Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Portando teorias entre assistentes de prova: Um estudo de caso

Aluna: Thayonara de Pontes Alves (tpa@cin.ufpe.br)
Orientador: Prof. Leopoldo Motta Teixeira (lmt@cin.ufpe.br)

Trabalho de Graduação

Recife
Julho de 2018

Universidade Federal de Pernambuco
Centro de Informática

Thayonara de Pontes Alves

Portando teorias entre assistentes de prova: Um estudo de caso

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Leopoldo Motta Teixeira

Recife
Julho de 2018

Aos meus pais, Dalva Pontes e Marcos Alves. Sem vocês, não.

Agradecimentos

Ao Deus triuno, pessoal e infinito, fonte de todo conhecimento e refrigério em minha caminhada. Agradeço por seu chamado eficaz em minha vida e pelas suas providências que me fizeram chegar até aqui.

Às grandes famílias Rodrigues de Pontes e Emiliano Alves, que enchem minha vida de boas recordações. Agradeço especialmente aos meus pais, por todo suporte, renúncias e orações incessantes que me impulsiona a alcançar os meus objetivos. Ao meu irmão Marcos e ao meu primo Matheus, que tenho um amor fraternal, pelo companheirismo de todos esses anos. Às minhas priminhas, em particular à Larissa, por sua capacidade de trazer tanta leveza em minha vida, principalmente nos momentos de tensão.

Ao Alessandro, meu amor e melhor amigo. Agradeço por estar sempre ao meu lado, por suportar os dramas que antecedem os períodos de provas e de entregas de projeto, por abrir mão de todo seu espaço quando eu precisava permanecer em Recife e pelos inúmeros cafés da manhã que você preparou para mim durante esses dias.

Ao meu orientador Leopoldo, que tem sido um grande exemplo (não apenas academicamente) pra mim nesses últimos anos, conduzindo aquilo que faz com enorme competência e indo sempre além do que se espera dele. A respeito desse trabalho, agradeço pelos conhecimentos passados, pelo suporte com a formalização, além de toda compreensão, paciência e tranquilidade passada durante esse período.

Ao professor Márcio Cornélio, por ter aceitado o convite para participar da banca e sugerir melhorias para esse trabalho.

Aos demais professores do CIn, por me fazerem ter a convicção que escolhi o melhor curso, em um centro de excelência, para fazer a minha graduação.

Aos meus professores do ensino médio e técnico, principalmente tio Lauro e Luiz Carlos, por me darem o amor à computação e serem minhas inspirações para escolha do meu curso.

Ao Leet, melhor célula de estudos, representada no CIn por Avyner, Guilherme e Ricardo, por termos sonhado com isso e por estarmos realizando juntos.

A todos amigos feitos no CIn e que foram participantes comigo dos sofrimentos impostos pelas deadlines de projetos: Késsia, Pedro Henrique, Raquel, Albertinin, Pedro Sereno, Bárbara, Deyvson, Eduardo, Márcia, João Luiz, João Santos, Egberto, Lucas, Mário, Layon, entre outros. Vocês marcaram minha Graduação!

Aos pastores, líderes, funcionários e irmãos da Igreja Presbiteriana A Ponte, grandes exemplos pra mim, por estimularem meu amadurecimento e pelo incentivo do exercício de nossas vocações - não importando qual seja. Agradeço especialmente ao meu GR, pelo apoio e orações constantes.

À Associação Brasileira de Cristãos na Ciência (ABC²), por solidificar por meio de palestras, seminários, reuniões do grupo local, treinamento, livros e vários artigos, minhas convicções a respeito do bom convívio da Fé e da Ciência, e por auxiliar os cristãos que

pertencem aos dois campos (Fé Cristã e Científico) a melhor integrar suas vocações científica e espiritual. Graças a ABC², pude lidar melhor com os questionamentos que surgiram no ambiente acadêmico e ajudar outros com essas questões também.

No demais, agradeço a todos meus amigos que suportam minhas ausências com paciência. Aos que não têm paciência, também agradeço. Só não desistam de mim!

“It makes sense that there is no sense without God.”

Edith Schaeffer

Resumo

Linhas de Produtos de Software (LPS) são famílias de produtos de software onde produtos semelhantes são criados a partir de uma característica comum, maximizando reuso de software, diminuindo os custos e aumentando a qualidade dos produtos desenvolvidos. Elas são representadas formalmente por uma tripla de elementos: (i) um modelo de features contendo as features e as dependências entre elas; (ii) um asset mapping, que contém conjuntos de artefatos e nomes de ativos; (iii) um configuration knowledge, que permite que as features sejam relacionados aos artefatos.

Em LPS, há também muitos desafios. Os sistemas tendem a crescer com o tempo, o que aumenta a complexidade de evoluir a LPS. Ao evoluir, em muitas situações, gostaríamos de ter a garantia de que podemos alterar com segurança uma LPS no sentido de que o comportamento dos produtos existentes é preservado após a alteração. Por isso, é importante ter uma noção de refinamento da linha de produtos que garanta a preservação do comportamento dos produtos da linha de produtos original. A teoria de refinamento de linhas de produtos formaliza essa noção. Para provar a solidez dessa teoria, este trabalho descreve um esforço de sua formalização, usando o assistente de provas Coq, portando essa formalização do trabalho já realizado no provador de teoremas PVS.

Palavras-chave: Linhas de produtos de software, refinamento de linha de produtos, assistentes de provas, Coq.

Abstract

Software Product Line (SPL) are families of software products where similar products are created from a common feature, maximizing software reuse, reducing development costs, enhancing the quality of the developed products. They are formally represented as a triple: (i) a feature model that contains features and dependencies among them, (ii) an asset mapping, that contains sets of assets and asset names, (iii) a configuration knowledge, that allows features to be related to assets.

There are several challenges in the SPL development context. SPLs tend to increase over time, and the larger a SPL becomes, the higher is the complexity to evolve it. When evolving, in some cases, it is desirable to provide some assurance that we can safely change a SPL in the sense that the behaviour of existing products is preserved after the change. For this, it is important to have a notion of product line refinement that assures behavior preservation of the original product line products. The theory of product line refinement formalizes this notion. To specify and prove soundness of the theories, this work describes an effort to formalize them using the Coq proof assistant, porting the existing formalization already realized in the PVS theorem prover.

Keywords: Software product lines, product line refinement, proof assistants, coq.

Sumário

Lista de Figuras

Lista de Quadros

Capítulo 1

Introdução

1.1 Contexto

Software permitiu conquistas importantes para o mundo moderno. Seja neste planeta ou mesmo no espaço, ele está cada vez mais presente, moldando nossa sociedade e servindo como um dos principais aliados quando lidamos com as questões que nos cercam. No final dos anos 60, já tendo dimensão da importância dos sistemas de software em muitas atividades daquela época e reconhecendo o fracasso em áreas do campo, devido a falta de uma abordagem sistemática no desenvolvimento desses sistemas, foi adotado o termo engenharia de software, projetando a analogia a outros campos da engenharia [Naur and Randell, 1969]. Portanto, a engenharia de software traz aplicação de técnicas de engenharia e não se limita aos processos técnicos do desenvolvimento do software mas foca em todos os seu aspectos, incluindo desenvolvimento de ferramentas, métodos e teorias para apoiar a produção de software [Sommerville, 2010].

Com a intenção de trazer benefícios como maior qualidade, menor custo e tempo de produção, as empresas vêm investindo em abordagens que maximizem o reúso de softwares existentes, tendo a Linha de Produtos de Software (LPS) como uma das principais delas. A ideia das LPS é unir os benefícios da customização em massa, construindo soluções individuais a partir de um conjunto de peças reutilizáveis e da produção em massa, construídos em grande escala, muitas vezes de maneira automatizada. Isto requer um investimento inicial significativo que sobressai ao do investimento de desenvolver um único produto, mas que se torna a melhor opção quando pensamos a longo prazo [Apel et al., 2013].

É esperado que software esteja sempre evoluindo, e para que a LPS obtenha sucesso, é necessário que sua infraestrutura seja, a longo prazo, um meio adequado para o lançamento de novos produtos no mercado de maneira eficiente [Van der Linden et al., 2007]. No entanto, problemas podem ocorrer quando a LPS está progredindo em sua evolução, principalmente quando isso se dá através de extração e modificação manual de diferentes parte do código e de outros artefatos. A LPS pode escalar até ser capaz de gerar um número elevado de combinações que resultam em produtos, estando mais sujeito à geração de produtos inválidos e a introdução de erros [Teixeira, 2010].

A checagem manual relacionada à composição segura, torna-se uma tarefa maçante em LPS, quando não impraticável, e é benéfico adotar meios que previnam que os produtos

gerados estejam com propriedades inválidas [Teixeira, 2010]. Felizmente, as LPS podem se beneficiarem de processos que geram automaticamente produtos. Para isso, são necessários introduzir elementos extras como: *Feature Models* (FMs, ou modelos de feature) [Kang et al., 1990], *Asset Mapping* (AM) e *Configuration Knowledge* (CK, ou modelos de configuração) [Czarnecki and Eisenecker, 2000]. FMs normalmente são expressos em forma de árvores, e contém as features como nós e as informações de como elas se relacionam. AM é o mapeamento um-para-um de nomes de artefatos de software para artefatos reais. Enquanto que CKs tratam de mapeamentos entre features e artefatos [Borba et al., 2012].

A respeito disso, teorias relacionadas à evolução segura de LPS vem sendo desenvolvidas, das quais tem ênfase a teoria do refinamento de LPS [Borba et al., 2012], já formalizada utilizando o provador de teoremas PVS, que estabelece a independência da noção de refinamento em relação às linguagens usadas para descrever artefatos, FMs ,AMs e CKs.

Um outro assistente de provas que vem ganhando popularidade entre acadêmicos, pesquisadores e engenheiros é o Coq. Coq ajuda com: notações avançadas, pesquisa de prova e desenvolvimentos modulares. Ele também permite extrair código para linguagens como Ocaml e Haskell [Mohring, 2011]. Vários projetos em diferentes áreas foram realizados com essa ferramenta. Um exemplo na área da matemática, foi a prova do teorema de Feit-Thompson, realizada por Georges Gonthier juntamente com colegas da Microsoft Research e INRIA [Gonthier, 2011]. Em segurança computacional, Barthe e outros usaram Coq para desenvolver *Certicrypt*, um ambiente de prova formais de criptografia computacional [Barthe, 2009]. Um outro exemplo está relacionado as empresas *Gemalto* e *Trusted Logic*. Elas obtiveram o mais alto *Nível de Garantia de Avaliação* (EAL 7) pela formalização das propriedades de segurança da plataforma *JavaCard* , utilizando Coq em sua formalização [Chetali et al., 2008].

Por esses fatos e características atraentes de Coq, resolvemos especificar teorias que já estavam formalizadas em PVS neste sistema, como forma de estudo e avaliação do mesmo.

1.2 Objetivo

Este trabalho tem como objetivo especificar teorias de modelos específicos de uma LPS, como FM e CK, sendo um primeiro passo para especificação da teoria de refinamento de LPS em Coq. Estas teorias já foram especificadas em PVS. Apesar disso, valendo-se que já existem linguagens de programação formalizadas em Coq, há o potencial de integração das formalizações destes modelos com estas linguagens, o que pode nos levar a especificações e provas mais expressivas. Adicionalmente, pretendemos refletir brevemente sobre este processo e diferenças entre os sistemas.

1.3 Estrutura do trabalho

A presente monografia está organizada em 4 Capítulos e 2 apêndices contendo as provas dos lemas.

O Capítulo 2 contém a fundamentação teórica necessária para o entendimento dos conceitos abordados no trabalho e as razões que justificam o mesmo. Nele, incluímos conceitos, benefícios, desafios e elementos que compõe as Linhas de Produtos de Software, definição da teoria de Refinamento de Linhas de Produtos e uma introdução ao assistente de provas Coq, dando pequenos detalhes de algumas definições permitidas nesse sistema e que foram usadas para elaboração deste trabalho.

No Capítulo 3 descrevemos a implementação da formalização que realizamos no assistente de provas Coq, relacionando com conceitos apresentados no capítulo anterior. Adicionamos a esse capítulo um passo a passo de uma das provas que construímos, além de dar um breve relato das diferenças sentidas com o desenvolvimento em Coq para o realizado em PVS.

Por fim, no Capítulo 4, concluímos o trabalho e descrevemos o que planejamos realizar como continuidade do que realizamos até aqui.

Capítulo 2

Fundamentação teórica

Neste capítulo são descritos os conceitos de linhas de produtos de software, refinamento de linhas de produtos e de Coq, necessários para o discernimento deste trabalho.

2.1 Linha de Produtos de Software

A necessidade de uma rápida entrega de produtos unido à introdução do individualismo à produção, pressionou o mercado a adotar abordagens que viabilizam a diversidade dos portfólios de produtos. Reutilizar artefatos já existentes ao invés de começar a produção do zero, se mostrou uma excelente forma de seguir nessa direção. A respeito disso, as linhas de produtos vem proporcionando aos seus adeptos, uma maneira mais eficaz de produção a partir do reúso de componentes, fornecendo-lhes vantagens competitivas. Semelhantemente, o reúso tem sido incentivado a fim de maximizar o retorno sobre os investimentos em software [Sommerville, 2010], e as linhas de produtos de software (LPS) vem ganhando força na indústria e também na academia.

LPS, conhecida também como família de produtos de software, é uma abordagem para o reúso sistemático do software que busca ter as vantagens da produção em massa com os benefícios da customização em massa. Esse reúso resulta então, não apenas em uma maior quantidade de produtos, mas em uma maior qualidade, por levar em consideração os desejos individuais de seus clientes e apresentando um número mínimo de erros, defeitos e falhas possíveis, por dar a garantia de que eles são verificados e testados um número maior de vezes.

Uma LPS consiste em um conjunto de sistemas de software que compartilham de funcionalidades em comum, mas que diferem uma das outras. Para isso, se é pensado em componentes já existentes no portfólio ou que ainda serão produzidos, mas que possam fazer parte do conjunto de componentes comuns reutilizáveis por todos os sistemas que são membros da mesma família. A partir do momento em que se deseja criar um outro produto, usam-se os componentes comuns à família e insere, se necessário, outras funcionalidades a fim de atender as necessidades de um determinado segmento de mercado. Assim, o novo produto é construído com menos esforço uma vez que ele faz parte de uma LPS.

Há vários casos de sucesso relatados de organizações de pequena à grande porte que

incluíram LPS em suas produções. Um exemplo é a Overwatch, empresa que fornece produtos relacionados a análises geoespacial, inteligência multi-source e inteligência personalizada. Em 2003, a empresa adotou o método LPS produzindo a linha *Overwatch Intelligence Center*. Desse ano até o ano de 2009, a linha subiu de dois para dez produtos e aumentou a receita em um fator de 3,6 [Mcgregor et al., 2010]. A empresa garante que isso não teria sido possível sem a diminuição dos custos de produção e o "*time to market*" fornecidos por essa abordagem.

Um outro caso é a linha de produtos core Flight Software System (cFS). Lançado como código aberto em 2015, foi inicialmente desenvolvido pelo Centro de Voo Espacial Goddard (GSFC) da NASA e vem servindo como base para softwares que permitem missões espaciais que, sem o uso dele, levaria anos para ser concluído [1]. A linha cFS vem trazendo benefícios como redução do tempo de implantação de software, redução do cronograma de projeto e ajudando a traçar o orçamento com mais certezas e facilitam diretamente a reutilização formal do software. Outras empresas como a Hewlett Packard, General Motors, Boeing, Nokia, e Philips aplicam essa abordagem na produção de seus produtos.

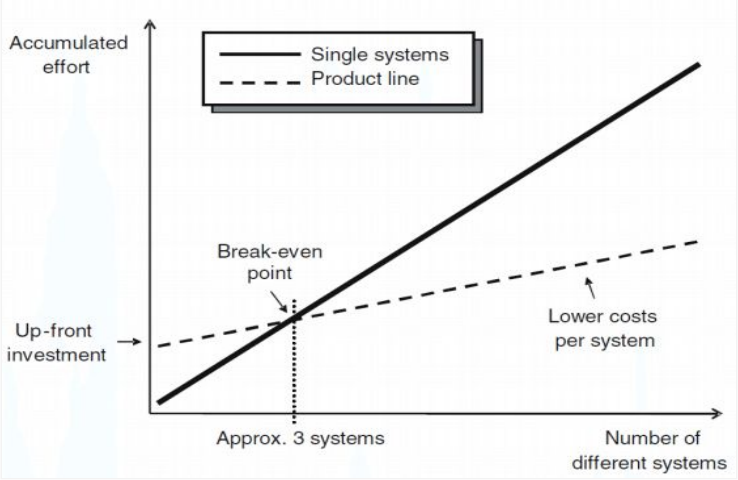

Figura 2.1 Custos para desenvolver n sistemas entre sistemas tradicionais e em LPS. Fonte: [Linden et al., 2005].

Infelizmente, esses benefícios não vêm de graça, é preciso fazer um investimento inicial superior ao das estratégias tradicionais [Linden et al., 2005] . A Figura 2.1 retrata isso. Nela, vemos a comparação dos custos acumulados em relação ao número de sistemas diferentes dentro de uma LPS. O investimento inicial é maior para uma família de sistemas, comparado ao desenvolvimento de um sistema único. No entanto, observamos que a partir de três produtos, que é o ponto em que gastos se equivalem, os custos de sistemas únicos

crescem mais rápido que os de família de sistema, o que torna LPS uma estratégia mais rentável que as tradicionais.

Neste trabalho, adotamos uma representação de LPS através de 3 elementos:

- Modelo de feature, composto pelas features e as dependências entre elas.
- Asset mapping, que relaciona os nomes dos assets aos assets propriamente dito.
- Configuration Knowledge, que mapeia expressões de features para seus respectivos assets.

No restante deste capítulo, fornecemos mais detalhes sobre estes elementos.

2.1.1 Feature Model

Por capturar intenções dos stakeholders de uma linha de produtos, além de conceitos de *design* e de implementação usados para estruturar, reutilizar e mudar artefatos de software, existem muitas definições para *feature* [Apel et al., 2013]. Uma delas é que se trata de um elemento que amplia e modifica a estrutura de um determinado programa para satisfazer os requisitos de um ou mais *stakeholders* [Apel et al., 2013].

Features são usadas para distinguir produtos em uma LPS e exercem um papel importante quanto ao gerenciamento de variabilidade na abordagem *Feature-Oriented Domain Analysis* (FODA) [Kang et al., 1990].

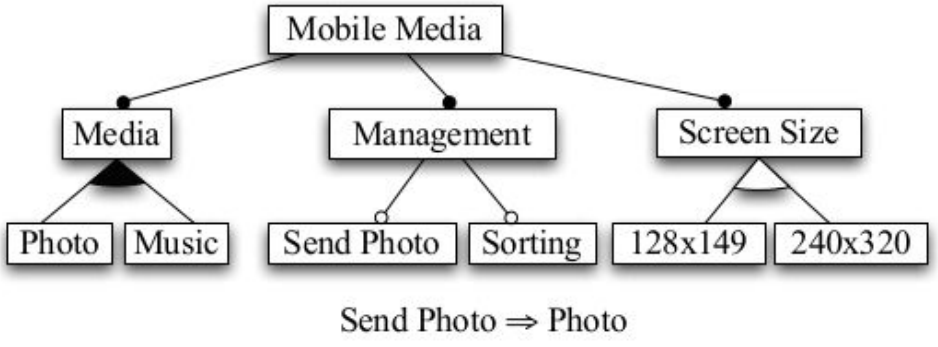

Figura 2.2 Feature Model. Fonte: [Neves et al., 2012]

Combinações diferentes de *features* resultam em produtos diferentes em uma LPS. Essas combinações são bem capturadas por *feature models* (FMs), geralmente, representados através de árvores, onde é possível expressar os relacionamentos das features. As *features* têm nomes distintos e grupos abstratos de requisitos associados [Kang et al., 1990], e podem ser de 4 tipos:

- **Obrigatória**: representado graficamente por um círculo preenchido, a *feature* tem que estar sempre presente, caso a *feature* pai seja selecionada;
- **Opcional**: representado graficamente por um círculo vazio, a *feature* pode estar presente ou não;
- **Alternativa**: representado graficamente por um arco preenchido, *features* desse tipo fazem parte de um grupo no qual somente uma pode ser selecionada;
- **Ou**: representado graficamente por um arco vazio, *features* desse tipo pertencem a um grupo no qual mais de uma feature pode ser selecionada.

As combinações de features, em uma LPS não devem ser realizadas de qualquer forma, devem ser respeitadas as restrições impostas pelo FM, sejam elas expressas através das relações entre *features* visíveis diretamente pelo diagrama ou por meio de fórmulas. Na Figura 2.2, temos o exemplo do FM de Mobile Media, contendo 10 features e uma fórmula. Nela, `Media`, `Management` e `Screen Size` são features obrigatórias, `Send Photo` e `Sorting` são opcionais, `Photo` e `Music` são alternativas e `128x149` e `240x320` são ou. A fórmula abaixo do modelo da Figura 2.2 diz que em todo produto gerado em que `Send Photo` tiver presente, `Photo` também estará. Sendo assim, uma configuração como `{Send Photo, Sorting, Music}`, é uma composição de features inválida, por não respeitar a fórmula do FM. Outro exemplo de configuração inválida seria `{Send photo, Photo, 128x149, 240x320}`, por não respeitar a restrição de `128x140` e `240x320` serem alternativas.

2.1.2 Asset Mapping

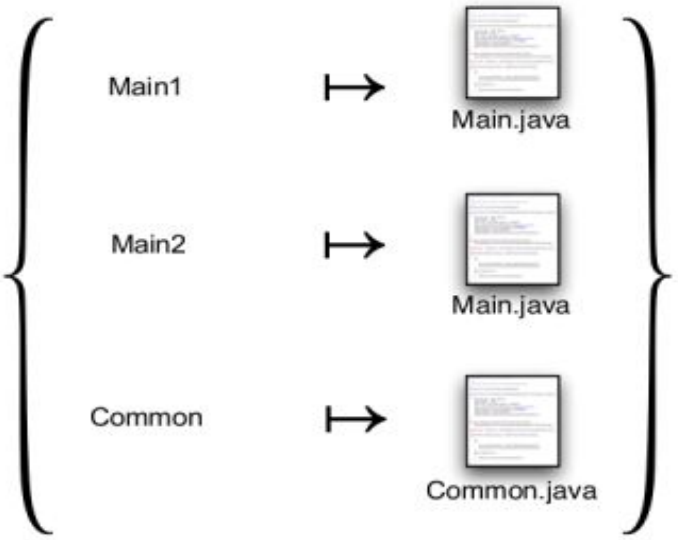

Figura 2.3 Asset Mapping. Fonte: [Neves et al, 2012]

Em uma LPS, com base em um conjunto de artefatos (*asset*) reutilizáveis, um desenvolvedor de software pode gerar um produto implementando funcionalidades com base nos requisitos de um cliente. A respeito disso, devemos considerar linguagens diferentes para a especificação e implementação de *features*, como documentos de requisitos, modelos de design, código, modelos, testes, arquivos de imagem, arquivos XML e assim por diante [Teixeira, 2014]. Apesar dos conceitos e exemplos na teoria de refinamento de linhas de produtos de software focarem em assets de código, os conceitos podem ser aplicados a outros *assets*.

Em um AM, os nomes dos *assets* são mapeados para esses artefatos reais. É um mapeamento único, onde um nome de *asset* estará relacionada a apenas um *asset*. No exemplo da Figura 2.3, temos os nomes dos assets do lado esquerdo sendo relacionados aos seus respectivos *assets*, ao lado direito. Existem dois **Main.java**, no entanto, um está relacionado ao nome de asset **Main1** e o outro a **Main2**, o que é possível por se tratar de arquivos distintos e em locais diferentes. AM elimina ambiguidades, dependendo da funcionalidade de interesse, um deles será selecionado, enquanto o outro não.

2.1.3 Configuration Knowledge

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

Figura 2.4 Configuration Knowledge. Fonte: [Neves et al, 2012]

Configuration Knowledge (CK) é uma relação de expressões de features aos assets [Borba et al., 2012]. Para que um CK seja bem formado, é preciso que os nomes das *features* estejam presentes no FM e os nomes de artefatos presentes no AM.

Assim que um produto novo é planejado, primeiramente, são selecionadas as features válidas de acordo com um FM, logo depois o CK é processado para se ter os assets que implementam essas features e obtemos o produto final [Sampaio, 2017]. Por exemplo, segundo a Figura 2.4, ao selecionarmos `Photo`, `Photo.java`, entre outros que não aparecem na figura, deverá estar presente no produto final. Se `Photo` e `Music` forem selecionados, `Common.aj`, `Music.java`, `Photo.java`, entre outros, fazem parte do novo produto.
Apresentado os três elementos que compõem uma linha de produtos, damos sua definição formalizada a seguir.

Definição 1: < Linha de produtos >

Para um Feature Model F, um Asset Mapping A e um Configuration Knowledge K, dizemos que a tupla

$$(F, A, K)$$

é uma linha de produtos quando, para todo $c \in [\![K]\!]$, desde que

$$wf\,([\![K]\!]_c^A)$$

onde $[\![K]\!]_c^A$ é o conjunto de todas as configurações válidas da semântica de um FM como $[\![F]\!]$ da função semântica de um CK como K, de um AM como A e de uma configuração de produto c.
A restrição exige que $[\![K]\!]_c^A$ seja um conjunto bem formado de artefatos. A restrição de boa formação na definição se faz necessária, pois a falta de uma entrada em um CK pode ocasionar em conjuntos de artefatos em que partes necessárias do produto não estejam presentes e, portanto, não são considerados produtos válidos.

2.2 Refinamento de Linha de Produtos

Ao evoluir uma LPS, há uma preocupação quanto a introdução de defeitos ou alteração de comportamento indesejados entre os seus produtos, podendo haver necessidade de testar toda LPS. Em particular, para que não se perca os benefícios que essa abordagem oferece, é requerido que haja garantia da preservação do comportamento dos produtos fabricados durante um cenário de evolução. Isso pode ser bastante desafiador, uma vez que temos que levar em consideração os artefatos, FMs e CKs. Para dar suporte aos desenvolvedores ao lidar com as preocupações relacionadas à complexidade de desenvolver LPS, é necessário

levar em conta a noção de refinamento de LPS [Borba et al., 2012], que já foi codificada e comprovada usando o sistema PVS.

Assim como o refinamento de programa e modelo [Borba et al., 2004] [Gheyi et al., 2005], o refinamento de PL preserva o comportamento, permitindo adicionar novos produtos preservando os existentes [Borba et al., 2012]. A Definição 2 apresenta a formalização de refinamento de linha de produtos.

Definição 2: < Refinamento de Linha de Produtos >

Para as linhas de produtos (F, A, K) e (F', A', K'), a segunda refina a primeira, denotada como

$$(F, A, K) \sqsubseteq (F', A', K')$$

sempre que

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^A$$

Aqui, $[\![K]\!]_c^A \sqsubseteq [\![K]\!]_c^A$ indica o refinamento do conjunto de artefatos. Esta definição relaciona duas LPS, por isso todos os produtos da nova LPS (F', A', K') também são bem formados. Por ser composicional, essa definição dá a garantia que refinar um único elemento de uma LPS, implica em refinar toda a LPS. Isso é essencial em termos práticos, quando desenvolvedores alteram apenas um dos elementos em um cenário de evolução.

2.3 Assistente de provas Coq

Coq é uma ferramenta gratuita que permite o que é chamado de *prova assistida por computador*, onde as provas são parcialmente geradas de forma automática, mas necessitando da intervenção humana para guiar a construção dessas provas. Nesse contexto, Coq faz uso de duas linguagens com propósitos diferentes: Gallina e Vernacular. Gallina é a linguagem que permite desenvolver teorias que são construídas a partir de axiomas, hipóteses, parâmetros, lemas, teoremas, definições de constantes, funções, predicados e conjuntos [2]. Vernacular, por sua vez, é a linguagem de comandos, usada para definir objetos, construção de scripts de prova e prover a interação com o usuário.

Coq tem várias propriedades interessantes:

- **Baseado em uma linguagem de programação funcional de ordem superior**: Isso permite que façamos uso dessa ferramenta para desenvolver programas comuns, sem utilizar os recursos relacionados às provas, além de permitir expressar naturalmente propriedades de ordem superior.
- **Tipos dependentes**: Tipos dependentes associam tipos a valores de forma a possibilitar a construção de programas mais seguros e eficientes, fornecendo um maior controle sobre os dados utilizados nesses programas.
- **Uma linguagem de prova do Kernel *easy-to-check***: Atendendo ao "critério de Bruijn", podemos ignorar a possibilidade de erros durante a construção de uma prova e confiar apenas em um *kernel* (relativamente pequeno) de verificação de provas [Chlipala, 2013].
- **Automatização de provas programável**: Quase qualquer problema de verificação interessante é indecidível, então é importante apoiar os usuários, possibilitando que eles criem seus próprios procedimentos para resolver os problemas restritos que encontram em determinados teoremas [Chlipala, 2013].

O coração de Coq é o seu algoritmo de *type-checking*, que nos garante que as provas foram corretamente construídas, portanto, garantindo que os programas satisfaçam as suas especificações correspondentes. Isso faz de Coq uma opção, não apenas para o desenvolvimento de teorias matemáticas abstratas, mas também para o desenvolvimento de programas que atendam as especificações, principalmente se tratando de sistemas que se exige uma confiança absoluta: por exemplo, sistemas de telecomunicações, bancos, transporte, etc [Bertort et al., 2010].

Esse sistema faz parte de uma grande família de ferramentas de provas assistidas por computador, dentre elas: Automath, Nqthm, Mizar, LCF, Nuprl, Isabelle, Lego, HOL, PVS e ACL2 [Bertort et al., 2010]. Nessa seção, introduzimos os conceitos de definições permitidas em Coq.

2.3.1 Iniciando em Coq

2.3.1.1 CoqIDE

CoqIDE é uma ferramenta gráfica amplamente utilizada e seu principal objetivo é possibilitar ao usuário a navegação em um arquivo vernacular (arquivos no formato .v), executando comandos ou desfazendo-os. O exemplo da tela principal do CoqIDE é dado na Figura 2.5. À esquerda, contém os comandos que são interpretados sequencialmente. Conforme

executamos esses comandos, um feedback em cores é fornecido. Por exemplo, por padrão, a cor verde é usada para indicar os textos que já foram processados, enquanto que o vermelho aponta os trechos que contém erros. Na parte superior, ao lado direito, exibe as metas que ainda não foram provados, em caso de estarmos no modo de prova. Abaixo dela, temos a janela que exibe outputs dos comandos, além das mensagens de erros quando eles ocorrem. Também temos uma barra de menu e uma barra de ferramentas no canto superior, juntamente com uma barra de status no canto inferior.

Uma alternativa ao CoqIDE é usar emacs com proof-general.

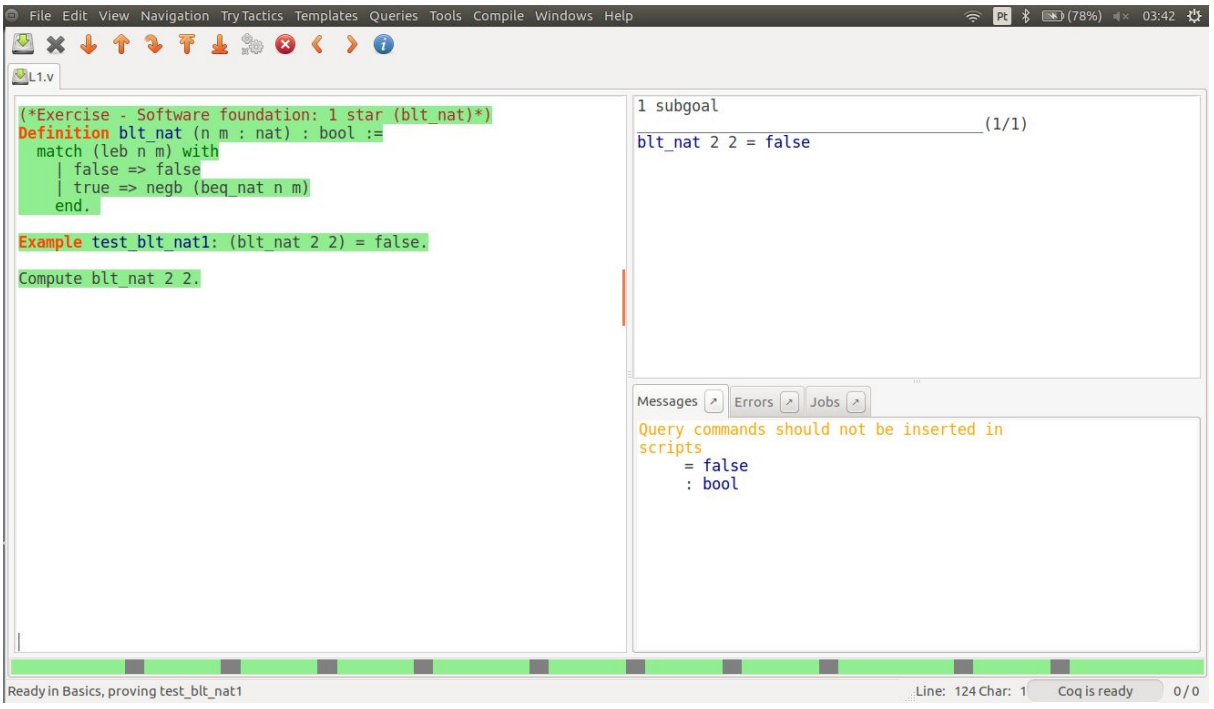

Figura 2.5 CoqIDE

2.3.1.2 Expressões, tipos e funções

Uma expressão em Coq é formada por um nome e um tipo [Pierce et al., 2018]. O comando `Check` verifica se a expressão é bem formada e solicita ao Coq para mostrar o seu tipo.

Check bool**.**

⇒ bool: Set

Check 3**.**

⇒ 3: nat

O objeto bool é um tipo predefinido para valores booleanos, possuindo o tipo Set. Já a constante 3 o tipo natural. Funções como `andb` em si também são valores de dados, como true e false .

```
Check andb.
⇒ andb
   : bool -> bool -> bool
```

`andb` tem o tipo bool -> bool -> bool, ela pode ser lida como "Dada duas entradas do tipo bool, a sua saída será do tipo bool".

2.3.1.3 Proposição

```
Check 1 < 10.
⇒ 1 < 10 : Prop
```

Coq usa uma variação muito expressiva do cálculo das construções indutivas [Mohring et al., 2011]. Uma característica importante desse cálculo é que todo tipo é também um termo e também tem um tipo [Bertort et al., 2010]. Proposições têm o tipo Prop, por exemplo, 1 < 10 é o tipo de todas as provas de que 1 é menor do que 10 e também é um termo do tipo Prop. Abaixo, uma síntese da sintaxe de Coq para proposições lógicas.

⊥	T	a = b	a ≠ b	¬ P	P ∧ Q	P ∨ Q	P → Q	P ⇔ Q
False	True	a = b	a<>b	~ P	P /\ Q	P \/ Q	P -> Q	P <->Q

2.3.1.4 Quantificadores

Para quantificar universalmente proposições como n > 0, onde n é um número natural, usamos o termo forall para representar o quantificador ∀.

```
Check (forall n: nat, n > 0).
⇒ forall n : nat, n > 0  : Prop
```

Para dizer que há algum x do tipo T, tal que alguma propriedade P, escrevemos ∃ x : T , P . Em vernacular, o quantificador existencial se dá pelo comando `exists`.

```
Check (exists n : nat , 4 = n + n).
⇒ exists n : nat, 4 = n + n : Prop
```

2.3.1.5 Tipos Enumerados

Em Coq, ao invés de usar apenas tipos de dados já definidos em seus recursos internos, podemos definir tipos a partir do zero [Pierce et al., 2018]. Tipos enumerados são uma maneira simples de declaração ou definição de tipos. Eles são usados para descrever conjuntos finitos [Bertort et al., 2010] e seus membros são representados pelos seus construtores.

```
Print bool.
⇒ Inductive bool : Set :=
| true  : bool
| false : bool
```

O comando `Print` exibe na tela informações a respeito do objeto declarado ou definido. Por meio desse comando, vemos, mais uma vez, que bool possui o tipo Set e é enumerado pelos construtores true e false.

No momento que definimos um tipo indutivo, Coq inclui, automaticamente, teoremas para que seja possível raciocinar e computar tipos enumerados. `bool_ind`, por exemplo, é o princípio de indução associado à definição indutiva. Para visualizá-lo, podemos usar o seguinte comando:

```
Check bool_ind.
```

E como resultado, temos o seguinte:

```
bool_ind
    forall P : bool -> Prop, P true -> P false -> forall b : bool, P b
```

Nesse teorema, há uma quantificação universal (`forall`) a respeito de uma propriedade P do tipo bool, seguida de implicações aninhadas, onde cada premissa trata-se de P aplicada a um de seus valores, vindo a conclusão que afirma que P vale para todos valores booleanos. De uma forma mais resumida, ela indica que, pra checar que uma propriedade é válida para todos os valores booleanos, basta checarmos se ela é válida para cada um deles.

O teorema `bool_rec`, o princípio de recursão, é semelhante a `bool_ind` e só se diferencia porque a quantificação inicial manipula uma propriedade cujo valor está em Set e não em Prop. Damos o comando `Check` mais um vez, dessa vez pra `bool_rec`.

```
Check bool_rec.
```

E observamos a saída:

```
bool_rec
     : forall P : bool -> Set, P true -> P false -> forall b : bool, P b
```

Com função `bool_rec`, somos capazes de definir uma função sobre o tipo bool simplesmente fornecendo os valores para cada booleano. Também podemos usar `bool_rec` para dar definições recursivas, enunciado na seção 2.3.1.7, sem recursão explícita do `Fixpoint`.

Por último, temos a função `bool_rect`, que é semelhante, mas desta vez o tipo final da proposição P é Type.

```
Check bool_rect.
```

```
bool_rect
  : forall P : bool -> Type, P true -> P false -> forall b : bool, P b
```

2.3.1.6 Módulos

Em um sentido mais amplo, módulos representam uma coleção de definições [Bertort et al., 2010] e são estruturas que ajudam, especialmente, na implementação de grandes projetos, proporcionando uma melhor organização.

```
Module A.

Inductive a : Set :=
| a1 : a
| a2 : a.

End A.

Module B.

Definition b := A.a.

End B.
```

No exemplo acima, temos definidos dois módulos através de `Module...End`. Nele, definimos um tipo indutivo no módulo `A`, e referenciamos-o em `B` através de `A.a` ao invés de apenas `a`.

2.3.1.7 Estrutura Recursiva

Coq permite recursão estrutural através do comando `Fixpoint`, onde a chamada recursiva acontece apenas em subtermos sintáticos de um de seus argumentos e podem ser capturados por meio de casamento de padrão. A construção `match...with...end` é uma macro que permite a escrita de expressões com casamento de padrões e análise de casos, em que cada caso consiste em um padrão, a seta "=>" e o termo do resultado.

```
Fixpoint factorial (n:nat) : nat :=
 match n with
  | 0 => S 0
  | S n' => mult n (factorial n')
  end.
```

`factorial` é um exemplo de função recursiva que gera o fatorial de um número n, onde n' é subtermo de n. Ao chamar `factorial`, os padrões serão testados e, de acordo com o resultado, executado ou não o termo do resultado. Para essa função, é feito o casamento de padrão no seu único argumento:

1. Se for zero, então o resultado é o sucessor de 0 (ou 1). Sendo esse o caso base, será finalizada a função.
2. Se for um sucessor, então o resultado é a multiplicação de n pelo fatorial de n'.

Vale destacar que Coq garante que todas as funções serão encerradas, logo, é exigido que algum dos argumentos esteja "diminuindo" [Pierce et al., 2018].

2.3.1.8 Parâmetros, Definições e Variáveis

```
Parameter p1 p2 p3 : bool.
```

Parâmetros são declarações globais com o formato `Parameter name : type`. Eles podem ser usados para declarar várias variáveis de uma só vez com o comando `Parameter`, como no exemplo acima [Bertort et al., 2010], e o efeito desse exemplo é adicionar (p1 p2 p3: bool) ao ambiente atual.

```
Definition d := 2.
```

Definições é dado no formato `Definition name args : type := term`, e se tratam de declarações de constantes globais. O efeito da definição do exemplo acima é adicionar d := 2: nat ao ambiente atual.

```
Variable v v1 v2: nat.
```

A declaração das variáveis é dada por `Variable name : type`, é semelhante aos parâmetros, porém, são declarações de variáveis locais.

2.3.2 Provas e táticas

Provas tem o propósito de tornar uma certa proposição verdadeira. Em Coq, elas são construídas utilizando as chamadas táticas, que alteram o estado atual da prova, transformando a meta em submetas. Algumas táticas estão resumidas nos quadros de 1 a 4 e fazemos uso de algumas delas na Seção 3.1.5.2.

Uma prova é bem sucedida quando não resta mais submetas a serem provados [Mohring et al., 2011]. As metas podem ser introduzidas através dos comandos `Theorem` (para teoremas) e `Lemma` (para lemas). Teoremas são proposições que se pode demonstrar

como sendo verdadeiras. Um teorema que auxilia na demonstração de outros resultados é chamado de lema [Rosen, 2009]. Já um axioma é uma sentença que se assume como verdadeira e que não precisamos fornecer uma demonstração disso. Em Gallina, axiomas são declarados pelo comando `axiom`.

Iniciamos uma prova através do comando `Proof` e através do comando `Qed` ou `save`, a prova é gravada no ambiente com nome dado e o seu tipo. A partir disso, é possível referenciar a sentença provada e usá-la através das táticas.

2.3.2.1 Resolvendo metas simples

Com as táticas resumidas no quadro 1, somos capazes de provar metas ou submetas consideradas simples de resolver.

Quadro 1: Resolvendo metas simples.

Meta	Descrição
`reflexivity`	Resolve se for uma igualdade trivial.
`contradiction`	Resolve qualquer meta se o contexto contém *False* ou hipóteses contraditórias.
`intuition`	Resolve uma variedade maior de metas fáceis.

Baseado em: [3]

2.3.2.2 Transformando metas em submetas

Durante a construção da prova, podemos querer realizar transformações em nossas metas ou hipóteses, com a finalidade de simplificá-las, incluir suposições no contexto, etc. Isso é obtido pela utilização das táticas do quadro 2.

Quadro 2: Transformando metas em submetas.

Meta	Descrição
`intros/intro`	Introduz variáveis que aparecem com forall e as premissas (lado esquerdo das implicações).
`simpl`	Simplifica a meta ou hipóteses relacionadas com o contexto.
`unfold`	Substitui a definição por seu corpo.
`apply`	Usa implicações para transformar metas e hipóteses.
`rewrite`	Substitui um termo por outro equivalente se a equivalência dos termos já tiver sido comprovada.
`inversion`	Deduz as igualdades que devem ser verdadeiras, dada a igualdade entre dois construtores.
`left/right`	Substitui uma meta que consiste em uma disjunção P \/ Q com apenas P ou Q.

Baseado em: [3]

2.3.2.3 Quebra de metas e hipóteses

As táticas do quadro abaixo é útil para quebrar metas e hipóteses e facilitar o prosseguimento da prova.

Quadro 3: Quebra de metas e hipóteses.

Meta	Descrição
`split`	Substitui uma meta que consiste em uma conjunção P /\ Q com duas submetas P e Q.
`destruct (and/or)`	Substitui uma hipótese P /\ Q por duas hipóteses P e Q . Alternativamente, se a hipótese é uma disjunção P V Q , gera duas submetas com diferentes contextos: um em que P se mantém e outro que Q se mantém.
`induction`	Gera uma submeta para cada construtor de um tipo indutivo e fornece uma hipótese de indução para construtores recursivamente definidos.

Baseado em: [3]

Capítulo 3

Formalização

Coq, como descrito na Seção 2.3, é projetado para desenvolver provas matemáticas, escrever programas e tem poder para expressar propriedades desejadas desses programas, ou seja, fornecer as especificações. Especificações constituem-se em um conjunto de teorias que, por sua vez, incluem um conjunto de teoremas, lemas, axiomas, definições, funções, tipos, entre outros.

Por ser baseado em lógica de ordem superior, é possível a escrita de programas que estejam em conformidade com sua especificação formal. O sistema padrão do Coq, conta também com a biblioteca Prelude, que inclui os módulos de notações, lógica, especificações básicas - conjuntos que podem conter informações lógicas, táticas, tauto, números e outros tipos de dados.

Neste capítulo, apresentamos os principais resultados de nosso trabalho, que é uma formalização de uma parte da teoria de refinamento de linhas de produtos no assistente de provas Coq. Essa formalização, como dito, foi inicialmente realizada em PVS pelo orientador deste trabalho, Leopoldo Teixeira.

3.1 Teoria de Feature Model

3.1.1 Módulo Name

Um produto em uma LPS é descrito por uma configuração do Feature Model. Para que essa configuração seja considerada válida, é necessário que atenda a todos as restrições de um dado FM. Há várias maneiras de representar uma configuração. Para o nosso contexto, o tipo associado a configurações está definido no módulo `Name` como sendo um conjunto de nomes de features, especificamente, um conjunto de `Name`. `Name` trata-se de um tipo não interpretado, sem informações concretas a respeito dele, o que é uma característica importante para raciocinar a respeito de valores arbitrários que satisfazem algumas especificações. Como não temos definição de conjuntos na biblioteca Prelude, importamos a biblioteca ListSet, uma biblioteca para conjuntos finitos, implementada com listas, para a construção das definições que realizam o seu uso, como é o caso da definição de configuração na linha 5 da especificação abaixo.

```
1.  Module Name.    (* Interactive Module Name started *)
2.  Require Import Coq.Lists.ListSet.
3.
4.
5.  Inductive Name : Type.   (* Name is defined *)
6.  Definition Configuration : Type := set Name. (* Configuration is defined *)
7.
8.
9.  End Name.   (* Module Name is defined *)
```

3.1.2 Módulo Form

Como já mencionamos na seção 2.1.1, uma configuração válida de um FM deve respeitar as restrições entre features, que em nossa especificação são expressas por meio de fórmulas proposicionais.

Antes de fornecermos a implementação do módulo em que especificamos como checar que uma configuração satisfaz as restrições, inicialmente apresentamos a construção do que seria uma fórmula, definida em nosso módulo `Form`.

```
1.  Module Form. (* Interactive Module Form started *)
2.
3.
4.  Import Name.
5.
6.
7.  Inductive Formula : Type :=
8.    | TRUE_FORMULA     : Formula
9.    | FALSE_FORMULA    : Formula
10.   | NAME_FORMULA     : Name     -> Formula
11.   | NOT_FORMULA      : Formula -> Formula
12.   | AND_FORMULA      : Formula -> Formula -> Formula
13.   | IMPLIES_FORMULA : Formula -> Formula -> Formula.  (* Formula is defined
14.                                                          Formula_rect is defined
15.                                                          Formula_ind is defined
16.                                                          Formula_rec is defined *)
17.
18.
19.
20.
21. End Form.  (* Module Form is defined *)
```

Uma fórmula é declarada em `Form` como sendo um novo conjunto de valores de dados - nesse caso, um tipo enumerado. Semelhantemente ao exemplo da seção 2.3.1.5, um tipo enumerado tem um nome do tipo e os membros. Aqui, o tipo é chamado de `Formula` e seus membros são `TRUE_FORMULA`, `FALSE_FORMULA`, `NAME_FORMULA`, `NOT_FORMULA`, `AND_FORMULA` e `IMPLIES_FORMULA`. Tais membros, são construtores que permitem a construção de Formula de forma que abranja à sintaxe abstrata de fórmulas proposicionais.

O comentário das linhas 11-14 é a saída fornecida por CoqIDE ao executarmos esse novo tipo. Pela saída gerada, vemos que foi definido automaticamente os teoremas `Formula_ind`, `Formula_rec` e `Formula_rect`. Teoremas com esses sufixos foram enunciados em na seção 2.3.1.5.

3.1.3 Módulo FeatureModel

Tendo definido tipos para fórmulas e para a representação de uma feature, podemos especificar a definição de FM. Representamos um FM como sendo um par de `features` e `formulae`, onde `features` é um conjunto de nomes de features e `formulae` é conjunto de fórmulas. Para facilitar a escrita de outras funções, de axiomas, teoremas, entre outros, construímos a função `names`, que recebe um FM e fornece as features, por meio da função fst (que retorna o primeiro elemento do par), além da função `formulas`, que requer um FM para devolver o segundo elemento deste FM, o conjunto de fórmulas.

```
Module FeatureModel.    (* Interactive Module FeatureModel started *)

Import Name Form.
Require Export Coq.Lists.ListSet.

Definition features : Type := set Name.  (* features is defined *)
Definition formulae : Type := set Formula.  (* formulae is defined *)

Definition FM : Type := features * formulae.  (* FM is defined *)

Definition names_ (fm : FM) : features := fst fm.   (* names_ is defined *)
Definition formulas (fm : FM) : formulae := snd fm.  (* formulas is defined *)

Definition  wfTree (fm: FM): Prop := True.  (* wfTree is defined *)

End FeatureModel.   (* FeatureModel is defined *)
```

3.1.4 Módulo Decidability

Um tipo tem igualdade decidível se quaisquer dois elementos desse tipo forem iguais ou diferentes. Em Coq, trabalhar com listas exige que os tipos de elementos usados nessas listas sejam decidíveis. Dado que ListSet é uma biblioteca de conjuntos implementadas como listas, ela faz essa imposição para os tipos fornecidos. Dessa forma, os tipos devem ser `forall x y:R, {x=y} + {~x=y}`, em que R é o tipo que forneceremos. Em nosso caso, isso foi feito para os tipos `Name`, `Formula` e `Configuration`, como descrito abaixo no módulo `Decidability`.

```
Module Decidability.  (* Interactive Module Decidability started *)

Import Name Form.

Axiom name_dec : forall x y: Name, {x = y} + {x <> y}. (* name_dec is declared *)
Axiom form_dec : forall x y: Formula, {x = y} + {x <> y}.  (* form_dec is declared *)
Axiom conf_dec : forall x y: Configuration, {x = y} + {x <> y}. (*conf_dec is declared *)

End Decidability.
```

Estas declarações adicionam axiomas de que todos os elementos do tipo `Name`, `Formula` e `Configuration` são iguais ou diferentes, tornando o seu tipo decidível.
A seguir, no módulo `FormulaTheory`, começamos a apresentar a teoria que diz respeito às fórmulas.

3.1.5 Módulo FormulaTheory

A função `names` é a nossa primeira função recursiva em Coq e cabe a ela retornar o conjunto de nomes das features usadas em uma dada fórmula. Em tipos indutivos, a definição de tais funções é feita pela utilização de um construtor especial, dito o operador de recursão do tipo, cuja criação em Coq é automática na hora da declaração de cada tipo, sendo o seu nome igual ao do tipo, acrescido do sufixo _rec, como mostrado na Seção 3.1.2 com o exemplo de `Formula_rec`.

Em funções recursivas, precisamos decrementar algum dos argumentos recebidos até que se chegue ao caso base, efetivamente encerrando a recursão e fornecendo o que é pretendido pela função. Em `names`, o argumento é o tipo indutivo `Formula f` e estamos realizando uma recursão estrutural sobre ele - isto é, fazemos chamadas recursivas apenas em valores estritamente menores que a fórmula `f` fornecida. É fundamental que o tipo seja indutivo, para que consigamos usar casamento de padrão a fim de identificar quando estamos ou não no caso base. Este requisito é uma característica fundamental do design de Coq. Em

particular, ele garante que todas as funções que podem ser definidas em Coq serão encerradas em todas as entradas.

```
Module FormulaTheory.  (* Interactive Module FormulaTheory started *)

Import Name Form Decidability FeatureModel.
...

Fixpoint names (f : Formula) : set Name :=
 match f with
  | TRUE_FORMULA => empty_set Name
  | FALSE_FORMULA => empty_set Name
  | NAME_FORMULA n1 => set_add name_dec n1 nil
  | NOT_FORMULA f1 => names f1
  | AND_FORMULA f1 f2 => set_union name_dec  (names f1) (set_diff name_dec
   (names      f2) (names f1))
 | IMPLIES_FORMULA f1 f2 => set_union name_dec  (names f1) (set_diff name_dec
 (names f2) (names f1))
 end.      (*  names is defined
                    names is recursively defined (decreasing on 1st argument *)
```

Como já especificamos anteriormente, o tipo `Formula` apresenta diversos construtores. Para que haja algum nome de feature, o construtor `NAME_FORMULA` deve aparecer na fórmula fornecida ou o resultado será um conjunto vazio. Por exemplo, caso a fórmula seja `NOT_FORMULA (FALSE_FORMULA))`, que é um tipo válido de `Formula`, durante o casamento de padrão, o caso `NAME_FORMULA` nunca é acionado para que se extraia um nome dessa fórmula.

```
Fixpoint wt (fs : features) (f : Formula) : Prop :=
  match f with
      | TRUE_FORMULA => True
      | FALSE_FORMULA => True
      | NAME_FORMULA n1 => In n1 fs
      | NOT_FORMULA f1 => wt fs f1
      | AND_FORMULA f1 f2 => (wt fs f1) /\ (wt fs f2)
      | IMPLIES_FORMULA f1 f2 => (wt fs f1) /\ (wt fs f2)
   end.  (* wt is defined
                   wt is recursively defined (decreasing on 2nd argument) *)
 Fixpoint wtFormulaeAux (fm : FM) (fs : formulae): Prop :=
  match fs with
      | nil => True
      | a1 :: x1 => (wt (names_ fm) a1) /\ (wtFormulaeAux fm x1)
  end. (* wtFormulaeAux is defined
```

```
                wtFormulaeAux is recursively defined (decreasing on 2nd argument) *)

Definition  wtFormulae (fm : FM) : Prop :=
  wtFormulaeAux fm (formulas fm).  (* wtFormulae is defined *)
```

Uma configuração válida deve respeitar as restrições de um FM em termos de fórmulas. Porém, antes disso, precisamos checar se as fórmulas são bem tipadas. Para isso, basta verificarmos se todos os nomes de features que estejam nessa fórmula, estão presente no conjunto de nomes de features do FM em questão. Isso é realizado pela função recursiva `wt`. O decremento, para `wt`, é realizado sobre o segundo argumento. Caso `f` apresente `AND_FORMULA f1 f2` ou `IMPLIES_FORMULA f1 f2`, ele irá fazer a checagem para `f1` e também para `f2`. Se for o caso de `f` apresentar `NOT_FORMULA f1`, a recursão será aplicada sobre `f1`. Se chegarmos em `NAME_FORMULA n1`, verificamos se esse `n1` está contido no conjunto de features de um FM. Se este for o caso, retorna True, caso contrário retorna False. Por último, `TRUE_FORMULA` e `FALSE_FORMULA` serão ambos True, uma vez que são trivialmente construtores de fórmulas bem tipadas.

A garantia das fórmulas serem bem tipadas, deve valer para todas as fórmulas do conjunto de fórmula do FM. Dessa forma, precisamos de uma função que verifique se, para cada fórmula de um dado FM, `wt` retorna True. Essa função é `wtFormulae` com o auxílio de `wtFormulaeAux`, que usa casamento de padrão para percorrer todos os elementos do conjunto de fórmulas de um FM. `wtFormulae` só retorna True, se `wtFormulaeAux` for True para todas as fórmulas do FM.

```
Fixpoint satisfies (f: Formula) ( c : Configuration) : Prop :=
 match f with
    | TRUE_FORMULA => True
    | FALSE_FORMULA => False
    | NAME_FORMULA n => set_In n c
    | NOT_FORMULA f1 => ~ (satisfies f1 c)
    | AND_FORMULA f1 f2 => (satisfies f1 c) /\ (satisfies f2 c)
    | IMPLIES_FORMULA f1 f2 => (satisfies f1 c) -> (satisfies f2 c)
 end.  (* satisfies is defined
              satisfies is recursively defined (decreasing on 1st argument)*)
```

A última função recursiva de `FormulaTheory` é a função `satisfies`, que indica se uma configuração satisfaz uma fórmula. As regras tradicionais de lógica proposicional se aplicam aqui. Por exemplo, uma configuração satisfaz a fórmula de conjunção p ∧ q se satisfazer p e q. Adicionalmente, e mais importante, uma configuração c satisfaz a fórmula de nome de features n se n é um valor em c.

3.1.5.1 Lemas

As últimas declarações que compõem o módulo `FormulaTheory` são as dos lemas `formNames`, `formNames2`, `wtFormSameFeature`, `satifies1` e `satisfies2`. Todos foram devidamente provados, alguns precisaram da declaração de outros lemas que auxiliassem em suas provas. Esses lemas auxiliares foram omitidos aqui, mas todos se encontram disponíveis no repositório[1] desse trabalho.

```
Lemma formNames : forall (fm : FM) (f : Formula),
    (wt (names_ fm) f) -> ( forall (n : Name),
        set_In n (names f) -> set_In n (names_ fm)).

Lemma formNames2 : forall (fm : FM) (f : Formula) (n: Name) , b
   (wt (names_ fm) f) /\
      ( ~ (set_In n (names_ fm)))
        -> ( ~ (set_In n (names f))).

Lemma wtFormSameFeature : forall (abs : FM) (con : FM), (names_ abs = names_ con
  /\ (wfTree abs)
     /\ (wfTree con)
       -> forall (f : Formula), (wt (names_ abs) f)
          ->  (wt (names_ con) f)).
```

Os primeiros lemas que definimos foram `formNames` e `formNames2`. Ambos garantem que uma fórmula bem tipada só irá conter nomes de features válidos em um dado FM. Estamos garantindo que, se `wt (names_ fm) f` for True, então pra qualquer nome de feature que esteja contido nos nomes de uma determinada fórmula `f`, este nome também estará contido no conjunto de nomes de um FM. Já `formNames2` afirma que se `wt (names_ fm) f` for True, se um nome de feature não estiver contido no conjunto de nomes de features de um FM, também não estará contido no conjunto de nomes da fórmula `f`, umas vez que `f` só possui nomes que estejam contidos no conjunto de nomes de features deste FM.

Um outro lema é `wtFormSameFeature`, que indica que se o conjunto de features de um FM `abs` e de um FM `con` forem as mesmas, então se uma determinada fórmula for bem tipada com relação as features de `abs`, será também para as features de `con`.

Note que esses 3 primeiros lemas estão relacionados com as implicações de uma fórmula ser bem tipada, fazendo o uso da função `wt`. Abaixo, apresentamos os lemas que fazem referência a função `satisfies`.

```
Lemma satisfies1 : forall (f: Formula) (c : Configuration) (n : Name),
 ~(set_In n (names f)) -> satisfies f c = satisfies f (set_add name_dec n c).
```

```
Lemma satisfies2 : forall (f: Formula) (c : Configuration) (n : Name),
  ~ (set_In n (names f))
    -> satisfies f c = satisfies f (set_remove name_dec n c).
```

O lema `satisfies1` garante que se um dado nome de feature `n` não está presente em uma determinada fórmula, o fato de adicionarmos esse `n` à uma configuração `c` não altera o resultado de `satisfies f c`. Já `satisfies2` garante que se um dado nome de feature `n` não está presente em uma determinada fórmula, ao removermos esse `n` de uma configuração, o resultado de `satisfies f c` permanece o mesmo.

3.1.5.2 Prova

Separamos o lema `formNames` para apresentar o passo-a-passo da construção de sua prova. As provas dos demais lemas estão no apêndice deste trabalho. Lembrando que a meta inicial é:

```
∀ (fm : FM) (f : Formula),
wt (names_ fm) f ->
∀ n : Name, set_In n (names f) -> set_In n (names_ fm)
```

Como `Formula` trata-se de um tipo indutivo previamente definido, usamos o método de prova por indução para esse caso. Para usar indução sobre `f`, usamos a seguinte tática em `f`:

```
induction f.
```

O resultado da aplicação dessa tática exige, agora, que apliquemos nossos esforços para provar 6 metas menores, ou submetas, que derivam do princípio de indução, definido automaticamente por Coq na definição do tipo indutivo Formula, já visto em 3.1.2. Essas submetas são:

```
_____________________________________(1/6)
wt (names_ fm) TRUE_FORMULA ->
∀ n : Name, set_In n (names TRUE_FORMULA) -> set_In n (names_ fm)
_____________________________________(2/6)
wt (names_ fm) FALSE_FORMULA ->
∀ n : Name, set_In n (names FALSE_FORMULA) -> set_In n (names_ fm)
_____________________________________(3/6)
wt (names_ fm) (NAME_FORMULA n) ->
∀ n0 : Name, set_In n0 (names (NAME_FORMULA n)) -> set_In n0 (names_ fm)
_____________________________________(4/6)
wt (names_ fm) (NOT_FORMULA f) ->
```

```
∀ n : Name, set_In n (names (NOT_FORMULA f)) -> set_In n (names_ fm)
_____________________________________(5/6)
wt (names_ fm) (AND_FORMULA f1 f2) ->
∀ n : Name, set_In n (names (AND_FORMULA f1 f2)) -> set_In n (names_ fm)
_____________________________________(6/6)
wt (names_ fm) (IMPLIES_FORMULA f1 f2) ->
∀ n : Name,
set_In n (names (IMPLIES_FORMULA f1 f2)) -> set_In n (names_ fm)
```

Com a finalidade de visualizarmos apenas a submeta atual a ser provada, damos o comando "+" seguido das demais táticas a serem aplicadas somente a essa submeta. As duas primeiras submetas são mais fáceis de provar, visto que o conjunto `names` de `TRUE_FORMULA` e `FALSE_FORMULA` é vazio, o que resulta em False nos dois casos. Sendo assim, podemos usar a tática `contradiction` aqui, uma vez que iremos ter uma hipótese contraditória (aplicando `simpl` antes de `contradiction` veremos mais claramente isso).

```
+ simpl. contradiction.
+ simpl. contradiction.
```

A próxima submeta a ser considerada é o 3/6. Para observarmos melhor nossas hipóteses e meta, para esse caso, tomamos a liberdade de usar as táticas simpl e intros, que introduz variáveis que aparecem com o quantificador universal ∀ e as premissas (lado esquerdo) de implicações.

```
+ simpl. intros.
```

Que nos resulta em:

```
fm : FM
n : Name
H : In n (names_ fm)
n0 : Name
H0 : n = n0 \/ False
_____________________________________(1/1)
set_In n0 (names_ fm)
```

Em lógica proposicional, ∀ P e Q, se Q = False, P V Q = P. Este é o caso de H0. Podemos usar a tática `intuition` para obter H0: n = n0, e assim substituir o n da hipótese H por n0, através da tática `rewrite H0 in H.` Finalmente, aplicamos a hipótese H a nossa meta, por meio de `apply H`, para concluir esse passo da prova.

```
intuition. rewrite H0 in H. apply H.
```

Com isso, restam as 3 últimas submetas a serem provadas. Para o 4/6, por `NOT_FORMULA` de uma fórmula `f` ser apenas uma chamada recursiva em `f`, uma simples aplicação das táticas `simpl` e `tauto`, resolvem-na.

```
+ simpl. tauto.
```

As submetas 5/6 e 6/6 são semelhantes em sua estrutura, portanto as táticas usadas funcionam para ambos.

```
+ simpl. intros H. destruct H. intros. apply set_union_elim in H1.
```

Usamos as táticas acima para manipular as submetas 5/6 e 6/6, introduzindo as variáveis e premissas de implicações, substituir a hipótese H por duas, por ela apresentar conjunção e simplificar H1, aplicando o lema `set_union_elim` de ListSet. O resultado será:

```
fm : FM
f1, f2 : Formula
IHf1 : wt (names_ fm) f1 ->
      ∀ n : Name, set_In n (names f1) -> set_In n (names_ fm)
IHf2 : wt (names_ fm) f2 ->
      ∀ n : Name, set_In n (names f2) -> set_In n (names_ fm)
H : wt (names_ fm) f1
H0 : wt (names_ fm) f2
n : Name
H1 : set_In n (names f1) \/
    set_In n (set_diff name_dec (names f2) (names f1))
_____________________________________(1/1)
set_In n (names_ fm)
```

Até esse ponto, vemos que H1 apresenta uma disjunção, e podemos aplicar `inversion` para que tenhamos uma hipótese que apresente apenas uma dessas condições, já que aplicamos `inversion` para descobrir outras condições necessárias para que uma hipótese seja verdadeira.

```
inversion H1.
```

E iremos obter:

```
fm : FM
f1, f2 : Formula
IHf1 : wt (names_ fm) f1 ->
      ∀  n : Name, set_In n (names f1) -> set_In n (names_ fm)
IHf2 : wt (names_ fm) f2 ->
      ∀  n : Name, set_In n (names f2) -> set_In n (names_ fm)
H : wt (names_ fm) f1
H0 : wt (names_ fm) f2
n : Name
H1 : set_In n (names f1) \/
    set_In n (set_diff name_dec (names f2) (names f1))
H2 : set_In n (names f1)
_____________________________________(1/2)
set_In n (names_ fm)
_____________________________________(2/2)
set_In n (names_ fm)
```

Temos 2 submetas iguais a serem provadas, já tendo em mente que as hipóteses indutivas IHF1 e IHF2 deverão ser utilizadas nos próximos passos. Essas submetas fazem parte de outra submeta ainda em andamento (5/6 e 6/6), portanto não usaremos o "+" para provar essas novas submetas separadamente, mas o comando "−".

```
- apply IHf1. apply H. apply H2.
```

Uma série simples de aplicações de hipóteses a submeta, incluindo IHF1, foi suficiente para provar esse primeiro passo.

```
- apply IHf2. apply H0.
```

Avançando com mais algumas aplicações de hipóteses e estando a um passo de provar todo lema, esse é o estado atual:

```
fm : FM
f1, f2 : Formula
IHf1 : wt (names_ fm) f1 ->
      ∀ n : Name, set_In n (names f1) -> set_In n (names_ fm)
IHf2 : wt (names_ fm) f2 ->
      ∀ n : Name, set_In n (names f2) -> set_In n (names_ fm)
H : wt (names_ fm) f1
H0 : wt (names_ fm) f2
```

```
n : Name
H1 : set_In n (names f1) \/
     set_In n (set_diff name_dec (names f2) (names f1))
H2 : set_In n (set_diff name_dec (names f2) (names f1))
_____________________________________(1/1)
set_In n (names f2)
```

Sabemos que, se uma determinada feature está contida na diferença dos conjuntos de features `f2` e `f1`, ela certamente estará presente no conjunto `f2`. É isso que pretendemos ter ao aplicar o lema `set_diff_elim1` em H2.

```
- apply set_diff_elim1 in H2. apply H2.
```

Portanto, teremos uma hipótese que é exatamente a nossa última submeta, resta aplicá-la e encerramos a prova desse lema.

3.1.6 Módulo FeatureModelSemantics

```
Module FeatureModelSemantics. (* Interactive Module FeatureModelSemantics started *)

Import Name FormulaTheory Form Decidability FeatureModel.

Definition wfFM (fm : FM) : Prop := (wfTree fm) /\
(wtFormulae fm).            (* wfFM is defined *)

Definition satImpConsts (fm : FM) (c : Configuration) : Prop :=
 forall n: Name, set_In n (c) -> set_In n (names_ fm).      (* satImpConsts is defined *)

Definition satExpConsts (fm : FM) (c : Configuration) : Prop :=
  forall f: Formula, set_In f (snd fm) -> (satisfies f c = True).   (* satExpConsts is defined *)
```

Começamos o módulo `FeatureModelSemantics` dando três definições de funções. A primeira delas é `wfFM`, responsável por afirmar se um FM é bem-formado. Para que seja bem-formado, deve ser um par, onde o primeiro elemento são as features e o segundo são as fórmulas. Além disso, todas as fórmulas deste FM devem apresentar apenas features que estejam presentes no primeiro elemento do par que representa o FM.

A próxima função é `satImpConsts` que recebe um FM e uma configuração e indica se todas as features que estão presentes em uma determinada configuração, também estão presentes nas features do FM. `satExpConsts` também exige um FM e uma configuração, mas indica se toda fórmula `f` que se encontra no conjunto de fórmulas deste

FM, também tem `satisfies f c` igual a True. Estas duas funções são úteis para determinarmos o conjunto de configurações válidas de um dado FM.

```
Fixpoint filter (fm:FM) (s: set Configuration) : set Configuration :=
 match s with
  | nil => nil
  | a1 :: x1 =>
      if Is_truePB ((satImpConsts fm a1) /\
          (satExpConsts fm a1)) then a1 :: filter fm x1
       else filter fm x1
 end.

Fixpoint genConf (fm : features) : set Configuration :=
 match fm with
 | nil => nil
 | x :: xs => (set_add conf_dec (set_add name_dec x (nil)) (genConf xs)) ++ (genConf xs)
 end.

Definition semantics (fm : FM) : set Configuration :=
 filter fm (genConf (names_ fm)).
```

O conjunto de todas as configurações válidas geralmente representa a semântica de um FM. Isso é útil para raciocinar sobre FMs em geral, por exemplo, para definir uma noção de equivalência entre FMs. Dois FMs são ditos equivalentes se tiverem a mesma semântica. Para isso, a função `semantics` deve ser capaz de gerar todas essas configurações válidas de um dado FM. Tendo este objetivo em mente, criamos a função `genConf` que será responsável por gerar o conjunto das partes das features do FM, ou seja, gera todas as configurações possíveis a partir de um dado conjunto de features, sem levar em conta as restrições do FM. Por fim, podemos usar a função `filter` para permanecer apenas com as configurações que respeitam `satImpConsts` e `satExpConsts`.

As propriedades de equivalência e refinamento justificam a evolução segura e gradual dos FMs. Em um refinamento de LPS, a LPS resultante deve ser capaz de gerar produtos que correspondem comportamentalmente aos produtos das LPS originais. Assim, os usuários de um produto original não observam diferenças comportamentais ao usar as mesmas funcionalidades do produto correspondente na nova LPS. Para o caso de FMs, o refinamento de um FM estabelece que qualquer configuração no FM original (`abs`) deve ter uma correspondente no FM modificado (`con`). É esta ideia que as funções `refines` e `refines2` capturam. Em `refines`, exigimos que exatamente a mesma configuração esteja presente em ambos. Já em `refines2`, usamos uma definição mais geral, que não exige a mesma configuração exatamente, e permitiria renomear nomes de features, por exemplo.

```
Definition refines (abs : FM) (con : FM) : Prop :=
```

```
 if andb (Is_truePB (wfFM abs)) (Is_truePB(wfFM con)) then
     forall (conf : Configuration), set_In conf (semantics abs)
       -> set_In conf (semantics con)
 else False.

Definition refines2 (abs : FM) (con : FM) : Prop :=
  forall (conf1 : Configuration), set_In conf1 (semantics abs) ->
    exists (conf2 : Configuration),  set_In conf2 (semantics con).
```

Por fim, `wtFormRefinement` garante que um refinamento é bem tipado e `notMember` garante que um FM bem formado não apresenta configurações inválidas.

```
Lemma wtFormRefinement : forall (abs : FM) (con : FM), forall (name : Name),
   set_In name (fst abs) -> set_In name (fst con) /\ (wfTree abs) /\
     (wfTree con)
        -> (forall (f : Formula), (wt (fst abs) f)
             -> (wt (fst con) f)).

Theorem notMember : forall (fm : FM), wfFM fm = True
  -> ( forall (opt : Name),
     ~(set_In opt ( fst fm))
        -> (forall (conf : Configuration),  set_In conf (semantics fm)
          -> ~ (set_In opt (conf)))).
```

3.2 Maps

As teorias de AM e CK estão em andamento e as omitiremos aqui. No entanto, para o AM, precisamos de uma estrutura Map que permita o mapeamento de um nome de feature a um artefato real,pelo que optamos por formalizar uma teoria que trate dessa estrutura. Por concisão, omitimos algumas declarações de nossa especificação.

```
Module Maps.
...

 Variable S T: Type.

 Definition pair    := prod S T.
 Definition map_  := list pair.
 ...
 Fixpoint isMappable (s: map_) (l: S) (r: T)
  (Seq_dec : forall x y: S, {x = y} + {x <> y})
   (Teq_dec : forall x y: T, {x = y} + {x <> y}): Prop :=
    match s with
```

```
    | nil => False
    | p :: ps => if Is_truePB (fst p = l) then
                   if Is_truePB (snd p = r) then True
                   else (isMappable ps l r Seq_dec Teq_dec)
                 else (isMappable ps l r Seq_dec Teq_dec)
  end.
```

O tipo `pair`, em `Maps`, é definido como o produto dos tipos não interpretados `S` e `T`, onde `S` representa uma chave e `T` o valor associado a essa chave. Map é dado como uma lista desses pares. `isMappable` é a função recursiva que indica se existe um mapeamento de um `S` e `T` fornecidos, em um dado map `s`.

```
Fixpoint maps (defaulT: T) s ls: set T :=
  match ls with
    | nil => nil
    | x :: xs => if Is_truePB (existsT s x Seq_dec)
                 then set_add Teq_dec (option_elim defaulT (getT s x)) (maps defaulT s xs)
                 else (maps defaulT s xs)
  end.
```

A função `maps` fornece o conjunto de valores `T` de um map fornecido, dado um conjunto de chaves. O valor `defaulT` servirá para o retorno de `option_elim`, quando não há um valor `T` para alguma chave do conjunto de chaves fornecido.

```
Fixpoint dom (m: map_) : set S :=
  match m with
    | nil => nil
    | p :: ps => set_add Seq_dec (fst p) (dom ps)
  end.

Fixpoint img (m: map_) : set T :=
  match m with
    | nil => nil
    | p :: ps => set_add Teq_dec (snd p) (img ps)
  end.
```

`dom` é a função que retorna o conjunto de chaves de um map - o domínio - , enquanto *img* fornece o conjunto de valores de um map - a imagem.
Lemas também foram definidos a fim de atender as restrições do mapeamento de assets. Alguns deles foram:

```
Lemma inDom :
```

```
  forall s l r,
   isMappable s l r Seq_dec Teq_dec -> set_In l (dom s).
```

O lema `inDom` indica que se há um mapeamento de uma chave `l` para um valor `r` qualquer, então o domínio desse map contém `l`.

```
Lemma uniqueUnion:
  forall m1 m2, (unique m1) /\ (unique m2) ->
   (forall l,
    set_In l (dom m1) -> ~ (set_In l (dom m2)))
     -> unique(app m1 m2).
```

O lema `uniqueUnion` afirma que se `l` está no domínio de um map *`m1`* implica em *`l`* não está no domínio de `m2`, então o mapeamento da junção desses dois maps é único.

```
Lemma unionMap:
  forall s ls1 ls2 (defaulT: T), unique s ->
   maps defaulT s (set_union Seq_dec ls1 ls2)
    = set_union Teq_dec (maps defaulT s ls1) (maps defaulT s ls2).
```

O map da união dos conjuntos de chaves `ls1` e `ls2` equivale à união dos maps desses conjuntos.

```
Lemma mapUnion:
  forall s (ls1 ls2: set S) (r: T) (defaulT: T), (unique s) ->
   set_In r (maps defaulT s (set_union Seq_dec ls1 ls2)) ->
    set_In r (maps defaulT s ls1) \/ set_In r (maps defaulT s ls2).
```

Ainda raciocinando sobre união de conjuntos, se um `r` pertence ao map da união dos conjuntos de chaves `ls1` e `ls2`, logo, `r` pertence ao maps de `ls1` ou de `ls2`.

3.3 Relato

Durante o processo de implementação, a princípio, procuramos uma biblioteca que desse suporte a construção de conjuntos em Coq, visto que, diferentemente do PVS, a biblioteca Prelude não nos proporciona isso. Chegamos a utilizar outras bibliotecas, mas optamos por refazer a formalização com ListSet por apresentar mais definições que apoiam este trabalho. Apesar disso, comparado ao PVS, ainda encontramos algumas dificuldades, como é o caso da construção da função *`semantics`*. Nela, precisamos encontrar uma alternativa para obter

o conjunto de todas configurações válidas, dada um FM. Além disso, algumas provas em PVS eram simplificadas com o uso da estratégia “grind”, o que não foi possível em Coq.

Estes problemas surgiram porque nos esforçamos em fazer um mapeamento direto do que foi formalizado em PVS para Coq. Dado isso, pensamos em reestruturar as teorias, adaptando as mesmas às características dessa última.

Coq, porém, apresentou facilidades em relação a formalização feita em PVS. Uma delas é a construção das funções recursivas. PVS permite uma forma restrita de definição recursiva, exigindo provas denominadas *Condições de Correção de Tipo* (*TCCs* em inglês), como a garantia de que a função termine. Para garantir isso, as funções recursivas devem ser especificadas com uma função auxiliar como medida, que é a expressão que segue a palavra-chave `MEASURE` e gera o TCC de terminação.

Apesar da sua popularidade, não é trivial encontrar uma documentação adequada sobre Coq. Ainda que existam comunidades que dão suporte aos desenvolvedores de Coq como o coq-club, além de problemas solucionados e esclarecidos no StackOverflow e no Theoretical Computer Science SE, ainda tivemos dificuldades em esclarecer algumas questões durante esse desenvolvimento. Temos conhecimento do The Coq Consortium, uma outra comunidade, que presta uma assistência maior aos membros assinantes como acesso à lista de discussão dedicada, suporte premium para bugs, entre outros, que podem dar um suporte maior aos usuários de Coq.

Capítulo 4

Conclusão

Nesse trabalho, usamos a ferramenta Coq para descrever a formalização de um subconjunto da teoria de refinamento de linhas de produtos de software. Essa formalização já havia sido realizada em PVS. Para conseguir isso, foi necessário (i) compreensão de LPS, seus benefícios e desafios, (ii) familiarização com a teoria de linha de produtos de software, (iii) conscientização da importância e benefícios da formalização matemática em contexto de software, (iv) ter conhecimento de um conjunto particular de fatos matemáticos e como aplicá-lo, (v) familiarização com o assistente de prova Coq e (vi) algum conhecimento de PVS.

Os pontos `i-iv` se deram, principalmente, através de revisão da literatura. Parte desses materiais foram usados aqui como referências bibliográficas. Para `v`, além de algumas leituras recomendadas, o volume 1 da série de livros *Software Foundations* foi usado e vários de seus exercícios foram resolvidos, o que proporcionou a base em Gallina e Vernacular para dar início ao trabalho. Como estamos portando a teoria já formalizada em PVS para Coq, o ponto `vi` é requerido. Nesse caso, as dúvidas quanto essa linguagem foram esclarecidas com o próprio responsável pela formalização em PVS.

O resultado foi um conjunto de definições e provas da teoria de Feature Models, além das teorias de Asset Mapping e Configuration Knowledge que estão em andamento. Isso justifica trabalhos [Borba et al., 2012][Neves et al., 2012][Teixeira et al., 2015] que proporcionam evolução segura e gradual de LPS, que pode ser realizada em apenas um dos elementos em um determinado cenário de evolução, graças à garantia fornecida pelos teoremas [Borba et al., 2012].

Como trabalhos futuros, pretendemos finalizar a formalização das teorias de AM e CK, além da formalização das demais teorias relacionadas ao refinamento de LPS. Uma outra tarefa a ser realizada quanto a formalização aqui apresentada, trata da simplificação das provas realizadas, dado que não foi o nosso foco neste trabalho.

Referências

[1] The core Flight Software System (cFS) has reduced the costly and time-consuming process of developing software for spaceflight missions. Disponível em: <http://coreflightsystem.org/about-the-technology-why-cfs/>. Acesso em: 20 abr. 2018.

[2] Reference Manual of Coq. Disponível em <https://coq.inria.fr/distrib/current/refman/> . Acesso em: 23 jun. 2018.

[3] Coq Tactics Cheatsheet: <https://www.cs.cornell.edu/courses/cs3110/2017fa/a5/coq-tactics-cheatsheet.html>.Acesso em: 29 jun. 2018.

[Apel et al., 2013] Apel, S., Batory, D., Kastner, C., Saake, G. Feature-Oriented Software Product Lines: Concepts and Implementation. 1. ed. [S.l.]: Springer, 2013. 320 p.

[Barthe et al., 2009] Gilles Barthe, Benjamin Grègoire, and Santiago Zanella Bèguelin. Formal certification of code-based cryptographic proofs. In 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pages 90–101. ACM, 2009. See also: CertiCrypt http://www.msr-inria.inria.fr/projects/ sec/certicrypt.

[Bertot et al., 2010] Yves Bertot , Pierre Castran, Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, Springer Publishing Company, Incorporated, 2010.

[Borba et al., 2012] Borba, P., Teixeira, L., and Gheyi, R. (2012). A theory of software product line refinement. Theor. Comput. Sci., 455:2–30.

[Borba et al., 2004] P. Borba, A. Sampaio, A. Cavalcanti, M. Cornélio, Algebraic reasoning for object-oriented programming, Science of Computer Programming 52 (2004) 53–100.

[Chetali et al, 2008] Boutheina Chetali and Quang-Huy Nguyen. About the world-first smart card certificate with EAL7 formal assurances. Slides 9th ICCC, Jeju, Korea, September 2008. www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf.

[Chipala, 2013] Adam Chlipala, Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant, The MIT Press, 2013.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. (2000). Generative programming: methods, tools, and applications. Addison-Wesley.

[Delaware et al., 2009] Benjamin Delaware , William R. Cook , Don Batory, Fitting the pieces together: a machine-checked model of safe composition, Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, August 24-28, 2009, Amsterdam, The Netherlands.

[Gheyi et al., 2005] R. Gheyi, T. Massoni, P. Borba, An abstract equivalence notion for object models, Electronic Notes in Theoretical Computer Science 130 (2005) 3–21.

[Gonthier et al., 2011] Georges Gonthier. Advances in the formalization of the odd order theorem. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, Interactive Theorem Proving - ITP 2011, volume 6898 of Lecture Notes in Computer Science, page 2. Springer, 2011.

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

[Linden et al., 2005] F. J. van der Linden, K. Pohl, G. Bockle, and. Software Product Line Engineering. Springer, 2005.

[Machay et al., 2012] Julian Mackay , Hannes Mehnert , Alex Potanin , Lindsay Groves, Nicholas Cameron, Encoding Featherweight Java with assignment and immutability using the Coq proof assistant, Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, p.11-19, June 12-12, 2012, Beijing, China.

[Mcgregor et al., 2010] Mcgregor, J., Muthig, D., Yoshimura, K., Jensen, P., (2010). Guest Editors' Introduction: Successful Software Product Line Practices. Software, IEEE. 27. 16 - 21. 10.1109/MS.2010.74.

[Mohring et al., 2011] Christine Paulin-Mohring (2011) Introduction to the Coq Proof-Assistant for Practical Software Verification, LASER Summer School.

[Naur and Randell, 1969] Naur, P., Randell, B (1969). Software engineering. Scientific Affairs Div., NATO.

[Neves et al., 2012] Laís Neves , Leopoldo Teixeira , Demóstenes Sena , Vander Alves , Uirá Kulezsa , Paulo Borba, Investigating the safe evolution of software product lines, ACM SIGPLAN Notices, v.47 n.3, March 2012

[Pierce et al., 2018] Pierce,B., Amorim, A., Casinghino, C .Software Foundation Volume 1 - Logical Foundation. version 5.5 2018. <https://softwarefoundations.cis.upenn.edu/lf-current/> Acesso em: 23 jun. 2018.

[Rosen, 2009] Rosen, K. Matemática Discreta e suas Aplicações,Grupo A Educação, 2009.

[Sampaio, 2017] Sampaio, G. Partially safe evolution of software product lines. 2017. Dissertação de Mestrado (Pós-graduação em Ciência da Computação)- Universidade Federal de Pernambuco, Recife, 2017.

[Sommerville, 2010] Sommerville, I. (2010). Software Engineering. Addison-Wesley, Harlow, England, 9 edition.

[Teixeira et al., 2015] Teixeira, L., Alves, V., Borba, P. and Gheyi, R. A product line of theories for reasoning about safe evolution of product lines. International Systems and Software Product Line Conference, 2015.

[Teixeira, 2014] Teixeira, L. Safe evolution of software product lines and sets of product lines. 2014. Tese de Doutorado (Pós-graduação em Ciência da Computação)- Universidade Federal de Pernambuco, Recife, 2014.

[Teixeira, 2010] Teixeira, L. Verification and refactoring of configuration knowledge for software product lines. 2010. 134 p. Dissertação de Mestrado (Pós-graduação em Ciência da Computação)- Universidade Federal de Pernambuco, Recife, 2010.

[Van der Linden et al., 2007] Van der Linden, F., Schmid, K., and Rommes, E. (2007). Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer.

Apêndice A : Provas dos lemas do módulo FormulaTheory

```
Lemma formNames2 : forall (fm : FM) (f : Formula) (n: Name) ,(wt (names_ fm) f) /\
 (~(set_In n (names_ fm))) -> (~(set_In n (names f))).
Proof.
 unfold not.
 induction f.
  + simpl. tauto.
  + simpl. tauto.
  + simpl. intros. intuition. rewrite H in H1. apply H2. apply H1.
  + simpl. tauto.
  + simpl. intros. destruct H. destruct H. apply set_union_elim in H0. inversion H0.
    - apply (IHf1 n). intuition. apply H3.
    - apply set_diff_elim1 in H3. apply (IHf2 n). intuition. apply H3.
  + simpl. intros. destruct H. destruct H. apply set_union_elim in H0. inversion H0.
    - apply (IHf1 n). intuition. apply H3.
    - apply set_diff_elim1 in H3. apply (IHf2 n). intuition. apply H3.
Qed.
```

```
Lemma set_union_elim_not :
  forall (a:Name) (x y:set Name),
   ~(set_In a (set_union name_dec x y)) ->
     ~(set_In a x) /\ ~(set_In a y).
Proof.
 intros. split.
  + intuition. apply H. apply set_union_intro1. apply H0.
  + intuition. apply H. apply set_union_intro2. apply H0.
Qed.
```

```
Lemma set_union_elim_not2 :
  forall (a:Name) (x y:set Name),
   ~(set_In a x) /\ ~(set_In a y) ->
    ~(set_In a (set_union name_dec x y)).
Proof.
 intros. destruct H.
 intuition. apply H.
 apply set_union_elim in H1.
 generalize H1. tauto.
Qed.
```

```
Lemma satisfies1 : forall (f: Formula) (c : Configuration) (n : Name),
 ~(set_In n (names f)) -> satisfies f c = satisfies f (set_add name_dec n c).
Proof.
induction f.
 + simpl. intros. reflexivity.
 + simpl. intros. reflexivity.
 + simpl. intros. intuition. apply H1 in H0.
   - contradiction.
   - rewrite n. rewrite n0. reflexivity.
 + simpl. intros. apply not_compat. apply (IHf c). apply H.
 + simpl. intros. apply set_union_elim_not in H. destruct H as [H1 H2].
   specialize (IHf1 c n). specialize (IHf2 c n).
   apply set_diff_elim_not in H2. inversion H2.
   -  apply IHf1 in H1. apply IHf2 in H. rewrite H1.
     rewrite H. reflexivity.
   - contradiction.
 + simpl. intros. apply set_union_elim_not in H. destruct H as [H1 H2].
   specialize (IHf1 c n). specialize (IHf2 c n).
   apply set_diff_elim_not in H2. inversion H2.
   -  apply IHf1 in H1. apply IHf2 in H. rewrite H1.
     rewrite H. reflexivity.
   - contradiction.
Qed.
```

```
Lemma satisfies2 : forall (f: Formula) (c : Configuration) (n : Name),
 ~(set_In n (names f)) -> satisfies f c = satisfies f (set_remove name_dec n c).
Proof.
induction f.
 + simpl. intros. reflexivity.
 + simpl. intros. reflexivity.
 + simpl. intros. intuition. apply H1 in H0.
   - contradiction.
   - rewrite n. rewrite n0. reflexivity.
 + simpl. intros. apply not_compat. apply (IHf c). apply H.
 + simpl. intros. apply set_union_elim_not in H. destruct H as [H1 H2].
   specialize (IHf1 c n). specialize (IHf2 c n).
   apply set_diff_elim_not in H2. inversion H2.
   -  apply IHf1 in H1. apply IHf2 in H. rewrite H1.
     rewrite H. reflexivity.
   - contradiction.
 + simpl. intros. apply set_union_elim_not in H. destruct H as [H1 H2].
   specialize (IHf1 c n). specialize (IHf2 c n).
```

```
  apply set_diff_elim_not in H2. inversion H2.
  - apply IHf1 in H1. apply IHf2 in H. rewrite H1.
    rewrite H. reflexivity.
  - contradiction.
Qed.
```

```
Lemma wtFormSameFeature : forall (abs : FM) (con : FM), (names_ abs = names_ con
 /\ (wfTree abs) /\ (wfTree con) -> forall (f : Formula), (wt (names_ abs) f)
 ->  (wt (names_ con) f)).
Proof.
 intros.
 destruct H as [equals_abs_con wf_abs_con].
 destruct wf_abs_con as [wf_abs wf_con].
 induction f.
  + rewrite equals_abs_con in H0. apply H0.
  + rewrite equals_abs_con in H0. apply H0.
  + simpl. simpl in H0. rewrite equals_abs_con in H0. apply H0.
  + apply IHf. apply H0.
  + rewrite equals_abs_con in H0. apply H0.
  + rewrite equals_abs_con in H0. apply H0.
Qed.
```

Apêndice B : Provas dos lemas do módulo FeatureModelSemantics

```
Lemma wtFormRefinement : forall (abs : FM) (con : FM), forall (name : Name),
set_In name (fst abs) -> set_In name (fst con) /\ (wfTree abs) /\
(wfTree con) -> (forall (f : Formula), (wt (fst abs) f) -> (wt (fst con) f)).
Proof.
induction f.
 + simpl. tauto.
 + simpl. tauto.
 + destruct H0. intros. simpl. rewrite name in H0.
   rewrite n. apply H0.
 + simpl. tauto.
 + simpl. intros. destruct H1. split.
   - apply IHf1. apply H1.
   - apply IHf2. apply H2.
 + simpl. intros. split.
   - apply IHf1. destruct H1. apply H1.
   - apply IHf2. destruct H1. apply H2.
Qed.
```

```
Theorem notMember : forall (fm : FM), wfFM fm = True -> ( forall (opt : Name),
 ~(set_In opt ( fst fm)) -> (forall (conf : Configuration),
  set_In conf (semantics fm) -> ~ (set_In opt (conf)))).
  Proof.
  intros.
  unfold not in H.
  unfold not.
  destruct opt.
  intro H2.
  destruct conf in H1.
  + destruct fm. destruct f.
    - simpl in H1. apply H1.
    - destruct n. destruct f;
      simpl in H; apply H0; left; reflexivity.
  + destruct fm. destruct f.
    - simpl in H1. apply H1.
    - simpl in H. apply H0. left. rewrite n0. reflexivity.
Qed.
```