Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Lidando com a legibilidade de provas em Isabelle

Rafael M. Rodrigues

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Cláudia Nalon

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Lidando com a legibilidade de provas em Isabelle

Rafael M. Rodrigues

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Cláudia Nalon (Orientadora)
CIC/UnB

Prof. Dr. Mauricio Ayala-Rincón
Universidade de Brasília

Dr.a Ariane Alves de Almeida
Cargill,S2C2

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 25 de Julho de 2023

Dedicatória

Este trabalho é dedicado a Eduardo Marques Monteiro, que tanto me apoiou durante a minha graduação e o desenvolvimento deste trabalho mas, tragicamente, não pode ver sua conclusão.

Agradecimentos

Agradeço à minha orientadora, a professora doutora Cláudia Nalon, pelo seu apoio, ensinamentos e companheirismo durante o desenvolvimento desse trabalho.

Agradeço, também, aos meus pais, Sylvia e Antonio, que me apoiaram incondicionalmente durante tantos desafios.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O raciocínio automatizado é a área de inteligência artificial que busca criar ferramentas capazes de simular o raciocínio humano. Estas ferramentas são bastante úteis no domínio da prova automática de teoremas. No entanto, um fator a ser considerado é a possível dificuldade que provas automaticamente encontradas podem apresentar quanto à sua legibilidade.

A legibilidade de provas é um tema de grande importância, pois está diretamente ligada com a divulgação de resultados científicos. Neste trabalho, o problema central abordado é o aprimoramento da legibilidade de provas produzidas por meio da interação com assistentes de provas, com um foco em provas produzidas com o auxílio de ferramentas de automação do raciocínio, utilizando técnicas de melhoria da legibilidade propostas por Zammit [1] e Lamport [2].

Como base teórica, este trabalho utiliza a lógica de predicados de ordem superior e a dedução natural. Este alicerce teórico é usado por meio da ferramenta Isabelle, um assistente de provas que utiliza a lógica de ordem superior, e que neste projeto serviu como ferramenta principal.

Para ilustrar os conceitos abordados, este projeto apresenta um estudo de caso onde se faz a especificação e verificação formal de um sistema de comando e controle. Esta verificação é feita por meio da prova de alguns teoremas que demonstram propriedades da especificação. O central do trabalho consiste na reescrita dessas provas, inicialmente feitas com um alto grau de automação, para provas mais legíveis para humanos, utilizando técnicas de aprimoramento da legibilidade extraídas da literatura a respeito do tema.

Palavras-chave: Métodos formais, comando e controle, legibilidade de provas

Abstract

Automated reasoning is the area of artificial intelligence that seeks to create tools capable of simulating human reasoning. These tools are very useful in the domain of automated theorem proving. However, one factor to be considered is the possible difficulty that automatically produced proofs may present regarding their readability.

The readability of proofs is a topic of great importance, as it is directly linked to the dissemination of scientific results. In this work, the central problem addressed is the improvement of the readability of proofs produced through the interaction with proof assistants, with a focus on proofs produced with the aid of automated reasoning tools using techniques for the improvement of proof legibility proposed by Zammit [1] and Lamport [2]..

This work uses higher-order predicate logic and natural deduction as it's theoretic foundation. These concepts are used in practice with Isabelle, a proof assistant that uses higher-order logic.

To illustrate the concepts covered, this project presents a case study regarding the formal specification and verification of a command and control system. This verification is done through the proof of several theorems that demonstrate properties of the specification. The central part of the work consists in rewriting these proofs, initially done with a high degree of of automation, to more human-readable proofs using techniques for improving the legibility of proofs taken from the literature available on this topic.

Keywords: Formal methods, command and crontrol, proof legibility

Sumário

Capítulo 1

Introdução

O raciocínio automatizado é a área da inteligência artificial que busca produzir artefatos computacionais capazes de raciocinar de maneira parcial ou totalmente automática. Por raciocínio, nesse contexto, entende-se o processo pelo qual partimos de algum conjunto de fatos expressos em uma linguagem formal e, por meio de procedimentos de manipulação simbólica e reescrita das sentenças dessa linguagem, obtêm-se novos fatos.

Neste trabalho será explorado o processo de produção de provas auxiliado por sistemas computacionais. Mais especificamente, será analisada a interação entre o usuário deste tipo de sistema e as provas neles produzidas.

A produção de provas matemáticas é de suma importância para o avanço das ciências. Por conta disso, é também importante que o processo de divulgação científica seja eficaz e que o entendimento destes resultados seja também considerado com cuidado.

Portanto, este trabalho se propõe a investigar como provas produzidas com o auxílio de artefatos computacionais podem ter sua legibilidade melhorada por meio de boas práticas extraídas da literatural acerca do assunto., de forma que os resultados obtidos possam ser lidos e compreendidos mais facilmente por seus leitores.

1.1 Raciocínio automatizado

O raciocínio automatizado tem como fundação operacional a lógica formal e suas muitas variações. Estas são empregadas como as linguagens formais nas quais fatos são expressos como sentenças lógicas. A partir disto, utilizando seus respectivos cálculos dedutivos, podemos computar novos fatos a partir da manipulação destas sentenças.

Pode-se traçar a história do raciocínio automatizado até seu aparente surgimento em 1956, com a criação de um programa intitulado *Logic Theorist*, produzido por Allen Newell, Herbert A. Simon e Cliff Shaw [3, Página 36]. O programa em questão foi capaz

de provar 38 dos primeiros 52 teoremas postulados por Bertrand Russel e Alfred North Whitehead nos *Principia Mathematica* [4, Página 167].

1.2 Aplicações do raciocínio automatizado

A prova automática de teoremas consiste em desenvolver sistemas computacionais, chamados provadores automáticos, que mostram que alguma sentença pode ser deduzida de um conjunto de axiomas e hipóteses, representados por outras sentenças. Provadores automáticos são utilizados em uma variedade de aplicações. Além das aplicações óbvias à matemática, a prova automática de teoremas também pode ser usada por projetistas de *hardware*, por exemplo, para mostrar que a especificação lógica da implementação de um circuito possui propriedades condizentes com a especificação lógica do mesmo. O mesmo pode se feito, também, com sistemas de *software* e suas respectivas especificações.

Ferramentas de prova automática podem utilizar uma grande variedade de linguagens formais, de forma que a especificação do sistema e suas propriedades pode tomar diversas formas. Estas linguagens permitem a descrição precisa e não ambígua da informação necessária ao trabalho do desenvolvedor, de tal maneira que este se exime de algumas das dificuldades e armadilhas que surgem na solução do problema em decorrência do uso de linguagens informais. É interessante notar que, muitas vezes, o próprio fato de se expressar, ou tentar expressar, fatos sobre um sistema em uma linguagem não ambígua pode proporcionar ao redator um maior entendimento do problema, uma vez que tal atividade o força a pensar de forma sistemática.

As provas em si, como produzidas por um provador automático, descrevem como e porquê uma determinada conjectura pode ser derivada do conjunto de axiomas e hipóteses, de tal maneira que não haja espaço para dúvidas quanto a sua validade. Por vezes, uma prova assim produzida representa não apenas um argumento à veracidade de uma sentença, mas também uma descrição do processo pelo qual o problema é resolvido, no caso de provas construtivas.

Desta maneira, fica claro que provadores automáticos são sistemas computacionais bastante poderosos, capazes de resolver problemas de grande complexidade e importância. Por conta desta capacidade, seu uso pode ser, por vezes, difícil, sendo relegado a especialistas em algum domínio do saber cujas necessidades e desafios produzam motivos para seu uso.

1.2.1 Aplicações matemáticas

Já há diversas aplicações maduras da prova automatizada de teoremas. No ramo da matemática, por exemplo, já obteve-se provas na linguagem proposta por Russel e

Whitehead, como foi mencionado no início desta seção. Foi produzida também uma prova para o teorema das quatro cores, por George Gonthier, utilizando o sistema Coq [5] como artifício prático para a prova semiautomática [6, páginas 1382 à 1393].

1.2.2 Verificação de *software*

A prova automatizada de teoremas também encontra adoção prática na verificação de *software*. O verificador interativo Karlsruhe, desenvolvido como plataforma experimental para a verificação de *software* na Universidade de Karlsruhe, é um exemplo interessante deste caso de uso. Este provador foi utilizado para, dentre outras coisas, auxiliar no desenvolvimento de um compilador de Prolog [7] para WAM (*i.e. Warren Abstract Machine*), uma máquina abstrata.

1.2.3 Verificação de *hardware*

O ramo da verificação de *hardware* é, talvez, onde há a adoção industrial da prova automática de teoremas em maior escala. O sistema ACL2 [8], por exemplo, foi utilizado na produção de provas de correção da unidade de ponto flutuante do microprocessador AMD5k86, da AMD [9].

1.3 Limitações do raciocínio automatizado

Há limitações no processo de raciocínio automatizado. Uma destas limitações é a natureza indecidível do problema de busca por provas em muitas da linguagens lógicas. Para o caso da lógica proposicional, por exemplo, o problema é decidível porém co-NP-completo e, portanto, supõe-se que apenas soluções de tempo exponencial poderiam existir para provas genéricas neste contexto [10, página 56]. Para a lógica de predicados de primeira ordem, por outro lado, o primeiro teorema de incompletude de Gödel [11] aponta uma limitação mais severa. Este teorema mostra que a lógica de predicados de primeira ordem é sintaticamente incompleta, ou seja, há fórmulas verdadeiras cujas provas não podem ser encontradas, seja pelos meios tradicionais, seja pelo uso se ferramentas de raciocínio automatizado.

Segue daí que um provador automático pode entrar em um ciclo interminável de computação durante a busca por uma prova quando isto for uma instância de um problema indecidível, mesmo que a sentença seja verdadeira. Apesar desta limitação, provadores de teoremas são capazes de resolver muitos problemas difíceis, mesmo que o cálculo implementado seja incompleto.

Outra questão que surge é a legibilidade de provas. Por vezes, a maneira como um sistema de automatização de provas opera pode produzir uma cadeia dedutiva correta, mas pouco compreensível. Desta forma, é possível que um usuário não consiga ler os resultados e imediatamente seguir o raciocínio. Isso representa um enorme problema pois, frequentemente, as razões e argumentos pelos quais uma prova é construída são tão esclarecedores sobre o problema quanto o resultado da prova em si, podendo trazer elucidações sobre a natureza do fenômeno estudado que sirvam como direcionamento ao progresso do estudo ou projeto. Esta questão é o tema central do trabalho relatado nesta monografia.

1.4 Assistentes de provas

Assistentes de provas são sistemas computacionais cujo propósito é permitir a realização de atividades envolvendo raciocínio lógico sobre definições e propriedades com o auxílio de computadores. Especificamente, assistentes de provas são utilizados para auxiliar o desenvolvedor com a produção de definições, provas e teoremas por meio de linguagens computacionais especializadas em uma dinâmica de utilização interativa. Muitos assistentes de provas contam com linguagens capazes também de expressar computações, de forma que pode-se integrar aspectos computacionais em meio aos aspectos descritivos da linguagem. É comum, também que sistemas assistentes de provas contenham grandes bibliotecas de teoremas e definições que podem ser importados em uma teoria e utilizados nas provas que o usuário desenvolver [12].

Uma capacidade interessante dos assistentes de provas é que, ao passo que produzir provas de forma completamente automática em lógicas cujos cálculos são indecidíveis é computacionalmente impossível, checar a correção de provas existentes é um processo simples. Desta maneira, além de prover suporte à escrita e organização de provas, um assistente de provas pode, também, agir como um corretor, checando cada passo da prova informado pelo usuário e o alertando quanto ao surgimento de erros.

Neste trabalho o problema da legibilidade de provas será explorado em etapas. Primeiro, no Capítulo 2, será apresentadas a lógica de ordem superior e dedução natural, que constituem a infraestrutura teórica para o desenvolvimento seguinte. No Capítulo 3 será introduzido o o assistente de provas Isabelle, que servirá como ferramenta principal neste trabalho. No Capítulo 4 o problema da legibilidade de provas será apresentado em mais detalhes e será mostrada uma abordagem para a escrita de provas legíveis com base em técnicas presentes na literatura científica acerca do tema. Finalmente, no Capítulo 5, será apresentado um estudo de caso onde estes elementos são aplicados à verificação formal de um sistema de comando e controle.

Capítulo 2

Lógica

2.1 Lógica

A lógica é o estudo dos padrões de raciocínio corretos. Raciocínio pode ser entendido como a atividade que busca produzir argumentos passo a passo, onde cada argumento é composto por um conjunto de premissas, os passos intermediários e uma conclusão. Em um argumento lógico a veracidade da conclusão segue a partir da veracidade das premissas. O interesse central do estudo da lógica é justamente determinar se argumentos são válidos, ou seja, se as premissas apoiam a verdade da conclusão [13, Páginas 13 e 39].

Desta forma, a lógica constitui a infraestrutura teórica sobre a qual repousa o campo do raciocínio automatizado. O raciocínio automatizado pode então também ser entendido como o empenho em desenvolver sistemas capazes de automatizar em todo ou em parte o processo de raciocínio por meio da automação total ou parcial do processo de inferência sobre construções lógicas.

É importante constatar que lógica é composta por uma pluralidade de diferentes sistemas. Há várias linguagens, cada uma com suas peculiaridades materializadas na forma de diferentes sintaxes, semânticas e cálculos dedutivos. Cada uma com sua forma particular de escrever sentenças "bem formadas", interpretá-las e manipulá-las.

2.1.1 Lógicas de ordem superior

Apresentaremos as lógicas de ordem superior de maneira comparativa. Para apoiar a elucidação do tema, apresentaremos inicialmente a lógica de predicados de primeira ordem e, em seguida, discutiremos como as lógicas de ordem superior podem ser construídas como extensões da lógica de primeira ordem.

A lógica de predicados de segunda ordem é bastante similar à lógica de primeira ordem, porém com uma diferença crucial. Ao passo que a lógica de primeira ordem permite apenas

quantificação sobre indivíduos, a lógica de segunda ordem permite também quantificação sobre relações e propriedades [14, página 1]. Em outras palavras, ao passo que na lógica de predicados de primeira ordem o conjunto que representa o domínio de discurso pode conter apenas entidades atômicas, *i.e.* entidades que não podem ser decompostas em partes; na lógica de segunda ordem este pode conter também conjuntos, que são entidades formais compostas de partes.

Nas lógicas de ordem superior, a construção da lógica de segunda ordem é estendida para permitir quantificação sobre entidades de ordem superior, como relações de relações ou propriedades de relações.

Sintaxe

Inicialmente, definimos a sintaxe da lógica de predicados de primeira ordem.

Definição 1

O vocabulário básico da lógica de predicados de primeira ordem é construído pelos seguintes símbolos [14, página 1]:

- Constantes individuais $a, b, c, \ldots$
- Variáveis individuais $x, y, z, \ldots$
- Símbolos para relações $P, Q, \ldots$. A cada símbolo de relação é atribuído um número natural $n \geq 0$, representando sua multiplicidade. Uma relação de multiplicidade n é chamada n-ária. Um símbolo de relação de multiplicidade 1 é também chamado de símbolo de predicado.
- Símbolos para funções $f, g, h, \ldots$. A cada símbolo de função é atribuído um número natural $n \geq 0$, representando sua multiplicidade. Uma relação de multiplicidade n é chamada n-ária.
- Conectivos lógicos $\vee, \wedge, \neg, \rightarrow$ e $\leftrightarrow$.
- Quantificadores $\forall$ e $\exists$.
- Símbolo de igualdade $=$
- Sinais de pontuação $(,)$ e $[,]$

Em seguida, trabalhado sobre este vocabulário, podemos então definir as estruturas principais da sintaxe da lógica de predicados de primeira ordem, *i.e* os termos e fórmulas.

Definição 2

Os termos da lógica de predicados de primeira ordem são definidos da seguinte maneira [14, página 1]:

- Qualquer variável individual é um termo.
- Se f é uma função n-ária e $t_1, \ldots, t_n$ são termos, então $ft_1, \ldots, t_n$ é um termo.
- Nada mais além das duas construções acima é um termo.

Com a definição dos termos podemos, finalmente, definir a estrutura das fórmulas da lógica de predicados de primeira ordem.

Definição 3

As fórmulas na lógica de predicados de primeira ordem são definidas da seguinte maneira [14, página 2]:

- Um símbolo de relação n-ário seguido por n termos é uma fórmula. Este tipo de fórmula se denomina fórmula atômica.
- Qualquer expressão da forma $s = t$, onde s e t são termos é uma fórmula. Este tipo de fórmula se denomina fórmula atômica.
- Se ϕ e ψ são fórmulas, então também são fórmulas as expressões $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, e $\phi \leftrightarrow \psi$.
- Se ϕ é uma fórmula e x é uma variável individual, então tanto $\forall x\phi$ e $\exists x\phi$ são fórmulas. Nestas duas fórmulas as ocorrências da variável x são ditas ligadas. Todas as variáveis não ligadas são ditas livres. Uma fórmula onde todas as variáveis são ligadas será chamada de *sentença*.
- Nada mais além das construções acima é uma fórmula.

A partir destas definição que formam a sintaxe da lógica de primeira ordem, pode-se estendê-las para construir a lógica de predicados de de segunda ordem. Para isso, agora permite-se também a quantificação sobre relações. O vocabulário de segunda ordem é produzido a partir de uma extensão do vocabulário de primeira ordem onde se adiciona símbolos $X, Y, \ldots$, chamados de relações, ou variáveis de segunda ordem.

As fórmulas de um vocabulário de segunda ordem são obtidas pela expansão da definição das fórmulas de primeira ordem. Agora, admite-se que qualquer variável de relação de multiplicidade n seguida por n termos é uma fórmula. Admite-se também que, dadas uma variável de relação X e uma fórmula ϕ, as construções $\forall X\phi$ e $\exists X\phi$ são fórmulas da lógica de segunda ordem.

Semântica

A semântica da lógica permite atribuir significado às fórmulas. Em outras palavras, constrói-se a partir da semântica uma maneira de expressar formalmente a interpretação que as fórmulas devem ter em algum contexto.

Dado um vocabulário de primeira ordem $\mathbf{L}$, suponha que as variáveis individuais de $\mathbf{L}$ podem ser dispostas como a lista $v_0, v_1, \ldots, v_n, \ldots$, que os símbolos de relações, funções e constantes individuais são, respectivamente, dispostas como as listas $(P_i, i \in I)$, $(f_j, j \in J)$, $(c_k, k \in K)$ e que para cada $i \in I, j \in J$, as multiplicidades de P_i e f_j são, respectivamente, os números naturais $n(i)$ e $m(j)$. A semântica da lógica de predicados de primeira ordem é dada pela seguinte estrutura, chamada de uma *interpretação* de $\mathbf{L}$ [14, página 3].

$$\mathfrak{A} = (A, \{R_i : i \in I\}, \{g_j : j \in J\}, \{e_k : k \in K\})$$

Onde A, chamado de *domínio* ou *universo* de $\mathfrak{A}$, é um conjunto enumerável não vazio. O conjunto $\{R_i : i \in I\}$ representa uma coleção de relações em A onde, para cada $i \in I$, R_i é uma relação de multiplicidade $n(i)$. O conjunto $\{g_j : j \in J\}$ representa uma coleção de operações em A onde cada g_i tem multiplicidade $m(j)$. Finalmente, o conjunto $\{e_k : k \in K\}$ contém elementos de A denominados *elementos designados de* $\mathfrak{A}$. Nesta definição, chamamos os elementos R_i, g_j, e_k de *denotações* de P_i, f_j, c_k em $\mathfrak{A}$. Suponha a existência da sequência contável de elementos de A dada por $a = (a_0, a_1, \ldots)$. A tal sequência daremos o nome de *A-sequência.*

Definição 4

Para qualquer termo t, definimos a interpretação $t^{(\mathfrak{A},a)}$ *em* $(\mathfrak{A}, a)$ da seguinte forma:

- $c_k^{(\mathfrak{A},a)} = e_k$
- $v_n^{(\mathfrak{A},a)} = a_n$

- Para cada $j \in J$ e os termos $t_1, \ldots, t_{m(j)}, f_j t_1 \ldots t_{m(j)}{}^{(\mathfrak{A},a)} = g_j(t_1{}^{(\mathfrak{A},a)}, \ldots, t_{m(j)}{}^{(\mathfrak{A},a)})$

Agora, para cada número natural n e $b \in A$, nós definimos:

$$[n|b]a = (a_0, a_1, \ldots, a_{n-1}, b, a_{n+1}, \ldots)$$

Finalmente, definimos a satisfatibilidade na lógica de predicados de primeira ordem.

Definição 5

Para cada fórmula ϕ, definimos a relação a satisfaz ϕ em $\mathfrak{A}$, denotada por $\mathfrak{A} \models_a \phi$, como:

- Para os termos t e u, $\mathfrak{A} \models_a t = u \Leftrightarrow t^{(\mathfrak{A},a)} = u^{(\mathfrak{A},a)}$
- Para os termos $t_1, \ldots, t_{n(i)}$, $\mathfrak{A} \models_a P_i t_1 \ldots t_{n(i)} \Leftrightarrow R_i(t_1{}^{(\mathfrak{A},a)} \ldots, t_{n(i)}{}^{(\mathfrak{A},a)})$
- $\mathfrak{A} \models_a \neg\phi \Leftrightarrow$ não $\mathfrak{A} \models_a \phi$
- $\mathfrak{A} \models_a \phi \wedge \psi \Leftrightarrow \mathfrak{A} \models_a \phi$ e $\mathfrak{A} \models_a \psi$
- $\mathfrak{A} \models_a \phi \vee \psi \Leftrightarrow \mathfrak{A} \models_a \phi$ ou $\mathfrak{A} \models_a \psi$
- $\mathfrak{A} \models_a \phi \rightarrow \psi \Leftrightarrow$ se $\mathfrak{A} \models_a \phi$ então $\mathfrak{A} \models_a \psi$
- $\mathfrak{A} \models_a \phi \leftrightarrow \psi \Leftrightarrow \mathfrak{A} \models_a \phi$ se, e somente se $\mathfrak{A} \models_a \psi$
- $\mathfrak{A} \models_a \exists v_n \phi \Leftrightarrow$ se para algum $b \in A, \mathfrak{A} \models_{[n|b]\alpha} \phi$
- $\mathfrak{A} \models_a \forall v_n \phi \Leftrightarrow$ se para todo $b \in A, \mathfrak{A} \models_{[n|b]\alpha} \phi$

Dizemos que uma fórmula ϕ é *verdadeira* em $\mathfrak{A}$ se $\mathfrak{A} \models_a \phi$ se para toda sequência de elementos a extraídos de A e *satisfatível* em $\mathfrak{A}$ se $\mathfrak{A} \models_a \phi$ para alguma sequência de elementos a extraídos de A.

Dizemos também que, se uma sentença σ é verdadeira em $\mathfrak{A}$, $\mathfrak{A}$ é um *modelo* de σ, escrito $\mathfrak{A} \models \sigma$.

Seja $\Sigma \subseteq \mathbf{L}$. $\mathfrak{A}$ é um *modelo* de Σ, escrito $\mathfrak{A} \models \Sigma$, se cada elemento de Σ é verdadeiro em $\mathfrak{A}$. A sentença σ é *consequência lógica* de Σ, escrita $\Sigma \models \sigma$, se σ é verdadeira em todos os modelos de Σ. A sentença σ é logicamente válida se for verdadeira em todas as interpretações de $\mathbf{L}$, *i.e.* $\emptyset \models \sigma$.

Agora, dada a semântica para a lógica de primeira ordem, podemos estendê-la para obter uma definição para a semântica da lógica de segunda ordem. Para isso, partirmos

de um vocabulário $\mathbf{L}'$ que estende o vocabulário de primeira ordem $\mathbf{L}$. Suponha que, para cada $n \geq 1$, as variáveis de relação de ordem n de $\mathbf{L}'$ são enumeradas como $V_0^{(n)}, V_1^{(n)}, \ldots$. Dado um conjunto A, podemos construir a sequência de relações de A $\mathbf{R} = (R_m^{(n)} : m = 0, 1, \ldots n = 1, 2, \ldots)$ tal que, para cada n, $R_m^{(n)}$ tem multiplicidade n.

Definimos também que para cada número natural n e relação Q sobre A a notação $[m|Q]\mathbf{R}$ representa a substituição de $R_m^{(n)}$ por Q em $\mathbf{R}$.

Finalmente, se $\mathfrak{A}$ é uma $\mathbf{L}$-estrutura e $\mathbf{R}$ é uma $\mathbf{L}'$-sequência de relações sobre A, a noção de satisfação das fórmulas de $\mathbf{L}'$ pode ser estendida para formar uma interpretação de segunda ordem pelas seguintes regras:

- $\mathfrak{A} \models_{\mathbf{R}} \exists V_m^{(n)} \phi \Leftrightarrow$ para alguma relação Q sobre A de multiplicidade n $\mathfrak{A} \models_{[m|Q]\mathbf{R}} \phi$
- $\mathfrak{A} \models_{\mathbf{R}} \forall V_m^{(n)} \phi \Leftrightarrow$ para qualquer relação Q sobre A de multiplicidade n $\mathfrak{A} \models_{[m|Q]\mathbf{R}} \phi$

Este processo de extensão da linguagem pode ser seguido indefinidamente. Tal qual a lógica de predicados de primeira ordem foi estendida para uma lógica de segunda ordem por meio das modificações supracitadas à sua estrutura e interpretação, a lógica de predicados de segunda ordem pode ser estendida para uma de terceira ordem por meio de modificações similares para adicionar símbolos de terceira ordem e os devidos ajustes a sua sintaxe e semântica para os atender.

2.2 Teoria de provas

A teoria de provas é o ramo da lógica que visa estudar provas e suas propriedades. A teoria de provas divide-se, a princípio, em duas áreas, a teoria estrutural de provas e a teoria da interpretação de provas [15, página 1].

A teoria estrutural de provas foca na análise da estrutura das provas, que são definidas como objetos combinatórios, *i.e.* uma estrutura conceitual composta de objetos discretos. Já a teoria da interpretação de provas estuda as traduções que podem existir entre teorias formais.

O estudo da estrutura das deduções tem enfoque em *cálculos dedutivos*, esquemas utilizados para construir deduções para as fórmulas de uma linguagem. Formalmente, um cálculo dedutivo é um par (A, R), onde A representa um conjunto de *axiomas* e R representa um conjunto de *regras de inferência*. A princípio, trataremos de um tipo de cálculo dedutivo denominado *dedução natural*.

Os objetos que representam provas de sentenças lógicas serão chamado neste trabalho de *deduções*. Há diferentes formas de representar uma dedução. Utilizaremos a representação de árvore. Os nós são representados por fórmulas. As fórmulas nos nós acima de um determinado nó são suas premissas, e este nó é a conclusão. A fórmula na raiz da

árvore representa a conclusão final da dedução [15, página 22]. É interessante mencionar que, neste contexto, as árvores são representadas em uma orientação refletida no eixo horizontal em relação à representação usual. As árvores neste contexto são construídas com a raiz na base e as folhas no topo.

Um *axioma* é uma fórmula bem formada aceita a priori como verdadeira. Uma *regra de inferência* é algum mecanismo que permite obter uma fórmula bem-formada a partir de um conjunto de fórmulas bem formadas.

2.2.1 Dedução natural

A dedução natural é um cálculo dedutivo no qual a dedução de uma fórmula é representada por uma árvore. O nó da raiz da árvore é representado pela fórmula cuja dedução está sendo empreendida, as folhas representam hipóteses. Os nós intermediários representam fórmulas intermediárias, obtidas das hipóteses por meio da aplicação de regras de inferência. Na dedução natural o conjunto de axiomas é vazio.

As *suposições* são fórmulas temporariamente tidas como verdadeiras, tal que estas podem ser utilizadas em conjunto com regras de inferência para a dedução de novas fórmulas. Na dedução natural, as suposições devem ser marcadas por um símbolo que as diferencia das demais fórmulas. Todas as suposições da mesma forma devem ser marcadas com o mesmo símbolo. O conjunto de fórmulas marcadas com o mesmo símbolo é chamado de *classe de suposição*. Suposições distintas, por outro lado, devem ser marcadas com símbolos diferentes. As suposições de uma dedução podem estar *abertas* ou *fechadas*. Quando uma suposição é utilizada na aplicação de uma regra de inferência diz-se que, a partir desta aplicação, a suposição é *fechada*. Todas as suposições que não forem explicitamente fechadas são ditas *abertas*. Todas as suposições de uma classe de suposição devem ser fechadas simultaneamente [15, página 36].

Definição 6

A dedução natural é indutivamente definida da seguinte maneira. Suponha as deduções, D_1, D_2 e D_3. Suponha as classes de suposições $[A]^u$, $[B]^v$ e $[\neg A]^x$ que contém suposições abertas para as premissas da fórmula cuja dedução é está sendo feita, mas são fechadas quando considera-se a dedução como um todo. Uma dedução no sistema da dedução natural é feita a partir de aplicações sucessivas das seguintes regras de inferência:

- Uma fórmula A representa uma dedução natural da suposição aberta A, formando uma árvore de um único nó.

$$[A]^u$$

- A introdução da negação produz, a partir de uma suposição aberta A que, por meio de uma dedução D_1, produz uma fórmula $\bot$, a conclusão $\neg A$, fechando a hipótese aberta A.

$$\dfrac{\begin{array}{c}[A]^u \\ D_1 \\ \bot\end{array}}{\neg A}\,\mathrm{I}\neg, u$$

- A eliminação da negação produz a partir de duas deduções D_1 e D_2 e duas premissas $\neg A$ e A, a conclusão $\bot$.

$$\dfrac{\begin{array}{cc}D_1 & D_2 \\ \neg A & A\end{array}}{\bot}\,\mathrm{E}\neg$$

- A introdução da conjunção produz, a partir de duas premissas A e B, produzidas pelas deduções D_1 e D_2, respectivamente, uma nova fórmula constituída pela conjunção de A e B, *i.e.* $A \wedge B$

$$\dfrac{\begin{array}{cc}D_1 & D_2 \\ A & B\end{array}}{A \wedge B}\,\mathrm{I}\wedge$$

- A eliminação da conjunção produz, a partir de uma premissa na forma de uma conjunção, produzida por uma dedução D_1, uma fórmula correspondente a um dos termos da conjunção. Há duas regras de eliminação da conjunção, uma para cada fórmula que compõe a conjunção.

$$\dfrac{\begin{array}{c}D_1 \\ A \wedge B\end{array}}{A}\,\mathrm{E}\wedge_R$$

$$\dfrac{\begin{array}{c}D_1 \\ A \wedge B\end{array}}{B}\,\mathrm{E}\wedge_L$$

- A introdução da implicação produz, a partir de uma suposição aberta A que, por meio de uma dedução D_1, produz uma fórmula B, a conclusão $A \to B$,

fechando a hipótese aberta A.

$$\dfrac{\begin{array}{c}[A]^u\\ D_1\\ B\end{array}}{A \to B}\,\mathrm{I}{\to}, u$$

- A eliminação da implicação, também conhecida por *modus ponens*, produz a partir de duas deduções D_1 e D_2 e duas premissas $A \to B$ e A, a conclusão B.

$$\dfrac{\begin{array}{cc}D_1 & D_2\\ A \to B & A\end{array}}{B}\,\mathrm{E}{\to}$$

- A introdução da disjunção produz, a partir de uma premissa A, produzida por uma dedução D_1, uma nova fórmula composta pela disjunção $A \vee B$. Esta regra de inferência possui, também, uma variante à esquerda e outra à direita, como ilustrado abaixo.

$$\dfrac{\begin{array}{c}D_1\\ A\end{array}}{A \vee B}\,\mathrm{I}{\vee_R}$$

$$\dfrac{\begin{array}{c}D_1\\ B\end{array}}{A \vee B}\,\mathrm{I}{\vee_L}$$

- A eliminação da disjunção produz, a partir de uma disjunção $A \vee B$, por sua vez produzida por uma dedução D_1, e duas ocorrências da premissa C, uma produzida por uma dedução partindo de uma suposição aberta A e outra produzida por uma dedução partindo de uma suposição aberta B, a conclusão C, fechando as suposições abertas A e B.

$$\dfrac{\begin{array}{ccc} & [A]^u & [B]^v\\ D_1 & D_2 & D_3\\ A \vee B & C & C\end{array}}{C}\,\mathrm{E}{\vee}, u\ v$$

- A introdução do quantificador universal produz, a partir de uma premissa $A[x/y]$, produzida por sua vez por uma dedução D_1, a conclusão $\forall x A$. Deve-se ressaltar aqui que, ou $x \equiv y$, ou y não é uma variável livre ocorrendo em A

nem em qualquer suposição aberta em D_1.

$$\dfrac{\begin{array}{c}D_1\\A[x/y]\end{array}}{\forall x A}\,\mathrm{I}\forall$$

- A eliminação do quantificador universal produz, a partir de uma premissa $\forall x A$, por sua vez produzida por uma dedução D_1, a conclusão $A[x/y]$.

$$\dfrac{\begin{array}{c}D_1\\\forall x A\end{array}}{A[x/y]}\,\mathrm{E}\forall$$

- A introdução do quantificador existencial produz, a partir de uma premissa $A[x/t]$, produzida por sua vez por uma dedução D_1, a conclusão $\exists x A$.

$$\dfrac{\begin{array}{c}D_1\\A[x/t]\end{array}}{\exists x A}\,\mathrm{I}\exists$$

- A eliminação do quantificador universal produz, a partir de uma premissa $\exists x A$, produzida por sua vez pela dedução D_1, e da premissa C, produzida a partir da suposição aberta $[A[x/y]]$ e da dedução D_2, a conclusão C, fechando a suposição $[A[x/y]]$. Deve-se ressaltar aqui que, ou $x \equiv y$, ou y não é uma variável livre ocorrendo em A, y não é uma variável livre em C e nem em qualquer suposição aberta em D_2, exceto na suposição $[A[x/y]]$.

$$\dfrac{\exists x A \qquad \begin{array}{c}[A[x/y]]^u\\D_2\\C\end{array}}{C}\,\mathrm{E}\exists,\ u$$

- A eliminação clássica do absurdo produz, a partir de uma premissa $\bot$, por sua vez produzida por uma suposição aberta $[\neg A]$ e uma dedução D_1, a conclusão A, fechando a suposição $[\neg A]$

$$
\begin{array}{c}
[\neg A]^u \\
D_1 \\
\dfrac{\bot}{A}\ \bot_c
\end{array}
$$

Por meio da aplicação sucessiva das regras de inferência acima descritas, pode-se construir qualquer dedução no sistema da dedução natural. A título de exemplo, considere a dedução a seguir de $\neg\forall x \neg A(x) \rightarrow \exists x A(x)$:

$$
\dfrac{\dfrac{\dfrac{\neg\forall x\neg A(x)^w \qquad \dfrac{\dfrac{\dfrac{\neg\exists x A(x)^u \qquad \dfrac{A(x)^v}{\exists x A(x)}\ \mathrm{I}\exists}{\bot}\ \mathrm{E}\neg}{\neg A(x)}\ \mathrm{I}\neg, v}{\forall x \neg A(x)}\ \mathrm{I}\forall}{\bot}\ \mathrm{E}\neg}{\exists x A(x)}\ \bot_c, u}{\neg\forall x\neg A(x) \rightarrow \exists x A(x)}\ \mathrm{I}{\rightarrow}, w
$$

Com estes conceitos estabelecidos, temos agora parte da infraestrutura necessária para o desenvolvimento deste trabalho. Utilizando a lógica de ordem superior e o cálculo dedutivo descritos neste capítulo podemos expressar fatos sobre algum domínio de discurso e empreender deduções de novos fatos a partir destes, amparados pela rigidez da sintaxe e semântica elaboradas.

No próximo capítulo, voltaremos a nossa atenção à exploração de uma ferramenta computacional que permite trabalhar com estes conceitos de maneira interativa e automatizada. Esta ferramenta computacional permite a elaboração de especificações a respeito do domínio de discurso utilizando a lógica de ordem superior. Esta ferramenta permite a escrita de provas utilizando artifícios que auxiliam o usuário por meio da automação de alguns aspectos da dedução, servindo assim como um assistente de provas.

Capítulo 3

Isabelle e assistentes de provas

3.1 Isabelle

Isabelle [16] é um assistente de provas genérico que permite ao usuário especificar problemas em uma linguagem lógica e provar propriedades sobre o problema utilizando uma série de ferramentas voltadas para a prova automática ou interativa de teoremas. As aplicações principais do Isabelle são a formalização de provas matemáticas e a verificação formal de sistemas computacionais, que consiste na demonstração da corretude de propriedades de especificações formais de *hardware* e *software*, tal qual o funcionamento de circuitos combinatórios, a semântica de linguagens computacionais e o funcionamento de compiladores [17].

A variante mais popular do Isabelle é Isabelle/HOL, que implementa um ambiente de provas baseado na lógica de ordem superior (*HOL* é um acrônimo para *Higher Order Logic*), apropriada para o uso na especificação e verificação de sistemas complexos. Isabelle/HOL implementa ferramentas de especificação poderosas, como tipos de dados algébricos, definições indutivas e funções recursivas com casamento de padrões.

Para a produção de provas, o Isabelle incorpora diversas ferramentas para auxiliar o usuário. O Isabelle conta com provadores capazes de processar longas cadeias de raciocínio para encontrar provas para fórmulas. Quando estas ferramentas se mostram insuficientes, o Isabelle conta também com uma ferramenta externa chamada *Sledgehammer*, que faz uso de diversos provadores de primeira ordem com o intuito de encontrar provas automaticamente para sentenças lógicas.

Uma outra facilidade implementada é a capacidade de transformar especificações escritas em Isabelle em programas escritos em outras linguagens, como OCaml [18], Haskell [19] e Scala [20]. Desta forma, torna-se possível especificar e verificar um artefato de *software* em Isabelle e traduzi-lo em um programa correto em OCaml, por exemplo, que pode ser compilado e executado normalmente.

O sistema Isabelle conta também com uma extensa biblioteca de definições e teoremas matemáticos formalmente verificados, incluindo fatos sobre teoria de números, análise, álgebra e conjuntos [17]. Há, também, uma vasta coleção suplementar de definições e provas sobre diversos assuntos, disponível online em um repositório denominado *Archive of Formal Proofs*, que podem ser livremente usados [21].

As especificações em Isabelle são estruturadas na forma de arquivos de código fonte contendo definições de tipos, teoremas e provas. Vejamos agora como se dá a representação de especificações em Isabelle.

3.1.1 Tipos, termos e fórmulas

Em Isabelle/HOL há diversos tipos de entidades linguísticas que podem ser usadas na criação de especificações, como veremos a seguir.

Tipos

HOL, a lógica implementada pelo Isabelle/HOL, é uma lógica tipada e seus tipos, em linhas gerais, se dão da seguinte maneira:

- **tipos base**: são tipos primitivos, como os booleanos, denotados por *bool*, e os naturais, denotados por *nat*;
- **construtores de tipos**: são o mecanismo linguístico utilizado na construção de tipos compostos, a partir de tipos simples. *E.g.* o tipo das listas é dado por meio do construtor *list*, de tal maneira que uma lista de naturais tem o tipo *nat list*;
- **tipos funcionais**: tipos funcionais, denotados pelo símbolo "⇒", representam funções, tal qual são comumente encontradas em linguagens de programação funcionais;
- **variáveis de tipo**: variáveis de tipo, denotadas por letras precedidas de apóstrofos, tal como *'a, 'b,* ..., são utilizadas como mecanismo para polimorfismo e sua aplicação se tornará clara adiante, com a análise de exemplos.

Termos

Termos são formados por meio da aplicação de funções a argumentos, que podem ser membros de qualquer tipo supracitado, desde que haja um casamento entre o tipo da função e o tipo de seu argumento. *E.g.* suponha que f é uma função de tipo *'t1* ⇒ *t2* e t é um membro do tipo *'t1*, então a construção $f\ t$ tem tipo *'t2*.

Termos podem, também, conter construções funcionais de controle, como (*if* ... *then* ... *else* ...), (*let* ... *in* ...) e (*case* ... *of* ...), cujos significados ficarão aparente à medida que forem observados em exemplos.

Além disso, termos também podem conter abstrações lambda, da forma $\lambda x.t$, que tem o mesmo funcionamento de termos no cálculo lambda e servem para expressar funções anônimas.

Fórmulas

Em Isabelle, fórmulas são termos de tipo *bool*. Fórmulas admitem dois possíveis valores, representados pelos valores *True* e *False* e podem ser utilizadas na construção de fórmulas compostas por meio do uso dos conectivos usuais, *i.e.* $\neg$, $\wedge$, $\vee$, $\rightarrow$. Fórmulas são, também, passíveis de prova, como será explicado mais a diante.

Igualdade

Em Isabelle, a igualdade é denotada pelo símbolo infixo =, que é um membro do tipo *'a* $\Rightarrow$ *'a*. Para fórmulas, a igualdade tem o significado de "*se, e somente se ...*".

Quantificadores

Há, também, os quantificadores existencial e universal usuais, denotados por $\forall x.P$ e $\exists x.P$, onde P é uma fórmula, que tem o significado e uso esperado.

Anotações de tipo

Ao passo que Isabelle tem a capacidade de fazer inferência de tipos em quase todas as construções válidas em sua linguagem, há também um mecanismo de anotação de tipos que pode ser usado para fazer a coerção de um tipo a um determinado termo ou, simplesmente, criar uma indicação visual com a finalidade de auxiliar na leitura do código.

As anotações de tipo são feitas por meio da seguinte notação: $t :: '\tau$ indica que um termo t tem tipo $'\tau$.

3.1.2 Teorias

Uma *teoria* é uma coleção de *tipos*, *funções* e *teoremas*, objetos estes que serão apresentados em detalhes brevemente.

Em Isabelle, uma teoria segue a seguinte estrutura:

theory MyTheoryName
  imports Theory1 Theory2
begin

(*Aqui vão as definições de tipos,
funções e teoremas. Isto é, a
propósito, um comentário em
Isabelle. *)

end

No bloco de código acima, há alguns pontos importantes a serem detalhados:

- A palavra reservada **theory** precede, obrigatoriamente, o nome dado à teoria. No exemplo acima, a teoria foi nomeada *MyTheoryName*, porém este nome é arbitrário e pode ser decidido pelo programador *ad libitum*. É importante reforçar, no entanto, que um arquivo de especificações em Isabelle deve ser nomeado, obrigatoriamente com o mesmo nome dado à teoria que o mesmo contem.
- A palavra reservada **imports** precede os nomes das teorias importadas, cujos conteúdos se tornam visíveis e disponíveis no contexto da teoria principal. A implementação Isabelle/HOL contém uma teoria denominada *Main*, que contém diversas definições e teoremas úteis sobre fatos matemáticos e lógicos.
- Por fim, as palavras reservadas **begin** e **end** agem como delimitadores, indicando explicitamente o começo e o final do corpo da teoria.

É importante agora comentar brevemente sobre o uso de aspas no código fonte em Isabelle/HOL. Grosso modo, todas as construções que fizerem parte da linguagem HOL, especificamente, devem ser circunscritos por aspas, exceto quado se tratar de entidades linguísticas unitárias, sem espaços, onde o uso de aspas é facultativo. Estes pormenores ficarão claros à medida que exemplos forem apresentados.

3.1.3 Definições de tipos e funções

Em Isabelle, tipos de dados podem ser indutivamente definidos por meio da seguinte sintaxe:

A seguinte sintaxe, fornecida ao assistente de provas Isabelle/HOL, produzirá uma nova definição de tipo, a partir da qual seus elementos poderão ser também produzidos:

$$\begin{aligned}\textbf{datatype}\ (`a1, \ldots, `an)\ typeName = {} & C_1\ \text{“}t_{11}\text{”} \ldots \text{“}t_{1n}\text{”} \\ | & \vdots \quad \ddots \quad \vdots \\ | & C_k\ \text{“}t_{k1}\text{”} \ldots \text{“}t_{kn}\text{”}\end{aligned}$$

Esta definição é composta das seguintes partes:

- O tipo, aqui denominado *typeName*, deve ser precedido da palavra reservada *datatype*, que inicia a declaração de tipo, e (*'a1, ..., 'an*), uma lista de variáveis de tipo, utilizadas na interpretação do tipo como entidade linguística polimórfica.
- Os símbolos $C_1, \ldots, C_n$ são denominados os *construtores* do tipo, utilizados como elementos estruturantes dos membros do tipo.
- Finalmente, os vários elementos t_{mn} são termos, escritos na sintaxe HOL, que formam o corpo de um elemento do tipo.

Por mérito de ilustração, segue uma definição de tipo de uma lista paramétrica:
datatype $`a\ list = Nil$
$\quad\quad\quad\quad |Cons\ `a$ "'$a\ list$"

Neste exemplo, definimos um tipo chamado *list*, que representa uma lista de elementos de tipo 'a. Sua definição se dá pela criação de dois construtores. O construtor Nil representa uma lista vazia. O construtor $Cons$, que associa dois elementos, um de tipo 'a e um de tipo 'a $list$, em uma relação que pode ser explicada como a colocação de um elemento de um determinado tipo como primeiro elemento de uma lista de elementos do mesmo tipo, como se a lista estivesse sendo "aumentada" na posição do seu primeiro elemento.

Alguns membros deste tipo de lista são:

$(Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil)))$ :: *nat list*
$(Cons\ True\ (Cons\ False\ Nil))$ :: *bool list*

Acima, vemos duas listas construídas com os construtores previamente definidos. Vê-se que a forma como estes são usados produz listas que são criadas elemento a elemento a partir de uma lista vazia.

Quanto à definição de funções, há dois tipos: *definições*, usadas para criar funções não recursivas, e *funções*, usadas quando se faz necessário o emprego de recursão. Vejamos em detalhes como se dá a utilização de cada um desses tipos:

Uma definição, em Isabelle/HOL, é usada para definir procedimentos não recursivos, seguindo esta sintaxe:

definition $myDefinition$ :: " 'a_1 => 'a_2 => ... => 'a_n" **where**
"$myDefinition\ a_1\ a_2\ \ldots a_{n-1}\ =\ \ldots$"
(*Aqui vai uma descrição das operações feitas com os parâmetros da definição*)

Nesta construção, notam-se alguns pontos de interesse:

- A definição, aqui denominada $myDefinition$, é precedida do termo **definition**, que dá início à criação da definição.
- O nome da definição é seguido de ::, indicando que o que precede é uma representação de seu tipo, e este símbolo por sua vez é seguido de uma declaração de tipo funcional, na forma $`a_1 => `a_2 => \ldots => `a_n$, que representa um tipo funcional ordenadamente a_n representa o tipo de dados retornado pela definição e os tipos $a_1\ a_2\ \ldots a_{n-1}$ representam os tipos dos parâmetros da definição.
- Esta declaração é seguida da palavra reservada **where**, que dá início à declaração de como as entradas são operadas entre si para produzir a saída, o que deve ser descrito logo em seguida, utilizando a linguagem HOL.

A título de exemplo, considere a seguinte definição, que determina o quadrado de um número natural:

definition $square$ :: "$nat => nat$" **where**
"$square\ n = n * n$"

Na definição acima, criamos uma função não recursiva denominada $square$, que recebe um número natural e produz outro. Seu funcionamento é descrito por uma equação, onde diz-se que a aplicação do nome $square$ a um número natural `n` deve ser empreendida por meio da multiplicação de n por si próprio.

Uma função em Isabelle/HOL é, por sua vez, um procedimento recursivo sobre tipos indutivamente definidos. Sua criação é produzida pela seguinte sintaxe:

fun $myFunction$ :: "$`a_1 => \ldots => `a_n$" **where**
"$myFunction\ T_{11} \ldots T_{n1} = \ldots$"|
"$myFunction\ T_{21} \ldots T_{n2} = \ldots$"|
$\vdots \qquad \ddots \qquad \vdots$
"$myFunction\ T_{n1} \ldots T_{nn} = \ldots$"

Vejamos, agora, os significados de cada um dos elementos linguísticos introduzidos acima:

- A definição de uma função, aqui denominada por $myFunction$, é iniciada pela ocorrência da palavra reservada **fun**. O nome da função é seguido, também, da declaração de seu tipo, como visto anteriormente, e da palavra **where**.
- Na definição de uma função recursiva sobre um tipo indutivo, deve haver várias equações, uma para cada construtor do tipo, separadas pelo carácter |, onde, em

cada uma, atribui-se à aplicação da função a termos T, por meio do sinal de igualdade, uma representação computacional escrita na linguagem HOL.

A título de exemplo, considere a seguinte função em Isabelle/HOL, utilizada na concatenação de duas listas, cuja estrutura foi definida em um exemplo anterior:

fun *concat* :: "'*a list* => '*a list* => '*a list*" **where**
"*concat Nil* *ys* = *ys*" |
"*concat* (*Cons x xs*) *ys* = *Cons x* (*concat xs ys*)"

No exemplo acima é definida uma função recursiva denominada *concat*, que produz uma lista a partir de outras duas, todas contendo elementos de um mesmo tipo '*a*, que é verificado por meio de casamento de padrões.

Essa função faz casamento de dois tipos de padrão estrutural para as listas que recebe. O primeiro padrão é a aplicação da função a uma lista vazia e uma outra qualquer, que produz a segunda inalterada. No segundo padrão a função recebe uma lista com pelo menos um elemento, representado pela cadeia (*Cons x xs*).

A concatenação neste segundo padrão ocorre por meio da aplicação recursiva de *concat* entre as listas *xs* e *ys*, produzindo uma outra lista. Esta lista resultante, por sua vez, tem o elemento *x* inserido em seu início, por meio do construtor *Cons*. Desta maneira, através de sucessivas chamadas recursivas a *concat*, recoloca ordenadamente todos os elementos da primeira lista no início da segunda, produzindo assim a concatenação das duas listas.

3.1.4 Teoremas, lemas e provas em Isabelle

A definição de teoremas e suas respectivas provas são um aspecto central da utilização do Isabelle. Inicialmente, voltemos nossa atenção à definição de teoremas.

Em Isabelle, um teorema é definido pela seguinte sintaxe:

theorem *theoremName*[*options*] : "*some_HOL_formula*"
by *ProofRule*

Em detalhes:

- O teorema, aqui denominado *theoremName*, deve ser precedido pela palavra reservada **theorem**, que inicia a definição, e pode ser sucedido por uma lista de opções, que controlam a futura utilização do teorema pelos mecanismos de raciocínio automatizado de Isabelle. É interessante mencionar que no sistema Isabelle,

os termos **theorem** e **lemma** têm exatamente a mesma interpretação, assim como na matemática, servindo apenas para a diferenciação subjetiva de resultados para o programador.

- Após estes entes linguísticos, deve seguir uma fórmula, expressa em HOL.

Tão importante quanto a definição de teoremas, é a escrita de suas provas. A escrita interativa de provas em Isabelle se inicia com a definição de um teorema, que impele o interpretador de Isabelle a iniciar uma sessão de prova. Durante uma sessão de prova, o interpretador aguarda o fornecimento de comandos de prova, que especificam quais regras de inferência devem ser usadas a fim de que se produza a fórmula enunciada pelo teorema a partir de axiomas.

Há duas formas de se seguir com este processo em Isabelle, os chamados *apply scripts*, onde a prova se dá por meio do cálculo de sequentes regido pelo fornecimento de comandos ao interpretador que seguem a forma **apply**$(rule)$. Esta abordagem produz provas de difícil legibilidade e manutenção. Além deste mecanismo, Isabelle conta com outro sistema de provas, desenvolvido na próxima seção, chamado *ISAR*, que tem como propósito permitir a escrita de provas mais legíveis e cuja manutenção é mais prática. Este sistema de provas será utilizado no desenvolvimento deste trabalho.

Isabelle/ISAR

ISAR (*Intelligible Semi-Automated Reasoning* ou raciocínio inteligível semiautomatizado, em inglês) é uma abordagem utilizada na produção de provas formais legíveis. O ISAR tem como fundamentação preencher a lacuna linguística existente entre os processos e notação de prova internos ao sistema de prova interativa do Isabelle, coloquialmente chamados de "*apply scripts*", e o nível de abstração normalmente esperado pela audiência que eventualmente venha a ler as provas assim produzidas.

As provas em Isabelle/ISAR, quando escritas cuidadosamente e seguindo boas práticas, que serão oportunamente apresentadas ao longo deste trabalho, têm a característica interessante de serem mais legíveis, uma vez que a cada passo da prova deve-se enunciar o que está sendo provado e, também, quais regras de inferência devem ser aplicadas às fórmulas de cada passo da prova.

Uma prova em Isabelle/ISAR segue a seguinte estruturação básica:

lemma $myLemma$: "$someHOLformula$"
proof
  fix $var_1 \ldots var_n$
  assume 0 : "$proposition_1$"

from 0 **have** 1 : "$formula_0$" **by** $rule_0$
⋮
from n **show** "$formula_n$" **by** $rule_n$
qed

Devemos considerar alguns pontos na construção acima:

- Nota-se que a prova é iniciada pelo comando **proof** e terminada pelo comando **qed**. Dentre destes delimitadores vão os demais comandos de prova;
- Inicialmente, usa-se o comando **fix** para definir variáveis locais, que podem ser quantificadas durante o processo de raciocínio. Este comando é opcional;
- Pode-se também assumir propriedades, dadas por fórmulas, por meio do uso do comando **assume**. Esta fórmula é dada como verdadeira e fica associada à chave que sucede o termo **assume** e a precede. Neste exemplo, a fórmula "$proposition_1$" fica associada à chave 0. Este comando é opcional;
- Enfim, começam os comandos de prova obrigatórios. Nestes, usa-se a construção **from** a **have** b: para indicar que, da fórmula associada à chave a, obtêm-se uma nova fórmula, associada à chave b. Após essa construção, enuncia-se a fórmula obtida e a regra de inferência ou chamadas a um provador externo que a produziu, esta precedida pelo comando **by**. Das regras de inferência, há aquelas utilizadas para introduzir e eliminar conectivos lógicos, quantificadores, e igualdade. Dos provadores externos, estes são programas externos que servem para provar automaticamente fórmulas específicas. Falaremos mais destes em breve. Nota-se que no último passo da dedução utiliza-se o termo **show** em vez de **have**, para diferenciá-lo dos demais.

Voltamos agora nossa atenção a um exemplo prático, uma breve prova em Isabelle do teorema de Cantor, que diz que uma função cujo domínio é algum conjunto C e a imagem é o conjunto das partes de C não pode ser sobrejetora:

1. **lemma** "$\neg$ *surj(f :: 'a* $\Rightarrow$ *'a set)*"
2. **proof**
3. **assume** 0: "*surj f*"
4. **from** 0 **have** 1: "$\forall$ *A.* $\exists$ *a. A* $\neq$ *f a*" **by** (*simp add: surj_def*)
5. **from** 1 **have** 2: "$\exists$ *a.* $\{x.\ x \notin f\ x\} = f\ a$" **by** *blast*
6. **from** 2 **show** "*False*" **by** *blast*
7. **qed**

O pequeno teorema acima se dá por uma prova por contradição. Assume-se em 3 que f é uma função sobrejetora, ou seja, todos os elementos do contradomínio de f estão associados a pelo menos um elemento do domínio. Então, a partir disso, afirmamos em 4 que para todo conjunto A pode ser produzido a partir de um elemento a, ao qual f é aplicada. Isso se dá por meio de uma simplificação da fórmula utilizando a definição de sobrejetividade, como indicado pela regra de inferência aplicada. Disto, afirma-se em 5 que há algum elemento a tal que a aplicação de f a a produz um conjunto de elementos x tal que este não contém qualquer elemento produzido pela aplicação de f a x. Isto produz uma contradição, dado que a função f não produz o conjunto das partes de seu argumento, e o conjunto das partes de qualquer conjunto contém, necessariamente, ele próprio. Este passo se dá pela chamada ao provador externo *blast*, que faz uso de um procedimento de busca de provas para fórmulas envolvendo lógica e conjuntos. O passo final, 6, consiste na constatação da falsidade, representada pelo valor *False*, finalizando a prova por contradição via uma nova aplicação da regra *blast*.

Há ainda alguns termos utilizados na estruturação de uma prova em ISAR para reduzir a repetitividade do texto, agindo como "açúcar sintático" para algumas combinações comuns.

Um destes termos é *this*, que quando utilizado em um passo de alguma prova serve como referência ao resultado do passo anterior, *e.g.* **from** *this* **have** ... , de tal forma que não se faz necessário reafirmar a fórmula (ou a etiqueta associada a ela) obtida no passo anterior.

Há ainda outros três termos utilizados neste contexto, produzindo assim um grau ainda maior de simplificação do texto da prova. Estes termos são:

- **then**, que significa o mesmo que **from** *this*;
- **thus**, que significa o mesmo que **then show**;
- **hence**, que significa o mesmo que **then have**.

Utilizando esta sintaxe, a prova utilizada no exemplo anterior pode ser expressa da seguinte maneira:

lemma "$\neg$ *surj(f :: 'a* $\Rightarrow$ *'a set)*"
proof
  assume "*surj f*"
  hence "$\exists$ *a.* $\{x.\ x \notin f\ x\} = f\ a$" **by**(*auto simp: surj_def*)
  thus "*False*" **by** *blast*
qed

Há ainda duas outras variações sintáticas utilizadas na reestruturação de provas, estas com o intuito de mudar a ordem do enunciado de um passo na prova, botando ênfase na conclusão:

- **have**/**show** *prop* **using** *fact* é equivalente a **from** *fact* **have**/**show** *prop*.
- **with** *facts* = **from** *facts this*

Estas variações sintáticas, ao passo que optativas e "invisíveis" ao interpretador do Isabelle, por se tratarem de açúcar sintático, permitem uma maior legibilidade do código para o desenvolvedor.

Nos exemplos acima foram utilizados procedimentos de prova automática, tal qual *auto* e *blast*. Estes tipos de ferramenta são parte fundamental do sistema de provas utilizado em Isabelle e portanto trataremos agora destes mecanismos com um pouco mais de detalhe.

- O comando *simp* é utilizado para encontrar provas para um teorema por meio de reescrita utilizando equações que funcionam como *regras de simplificação*. Estas equações são definidas ao atribuir a um teorema enunciado como equação a marcação [*simp*], *e.g.* **theorem** *someTheorem* [*simp*]: "*some_formula*". A simplificação consiste em tentar reescrever ocorrências do termo à esquerda pelo termo à direita de uma equação. Este processo de reescrita é repetido enquanto for possível. A uma aplicação da regra *simp* podem ser acrescentadas regras de reescrita suplementares, por meio do comando *add*. *E.g.* (*simp add: rule*).
- O comando *auto* tenta encontrar provas por meio de simplificação, porém com maior flexibilidade e potência que *simp*, já que também inclui mecanismos de raciocínio sobre fatos de primeira ordem e aritmética.
- O comando *force* é utilizado na busca de provas em lógica clássica.
- Há a variante *fastforce*, que não é tão poderosa quanto *force*, porém tem uma execução mais rápida.
- O comando *blast* é utilizado para buscar provas para fatos sobre sentenças lógicas de primeira ordem ou sobre conjuntos.
- O comando *atrith* é utilizado para buscar provas para fatos sobre relações aritméticas.
- O comando *try* é utilizado para tentar todas as técnicas acima, iterativamente.

Além destas, há ainda outras regras de inferência e táticas de prova automatizada que serão explicadas individualmente à medida que se fizer oportuno por razão de seu uso em algum exemplo ou excerto de código.

Há ainda outro aspecto estrutural da sintaxe de provas em ISAR que deve ser apresentado, que é a sintaxe utilizada para produzir provas divididas por casos. Uma prova por casos segue, em linhas gerais, a estrutura:

proof (*cases var*)
    case (*case*_1)
    proof steps
        ⋮
next
    case (*case*_n)
    proof steps
        ⋮
qed

A prova acima é iniciada com a utilização do comando **proof**, acrescido de *cases var*, que sinaliza que a prova será feita por meio de uma análise de casos sobre o termo *var*. Em seguida, a análise de cada caso em particular se inicia pela utilização do comando **case**, seguida da representação textual do caso considerado. Após o final da análise de um caso particular, emite-se ao interpretador o comando **next** para indicar que o próximo caso será provado.

Provas por indução se dão, em grande medida, da mesma forma, conforme o esboço:

proof (*induction var*)
    case (*base*)
    proof steps
        ⋮
next
    case (*induction step*)
    proof steps
        ⋮
qed

Neste exemplo, utiliza-se a regra *induction var* após **proof** para sinalizar ao interpretador que se empreenderá uma prova por indução na variável *var* e, sem seguida, a mesma se dá, como de costume em provas por indução, pela prova com a quantidade de casos adequada para os objetos cujas propriedades estão sendo analisadas. O resto do exemplo segue da mesma forma que a prova por casos previamente demonstrada.

Há ainda muitos outros detalhes sobre o uso do provador Isabelle na especificação e prova de teoremas, mas a favor da brevidade terminamos agora esta introdução aos princípios fundamentais da utilização de Isabelle. Os temas e técnicas centrais foram apresentados e, à medida que se fizer necessário, os demais pormenores serão explicados *ad hoc* no contexto em que primeiro aparecerem.

Esta ferramenta será usada no Capítulo 5 para especificar e provar propriedades de um sistema que será oportunamente apresentado.

Capítulo 4

Legibilidade de provas

4.1 Da importância da legibilidade de provas

O problema da legibilidade de provas foi abordado por David Hilbert na forma de um vigésimo quarto problema que, originalmente, não foi divulgado junto ao seu famoso conjunto de 23 problemas para o "novo século". O vigésimo quarto problema de Hilbert se referia a encontrar a prova mais simples para um dado teorema [22, página 2]. O conceito de "prova mais simples" depende de questões subjetivas relacionadas à interpretação do leitor. Isso envolve conceitos imprecisos e, para contornar isso, Hilbert propôs um cálculo dedutivo sobre fórmulas de tal maneira que uma prova pudesse ser escrita na forma de uma sequência finita de fórmulas finitas, cada uma produzida a partir das anteriores por meio de uma operação de inferência [22, página 5].

Com estas considerações em vista, fica claro que é importante produzir provas legíveis, de tal maneira que o esforço requerido do leitor na interação com o texto e notação seja minimizado. Segundo Zammit [1, página 24], a legibilidade de provas produzidas em assistentes de provas pode ser aprimorada por meio de quatro estratégias.

A primeira consiste em apresentar as provas na forma de uma estrutura hierárquica. Essa abordagem é tratada em detalhes por Lamport. Segundo Lamport, o aprimoramento da legibilidade de provas depende de dois fatores: estrutura e nomenclatura [2, página 1]. A adoção de uma estrutura adequada facilita o entendimento do que cada passo representa na fórmula. Já a adoção de uma nomenclatura apropriada para os fatos tratados na prova faz com que seja fácil distinguir de quais fatos depende um determinado passo da prova [2, página 1].

Lamport comenta, também, o impacto do nível de detalhamento da prova na sua legibilidade. Muito detalhamento pode fazer com que as provas sejam difíceis de entender pela quantidade de informação que o leitor precisa manter em mente, já pouco detalhamento pode fazer com que o salto de um passo da prova para o próximo não seja intuitivo [2,

página 1]. Lamport propõe que as provas sejam estruturadas como uma sequência de fórmulas nomeadas por números naturais sequenciais. Cada fórmula deve ser seguida imediatamente pela prova que a produz [2, página 3].

A segunda estratégia proposta por Zammit consiste em explicar a prova em linguagem natural. Com esta abordagem, coloca-se a prova justaposta a um parágrafo escrito em linguagem natural explicando o processo de dedução desenvolvido. Desta maneira, alia-se a inteligibilidade da linguagem natural com a precisão da notação matemática de forma que, idealmente, uma complemente a outra em termos de facilitar a leitura e compreensão da prova [1, página 24].

A terceira estratégia envolve o emprego do paradigma de programação letrada [23] no desenvolvimento de provas. Neste contexto, utiliza-se uma ferramenta que permita a intercalação de pequenos *scripts* executáveis com elementos textuais em linguagem natural, de maneira que o documento possa ser tanto lido por humanos tal qual um texto comum, como também compilado e executado por uma máquina [1, página 24].

A quarta e última estratégia descrita por Zammit consiste em utilizar uma ferramenta computacional para o desenvolvimento de provas cuja linguagem é similar à linguagem natural e informal costumeiramente usada em textos matemáticos. Desta forma, o texto das provas seria, segundo o autor, mais facilmente lido e compreendido por um público amplo [1, página 25].

Neste trabalho, utilizaremos a primeira e terceira estratégia, pois estas mais se adequam ao problema e ferramenta que nos propomos a usar.

Capítulo 5

Estudo de caso

O presente trabalho empreende apresentar provas legíveis que constatem a verificação formal das especificações de um sistema de comando e controle utilizado pelo Exército Brasileiro. Estas provas serão produzidas paralelamente às demais provas produzidas pela equipe de pesquisadores empenhados no projeto de tal forma que ambos conjuntos de provas demonstrem exatamente os mesmos resultados.

Espera-se que o artifício apresentado neste trabalho apresente um grau superior de clareza e legibilidade, sendo assim útil a uma quantidade maior de pessoas que possam eventualmente valer-se das especificações realizadas no contexto do projeto.

5.1 Verificação de sistemas de comando e controle

Este trabalho se baseia em parte no relatório técnico intitulado *Requirements Specification and Analysis of Command and Control Systems* [24] que apresenta um método para a formalização e análise dos requisitos de sistemas de comando e controle. A formalização é feita em PVS e é aplicada a um exemplo real.

Sistemas de comando e controle são utilizados para representar um ambiente real e a comunicar a operadores de tal maneira que informações extraídas dessa representação possam servir de apoio a tomada de decisões. Sistemas de comando e controle são dotados de sensores que produzem medições a partir do ambiente e que são continuamente compiladas em representações abstratas da situação no ambiente. Um sistema de comando e controle suporta o processo de tomada de decisões no qual os operadores do sistema comparam o estado atual da representação do ambiente com algum estado desejado e, a partir disso, emitem comandos atuadores que agem sobre o ambiente.

A interface entre o sistema e o ambiente externo é modelada por eventos, que podem ser distinguidos entre eventos de entrada, produzidos pelo ambiente, e saída, produzidos pelo sistema. Eventos de entrada são, em geral, métricas produzidas por sensores ou comandos

de operadores. Eventos de saída são, em geral, comandos ou sinais de controle para os atuadores do sistema. Os eventos são considerados como objetos atômicos, ou seja, que ocorrem de forma instantânea, um após o outro. Eventos de entrada causam uma mudança no estado do sistema, algo que assumimos que também ocorre instantaneamente. Uma mudança no estado do sistema pode provocar, também, um evento de saída. O sistema será modelado por meio de uma máquina de Mealy modificada.

5.1.1 Modelo da máquina

Definição 7 Máquina de estados de Mealy [24, página 6]

Uma máquina de Mealy tradicional é definida como a tupla $(I, O, S, \delta, \lambda, s_0)$, onde:

1. I é um conjunto finito de símbolos de entrada.
2. O é um conjunto finito de símbolos de saída.
3. S é um conjunto finito de estados.
4. $\delta : S \times I \to S$ é a função de transição de estados.
5. $\lambda : S \times O \to S$ é a função produtora dos eventos de saída.
6. $s_0 \in S$ é o estado inicial.

No modelo adotado, algumas mudanças são feitas quanto à forma canônica apresentada acima:

1. Os conjuntos I, O e S podem ser infinitos.
2. A função de transição δ pode ser definida parcialmente ou de modo não determinístico.
3. A cada transição de estados, um conjunto de símbolos de saída é emitido e a composição deste conjunto depende dos estados envolvidos na transição.
4. Pode-se tomar um conjunto de estados iniciais (*i.e.* pode haver mais de um estado inicial).

Permitir um número infinito de estados aumenta drasticamente o poder expressivo do modelo [24, página 7]. Um estado pode ser visto como uma atribuição de valor à uma variável de estado.

Após essas modificações, o modelo de máquina fica da seguinte maneira:

Definição 8 Máquina de Mealy modificada

A máquina de Mealy é definida como a tupla $(I, O, S, \Delta, \Lambda, S_0)$ [24, página 7], onde:

- I é um conjunto possivelmente infinito de eventos de entrada.
- O é um conjunto possivelmente infinito de eventos de saída.
- S é um conjunto possivelmente infinito de estados.
- $\Delta : I \longrightarrow \mathcal{P}(S \times S)$ é a função de transição de estados.
- $\Lambda : O \longrightarrow \mathcal{P}(S \times S)$ é a função que desencadeia eventos de saída.
- $S_0 \subseteq S$ é um conjunto não vazio de possíveis estados iniciais.

Essa máquina apresenta o seguinte funcionamento: assumimos que, inicialmente, o sistema se apresenta em algum estado $s_0 \in S_0$. Assumimos também que, durante a execução, a máquina se encontra em algum estado $s \in S$. Nestas condições, quando ocorre um evento de entrada $i \in I$ o sistema transita, de maneira não determinística, a algum estado t, tal que $(s, t) \in \Delta(i)$, caso este exista. Durante esta transição todos os eventos de saída o tais quais $(s, t) \in \Lambda(o)$ são produzidos. Caso não exista tal estado t o sistema é bloqueado em *deadlock*.

Para facilitar a futura análise do sistema, alguns elementos são também adicionados. Estas adições são:

- A invariante $Inv \in \mathcal{P}(S)$, que representa condições de integridade do estado.
- A precondição sobre eventos de entrada $Pre : I \to \mathcal{P}(S)$, que representa fatos que assumimos sobre o ambiente.

A invariante é introduzida na especificação do sistema pois, com ela, podemos especificar a priori que alguma condição deve sempre valer e que a mesma é mantida a cada mudança de estados.

As precondições podem ser interpretadas como requisitos que o ambiente no qual o sistema opera deve atender de forma que, sempre que tais requisitos sejam satisfeitos, o sistema não entrará em *deadlock*. Estes requisitos sobre o ambiente se configuram aqui como uma propriedade sobre os eventos de entrada que podem ser gerados tais que estes não produzam uma transição de estados que leve a uma condição de *deadlock*.

A verificação do sistema é feita em duas etapas, A primeira etapa consiste em mostrar que há um estado inicial válido, *i.d.*

$$\exists s.s \in S_0 \cap Inv$$

Na segunda etapa, uma prova deve ser produzida para mostrar que para qualquer entrada i e qualquer estado s que obedeça tanto à invariante quanto à precondição quanto à entrada i, existe um estado t pertencente à invariante tal que a entrada i leve ao estado t.

$$\forall i \in I.\forall s \in Inv \cap Pre(i).\exists t \in Inv.(s,t) \in \Delta(i)$$

Quanto à validação, esta é feita por meio da inspeção do código e de "desafios" formais.

Uma forma de validação é a inspeção das especificações por especialistas no assunto sob consideração. Para isso, é necessário que a especificação seja acessível e legível. Outra abordagem é a imposição de teoremas ao sistema. A ideia é checar se algumas propriedades esperadas do sistema podem ser produzidas a partir das estruturas propostas pela especificação.

Para provar tais teoremas, utiliza-se um princípio de indução quanto à "alcançabilidade" dos estados do sistema. Um estado é alcançável se, partindo do estado inicial, o sistema pode o alcançar por meio de uma sequência de símbolos de entrada que satisfaçam a precondição. Formalmente, o conceito de alcançabilidade é expresso indutivamente da seguinte forma:

- $S_0 \cap Inv \subseteq R$
- se $s \in R$, $s \in Pre(i)$, $t \in Inv$ e $(s,t) \in \Delta(i)$, então $t \in R$

para o menor conjunto de estados S' possível.

A esta definição, associa-se o seguinte princípio de indução que permite demonstrar que alguma propriedade P vale para todos os estados alcançáveis:

- $\forall s \in S_0.P(s)$
- $\forall s,t \in S, i \in I.(s \in S' \wedge s \in Pre(i) \wedge t \in Inv \wedge (s,t) \in \Delta(i) \wedge P(s) \rightarrow P(t))$

A primeira expressão afirma que a propriedade P deve valer para todos os estados iniciais possíveis.

A segunda expressão afirma que para quaisquer estados s e t e para qualquer evento de entrada i, deve valer que, se s faz parte do conjunto de estados S', s atende à precondição quanto à entrada i, o estado t faz parte da invariante, a transição (s,t) é possível quanto

à transição provocada por i e a propriedade P vale para s, então a propriedade P deve valer também para t.

5.1.2 Estudo de caso

A especificação abordada neste trabalho é fundamentada num sistema de comando e controle baseado em *track fusion*. Descreveremos agora, em linhas gerais, o funcionamento de um sistema de *track fusion*.

Track fusion

Um registro é uma descrição de algum objeto do mundo real que conta com informações como posição, velocidade, identificação, etc [24, página 15]. Um registro existe no sistema em dois níveis distintos:

- Registros de sensores, que são produzidas por sensores a partir da sua interação com o ambiente.
- Registros de sistema, que são geradas pelo próprio sistema.

Um objeto do mundo real pode ser descrito simultaneamente por vários sensores. Para garantir que o sistema tenha uma visão coerente do ambiente é necessário que todos os registros de sensores que representam um mesmo objeto sejam unidos em um único registro de sistema por meio de um processo denominado fusão. Com este fim, é necessário também definir alguns critérios de correlação capazes de determinar se dois registros representam o mesmo objeto.

A entrada do sistema é definida por meio das ações que podem ser tomadas a partir de medições dos sensores. Um sensor pode iniciar, atualizar ou apagar registros de sensores. Uma representação do ambiente para o sistema consiste em conjuntos de registros de sensores e do sistema, as relações entre estes e informações sobre o ambiente derivadas destes dados. O sistema deve relatar aos operadores os registros de sistema, para que estes sirvam de apoio à tomada de decisões. É também necessário que o sistema produza alertas ao detectar uma decorrelação entre um registro de sensor e uma de sistema.

5.1.3 Interface do sistema *Track Fusion*

Inicialmente, definem-se dois tipos básicos para representar os registros, um tipo representando registros de sensores e outro registros de saída. A partir disto então definem-se os eventos de entrada. Há três tipos de eventos de entrada: criação de novos registros, atualização de registros existentes e a exclusão de registros existentes. Todos os eventos

de entrada dizem respeito à manipulação de registros de sensores. Cada evento carrega, também, parâmetros indicando registro afetado e, para os eventos de atualização e eliminação, também o estado do registro.

Os eventos de saída são definidos de maneira análoga. Há quatro tipos de eventos: criação, atualização e eliminação de registros e um quarto tipo de evento de saída referente à emissão de alertas de decorrelação. Todos os eventos de saída dizem respeito a registros de sistema.

5.1.4 Modelo de informação

O estado global do sistema é modelado como um registro contendo os seguintes dados: o conjunto dos registros de sensor recebidos até agora, o conjunto dos registros de sistema derivados destes, uma relação indicando quais registros de sensor e de sistema estão ligados entre si e uma função que produz o estado de cada registro de sistema.

Em seguida, um número de invariantes é formulado para fixar algumas propriedades globais do sistema. A relação entre os registros de sensor e sistema é estabelecida por meio destas invariantes. Em particular, deve ser fixado que a relação de ligação entre registros seja sobrejetora e total. Desta maneira, cada registro de sistema pode ser interpretada como um conjunto de um ou mais registro de sensor.

Definimos agora as seguintes condições:

Definição 9 Condição 1

A relação de ligação entre registros de sensor e sistema, sn e tn respectivamente, só pode existir entre registros que fazem parte do estado atual:

$$\forall sn, tn.joined(sn, tn) \rightarrow sensor\_ids(sn) \wedge system\_ids(tn)$$

Definição 10 Condição 2

Cada registro de sensor sn no estado é ligado a exatamente um registro de sistema tn:

$$\forall sn.sensor\_ids(sn) \rightarrow \exists! tn.joined(sn, tn)$$

Definição 11 Condição 3

Cada registro de sistema tn no estado é ligado a pelo menos um registro de sensor sn:

$$\forall tn.\exists sn.joined(sn, tn)$$

A invariante do sistema Inv é definida pela conjunção das três condições acima. Em um estado inicial, ambos os conjuntos de registros devem ser vazios.

5.1.5 Mapeamento entre entrada e saída

Para cada tipo de evento de entrada é necessário definir a transição de estados correspondente. Cada novo registro de sensores obtido pode ser ligado a um registro de sistema existente, contanto que alguns critérios de correlação sejam mantidos. Alternativamente, pode ocorrer que, após uma atualização em um registro de sensor, o registro com o qual este é ligado fique decorrelacionado. Este tipo de situação é decidido por meio de um conjunto de critérios de decorrelação. Para a obtenção de um sistema estável, os critérios de decorrelação não serão definidos como a exata negação dos critérios de correlação.

Inicialmente, os critérios de correlação e decorrelação serão introduzidos como relações não interpretadas entre registros de sensor e o estado de registros de sistema. Para evitar interpretações arbitrárias, adiciona-se também algumas restrições à estas condições: Deve ser possível iniciar algum registro de sistema em um registro de sensor e os critérios de correlação e decorrelação devem ser mutuamente exclusivos.

Definição 12 [Correlação e decorrelação]

Assumindo que as variáveis não ligadas são universalmente quantificadas e dados os registros de sensor e sistema, s e t, respectivamente, observam-se as duas condições:

$$\exists t.correlates(s, t)$$

$$\neg(correlates(s, t) \wedge (decorrelates(s, t)))$$

Pode-se agora definir as transições de estados provocadas por eventos de entrada. Quando um novo registro de sensor é inicializado, é necessário que este seja adicionado ao conjunto de sensores de sensor e que ele seja correlacionado ao registro de sistema ao qual ele é unido.

Quando um registro for apagado, é necessário que ele seja removido do conjunto de registros de sensor. Além disso, é necessário, também, que o conjunto de registros de

sistema e a relação de união tenham tamanhos estritamente decrescentes ao longo da execução do sistema.

Quando um registro de sensor for atualizada, o conjunto de registros de sensor deve permanecer o mesmo. Necessita-se também, que no próximo estado, os registros de sistema e sensor não fiquem decorrelacionadas.

Define-se em seguida as mudanças de estado que produzem eventos de saída. Dados os estados (x, y) antes e depois da transição, um registro de sistema é novo se ocorrer em y mas não em x. Atualizado se ocorrer de forma diferente em x e y. Apagado se ocorrer em x mas não em y. Finalmente detecta-se uma decorrelação do sistema se o registro for unido a registros de sistema diferentes em x e y.

Presupostos sobre o ambiente

A especificação é completada ao adicionarmos alguns pressupostos sobre as características do ambiente. Os principais pressupostos serão:

- Apenas registros de sensor existentes podem ser atualizados ou apagados.
- Apenas registros de sensor novos podem ser iniciados.
- Há sempre novos identificadores para registros de sistema.

5.1.6 Análise

Agora podemos dar início à verificação do sistema por meio da demonstração das obrigações de prova. Isso se dá com o apoio do sistema assistente de provas Isabelle. A teoria que especifica a máquina contém os seguintes parâmetros: I, O, S, que são parâmetros de tipos, e $Imap$, $Omap$, $init$, Inv e Pre, que são definidos, correspondem aos componentes da máquina de estados. O trabalho de verificação consiste em provar que estes pressupostos valem para os parâmetros da teoria, *i.e.* eventos de entrada, saída, etc. Neste ponto também é definido o conceito de alcançabilidade. Finalmente, prova-se que a invariante se mantém para todos os estados alcançáveis.

Em detalhes, a demonstração das obrigações de prova se dá em alguns passos. Primeiramente, deve-se checar se existe um estado inicial que satisfaz a invariante. Em outros termos, isso significa mostrar que para algum estado x vale a seguinte propriedade:

$$\exists x.init(x) \wedge inv(x)$$

A segunda obrigação de prova consiste, em termos gerais, em mostrar que transições de estados são possíveis sempre que as precondições são válidas. Para isso, é necessário antes

mostrar um lema auxiliar que afirma que, para cada evento de entrada, uma transição de estados é possível. Para estados x e y e evento de entrada i:

$$\forall ix.Pre(i)(x) \wedge Inv(x) \rightarrow \exists y.Inv(y) \wedge Input\_table(i)(x, y)$$

5.2 Refinamentos

Nesta seção, introduz-se dois refinamentos da especificação desenvolvida na última seção. O primeiro refinamento adiciona mais requisitos às transições de estados, tornando a especificação então mais operacionalmente concreta. Este refinamento produz uma especificação de dados mais estruturada e diminui o não-determinismo presente na especificação de base. O primeiro refinamento é, também, parametrizado para que possa ser relacionado posteriormente com o segundo refinamento.

O primeiro refinamento incorpora três mudanças quando comparado à especificação base:

- Requisitos operacionais são adicionados.
- A invariante é convertida em subtipos.
- A especificação é parametrizada.

A especificação base é bastante global. Nela, computações cinemáticas foram deixadas de fora da especificação e a conexão entre registros de sensor e sistema foi especificada apenas nas invariantes. Desta forma, é possível a existência de implementações indesejáveis. O propósito do primeiro refinamento é, justamente, adicionar requisitos suplementares para reduzir o conjunto de possíveis implementações. Isto é logrado por meio da especificação de como exatamente um dado estado é computado a partir do último estado. Um efeito colateral interessante é que, ao adicionar definições mais detalhadas das transições de estados, um número de propriedades que eram garantidas pela invariante na especificação base emergem como propriedades do sistema induzidas pela própria definição das transições.

Na especificação base, alguns tipos e constantes não foram definidas. Mantendo estas definições não interpretadas, se permite uma certa liberdade na implementação do sistema. Com a finalidade de refinar a especificação, é necessário agora especificar estes símbolos não interpretados como parâmetros da teoria. O que se obtém com isso é, de certa forma, um conjunto de especificações, uma para cada configuração na instanciação dos parâmetros. Um refinamento lógico pode ser visto neste contexto como uma instanciação parcial de alguns desses parâmetros, de maneira que o conjunto de possíveis especificações é reduzido.

Os critérios de correlação e decorrelação devem ser exclusivos. Um registro de sensor e o registro de sistema iniciado sobre ele devem estar correlacionados. Um teste de decorrelação em um registro de sistema atualizado não deve falhar, a não ser que tenha já falhado antes da atualização.

Refinamento das transições de estados

A implementação da criação de novos registros de sensor na especificação base permitia que, a qualquer momento, um novo registro de sistema fossem criados. Esse comportamento é eliminado no primeiro refinamento. Agora, a criação de um novo registro de sistema só pode ocorrer se todos os testes de correlação com os registros de sistema existentes falharem.

Similarmente, é agora requerido que ao atualizar um registro de sensor, a união que este tem com um registro de sistema só pode ser desfeita se houve decorrelacionamento. Além disso, todos os registros não envolvidos na operação devem permanecer inalterados. Portanto, criamos também um predicado sobre a não correlação, que indica que não há correlação entre um dado registro de sensor e qualquer registro de sistema.

Se um novo registro de sensor é recebido, um novo registro de sistema é então criado caso não seja possível unir o registro de sensor a um registro de sistema existente. Para produzir este efeito, duas novas transições de estado são criadas, uma para a criação de registro de sistema e uma para detectar a correlação com um registro de sistema.

Apagar um sensor envolve deletar o seu registro de identificação no sistema. A relação de união entre registros deve ser atualizada apropriadamente. Se o registro de sistema correspondente estava unido apenas ao registro de sensor apagado, ele se torna instável e, por isso, deve ser também removido.

Quando uma atualização a algum registro de sensor é recebida, o estado do registro de sistema associado deve também ser atualizado apropriadamente a não ser que seja detectada a decorrelação. Neste caso o registro de sensor deve então ser atribuído a outro registro de sistema, existente ou criado *ad hoc* para este fim.

5.3 Verificação formal em Isabelle

Finalmente, com esta infraestrutura teórica definida, pode-se partir para a especificação e verificação do sistema feita com o auxílio do sistema assistente de provas Isabelle. O trabalho apresentado nesta seção toma como base as especificações produzidas pela Dra. Cláudia Nalon e Dra. Ariane Alves de Almeida, que fizeram a especificação e as provas iniciais.

Esta apresentação se dará em dois momentos. Primeiro, definimos e validamos o modelo de máquina abstrata que será usada como base para o sistema. Segundo, especificaremos e analisaremos a uma especificação do sistema de comando e controle que representa a especificação base previamente descrita e refinada com o primeiro refinamento.

Especificação da máquina abstrata

A teoria *AbstractStateMachine* modela a especificação da máquina abstrata. Neste ponto, definem-se alguns parâmetros da teoria (precedidos pela palavra reservada **fixes**) e, em seguida definem-se duas propriedades sobre a máquina referentes à existência de um estado inicial e de um estado alvo para as transições válidas.

locale *stateMachineLcl* =
  fixes *stateTrans* :: "$'inE \Rightarrow ('state \Rightarrow' state \Rightarrow bool)$"
  and *triggerFunc* :: "$'outE \Rightarrow ('state \Rightarrow' state \Rightarrow bool)$"
  and *init* :: "$'state \Rightarrow bool$"
  and *inv* :: "$'state \Rightarrow bool$"
  and *preCond* :: "$'inE \Rightarrow' state \Rightarrow bool$"
assumes
  exInit: "$\exists x.(init\ x \wedge inv\ x)$" **and**
  exState: "$\forall i\ x.(preCond\ i\ x \wedge inv\ x \longrightarrow (\exists y.(inv\ y \wedge stateTrans\ i\ x\ y)))$"

Em seguida define-se uma propriedade de alcançabilidade. Esta propriedade é indutivamente definida sobre os estados e afirma que o estado inicial é alcançável e, também, que se um estado y é alcançável e existe uma transição válida de y a x, então x é também alcançável.

inductive *reachable* :: "$'state \Rightarrow bool$" **where**
  base: "$(init\ x \wedge inv\ x) \Longrightarrow reachable\ x$" |
  step: "$(\exists y\ i.(reachable\ y \wedge preCond\ i\ y \wedge inv\ x \wedge stateTrans\ i\ y\ x)) \Longrightarrow reachable\ x$"

Finalmente, prova-se que a invariante vale em relação a esta definição de máquina e a propriedade de alcançabilidade. Esta propriedade afirma que, se um dado estado x é alcançável, então ele faz parte da invariante.

Na primeira prova, utilizando um alto grau de automação, a prova é feita por meio de uma análise de casos sobre a definição de *reachable* e a dedução do resultado é feita pelo provador *metis*.

lemma *invariantHolds*:
"$\forall x.reachable\ x \longrightarrow inv\ x$"
proof -
show ?*thesis*
using *reachable.cases*
by *metis*
qed

Esta prova foi reescrita para aprimorar a sua legibilidade de acordo com as recomendações delineadas no capítulo anterior, assim como outras adaptações. Primeiramente, o enunciado da prova foi escrito utilizando as palavras reservadas **fixes**, **assumes** e **shows**. Esta construção permite mostrar de maneira explícita qual é a variável quantificada, quais fatos são assumidos e o que se quer mostrar.

A prova é feita por casos, mas agora os casos são individualmente e explicitamente provados. Por *reachable* ser uma definição indutiva, cada caso corresponde à uma das formas que *reachable* pode tomar. As demonstrações de cada passo seguem de maneira simples e dispensam maior elucidação.

lemma *invariantHolds*:
fixes x
assumes "$reachable\ x$"
shows "$inv\ x$"
proof ($cases\ rule : reachable.cases[of x]$)
show "$reachable\ x$" **using** *assms* **by** *simp*
next
assume 1: "$init\ x \wedge inv\ x$"
from 1 **show** 2: "$inv\ x$" **by** ($ruleconjE$)
next
assume 1: "$\exists y\ i.reachable\ y \wedge preCond\ i\ y \wedge inv\ x \wedge stateTrans\ i\ y\ x$"
then obtain y i **where** 2: "$reachable\ y \wedge (preCond\ i\ y \wedge inv\ x \wedge stateTrans\ i\ y\ x)$"
by *blast*
from 2 **have** 3: "$preCond\ i\ y \wedge inv\ x \wedge stateTrans\ i\ y\ x$"
by *simp*
then show "$inv\ x$"
by *simp*
qed

Especificação do sistema

Neste ponto, especificamos o sistema de comando e controle, que é construído com base na especificação da máquina abstrata previamente descrita.

Inicialmente, apresentam-se algumas definições sobre as propriedades e funcionamento do sistema. As primeiras três definições correspondem às condições previamente apresentadas. A quarta, corresponde à invariante e, a última, representa um predicado que determina se um dado estado *st* é um estado inicial.

definition *constraint*1:: "*state* ⇒ *bool*" **where**
"*constraint*1 *st* ≡ ∀*sn tn*.((*sn*, *tn*) ∈ *joined st* ⟶ (*sn* ∈ *senIds st*∧*tn* ∈ *sysIds st*))"

definition *constraint*2:: "*state* ⇒ *bool*" **where**
"*constraint*2 *st* ≡ ∀*sn*.(*sn* ∈ *senIds st* ⟶ (∃!*tn*.(*sn*, *tn*) ∈ *joined st*))"

definition *constraint*3:: "*state* ⇒ *bool*" **where**
"*constraint*3 *st* ≡ ∀*tn*.(*tn* ∈ *sysIds st* ⟶ (∃*sn*.(*sn*, *tn*) ∈ *joined st*))"

definition *invariant*:: "*state* ⇒ *bool*" **where**
"*invariant st* ≡ (*constraint*1*a*; *st*) ∧ (*constraint*2 *st*) ∧ (*constraint*3 *st*)"

definition *isInit*:: "*state* ⇒ *bool*" **where**
"*isInit st* ≡ *senIds* (*st*) = {}"

A axiomatização que segue define algumas propriedades sobre a correlação de registros. A propriedade *corex* afirma que todos os registros de sensor devem ser correlacionado à um registro de sistema. A propriedade *cordecor* postula que não podem haver dois registros *s* e *t* que sejam simultaneamente correlacionados e decorrelacionados.

axiomatization where *corex*: "∀*s*.∃*t*.*correlates s t*"
and *cordecor*: "∀ *s t*.¬(*correlates s t* ∧ *decorrelates s t*)"

Em seguida, faz-se as definições de criação, eliminação e atualização de registros. A criação de um novo registro de sensor consiste em adicioná-lo ao conjunto de sensores de um estado e checar que ao unir este registro de sensor a um registro de sistema a correlação é mantida. A eliminação de um registro de sensor, por outro lado, é lograda por meio da remoção do registro em questão do conjunto de registros do estado e da confirmação que todos os registros de sensor e sistema que permanecem ligados após a remoção já eram

ligados antes dela. Por fim, a atualização de um registro consiste em remover o registro desatualizado do estado e, em seguida, adicionar a sua "versão" atualizada.

definition $newSensTrack$:: "$sensorTrack \Rightarrow$
$sensorTrackState \Rightarrow$
$state \Rightarrow$
$state \Rightarrow$
$bool$"
where "$newSensTrack\ sn\ s\ x\ y \equiv$
$senIds(y) = sn \cup senIds(x) \wedge$
$(\forall tn \in sysIds(y).(sn, tn) \in joined\ y \longrightarrow$
$correlates\ s(sysInfo\ y\ tn))$"

definition $wipeSensTrack$:: "$sensorTrack \Rightarrow state \Rightarrow state \Rightarrow bool$" **where**
"$wipeSensTrack\ sn\ x\ y \equiv$
$senIds(y) = senIds(x) - \{sn\} \wedge$
$sysIds(y) \subseteq sysIds(x) \wedge$
$(\forall s\ t.(s, t) \in joined\ y \longrightarrow (s, t) \in joined\ x)$"

definition $upSensTrack$:: "$sensorTrack \Rightarrow$
$sensorTrackState \Rightarrow$
$state \Rightarrow$
$state \Rightarrow$
$bool$"
where "$upSensTrack\ sn\ s\ x\ y \equiv$
$senIds(y) = senIds(x) \wedge$
$(\forall tn.decorrelates\ s(sysInfo\ y\ tn) \longrightarrow$
$(sn, tn) \notin joined\ y)$"

As precondições são definidas para cada um dos três tipos de evento de entrada: criação, eliminação e atualização. A precondição associada à criação afirma que, para um registro de sensor sn e um estado s, a criação de um novo evento de entrada consistindo nestes dois elementos implica que sn não é um elemento do conjunto de registros de sensor de um estado x e que, para algum registro de sistema tn, tn não faz parte do conjunto de registro de sistema de x. Para a atualização, a precondição afirma que para um evento de entrada de atualização, um dado registro de sensor sn é parte do conjunto de registros de sensor de x e há algum registro de sistema tn que não faz parte do conjunto de registros

de sistema de x. Finalmente, a precondição associada à eliminação de um registro afirma apenas que tal registro faz deve fazer parte do conjunto de registro de sensor do estado x.

definition *preconditions*:: "$inEv \Rightarrow state \Rightarrow bool$" **where**
"$preconditions\ ie\ x \equiv$
case ie **of**
$newInEv\ sn\ s \Rightarrow sn \notin senIds(x) \wedge (\exists tn.tn \notin sysIds(x))|$
$upInEv\ sn\ s \Rightarrow sn \in senIds(x) \wedge (\exists tn.tn \notin sysIds(x))|$
$wipeInEv\ sn \Rightarrow sn \in senIds(x)$"

Agora, empreende-se as provas necessárias para verificar que a especificação obedece às propriedades esperadas. A primeira prova, feita com o auxílio de mecanismos de automação, afirma que, para quaisquer st, sn e s, caso st respeite a invariante e os três termos respeitem a precondição associada à um evento de entrada, então há algum registro $st1$ tal que este respeite à invariante à condição de criação de registros $newSensTrack$.

lemma *newNext*:
fixes $st\ sn\ s$
assumes "$invariant\ st$"
and "$preconditions(newInEv\ sn\ s)st$"
shows "$\exists st1.invariant\ st1 \wedge newSensTrack\ sn\ s\ st\ st1$"
proof -
from $corex$ **have** 3: "$\exists t.correlates\ s\ t$"
by $auto$
then obtain t **where** "$correlates\ s\ t$"
by $auto$
from $assms$ **obtain** tn **where** "$tn \notin sysIds\ st$"
using $preconditions\_def$ **by** $auto$
define new **where** "$new = (senIds = insert\ sn(senIds\ st),$
$sysIds = insert\ tn(sysIds(st)),$
$sysInfo = (sysInfo\ st)(tn := t),$
$joined = insert(sn, tn)(joined\ st))$"
have "$invariantnew$"
using $assms(1)$
$assms(2)$
$constraint1\_def$
$constraint2\_def$

*constraint*3_*def*
insert_iff
invariant_def
new_def
preconditions_def
by *auto*
then have "*newSensTrack sn s st new*"
using ‹*correlates s t*›
assms(1)
assms(2)
*constraint*1_*def*
invariant_def
newSensTrack_def
new_def
preconditions_def
by *auto*
show ?*thesis*
using ‹*invariantnew*›
‹*newSensTracksn s st new*›
by *auto*
qed

Esta prova foi reescrita tal qual a apresentação abaixo com o intuito de melhorar a sua legibilidade. Novamente, a prova foi reescrita de maneira que, entre cada passo que é explicitamente escrito, há menos passos implícitos. Também foi empregada a estratégia de utilizar a enunciação explícita de algumas propriedades onde estas são usadas, para que fique visualmente claro como uma fórmula pode ser deduzida a partir da propriedade usada em tal dedução.

lemma *newNext*:
fixes *st sn s*
assumes "*invariant st*"
and "*preconditions*(*newInEv sn s*)*st*"
shows "∃*st*1.*invariant st*1 ∧ *newSensTrack sn s st st*1"
proof -
have 1: "∃*t*.*correlates s t*" **using** *corex*
by *simp*

then obtain t **where** 2: "$correlates\ s\ t$" **by** $auto$
obtain tn **where** 3: "$tn \notin sysIds\ st$"
using $assms\ preconditions\_def$ **by** $auto$
define new **where** 4: "$new = (senIds = insert\ sn(senIds\ st),$
$sysIds = insert\ tn(sysIds(st)),$
$sysInfo = (sysInfo\ st)(tn := t),$
$joined = insert(sn, tn)(joined\ st))$"
have 5: "$constraint1st \wedge constraint2st \wedge constraint3st$"
using $assms(1)\ invariant\_def$ **by** $blast$
have 6: "$constraint1\ st$"
by $(simp\ add$: "5")
have 7: "$''\forall sn\ tn.(sn, tn) \in joined\ new \longrightarrow\ sn \in senIds\ new \wedge tn \in sysIds\ new$"
using "4" "6" $constraint1\_def$ **by** $auto$
have 8: "$constraint1\ new$"
by $(simp\ add$: "7" $constraint1\_def)$
have 9: "$sn \notin senIds\ st \wedge (\exists tn.tn \notin sysIds\ st)$"
using $assms(2)\ preconditions\_def$ **by** $auto$
have 10: "$constraint2\ st$"
by $(simp\ add$: "5")
have 11: "$\exists!\ tn.(sn, tn) \in joined\ new$"
using "4" "6" "9" $constraint1\_def$ **by** $auto$
have 12: "$sn \in senIds\ new \longrightarrow (\exists! tn.(sn, tn) \in joined\ new)$"
by $(simp\ add$: "11")
have 13: "$\forall sn.sn \in senIds\ new \longrightarrow (\exists! tn.(sn, tn) \in joined\ new)$"
using "10" "11" "4" $constraint2\_def$ **by** $auto$
have 14: "$constraint2\ new$"
by $(simp\ add$: "13" $constraint2\_def)$
have 16: "$\forall tn.tn \in sysIds\ new \longrightarrow (\exists sn.(sn, tn) \in joined\ new)$"
using "4" "5" $constraint3\_def$ **by** $auto$
have 17: "$constraint3\ new$"
using "16" "4" $constraint3\_def$ **by** $auto$
have 18: "$constraint1\ new \wedge constraint2\ new \wedge constraint3\ new$"
by $(simp\ add$: "14" "17" "8")
have 19: "$invariant\ new$"
by $(simp\ add$: "14" "17" "8" $invariant\_def)$
have 20: "$senIds\ new = \{sn\} \cup senIds\ st$"
by $(simp\ add$: "4")

have 21: "$\forall tn \in sysIds\ new.(sn, tn) \in joined\ new \longrightarrow correlates\ s\ (sysInfo\ new\ tn)$"
using "12" "2" "4" **by** *auto*
have 22: "$senIds\ new = \{sn\} \cup senIds\ st \wedge$
$(\forall tn \in sysIds\ new.(sn, tn) \in joined\ new \longrightarrow$
$correlates\ s(sysInfo\ new\ tn))$"
using "20" "21" **by** *blast*
have 23: "$newSensTrack\ sn\ s\ st\ new$"
using "20" "21" $newSensTrack\_def$ **by** *blast*
have 24: "$invariant\ new \wedge newSensTrack\ sn\ s\ st\ new$"
by ($simp\ add$: "19" "23")
show ?*thesis*
using "24" **by** *blast* **qed**

A próxima prova consiste em mostrar que, para *st* e *sn* quaisquer e, assumindo que a invariante vale para *st* e a precondição referente à eliminação de registros quanto a estes termos vale, então existe algum registro $st1$ que respeita a invariante e para o qual a condição de eliminação de registros vale.

lemma $wipeNext$:
fixes *st sn*
assumes "$invariant\ st$"
and "$preconditions\ (wipeInEv\ sn)\ st$"
shows "$\exists st1.invariant\ st1 \wedge wipeSensTrack\ sn\ st\ st1$"
proof -
have "$sn \in senIds\ st$" **using** $assms(2)\ preconditions\_def$ **by** *auto*
define $removedSysIds$ **where** "$removedSysIds = \{tns.tns \in sysIds\ st \wedge$
$(\forall sns.(sns, tns) \in joined\ st \longrightarrow\ sns = sn)\}$"
define $newSt$ **where** "$newSt = (senIds = senIdsst - \{sn\},$
$sysIds = \{tn.tn \in sysIds(st) \wedge tn \notin removedSysIds\},$
$sysInfo = (sysInfost),$
$joined = \{(sn', tn').(sn', tn') \in joinedst \wedge$
$sn' \neq sn \wedge$
$tn' \notin removedSysIds\})$"
have $c1$: "$constraint1\ newSt$"
using $assms(1)\ constraint1\_def\ invariant\_def\ newSt\_def$ **by** *auto*
have $c2$: "$constraint2\ newSt$"
using $assms(1)$

case_prod_conv
constraint2_def
invariant_def
newSt_def
removedSysIds_def
by *force*
have *c*3: "*constraint*3 *newSt*"
by (*simp add* : *constraint*3_*def newSt_def removedSysIds_def*)
have "*invariant newSt*" **using** *invariant_def* "*c*1" "*c*2" "*c*3" **by** *simp*
have "*wipeSensTrack sn st newSt*"
using *newSt_def wipeSensTrack_def* **by** *auto*
show ?*thesis*
using ‹*invariant newSt*› ‹*wipeSensTrack sn st newSt*› **by** *auto*
qed

Esta prova foi reescrita com o intuito de melhorar a sua legibilidade. Assim como na última prova, esta foi reescrita de forma que a largura de cada passo da dedução é reduzido a uma quantidade mínima de operações determinada *ad hoc*. Novamente foi empregada a estratégia de enunciar explicitamente algumas propriedades perto do passo onde é usada, Desta forma, fica claro como um determinado fato produz o próximo por meio da regra de inferência aplicada.

lemma *wipeNext*:
fixes *st sn*
assumes "*invariant st*"
and "*preconditions* (*wipeInEv sn*) *st*"
shows "∃*st*1.*invariant st*1 ∧ *wipeSensTrack sn st st*1"
proof -
have 1: "*sn* ∈ *senIds st*" **using** *assms*(2) *preconditions_def* **by** *auto*
define *removedSysIds* **where** 2:"*removedSysIds* = {*tns*.*tns* ∈ *sysIds st* ∧
(∀*sns*.(*sns*, *tns*) ∈ *joined st* ⟶ *sns* = *sn*)}"
define *newSt* **where** 3:"*newSt* = (*senIds* = *senIds st* − {*sn*},
sysIds = {*tn*.*tn* ∈ *sysIds*(*st*) ∧ *tn* ∉ *removedSysIds*},
sysInfo = (*sysInfo st*),
joined = {(*sn*′, *tn*′).(*sn*′, *tn*′) ∈ *joined st* ∧ *sn*′ ≠ *sn* ∧ *tn*′ ∉ *removedSysIds*})"
have 4: "*constraint*1 *st* ∧ *constraint*2 *st* ∧ *constraint*3 *st*"
using *assms*(1) *invariant_def* **by** *blast*

have 5: "$constraint1\ st$"
by ($simp\ add$: "4")
have 6: "$\forall sn\ tn.(sn, tn) \in joined\ newSt \longrightarrow sn \in senIds\ newSt \wedge tn \in sysIds\ newSt$"
using "3" "5" $constraint1\_def$ **by** $auto$
have 7: "constraint1 newSt"
by ($simp\ add$: "6" $constraint1\_def$)
have 8: "$\forall sn.sn \in senIds\ newSt \longrightarrow (\exists! tn.(sn, tn) \in joined\ newSt)$"
using "2" "3" "4" $constraint2\_def$ **by** $force$
have 9: "$constraint2\ newSt$"
by ($simp\ add$: "8" $constraint2\_def$)
have 10: "$\forall tn.tn \in sysIds\ newSt \longrightarrow (\exists sn.(sn, tn) \in joined\ newSt)$"
by (simp add: "23")
have 11: "$constraint3\ newSt$"
by ($simp\ add$: "10" $constraint3\_def$)
have 12: "$invariant\ newSt$"
by ($simp\ add$: "11" "7" "9" $invariant\_def$)
have 13: "$wipeSensTrack\ sn\ st\ newSt$"
by ($simp\ add$: "3" $wipeSensTrack\_def$)
show $?thesis$
using "12""13" **by** $auto$
qed

A próxima propriedade provada diz respeito à precondição relativa à atualização de registros. O teorema demonstrado afirma que, para st, sn e s quaisquer, assumindo que a invariante é válida quanto a st e a precondição relativa à atualização de registros vale para st, sn e s, então existe algum registro $st1$ tal que a invariante vale para este e $upSensTrack$ vale para sn, s, st e $st1$.

lemma $updateNext$:
fixes $st\ sn\ s$
assumes "$invariant\ st$"
and "$preconditions\ (upInEv\ sn\ s)\ st$"
shows "$\exists st1.invariant\ st1\ \wedge upSensTrack\ sn\ s\ st\ st1$"
proof -
from $assms$ **have** 1: "$sn \in senIds\ st \wedge (\exists tn.tn \notin sysIds\ st)$"
using $preconditions\_def$ **by** $auto$
then have $cWipe$: "$sn \in senIds\ st$" **by** $auto$

have *cNext*: "(∃*tn.tn* ∉ *sysIds st*)" **using** "1" **by** *auto*
then obtain *tn′* **where** *cNew*: "*tn′* ∉ *sysIds st*" **by** auto
have "∃ *stWipe.invariant stWipe* ∧ *wipeSensTrack sn st stWipe*"
using *assmspreconditions_def cWipe wipeNext* **by** *simp*
then obtain *stWipe* **where** *AssmNew*1: "*invariant stWipe*∧*wipeSensTrack sn st stWipe*"
by *auto*
then have *AssmNew*2: "*preconditions* (*newInEv sn s*) *stWipe*"
using *cNew preconditions_def wipeSensTrack_def* **by** *auto*
show ?*thesis* **using** *AssmNew*1 *AssmNew*2 *newNext invariant_def*
newSensTrack_def upSensTrack_def wipeSensTrack_def cordecor
by (*smt*(*verit*, *del_insts*)
cWipe
*constraint*1*_def*
insert_Diff_single
isert_absorb
insert_is_Un)
qed

Esta prova foi reescrita como as anteriores. Novamente, as estratégias adotadas para melhorar a legibilidade da prova consistem em tornar os passos da prova menores e os passos dedutivos mais diretos em relação aos que se seguem. Além disso, novamente, mostrou-se proveitoso mostrar explicitamente como uma determinada definição é usada para construir a fórmula em um determinado passo da dedução.

lemma *updateNext*:
fixes *st sn s*
assumes "*invariant st*"
and "*preconditions* (*upInEv sn s*) *st*"
shoes "∃*st*1.*invariant st*1 ∧ *upSensTrack sn s st st*1"
proof -
have 1: "*sn* ∈ *senIds st* ∧ (∃*tn.tn* ∉ *sysIds st*)" **using** *assms preconditions_def* **by** *auto*
have 2: "*sn* ∈ *senIds st*" **using** "1" **by** *auto*
have 3: "(∃*tn.tn* ∉ *sysIds st*)" **using** "1" **by** *auto*
obtain *tn′* **where** 4: "*tn′* ∉ *sysIds st*" **using** "3" **by** *auto*
have 5: "∃*stWipe.invariant stWipe* ∧ *wipeSensTrack sn st stWipe*"
using *assms preconditions_def* 2 *wipeNext* **by** *simp*

obtain *stWipe* **where** 6: "*invariant stWipe* ∧ *wipeSensTrack sn st stWipe*" **using** "5" **by** *auto*

have 7: "*wipeSensTrack sn st stWipe*"

using "6" **by** *blast*

have 8: "*senIds stWipe* = *senIds st* − {*sn*} ∧ *sysIds stWipe* ⊆ *sysIds st* ∧ (∀*s t*.(*s*, *t*) ∈ *joined stWipe* ⟶ (*s*, *t*) ∈ *joined st*)"

using "7"*wipeSensTrack_def* **by** *force*

have 9: "*senIds stWipe* = *senIdsst* − {*sn*}"

using "8" **by** *auto*

have 10: "*sn* ∉ *senIds stWipe*"

by (*simp add* : "9")

have 11: "*sysIds stWipe* ⊆ *sysIds st*"

by (*simp add* : "8")

have 12: "(∃*tn*.*tn* ∉ *sysIds stWipe*)"

using "3" "11" **by** *blast*

have 13: "*sn* ∉ *senIds stWipe* ∧ (∃*tn*.*tn* ∉ *sysIds stWipe*)"

by (*simp add* : "10""12")

have 14: "*preconditions* (*newInEv sn s*) *stWipe*"

by (*simp add* : "13" *preconditions_def*)

show "∃*st*1.*invariant st*1 ∧ *upSensTrack sn s st st*1"

by (*smt*(*verit*) "14" "2" "6"

*constraint*1*_def*

cordecor

insert_Diff

insert_is$_U$*n*

invariant_def

newNext

newSensTrack_def

upSensTrack_def

wipeSensTrack_def)

qed

Finalmente, a interpretação completa a especificação do sistema refinado. Esta construção instancia a teoria paramétrica da máquina abstrata no contexto da especificação do sistema. Isto se segue então de uma prova mostrando que as condições da teoria se mantém com os parâmetros utilizados na instanciação.

interpretation $sp1 : stateMachineLcl$
$inputTable$
$outputTable$
$isInit$
$invariant$
$preconditions$
proof -
define $myInitialState$ **where** "$myInitialState = (senIds = \{\},$
$sysIds = \{\},$
$sysInfo = \lambda tn.(SOME\ t.True),$
$joined = \{\})$"
then have $c1State$: "$constraint1\ myInitialState$"
by $(simp\ add : constraint1\_def\ myInitialState\_def)$
then have $c2State$: "$constraint2\ myInitialState$"
by $(simp\ add : constraint2\_def\ myInitialState\_def)$
then have $c3State$: "$constraint3\ myInitialState$"
by $(simp\ add : constraint3\_def\ myInitialState\_def)$
then have $invState$: "$invariant\ myInitialState$"
using $c1State\ c2State\ c3State\ invariant\_def$ **by** $auto$
then have $initState$: "$isInit\ myInitialState$"
by $(simp\ add : isInit\_def\ myInitialState\_def)$
then have $eq\_pvs\_TCC\_1$: "$\exists x.isInit\ x \wedge invariant\ x$"
using $invState\ initState$ **by** $auto$
then have $eq\_pvs\_TCC\_2$: "$\forall i\ x.preconditions\ i\ x \wedge invariant\ x \longrightarrow$
$(\exists y.invariant\ y \wedge inputTable\ i\ x\ y)$"
using $newNext\ wipeNext\ updateNext\ preconditions\_def\ invariant\_def\ inputTable\_def$
by $(smt\ (verit, best)\ inEv.exhaust\ inEv.simps(10)\ inEv.simps(11)\ inEv.simps(12))$
then show "$stateMachineLcl\ inputTable\ isInit\ invariant\ preconditions$"
using $eq\_pvs\_TCC\_1\ eq\_pvs\_TCC\_2\ stateMachineLcl.intro$ **by** $simp$
qed

A reescrita desta prova segue a mesma linha das anteriores. Nesta prova, os passos da prova foram construídos e organizados com o intuito de tornar o mais claro possível o encadeamento lógico na dedução. O enunciado dos fatos foi escolhido cuidadosamente para guiar o leitor no processo de compreender a maneira como os pressupostos do teorema podem ser utilizados para construir a conclusão por meio de um processo de análise de seus componentes e síntese destes nas fórmulas seguintes da dedução.

interpretation $sp1 : stateMachineLcl$
$inputTable$
$outputTable$
$isInit$
$invariant$
$preconditions$
proof -
define $myInitialState$ **where** 1:"$myInitialState = (senIds = \{\},$
$sysIds = \{\},$
$sysInfo = \lambda tn.(SOME\ t.True),$
$joined = \{\})$"
have 2: "$\forall sn\ tn.(sn, tn) \in joined\ myInitialState \longrightarrow$
$sn \in senIds\ myInitialState \wedge$
$tn \in sysIds\ myInitialState$"
by ($simp\ add$: "1")
have 3: "$constraint1\ myInitialState$"
by ($simp\ add$: "2"$constraint1\_def$)
have 4: "$\forall sn.sn \in senIds\ myInitialState \longrightarrow$
$(\exists! tn.(sn, tn) \in joined\ myInitialState)$"
by ($simp\ add$: "1")
have 5: "$constraint2\ myInitialState$"
by ($simp\ add$: "4" $constraint2\_def$)
have 6: "$\forall tn.tn \in sysIds\ myInitialState \longrightarrow$
$(\exists sn.(sn, tn) \in joined\ myInitialState)$"
by ($simp\ add$: "1")
have 7: "$constraint3\ myInitialState$"
by ($simp\ add$: "6" $constraint3\_def$)
have 8: "$constraint1\ myInitialState \wedge$
$constraint2\ myInitialState \wedge$
$constraint3\ myInitialState$"
by ($simp\ add$: "3" "5" "7")
have 9: "$invariant\ myInitialState$"
using 3 5 7 $invariant\_def$ **by** $auto$
have 10: "$isInit\ myInitialState$"
by ($simp\ add$: "1" $isInit\_def$)
have 11: "$isInit\ myInitialState \wedge invariant\ myInitialState$"

by (*simp add* : "10" "9")
have *eq_pvs_TCC_1*: "$\exists x.isInit\ x \wedge invariant\ x$"
using "11" **by** auto
then have *eq_pvs_TCC_2*: "$\forall i\ x.preconditions\ i\ x \wedge invariant\ x \longrightarrow (\exists y.invariant\ y \wedge inputTable\ i\ x\ y)$"
using *newNext*
wipeNext
updateNext
preconditions_def
invariant_def
inputTable_def
by (*smt* (*verit*, *best*) *inEv.exhaust inEv.simps*(10) *inEv.simps*(11) *inEv.simps*(12))
then show "*stateMachineLcl inputTable isInit invariant preconditions*"
using *eq_pvs_TCC_1 eq_pvs_TCC_2 stateMachineLcl.intro* **by** *simp*
qed

Desta maneira, fica ilustrado como as técnicas de aprimoramento da legibilidade de provas apresentadas no último capítulo puderam ser aplicadas ao problema da verificação do sistema de comando e controle aqui aplicado. Por meio da reescrita das provas seguindo estas boas práticas de legibilidade, foram produzidas provas cuja leitura e entendimento são mais diretos e intuitivos e menos dependentes na familiaridade com o sistema Isabelle e seus mecanismos de automação do processo dedutivo.

Capítulo 6

Conclusão

Neste trabalho buscou-se melhor compreender o processo de produção de provas auxiliado por sistemas computacionais. Buscou-se também melhor entender a interação entre o usuário deste tipo de sistema, agora no papel de leitor, e as provas neles produzidas. Este último aspecto representa o foco central do trabalho.

Foi adotada uma abordagem em camadas. Inicialmente o foco recaiu sobre a lógica de ordem superior e dedução natural, que constituem a infraestrutura teórica sobre a qual se apoia todo o resto. Esta etapa do estudo aqui relatado demonstra como novo conhecimento pode ser produzido a partir da manipulação criteriosa de fatos já estabelecidos.

Em seguida, foi introduzida a ferramenta Isabelle, que permite aumentar a produtividade do trabalho com a lógica, assim como conferir também segurança nos resultados, que no ambiente Isabelle podem ser formalmente checados. Tendo introduzido os "insumos" necessários, em sua forma teórica e prática, pode-se então voltar a atenção ao problema de legibilidade de provas em si.

Neste ponto, foi estabelecida a importância da legibilidade de provas. Foi também apresentada um pouco da história deste problema. Por fim se estabeleceu, com base na literatura científica disponível, um conjunto de heurísticas que podem ser aplicadas para aprimorar a legibilidade de provas.

Por fim, foi apresentada uma aplicação prática de tudo que foi até então na especificação e verificação de um sistema de comando e controle. Nesta etapa, diversas provas foram produzidas com o intuito de demonstrar que a especificação criada respeita algumas condições predefinidas. A estas provas foram aplicadas as técnicas de aprimoramento de legibilidade previamente discutidas, produzindo assim um resultado mais robusto no que diz respeito à acessibilidade dos resultados a quem que eventualmente os consulte.

Neste trabalho foram abordados, de maneira panorâmica, vários aspectos do trabalho com métodos formais no que diz respeito à verificação de *software*. Foi desenvolvida uma proposta de metodologia na verificação de *software* fortemente embasada na utilização

de ferramentas computacionais que auxiliam e conferem mais eficiência e segurança ao trabalho do desenvolvedor.

Pode-se traçar, ainda, uma possível direção para futuros trabalhos. Ainda há muito a investigar no processo de desenvolvimento de provas legíveis. Um caminho a seguir é certamente a análise comparativa de diferentes estratégias de aprimoramento de legibilidade, que pode ser experimentalmente explorada em um futuro momento. Outra questão interessante é a investigação quanto a possíveis formas de automatizar o processo de aprimoramento da legibilidade, uma vez que critérios e técnicas adequadas tenham sido identificadas e descritas.

Referências

[1] Zammit, Vincent: *On the Readability of Machine Checkable Formal Proofs*. Tese de Doutoramento, University of Kent, Division of Computing, Engineering and Mathematical Sciences, School of Computing, March 1999. `https://kar.kent.ac.uk/21861/`. v, vi, 29, 30

[2] Lamport, Leslie: *How to write a 21st century proof*. Journal of Fixed Point Theory and Applications, 11:43–63, 2012. v, vi, 29, 30

[3] Russell, Stuart J. e Peter Norvig: *Artificial Intelligence: A Modern Approach*. Prentice Hall, NJ, EUA, 4a edição, 2021, ISBN 978-1292401133. 1

[4] McCorduck, Pamela: *Machines Who Think*. A. K. Peters, MA, EUA, 2a edição, 2004, ISBN 1-56881-205-1. 2

[5] *Coq*. https://coq.inria.fr/. Accessed: 2023-02-7. 3

[6] Gonthier, Georges: *Formal proof—the four color theorem*. Notices of the AMS, 55(11):1382 – 1393, 2008. 3

[7] Schellhorn, Gerhard e Wolfgang Ahrendt: *The wam case study: Verifying compiler correctness for prolog with kiv*. Em Bibel, Wolfgang e Peter H. Schmitt (editores): *Automated Deduction — A Basis for Applications: Volume III Applications*, páginas 165–194. Springer Netherlands, Dordrecht, 1998, ISBN 978-94-017-0437-3. `https://doi.org/10.1007/978-94-017-0437-3_7`. 3

[8] *ACL2*. https://www.cs.utexas.edu/users/moore/acl2/. Accessed: 2023-02-7. 3

[9] Moore, J Strother., Thomas W. Lynch e Matt Kaufmann: *A mechanically checked proof of the AMD5K86TM floating-point division program*. IEEE Transactions on Computers, 47(09):913–926, 1998, ISSN 1557-9956. 3

[10] Arora, Sanjeev e Boaz Barak: *Complexity Theory: A Modern Approach*. Cambridge University Press, 2009, ISBN 978-0-521-42426-4. 3

[11] Gödel, Kurt: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. Monatsh. f. Mathematik und Physik, 38:173–198, 1931. 3

[12] Geuvers, Herman: *Proof assistants: History, ideas and future*. Sãdhanã, Academy Proceedings in Engineering Sciences, 34:3–25, 2009. 4

[13] Jaakko, Hintikka e Gabriel Sandu: *What is logic?* Em Jacquette, Dale (editor): *Philosophy of Logic*, páginas 13–39. North Holland, 2006. 5

[14] Bell, John L.: *Higher-Order Logic and Type Theory*. Elements in Philosophy and Logic. Cambridge University Press, 2022. 6, 7, 8

[15] Troelstra, Anne Sjerp e Helmut Schwichtenberg: *Basic Proof Theory*. Cambridge University Press, 2ª edição, 2000, ISBN 0-521-77911-1. 10, 11

[16] *Isabelle*. https://isabelle.in.tum.de/. Accessed: 2023-02-7. 16

[17] *Overview of the isabelle proof assistant*. https://isabelle.in.tum.de/overview.html. Accessed: 2023-01-20. 16, 17

[18] *OCaml*. https://ocaml.org/. Accessed: 2023-02-7. 16

[19] *Haskell*. https://www.haskell.org/. Accessed: 2023-02-7. 16

[20] *Scala*. https://www.scala-lang.org/. Accessed: 2023-02-7. 16

[21] *Archive of formal proofs*. https://www.isa-afp.org/. Accessed: 2023-01-20. 17

[22] Thiele, Ruediger e Larry Wos: *Hilbert's twenty-fourth problem*. Journal of Automated Reasoning, 29:67–89, 2002. 29

[23] Knuth, Donald E.: *Literate programming*. The Computer Journal of the British Computer Society, 27(2):97 – 111, 1984. 30

[24] van de Pol, Jaco, Jozef Hooman e Edwin de Jong: *Requirements specifcation and analysis of command and control systems*. Relatório Técnico, Technische Universiteit Eindhoven, 1999. 31, 32, 33, 35