *V* é uma variável do tipo *C*.
(←) Não há no escopo de *funcs*, *mets* ou *preds* membro com o identificador *m*.

Lei 5 - Eliminação de chamada de função método

Considere a seguinte declaração de classe

class C {
atbs
function method *f* (*prs*) : *rt @spec_cases* { *fbody* }
funcs
mets
preds }

está incluída em *mod* e que *mod, A ▷ le : C,* onde *le* é do tipo *C* na classe *A*. Onde

mod, A ▷ ***var*** *a: rt* := *le.f(e)* =

var *a: rt*;
assert *le* != **null**;
vardecs(*prs*, *e*);
assert *fpre*(*C*[*m*(*prs*)])[*le*/**this**];
rt := *fbody*[*le*/**this**];
assert *fpos*(*C*[*m*(*prs*)])[*le*/**this**];

desde que:

(→) (1) *fbody* não contém chamadas recursivas; (2) *prs* não ocorre em e; (3) todos os atributos acessados assim como chamadas a procedimentos que ocorrem em *fbody* são do tipo **this**.*a* e **this**.*p*(*e*), respectivamente.

Lei 6 - Eliminação de função método

class C {
atbs
function
method *f* (*prs*) :
r*t*
{ *fbody* }
funcs
mets
preds
}

$=_{mod}$

class C {
atbs
funcs
mets
preds
}

desde que:

(→) *V.f(e)* não ocorre em *mod*, *funcs*, *mets* ou *preds*, , onde *V* é uma variável do tipo *C*.
(←) Não há no escopo de *funcs*, *mets* ou *preds* membro com o nome *f*.

Lei 7 - Eliminação de chamada de predicado método

Considere a seguinte declaração de classe

class C {
atbs
funcs
mets

predicate method *p (prs) @spec_cases { pbody }*
preds
}

está incluída em *mod* e que *mod, A ▷ le : C,* onde *le* é do tipo *C* na classe *A*. Onde

mod, A ▷ ***var*** *a:* ***bool*** := *le.p(e)* =

var *a: bool*;
assert *le* != **null**;
vardecs(*prs*, *e*);
assert *fpre*(*C*[*m*(*prs*)])[*le*/**this**];
a = pbody[*le*/**this**];
assert *fpos*(*C*[*m*(*prs*)])[*le*/**this**];

desde que:

(→) (1) *pbody* não contém chamadas recursivas; (2) *prs* não ocorre em e; (3) todos os atributos acessados assim como chamadas a procedimentos que ocorrem em *pbody* são do tipo **this**.*a* e **this**.*p*(*e*), respectivamente.

Lei 8 - Eliminação de método predicado

class C {
atbs
funcs
mets
predicate *p* (*prs*) { *pbody* }
preds
}

$=_{mod}$

class C {
atbs
funcs
mets
preds
}

desde que:

(→) *V.p(e)* não ocorre em *mod*, *funcs*, *mets* ou *preds*, onde *V* é uma variável do tipo *C*.
(←) Não há no escopo de *funcs*, *mets* ou *preds* membro com o identificador *p.*

A **Lei 9** realiza e remoção de múltiplos pontos de retorno em um método, essa pode ser uma etapa realizada antes da aplicação da **Lei 3**. A única condição para essa remoção é que esses pontos de retorno estejam em lugares guardados por condicionais de exclusão mútua, ou seja, a remoção da cláusula de retorno não deve interferir no fluxo do código.

Lei 9 - Eliminação de múltiplos pontos de retorno

class C {
atbs
funcs
method *m* (*prs*) (rts)
{ *mbody* }
mets
preds
}
	$=_{mod}$	**class** C {
		atbs
		funcs
		method *m* (*prs*) (rts) {
		mbody [*res := e*/**return** *e*]
		return *res;*
		}
		mets
		preds
		}

onde:
res representa a lista de variáveis de retorno do método.

desde que:
(→) As cláusulas de retorno estejam presente apenas dentro de condicionais mutuamente exclusivas.

3.1.4. Especificações

Nessa subseção serão apresentadas as leis que definem mudanças relacionadas às especificações de código: pré-condições, pós-condições, invariantes e asserções.

A **Lei 10** enfraquece uma pré-condição. Um predicado Ψ_1 é dito mais fraco em relação a um outro predicado Ψ_1', se $\Psi_1' \Rightarrow \Psi_1$, ou seja, Ψ_1' implica em Ψ_1. Já a **Lei 11** fortalece a pós-condição Ψ_2 de um método *m*, ou seja, $\Psi_2' \Rightarrow \Psi_2$.

A **Lei 12** e **Lei 13** representam a formalização entre as equivalências de açucares sintáticos da linguagem, presente em Java-JML e em Dafny. Essas leis (**Lei 10**, **Lei 11**, **Lei 12**, **Lei 13**) foram descritas considerando especificações de um método, contudo podem ser aplicadas para outros procedimentos como funções e predicados.

Lei 10 - Enfraquecer pré-condição

class C {

atbs
funcs

method *m (prs) (rts)*
requires Ψ_1;
modifies ω;
ensures Ψ_2;
{ mbody }

mets
preds
}

desde que:

$\Psi_1 \Rightarrow \Psi_1'$

$=_{mod}$

class C {
atbs
funcs

method *m (prs) (rts)*
requires Ψ_1';
modifies ω;
ensures Ψ_2;
{ mbody }

mets
preds
}

Lei 11 - Fortalecer pós-condição

class C {

atbs
funcs

method *m (prs) (rts)*
requires Ψ_1;
modifies ω;
ensures Ψ_2;
{ mbody }

mets
preds
}

desde que:

$\Psi_2' \Rightarrow \Psi_2$

$=_{mod,}$

class C {
atbs
funcs

method *m (prs) (rts)*
requires Ψ_1;
modifies ω;
ensures Ψ_2';
{ mbody }

mets
preds
}

Lei 12 - Agregar pré-condições

class C {

atbs
funcs

method *m (prs) (rts)*
requires Ψ_{11};
requires Ψ_{12};
…
requires Ψ_{1n};
modifies ω;
ensures Ψ_{2};
{ mbody }

mets
preds
}

$=_{mod,}$

class C {
atbs
funcs

method *m (prs) (rts)*
requires Ψ_{11} && Ψ_{12} && … && Ψ_{1n};
modifies ω;
ensures Ψ_{2};
{ mbody }

mets
preds
}

Lei 13 - Agregar pós-condições

class C {

atbs
funcs

method *m (prs) (rts)*
requires Ψ_{11};
modifies ω;
ensures Ψ_{21};
ensures Ψ_{22};
…
ensures Ψ_{2n};
{ mbody }

mets
preds
}

$=_{mod,}$

class C {
atbs
funcs

method *m (prs) (rts)*
requires Ψ_{1};
modifies ω;
ensures Ψ_{21} && Ψ_{22} && ... && Ψ_{2n} ;
{ mbody }

mets
preds
}

Em Java-JML a cláusula *invariant* determina um estado, em relação à atributos, a ser obedecido em todos os estados de um objeto de uma classe. Há também a cláusula *loop invariant,* onde o conceito de invariante aparece para monitorar a execução de loops. É uma propriedade que é preservada em cada execução do loop, ou seja, é satisfeita ao entrar no loop, e após cada execução do mesmo.

A *Law 5* [3] refere-se a *invariant* de JML, contudo Dafny não apresenta o conceito de *invariant*, então considerando as semelhanças com o *loop invariant,* essa lei foi adaptada para a **Lei 14**. Essa lei representa mais um açúcar sintático de Dafny, semelhante a Java-JML. Um invariante escrito em mais de uma cláusula invariante pode ser simplificada em uma clausula única, apenas fazendo conjunção dos predicados utilizando o operador '&&'.

Lei 14 - Agregar invariante de loop

```
class C {                                   class C {
    atbs                                        atbs
    funcs                                       funcs

    method    m(prs)                            method m(prs) (rts)
(rts) @spec_cases  {                            @spec_cases  {
    …                                           …
    while(Ψ1)                       =mod,       while(Ψ1)
    invariant Ψ21;                              invariant  Ψ21  &&  Ψ22
    invariant Ψ22;                              &&… && Ψ2n;
    …                                           {…}
    invariant Ψ2n;                              …}
    {…}
    …}

    mets                                        mets
    preds                                       preds
}                                           }
```

3.1.5. Sumário das Leis

Nessa subseção estão resumidas as leis adaptadas nessa seção. Aqui nos mapeamos cada lei com as leis originais que serviram como base para essas. As leis são apresentadas na **Tabela 1**.

Funcionalidade	Lei em Dafny	Lei Java-JML
Classes	Lei 1 - Eliminação de Classe	*Law 38*
Procedimentos	Lei 2 - Eliminação de chamada de método void	*Law 83*
	Lei 3 - Eliminação de chamada de método não void	*Law 84*
	Lei 4 - Eliminação de método	*Law 75*
	Lei 5 - Eliminação de chamada de função	*Law 84*
	Lei 6 - Eliminação de função	*Law 75*
	Lei 7 - Eliminação de chamada de predicado método	*Law 84*
	Lei 8 - Eliminação de método predicado	*Law 75*
	Lei 9 - Eliminação de múltiplos pontos de retorno	*Law 65*
Especificações	Lei 10 - Enfraquecer pré-condição	*Law 19*
	Lei 11 - Fortalecer pós-condição	*Law 20*
	Lei 12 - Agregar pré-condições	*Law 28*
	Lei 13 - Agregar pós-condições	*Law 29*
	Lei 14 - Agregar invariante de loop	*Law 5*

Tabela 1 Sumário das Leis para Dafny baseadas nas Leis para Java-JML

3.2. Adaptação das regras de refatoração da linguagem funcional

A linguagem Dafny tem vários aspectos semelhantes aos de linguagens funcionais como Haskell. Levando isso em consideração foi tomado como base o desenvolvimento de uma ferramenta de verificação para Haskell, o HaRe [7]. No desenvolvimento da ferramenta de refatoração HaRe, foram analisados especificação e a verificação da validade de refatorações. Apesar de atualmente apenas uma pequena fração dos casos de refatoração serem suportados pela ferramenta, o estudo cataloga um número bem maior refatorações. As refatorações definidas para Haskell foram analisadas a fim de determinar quais delas poderiam ser adaptadas para Dafny.

Diferentemente do trabalho feito para *Java-JML* [3], para *Haskell* [7] não há a descrição formal de todas as regras, boa parte dessas refatorações são apenas descritas e as condições de aplicação apresentadas, apenas em alguns casos é que são tratados em detalhes. Contudo, considerando essas descrições e as condições, e adotando como base o que é feito nas leis para *Java-JML*, essas refatorações serão apresentadas em formato de lei, a fim de sistematizar o processo e esclarecer os detalhes de cada refatoração.

Algumas das refatorações de *Haskell* representam a mesma refatoração já formalizada por alguma lei de *Java-JML*. Essas foram, então, utilizadas para confirmação da corretude da regra ou para aprimorar as condições de aplicação da mesma.

Serão consideradas as mesmas convenções utilizadas nas leis adaptadas de *Java-JML* na seção anterior. Assim como também classificaremos as regras de acordo com a funcionalidade da linguagem, agrupadas em três categorias, incluindo uma categoria para regras gerais.

3.2.1. Gerais

Renomear provavelmente é uma das refatorações mais básicas e úteis. Permite manter a consistência entre nome e representação no sistema [7]. E talvez por esse motivo seja uma das mudanças mais suportadas por ferramentas de refatoração. No HaRe [7], foi uma das primeiras refatorações suportadas e a mais testada.

A ideia geral do processo de renomear é semelhante, independentemente do escopo em que ocorre, seja módulo, classe ou procedimento. Contudo, as condições de aplicação em cada um desses escopos são distintas. Pensando nisso, foi especificada uma regra para cada uma dessas situações. A **Regra 1** trata o caso da renomeação de variável dentro de um procedimento. A regra está escrita considerando um método, entretanto pode ser aplicada com as mesmas condições

para os demais tipos de procedimento. Já a aplicação da **Regra 2** renomeia uma entidade de uma classe, seja ela método ou procedimento. A regra também pode ser aplicada da mesma forma para renomear atributos e procedimentos de um módulo, já que implicitamente essas entidades pertencem a uma classe chamada _default [15]. Já a **Regra 3**, a **Regra 4** e a **Regra 5**, referem-se, respectivamente, a renomeação de classe, datatypes e tipos em um módulo, entidades classificadas como de top-level em Dafny. Essas entidades só podem ser declaradas dentro de um módulo, mas podem ser referenciadas nas demais entidades.

Regra 1 - Renomear variável em procedimento

```
class C {                          =mod,    class C {
    atbs                                        atbs
    funcs                                       funcs
    method m (prs)                              method m(prs)
    {                                           {
        var x: T;                                   var x2: T;
        mbody                                       mbody[x2/x]
    }                                           }
    mets                                        mets
    preds                                       preds
}                                           }
```

Desde que:

(→) (1) Não há no escopo de *m* entidade com o identificador *x2* (2) Caso haja em *C* entidade com o nome *x2*, toda referência em *m* à essa variável deve ser do tipo ***this***.*x2* ou ***this***.*x2(e)*

Regra 2 - Renomear entidade em classe

```
class C {                          =mod,    class C {
    var x: T;                                   var x2: T;
    atbs                                        atbs
    funcs                                       funcs
    mets                                        mets
    preds                                       preds
}                                           }[this.x2/this.x]
```

Desde que:

(→) (1) Não há no escopo de *C* entidade com o nome *x2* (2) Toda referência a *x* em *funcs*, *mets* e *preds* deve ser do tipo ***this***.*x* ou ***this***.*x(e)*

Regra 3 – Renomear classe em módulo

module *M* **{** ***class*** *C* *{ classBody }* *clas* *dtys* *typs* *atbs* *funcs* *mets* *preds* }	=	**module** *M* { *clas* *dtys* *typs* *atbs* *funcs* *mets* *preds* }[C2/C]

Desde que:

(→) Não existe em *M* entidade *top-level* com o identificador *C2*

Regra 4 – Renomear datatype em módulo

module *M* **{** *clas* **datatype** D = constructors *dtys* *typs* *atbs* *funcs* *mets* *preds* }	=	**module** *M* { *clas* *dtys* *typs* *atbs* *funcs* *mets* *preds* }[D2/D]

Desde que:

(→) Não existe em *M* entidade *top-level* com o identificador *D2*

Regra 5 – Renomear type em módulo

module *M* **{** *clas* *dtys* **type** *T* *typs* *atbs* *funcs* *mets* *preds* }	=	**module** *M* { *clas* *dtys* *typs* *atbs* *funcs* *mets* *preds* }[T2/T]

Desde que:

(→) Não existe em *M* entidade *top-level* com o identificador *T2*

3.2.2. Procedimentos

A remoção de parâmetros de um procedimento serve para remover do código parâmetros não mais usados no corpo do procedimento. Pensando nisso a **Regra 6** foi definida.

Regra 6 - Remover parâmetro

module *M* **{**
moduleBody,
class *C* **{**
atbs
funcs
method *m* (*prs, x:* *T*)
@spec_cases
{ *mbody* }
mets
preds
}
}
	=	**module M {**
		moduleBody[le.m(e)/le.m(e, y)],
		class *C* {
		atbs
		funcs
		method *m*(*prs*)
		@spec_cases
		{ *mbody* }
		mets
		preds
		}[*m*(*e*)/*m*(*e*, *y*)]
		}

Onde:

le é uma entidade da classe C

y é o valor passado como argumento para o parâmetro *x* na chamada do método *m*

e é a lista dos argumentos da chamada do método sem x.
Desde que:
(→) *x* não ocorre em *mbody ou em @spec_cases*

A refatoração para eliminação de código duplicado, catalogada por Li [7] para Haskell, assemelha-se com uma refatoração bastante popular, a *Extract Method* [6]. A operação que deve ser realizada em etapas, e é semelhante à refatoração para remoção de procedimento. Inicialmente cria-se um procedimento com a porção de código duplicada e posteriormente essa parcela de código é substituída pela chamada ao procedimento. Aqui será considerado apenas duplicação de código em uma mesma classe.

A **Regra 7** formaliza a eliminação de código duplicado, a porção de código duplicado é detectada, cria-se um novo método, o código duplicado torna-se o corpo do novo método, as variáveis também utilizadas externamente são passadas como parâmetro do novo método, e então o código duplicado é substituído pela chamada do novo método.

Regra 7 - Eliminação de código duplicado

class C {
atbs
funcs
method *m*
(*prs*)
modifies ω_1
{
mbody1;
duplicated
mbody;
mbody2
}
mets
preds
}

$=_{mod,}$

class C {
atbs
funcs
method *m* (*prs*)
modifies ω_1
{
mbody1;
duplicated;
mbody2
}
method *m2*(*prs2*)
modifies ω_2
{ *duplicatedmbody* }
mets
preds
}[**this**.*m2*(*prs2*)/*duplicated*]

Onde:
prs2 é a lista de variáveis locais utilizadas em *duplicatedmbody* também utilizada em *mbody1* ou *mbody2*.
$\omega_2 = \omega_1 \cap \omega_{m2}$, onde ω_{m2} são as variáveis acessadas em *duplicatedmbody*

desde que:

(→) Não há no escopo de *funcs*, *mets* ou *preds* membro com o nome *m2*.

3.2.3. Variáveis e Atributos

Promover a definição de uma variável de um escopo para um mais amplo, de um procedimento para o de uma classe e então para de um módulo, pode permitir que esta seja usada por mais entidades do programa. A operação contrária, rebaixar definição, ajuda a agrupar definições relacionadas e limpar o programa. A **Regra 8** formaliza essa refatoração.

Regra 8 - Promover definição: de método para classe

class *C* {
atbs
funcs
method *m* (*prs*)
{
var x: T;
mbody
}
mets
preds
}
	$=_{mod,}$	**class** *C* {
		var *x*: *T*;
		atbs
		funcs
		method *m* (*prs*)
		modifies this
		{
		mbody
		}[**this**.*x*/*x*]
		mets
		preds
		}

Onde:

(→) Não há no escopo de *C* entidade com o identificador *x*

(←) (1) *x* não é utilizado em outro lugar em C exceto em *m* (2) Não há no escopo de *m* variável com o identificador *x*

3.2.4. Sumário das Regras

Nessa subseção estão resumidas as regras adaptadas nessa seção. Mapeamos cada regra com as que que serviram como base para elas. As regras são apresentadas na **Tabela 2**.

Funcionalidade	**Regra em Dafny**	**Regras da linguagem funcional [7]**
Gerais	Regra 1 - Renomear variável em procedimento	*Rename*
	Regra 2 - Renomear entidade em classe	*Rename*
	Regra 3 – Renomear classe em módulo	*Rename*
	Regra 4 – Renomear datatype em módulo	*Rename*
	Regra 5 – Renomear type em módulo	*Rename*
Procedimentos	Regra 6 - Remover parâmetro	*Remove an argument*
	Regra 7 - Eliminação de código duplicado	*Eliminating duplicated code*
Variáveis e Atributos	Regra 8 - Promover definição: de método para classe	*Demote a definition* e *Promote a definition*

Tabela 2 Sumário das regras para Dafny baseadas nas regras para Linguagem Funcional

4. Conclusão

Durante a vida útil de um software as mudanças do código são frequentes, seja para reestruturação interna ou para adaptação para novas funcionalidades. Para isso, é necessário a realização de refatorações, que tratam-se de mudanças internas no código sem que o comportamento externo do sistema seja afetado. Para verificar que o comportamento de um sistema não foi alterado, é preciso a realização de verificações do código. Uma forma de verificar a corretude do código é por meio do Projeto por Contrato (Design by Contract – DbC), que utiliza métodos formais com a definição de condições para verificação do código.

Para que as modificações não tenham efeitos colaterais um processo formal deve ser adotado. Uma das abordagens de formalização é através da aplicação de leis de programação, que descrevem cenário e condições de aplicação para que uma mudança seja feita. Com base em trabalhos já realizados e considerando as peculiaridades da linguagem escolhida, Dafny, as leis de programação e regras de refatoração foram adaptadas.

Dafny foi escolhida principalmente por apresentar suporte nativo para Projeto por Contratos, diferente de muitas das linguagens disponíveis. Uma outra vantagem, é que trata-se de uma linguagem de aprendizagem simples, sobretudo em aspectos relacionados com verificação de código.

Este capítulo apresenta as contribuições deste trabalho, bem como as limitações identificadas e as direções para trabalhos futuros.

4.1. Contribuições e Limitações

Neste Trabalho de Graduação foi investigado como mudanças em código fonte escrito na linguagem Dafny podem ser realizadas de maneira sistemática utilizando leis de programação. Dessa forma, foram levadas em consideração as construções de Dafny para adaptar leis de programação propostas para Java-JML [3], bem como transformações de programas escritas para linguagens funcionais como Haskell e Erlang [7] [8].

A partir da análise realizada, foi criado um catálogo com as leis de programação e regras de refatoração para Dafny. O catálogo é composto de 14 leis de programação e 8 regras de refatoração. Além de servir como fonte de consulta no processo de refatoração de programas em Dafny, o catálogo pode servir como trabalho inicial para o desenvolvimento de uma ferramenta de refatoração para a linguagem.

Esse trabalho apresenta algumas limitações. Algumas delas referentes ao cenário considerado na adaptação das leis. Foi considerado, por exemplo, a existência

de um módulo único, enquanto que em um sistema real, é provável a existência de múltiplos módulos. Para que sistema com vários módulos fosse considerado seria preciso tomar precauções quanto importações de módulo e uso de seus elementos, já que, por exemplo, funções e métodos não estáticos não podem ser referenciados fora do módulo o qual pertencem.

Apesar de apresentar a especificação de Frame, as leis e regras aqui descritas assumem que todas as declarações podem modificar todas as variáveis da classe ou módulo em que estão inseridas. Dessa forma, não consideramos os aspectos de *dynamic frame* suportados pela linguagem.

Algumas refatorações propostas em outros trabalhos, e que se enquadram com funcionalidade de Dafny, infelizmente não puderam ser consideradas. Algumas limitações quanto a implementação do parse de Dafny os considerava erro de sintaxe. Um desses casos foi com a refatoração catalogada por Li para Haskell [7], "Introduzindo casamento de padrão em um argumento". Dafny suporta casamento de padrão em tipos indutivos, mas não é possível o casamento de padrões de forma recursiva em argumentos do construtor. Ao tentar gerar um exemplo de prova, um erro de sintaxe é retornado.

Consideradas as contribuições e limitações deste trabalho, a próxima seção apresenta as direções e perspectivas futuras.

4.2. Trabalhos Futuros

Como trabalhos futuros, poderíamos citar a proposta de leis de programação e regras de refatoração para as construções de Dafny não cobertas pelas leis e regras propostas. Inclusive considerando as novas construções que estão por vir, já que Dafny é uma linguagem ainda em construção. Um exemplo é subtipos, que em futuras versões será suportado subtipos definidos pelo usuário [12], o que permite refatorações de alteração de tipos de atributos e parâmetros. Uma outra construção importante, e recém definida, é a de *Traits* [12], que comporta-se como uma superclasse abstrata ou interface.

Além disso, o refinamento das leis atuais para criação de outras leis atendendo outros cenários, bem como adaptação das leis atuais para considerar as limitações impostas nos cenários considerados nesse trabalho, considerando a existência de múltiplos módulo e da existência de *dynamic frames*.

O trabalho realizado, e os aqui listados são alguns dos passos para construção de uma ferramenta de refatoração. Que seria o trabalho futuro mais ambicionado pelo

trabalho aqui iniciado. Que poderia vir a gerar benefícios que poderiam influenciar diretamente na eficiência da operação.

4.3. Considerações Finais

O presente trabalho teve se objetivo principal alcançado ao ter fornecido um catálogo inicial de leis de programação e regras de refatoração para a linguagem Dafny. Que podem vir a servir para expansão para um catálogo mais completo compreendendo as demais construções da linguagem. Assim como pode também ser útil para apoiar no desenvolvimento de ferramentas para o suporte na refatoração de sistemas em Dafny.

O trabalho serviu para crescimento pessoal, no desenvolvimento acadêmico e cientifico ao estudar linguagens com suporte ao *Design by Contract*, leis de programação e refatoração.

Referências Bibliográficas

[1] G. F. Freitas, M. Cornélio, T. Massoni e R. Gheyi, “Object-oriented Programming Laws for Annotated Java Programs”.

[2] P. Borba, A. Sampaio, A. Cavalcanti e M. Cornélio, “Algebraic Reasoning for Object-Oriented Programming,” 2004.

[3] G. R. F. Freitas, “Refactoring Annotated Java Programs: A Rule-Based Approach,” 2009.

[4] J. Koenig e K. R. M. Leino, “Getting started with Dafny: a guide,” Marktoberdorf, 2012.

[5] K. R. M. Leino, “Developing Verified Programs with Dafny,” em *Proceedings of the 2013 International Conference on Software Engineering*, Redmond, 2013.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke e D. Roberts , Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.

[7] H. Li, “Refactoring Haskell Programs,” University of Kent, 2006.

[8] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg e T. Nagy, “Refactoring Erlang Programs,” The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden, 2006.

[9] G. T. Leavens e Y. Cheon, “Design by Contract with JML,” 2006.

[10] R. Henne-Wu, W. Mitchell e C. Zhang, “Support for Design by Contract TM in the C# Programming Language.,” *Journal of Object Technology,* vol. 4, nº 7, 2005.

[11] R. M. K. Leino, “Dafny: An Automatic Program Verifier for Functional Correctness,” Springer Berlin Heidelberg, 2010.

[12] K. R. M. Leino, “Types in Dafny,” Microsoft, 22 10 2014. [Online]. Available: http://research.microsoft.com/en-us/um/people/leino/papers/krml243.html. [Acesso em 01 2015].

M. L. Cornélio, “Refactorings as Formal Refinements,” Universidade
13] Federal de Pernambuco (UFPE), Recife, 2004.

H. Li e S. Thompson, “A Comparative Study of Refactoring Haskell and
14] Erlang Programs,” IEEE, 2006.

“Dafny Quick Reference,” Microsoft, 2014. [Online]. Available:
15] http://research.microsoft.com/en-us/projects/dafny/reference.aspx. [Acesso em 01 2015].

H. Li, C. Reinke e S. Thompson, “Tool Support for Refactoring Functional
16] Programs,” ACM, 2003.

APÊNDICE A. Códigos fonte de exemplo

As leis e regras aqui apresentadas não possuem uma prova formal. A fim de validar cada uma delas e comprovar sua solidez, foram escritos códigos de exemplo para simular situações em que elas se aplicariam. A seguir cada um desses códigos é apresentado.

A.1. Códigos fonte das Leis de Programação

Lei 1 - Eliminação de Classe

- Código antes da aplicação da lei

```
module Mod {
  class C {
    var f: int;
    method m()
  }
  datatype Option = A(int) | B(int)
  type T
  method m()
  function f(): int
}
```

- Código depois da aplicação da lei

```
module Mod {
  datatype Option = A(int) | B(int)
  type T
  method m()
  function f(): int
}
```

Lei 2 - Eliminação de chamada de método void

- Código antes da aplicação da lei

```
module Mod {
  class C {
    var f: int;
    method setF(newF: int)
     requires newF > 0;
     modifies this;
     ensures this.f > 0;
    {
      this.f := newF;
    }
  }
```

```
  method InitiatesC()
  {
    var le := new C;
    le.setF(10);
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
    class C {
    var f: int;
    method setF(newF: int)
     requires newF > 0;
     modifies this;
     ensures this.f > 0;
    {
      this.f := newF;
    }
  }

  method InitiatesC()
  {
    var le := new C;
    assert le != null;
    var newF := 10;
    assert newF > 0;
    le.f := newF;
    assert le.f > 0;
  }
}
```

Lei 3 - Eliminação de chamada de método não void

- Código antes da aplicação da lei

```
module Mod {
  class C {
    var f: int;
    method Square(newF: int) returns (square: int)
     requires newF > 0;
     modifies this;
     ensures this.f > 0;
    {
      this.f := newF * newF;
      return this.f;
    }
  }

  method InitiatesC()
```

```
  {
    var le := new C;
    var leF :=  le.Square(10);
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
    class C {
    var f: int;
    method Square(newF: int) returns (square: int)
     requires newF > 0;
     modifies this;
     ensures this.f > 0;
    {
      this.f := newF * newF;
      return this.f;
    }
  }

  method InitiatesC()
  {
    var le := new C;
    var square: int;
    assert le != null;
    var newF := 10;
    assert newF > 0;
    le.f := newF * newF;
    square :=  le.f;
    assert le.f > 0;
  }
}
```

Lei 4 - Eliminação de método

- Código antes da aplicação da lei

```
module Mod {
    class C {
    var f: int;
    method setF(newF: int)
     requires newF > 0;
     modifies this;
     ensures this.f > 0;
    {
      this.f := newF;
    }
  }

  method InitiatesC()
```

```
    {
      var le := new C;
      assert le != null;
      var newF := 10;
      assert newF > 0;
      le.f := newF;
      assert le.f > 0;
    }
}
```

- Código depois da aplicação da lei

```
module Mod {
     class C {
     var f: int;
   }

   method InitiatesC()
   {
     var le := new C;
     assert le != null;
     var newF := 10;
     assert newF > 0;
     le.f := newF;
     assert le.f > 0;
   }
}
```

Lei 5 - Eliminação de chamada de função método

- Código antes da aplicação da lei

```
module Mod {
  class C {
     function method Square(y: int): int
      requires y > 0;
      ensures Square(y) > 0;
     {
       y * y
     }
  }

   method InitiatesC()
   {
     var le := new C;
     var leF := le.Square(2);
   }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    function method Square(y: int): int
     requires y > 0;
     ensures Square(y) > 0;
    {
      y * y
    }
  }

  method InitiatesC()
  {
    var le := new C;
    var rt: int;
    var y: int := 2;
    assert y > 0;
    rt := y * y;
    assert rt > 0;
   }
}
```

Lei 6 - Eliminação de função método

- Código antes da aplicação da lei

```
module Mod {
  class C {
    function method Square(y: int): int
     requires y > 0;
     ensures Square(y) > 0;
    {
      y * y
    }
  }

  method InitiatesC()
  {
    var le := new C;
    var rt: int;
    var y: int := 2;
    assert y > 0;
    rt := y * y;
    assert rt > 0;
   }
}
```

- Código depois da aplicação da lei

```
module Mod {
```

```
  class C {

  }

  method InitiatesC()
  {
    var le := new C;
    var rt: int;
    var y: int := 2;
    assert y > 0;
    rt := y * y;
    assert rt > 0;
  }
}
```

Lei 7 - Eliminação de chamada de predicado método

- Código antes da aplicação da lei

```
module Mod {
  class C {
    predicate method IsEven(x: int)
     requires x > 0;
    {
      x % 2 == 0
    }
  }

  method InitiatesC()
  {
    var le := new C;
    var isEven := le.IsEven(2);
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    predicate method IsEven(x: int)
     requires x > 0;
    {
      x % 2 == 0
    }
  }

  method InitiatesC()
  {
    var le := new C;
    var x: int := 2;
```

```
    assert x > 0;
    var isEven := x % 2 == 0;
  }
}
```

Lei 8 - Eliminação de método

- Código antes da aplicação da lei

```
module Mod {
  class C {
    predicate method IsEven(x: int)
     requires x > 0;
    {
      x % 2 == 0
    }
  }

  method InitiatesC()
  {
    var le := new C;
    var x: int := 2;
    assert x > 0;
    var isEven := x % 2 == 0;
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {

  }

  method InitiatesC()
  {
    var le := new C;
    var x: int := 2;
    assert x > 0;
    var isEven := x % 2 == 0;
  }
}
```

Lei 9 - Eliminação de múltiplos pontos de retorno

- Código antes da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
```

```
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
    {
      if(x > y)
          { max := x; }
      else
          { max := y; }

      return max;
    }
  }
}
```

Lei 10 - Enfraquecer pré-condição

- Código antes da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
       requires x > 2
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
       requires x > 0
    {
      if(x > y)
          { return x; }
```

```
            else
              { return y; }
        }
      }
    }
```

Lei 11 - Fortalecer pós-condição

- Código antes da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
        ensures max >= x
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
        ensures max >= x && max >= y
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

Lei 12 - Agregar pré-condições

- Código antes da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
      requires x != y;
      requires x > 0;
    {
      if(x > y)
          { return x; }
        else
          { return y; }
```

```
    }
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
      requires x != y && x > 0;
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

Lei 13 - Agregar pós-condições

- Código antes da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
       ensures max >= x
       ensures max >= y
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

- Código depois da aplicação da lei

```
module Mod {
  class C {
    method Max(x: int, y: int) returns (max: int)
       ensures max >= x && max >= y
    {
      if(x > y)
          { return x; }
        else
          { return y; }
    }
  }
}
```

Lei 14 - Agregar invariante de loop

- Código antes da aplicação da lei

```
method m(n: nat)
{
    var i: int := 0;
    while (i < n)
        invariant 0 <= i;
        invariant i <= n;
    {
        i := i + 1;
    }
}
```

- Código depois da aplicação da lei

```
method m(n: nat)
{
    var i: int := 0;
    while (i < n)
        invariant 0 <= i && i <= n;
    {
        i := i + 1;
    }
}
```

A.2. Códigos fonte das regras de refatoração

Regra 1 - Renomear variável em procedimento

- Código antes da aplicação da lei

```
class C{
  var x2: int;
  method setX()
    modifies this;
  {
    var x: int := 10;
    this.x2 := x;
  }
}
```

- Código depois da aplicação da lei

```
class C{
  var x2: int;
  method setX()
    modifies this;
   {
```

```
      var x2: int := 10;
      this.x2 := x2;
    }
  }
```

Regra 2 - Renomear entidade em classe

- Código antes da aplicação da lei

```
class C{
  var x: int;
  method setX()
    modifies this;
  {
    var x2: int := 10;
    this.x := x2;
  }
}
```

- Código depois da aplicação da lei

```
class C{
  var x2: int;
  method setX()
    modifies this;
  {
    var x2: int := 10;
    this.x2 := x2;
  }
}
```

Regra 3 – Renomear classe em módulo

- Código antes da aplicação da lei

```
module M
{
  class C
  {
  }
  method A()
  {
    var x := new C;
  }
}
```

- Código depois da aplicação da lei

```
module M
{
  class A
```

```
  {
  }
  method A()
  {
    var x := new A;
  }
}
```

Regra 4 – Renomear datatype em módulo

- Código antes da aplicação da lei

```
module M
{
  datatype D = Empty

  method A()
  {
    var x : D;
  }
}
```

- Código depois da aplicação da lei

```
module M
{
  datatype A = Empty

  method A()
  {
    var x : A;
  }
}
```

Regra 5 – Renomear type em módulo

- Código antes da aplicação da lei

```
module M
{
  type T

  method A()
  {
    var x: T;
  }
}
```

- Código depois da aplicação da lei

```
module M
{
  type A

  method A()
  {
    var x: A;
  }
}
```

Regra 6 - Remover parâmetro

- Código antes da aplicação da lei

```
 method A(a: int)
  {
  }

  method B()
  {
    A(10);
  }
```

- Código antes da aplicação da lei

```
 method A()
  {
  }

  method B()
  {
    A();
  }
```

Regra 7 - Eliminação de código duplicado

- Código antes da aplicação da lei

```
class C {
    method m()
    {
      var x := 10;
      var y := x * 2;
    }
    method mm()
    {
      var x := 10;
      x := x * x;
```

```
      var y := x * 2;
    }
}
```

- Código depois da aplicação da lei

```
class C {
    method m()
    {
      var x := 10;
      m2(x);
    }
    method mm()
    {
      var x := 10;
      x := x * x;
      m2(x);
    }
    method m2(x: int)
    {
      var y := x * 2;
    }
}
```

Regra 8 - Promover definição: de método para classe

- Código antes da aplicação da lei

```
class C {
    method m()
    {
      var x: int;
      x := 10;
      var y := x * 2;
    }
}
```

- Código depois da aplicação da lei

```
class C {
     var x: int;

    method m()
    modifies this
    {
      x := 10;
      var y := x * 2;
    }
}
```