Universidade de Brasília

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

Verificação de Implementações em Hardware por meio de Provas de Correção de suas Definições Recursivas

por

Ariane Alves Almeida

Brasília
2014

Ariane Alves Almeida

Verificação de Implementações em Hardware por meio de Provas de Correção de suas Definições Recursivas

Dissertação apresentada ao Departamento de Ciência da Computação, para a obtenção de Título de Mestre em Informática

Área: Fundamentos e Métodos de Computação
Subárea: Teoria da Computação, Algoritmos e Métodos Formais

Orientador: Prof. Dr. Mauricio Ayala Rincon

Banca Avaliadora:

Prof. Dr. Mauricio Ayala Rincon

Prof. Dr. Flávio Leonardo Cavalcanti de Moura

Prof. Dr. Daniel Mauricio Muñoz Arboleda

Brasília
Junho, 2014

Alves Almeida, Ariane.

Verificação de Implementações em Hardware por meio de Provas de Correção de suas Definições Recursivas

83 páginas

Dissertação (Mestrado) - Universidade de Brasília. Instituto de Ciências Exatas. Departamento de Ciência da Computação.

1. Verificação Formal;
2. Métodos Formais;
3. Projeto de Hardware;
4. Especificação e Verificação Algébrica.

Agradecimentos

A Deus, pois sem ele nada é possível.

À minha família, em especial meus pais, Tânia e Evaldo, por todo incentivo e motivação que me impulsionaram a alcançar mais esta conquista, entre tantas outras.

Ao meu marido, José Rafael, por toda paciência e compreensão, essenciais quando precisei privá-lo de minha presença para conclusão de mais esta etapa.

Ao meu orientador, Prof. Mauricio Ayala, por toda ajuda e direcionamentos durante estes anos, muito importantes para meu crescimento e conclusão deste trabalho.

Aos professores Daniel Muñoz e Flávio Moura, por participarem da banca avaliadora deste trabalho, fornecendo valiosas contribuições e sugestões para sua correção.

Aos professores da UFG, André Galdino, Liliane Vale e Vaston Costa, pelo suporte, ajuda e incentivo a mim dispendidos.

E também a todos os amigos que direta ou indiretamente contribuíram para este trabalho, tenha sido em momentos de descontração fora do ambiente acadêmico ou contribuindo dentro dele. Em especial aqueles que integram/integraram o LaForCe e o GRACO, grupos envolvidos na realização deste projeto.

À CAPES e à UnB, por todo apoio fornecido.

Resumo

Uma abordagem é apresentada para verificar formalmente a corretude lógica de operadores algébricos implementados em *hardware*. O processo de verificação é colocado em paralelo ao fluxo convencional de projeto de *hardware*, permitindo a verificação de fragmentos da implementação do *hardware* tanto simultaneamente quanto após todo o processo de implementação ser concluído, evitando assim atrasos no projeto do circuito. A ideia principal para atestar a corretude de uma implementação em *hardware* é comparar seu comportamento operacional com uma definição formal de seu operador, analisando assim sua *equivalência funcional*; isto é, se ambas definições, de *hardware* e matemática, produzem os mesmos resultados quando fornecidas as mesmas entradas. A formalização dessa comparação é um desafio desta abordagem, já que as provas utilizadas para verificar a corretude e outras propriedades desses sistemas pode seguir esquemas indutivos, que proveem de maneira natural quando se trata com definições recursivas, usadas em linguagens de especificação e ferramentas de formalização. Já que Linguagens de Descrição de Hardware descrevem circuitos/sistemas de maneira imperativa, a abordagem se baseia na tradução conservativa de comandos iterativos presentes nessas linguagens em suas respectivas especificações recursivas. Esses esquemas de provas indutivas são baseados em garantir pré e pós-condições, bem como a preservação de invariantes durante todos os passos da execução recursiva, de acordo com a abordagem da lógica de Floyd-Hoare para verificação de procedimentos imperativos. A aplicabilidade da metodologia é ilustrada com um caso de estudo utilizando o assistente de prova de ordem superior PVS para fornecer prova de correção lógica de uma implementação em FPGA do algoritmo para inversão de matrizes de Gauss-Jordan (GJ). Essas provas em PVS são dadas em um estilo dedutivo baseado no Cálculo de Gentzen, aproveitando facilidades desse assistente, como tipos dependentes, indução na estrutura de tipos de dados abstratos e, é claro, suas linguagens de especificação e prova em lógica de ordem superior.

Palavras-chave: Verificação Formal, Métodos Formais, Projeto de Hardware, Especificação e Verificação Algébrica.

Abstract

An approach is introduced to formally verify the logical correctness of hardware implementations of algebraic operators. The formal verification process is placed sidelong the usual hardware design flow, allowing verification on fragments of the hardware implementation either simultaneously or after the whole implementation process finished, avoiding in this way hardware development delays. The main idea to state the correctness of a hardware implementation, is to compare its operational behavior with a formal definition of the operator, analysing their *functional equivalence*; that is, if both the hardware and the mathematical definition produce the same results when provided with the same entries. The formalization of this comparison is a challenge for this approach, since the proofs used to verify soundness and other properties of these systems might follow inductive schemata, that arise in a natural manner when dealing with recursive definitions, used in specifications languages of formalization tools. Since Hardware Description Languages describe circuits/systems in an imperative style, the approach is based on a conservative translation of iterative commands into their corresponding recursive specifications. The inductive proof schemata are then based on guaranteeing pre and post-conditions as well as the preservation of invariants during all steps of the recursive execution according to the Floyd-Hoare's logical approach for verification of imperative procedures. The applicability of the methodology is illustrated with a case study using the higher-order proof assistant PVS by proving the logical correction of an FPGA implementation of the Gauss-Jordan matrix inversion algorithm (GJ). These PVS proofs are given in a Gentzen based deductive style taking advantage of nice features of this proof assistant such as dependent types and induction in the structure of abstract data types, and, of course, of its higher-order specification and proof languages.

Keywords: Formal Verificafion, Formal Methods, Hardware Design, Algebraic Specification and Verification.

Sumário

Capítulo 1

Introdução

A complexidade e o tamanho dos sistemas computacionais para uso em diversas aplicações, criados tanto em software quanto em hardware, tem aumentado significativamente a cada dia. Dentre tais aplicações, pode-se destacar aquelas ligadas à engenharia, que utilizam várias operações algébricas para realizar suas funções e demandam grande quantidade de recursos computacionais, como tempo de processamento e consumo de memória. Visando uma maior eficiência para tais aplicações, se busca por abordagens que provêm melhor desempenho, permitindo o projeto de sistemas mais robustos, que apresentem uma maior velocidade de execução.

Uma alternativa para se melhorar o desempenho de uma aplicação, relacionado à sua velocidade de execução, é sua implementação em hardware, com o uso de dispositivos eletrônicos de propósito específico e arquitetura fixa, os ASIC's (*Application-Specific Integrated Circuits*), ou com arquiteturas híbridas, que apresentam tanto características de software quanto de hardware, que é o caso dos dispositivos programáveis de propósito geral de arquitetura reconfigurável, como *Field Programmable Gate Arrays* (FPGA's).

Implementar diretamente um hardware específico para uma funcionalidade faz que sua velocidade de execução seja muito maior que quando executada em um software. Essa melhora se deve à eliminação de um Processador de Propósito Geral (GPP), o que reduz as comunicações entre registros para sua execução, pois não é mais necessário buscar instruções em memória. Isso diminui a limitação de desempenho produzida pela grande quantidade de dados que necessita ser transmitida entre memória e processador, deixando-o ocioso, pois muitas vezes precisa aguardar operações de leitura/escrita destes dados para continuar seu processamento. Tal limitação é conhecida como *Gargalo de Von Newman*. Além disso, não são desperdiçadas instruções extra de processamento necessárias a um GPP e o paralelismo pode ser amplamente explorado.

1.1 Descrição do Problema

Assim, a utilização de implementações em hardware consegue aumentar o desempenho dos sistemas, resta deixá-los confiáveis, já que muitas vezes eles são utilizados em aplicações e ambientes críticos, cujas falhas podem levar a enormes perdas, financeiras

e/ou humanas. Para isso, é necessário que tais sistemas sejam validados de maneira sistemática, e não apenas por meio de simulações e testes. Isso porque ao aumentar o espaço de busca, aumenta-se a possibilidade de que erros sejam encontrados, e a validação de um sistema se dá ao conceber uma hipótese para o que se deseja verificar e não haver refutações quando realizadas análises sobre ele. É neste contexto que se insere a verificação formal, que possibilita a comprovação de que o sistema criado realmente satisfaz as propriedades lógicas e funcionais que se propõe por meio de técnicas dedutivas matemáticas.

1.2 Objetivo e Contribuições

O objetivo geral deste trabalho é propor uma abordagem de verificação formal para implementações em hardware de operadores algébricos e apresentar um caso de estudo para atestar sua viabilidade e utilização.

A formalização proposta se dá por meio de verificação da *equivalência funcional* entre a especificação do hardware produzido para implementar algum operador algébrico e a especificação formal deste operador. Para isso, será modelado o comportamento de ambos e verificado, por meio de provas de lemas que os relacionem, que ao fornecer os mesmos dados de entrada, os resultados produzidos são equivalentes. Essa abordagem objetiva fornecer certificados de garantia para o funcionamento lógico de uma arquitetura, podendo ser realizado em paralelo ao seu desenvolvimento, evitando atrasos do processo de projeto, bem como ao final de criação, validando o circuito e garantindo sua confiabilidade.

1.2.1 Objetivos Específicos

- O uso de tradução entre definições imperativas e definições recursivas, sendo as primeiras utilizadas para descrever hardware e as últimas para linguagens funcionais de especificação de assistentes de prova. Tais traduções são essenciais para prover meios de aplicar provas indutivas em funções baseadas em estruturas de repetição, bastante utilizadas em implementações de hardware. Também analisa-se o uso da Lógica de Floyd-Hoare para verificar a correção de tais estruturas e sua relação com provas indutivas;

- O uso combinado do Cálculo de Gentzen, esquemas de indução estrutural e elementos da lógica de ordem superior para modelagem;

- A aplicação dessas técnicas em um caso de estudo que formaliza uma implementação em hardware reconfigurável com FPGAs do algoritmo de Gauss-Jordan para inversão de matrizes com o assistente *Prototype Verification System* (PVS), atestando sua equivalência funcional em relação a esse algoritmo e também a viabilidade da abordagem proposta.

1.3 Organização do Trabalho

Para melhor entendimento do trabalho, inicialmente são apresentados conceitos básicos de álgebra linear e hardware reconfigurável no Capítulo 2. Nele também são abordados mecanismos de prova presentes no assistente de prova PVS, que é apresentado no Capítulo 3. No Capítulo 4 é apresentada a abordagem proposta neste trabalho, juntamente com sua contextualização e trabalhos relacionados. Então, um caso de estudo é apresentado no Capítulo 5 para exemplificar seu uso. Por fim, o Capítulo 6 traz as discussões e propostas de trabalho futuro.

Capítulo 2

Fundamentação Teórica

Este capítulo introduz alguns conceitos básicos de álgebra linear, arquiteturas de hardware e mecanismos de prova, necessários para se compreender a abordagem utilizada para verificação de sistemas de hardware, bem como do caso de estudo apresentado.

2.1 Álgebra Linear

Aqui são tratados conceitos e operações básicas de dois elementos essenciais da álgebra linear: matrizes e sistemas de equações lineares (adaptados de Boldrini (1986)). Também é apresentado o algoritmo de Gauss-Jordan para resolução de sistemas lineares e inversão de matrizes.

2.1.1 Matrizes e suas Operações

Muito utilizada para solução de problemas matemáticos e também como estrutura de dados em computação, uma matriz $A_{m\times n}$ é uma estrutura retangular com m linhas e n colunas (dimensão da matriz) de elementos $a_{i,j}$, onde $1 \leq i \leq m$ representa a i-ésima linha e $1 \leq j \leq n$ a j-ésima coluna dessa matriz, dando assim a posição do elemento na estrutura (cf. Equação 2.1).

$$A_{m\times n} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ddots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{2.1}$$

Operações entre duas matrizes $A_{m\times n}$ e $B_{m\times n}$, como soma e subtração, são feitas apenas pela aplicação dessas operações a cada um de seus respectivos elementos, ou seja, $(A \pm B)_{i,j} = A_{i,j} \pm B_{i,j}$. Para estas operações basta que as matrizes tenham a mesma dimensão. A multiplicação de matrizes, porém, exige que as dimensões das matrizes sejam tais que, para $A_{m\times n}$ e $B_{r\times t}$, onde $n = r$, formando a matriz $AB_{m\times t}$. Isso porque a multiplicação de matrizes é bem definida da forma da Equação 2.2.

$$ab_{i,j} = \sum_{k=1}^{n} (A_{i,k} * B_{k,j}) \tag{2.2}$$

Exemplo 2.1.1 (Multiplicação de matrizes)**.**

$$\begin{bmatrix} 2 & 1 & 3 \\ 0 & 2 & 4 \\ 5 & 4 & 1 \\ 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 5 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 20 & 22 \\ 26 & 26 \\ 22 & 27 \\ 5 & 7 \end{bmatrix}$$

Quando se tem uma matriz $A_{n\times m}|n = m$, diz-se que é uma *matriz quadrada*, denotada por A_n. Esse tipo de matriz possui propriedades e particularidades relevantes para este trabalho, como a presença de uma *diagonal principal* (DP), que é composta pelos elementos $a_{i,j}|i = j$. A presença da diagonal principal permite a particularização de matrizes quadradas tais como:

- **Matriz Diagonal:** onde todos os elementos que não pertencem à diagonal principal possuem valor zero;
- **Matriz Escalar:** matriz diagonal onde todo elemento $a_{i,i} = c$;
- **Matriz Identidade:** denotada por I_n, é uma matriz escalar onde $c = 1$;
- **Matriz Triangular Superior:** matriz onde todo $A_{i,j}|i > j = 0$;
- **Matriz Triangular Inferior:** matriz onde todo $A_{i,j}|i < j = 0$;

Outra particularidade interessante de matrizes quadradas, é que elas podem possuir sua *inversa*, ou seja, uma matriz A_n^{-1} tal que $A_n^{-1} \cdot A_n = A_n \cdot A_n^{-1} = I_n$ (cf. Exemplo 2.1.2). Tais matrizes são ditas *inversíveis* ou *não singulares*.

Exemplo 2.1.2 (Matriz Inversa)**.**

$$A_3 = \begin{bmatrix} 3 & 3 & 4 \\ 2 & 0 & 1 \\ 6 & 2 & 4 \end{bmatrix} \qquad A_3^{-1} = \begin{bmatrix} \frac{-1}{2} & -1 & \frac{3}{4} \\ \frac{-1}{2} & -3 & \frac{5}{4} \\ 1 & 3 & \frac{-3}{2} \end{bmatrix} \qquad \Longrightarrow \qquad A_3 \cdot A_3^{-1} = I_3$$

Matrizes inversíveis tem grande aplicação prática no âmbito da engenharia, economia, geologia e outras áreas. Isso faz com que métodos para encontrar a inversa de uma matriz inversível (*inversão*) sejam muito estudados, dentre eles, o método de Gauss-Jordan, que será visto na próxima seção.

2.1.2 Sistemas de Equações Lineares

Muito utilizadas principalmente para resolver problemas de otimização, em engenharia e outras áreas, equações lineares são aquelas que podem ser escritas da forma:

$$a_1 * x_1 + a_2 * x_2 + a_3 * x_3 + ... + a_n * x_n = c \tag{2.3}$$

onde $a_1, a_2, \cdots a_n$, $n > 0$, são os coeficientes de suas respectivas incógnitas $x_1, x_2, \cdots x_n$, e c é o termo independente da equação (constante). Note que uma equação linear pode ser reescrita, sem prejuízo, como:

$$a_1 * x_1 + a_2 * x_2 + a_3 * x_3 + ... + a_n * x_n + 1 * -c = 0 \tag{2.4}$$

Assim, um sistema de equações lineares pode ser representado como matrizes:

$$A_{m\times n} \cdot X_{n\times 1} = C_{n\times 1} \tag{2.5}$$

$$A'_{m\times n+1} \cdot X_{n+1\times 1} = 0 \tag{2.6}$$

Sendo 2.3 e 2.4 respectivamente equivalentes a 2.5 e 2.6, onde A é a matriz que contém os coeficientes, X as incógnitas e C os termos independentes das equações $A_i \cdot X = C_i$ que formam o sistema. Do mesmo modo, A' é a matriz A estendida com os termos independentes contidos em C com incógnitas X.

Um conjunto de equações lineares em que nenhuma delas pode ser obtida através de *combinações lineares* das outras, isto é, através de operações de soma de uma delas com produto de outra por um escalar, é dito *linearmente independente*. Sobre esses conjuntos, é possível aplicar *operações elementares*, ou seja, uma operação elementar que aplicada às linhas ou colunas de uma matriz, não altera sua independência linear. Essas operações podem ser:

- **Troca de linhas:** Dadas duas linhas k e l, transforma uma matriz $A_{m\times n}$ em outra $A'_{m\times n}$ onde:

$$A'_{i,j} = \begin{cases} A_{i,j} & \text{Se } i \neq k \text{ e } i \neq l \\ A_{k,j} & \text{Se } i = l \\ A_{l,j} & \text{Se } i = k \end{cases}$$

  Exemplo 2.1.3 (Troca de linhas)**.** Na matriz abaixo a linha de índice $i = 1$ é trocada pela linha de índice $i = 3$.

$$\begin{bmatrix} 3 & 7 & 4 & 0 \\ 1 & 5 & 6 & 2 \\ 5 & 4 & 9 & 3 \\ 7 & 2 & 9 & 1 \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} 5 & 4 & 9 & 3 \\ 1 & 5 & 6 & 2 \\ 3 & 7 & 4 & 0 \\ 7 & 2 & 9 & 1 \end{bmatrix}$$

- **Multiplicação de uma linha por um escalar diferente de zero:** Dado uma linha k e um escalar $c|c \neq 0$, transforma uma matriz $A_{m\times n}$ em outra $A'_{m\times n}$ onde:

$$A'_{i,j} = \begin{cases} A_{i,j} & \text{Se } i \neq k \\ A_{i,j} * c & \text{Se } i = k \end{cases}$$

Exemplo 2.1.4 (Multiplicação de linha por escalar diferente de zero)**.** A primeira linha é multiplicada pelo escalar 3.

$$\begin{bmatrix} 1 & 0 & 4 & 3 \\ 1 & 5 & 4 & 2 \\ 6 & 1 & 9 & 3 \\ 7 & 2 & 3 & 1 \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} 3 & 0 & 12 & 9 \\ 1 & 5 & 4 & 2 \\ 6 & 1 & 9 & 3 \\ 7 & 2 & 3 & 1 \end{bmatrix}$$

- **Soma de uma linha pelo produto de outra linha por um escalar diferente de zero:** Dadas duas linhas k e l, e um escalar c, transforma uma matriz $A_{m\times n}$ em outra $A'_{m\times n}$ onde:

$$A'_{i,j} = \begin{cases} A_{i,j} & \text{Se } i \neq k \\ A_{k,j} + A_{l,j} * c & \text{Se } i = k \end{cases}$$

Exemplo 2.1.5 (Soma de uma linha pelo produto de outra linha por um escalar diferente de zero)**.** A terceira linha é somada ao produto da segunda linha por 3.

$$\begin{bmatrix} 1 & 5 & 2 & 3 \\ 2 & 0 & 3 & 4 \\ 5 & 1 & 4 & 6 \\ 7 & 2 & 8 & 1 \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} 1 & 5 & 2 & 3 \\ 2 & 0 & 3 & 4 \\ 11 & 1 & 13 & 18 \\ 7 & 2 & 8 & 1 \end{bmatrix}$$

Quando uma matriz $A_{m\times n}$ pode ser obtida a partir de operações elementares a partir de uma matriz $B_{m\times n}$, diz-se que A e B são *linha-equivalentes*. Como operações elementares são utilizadas para resolver sistemas lineares (cf. 2.1.3), é importante observar que sua aplicação gera sistemas equivalentes, ou seja, que possuem exatamente a mesma solução, o que é dado pelo Teorema 2.1.1.

Teorema 2.1.1 (Equivalência sob operações elementares)**.** Sejam A e B matrizes linha-equivalentes, os sistemas de equações lineares $A \cdot X = 0$ e $B \cdot X = 0$ tem exatamente a mesma solução.

Demonstração. A prova deste teorema é dada demonstrando que as operações elementares que transformam A em B não alteram o conjunto de soluções. Supondo que B foi obtida de A pela aplicação de apenas uma operação elementar, tem-se:

1. **Troca de linhas:** Seja B o resultado de trocar as linhas i e k de A. Ao descrever os sistemas como $A \cdot X = 0$ e $B \cdot X = 0$, a única diferença observada entre os dois será que a equação representada pela i-ésima linha de $A \cdot X$ será correspondente à k-ésima linha de $B \cdot X$, e vice-versa. Ou seja, as equações obtidas são as mesmas, apenas organizadas de maneira diferente, o que produz o mesmo espaço de soluções.

2. **Multiplicação de linha por escalar:** Sendo B obtida de A pela multiplicação de um escalar $k|k \neq 0$ na i-ésima linha. Como as outras linhas não sofreram alterações, basta verificar a solução da i-ésima equação produzida por B. Seja

$$\begin{bmatrix} A_{i,1} & A_{i,2} & \cdots & A_{i,n} & -C_i \end{bmatrix}$$

a i-ésima linha de A, a equação resultante de $A_i \cdot X$ é dada por

(a) $x_1 * A_{i,1} + x_2 * A_{i,2} + \cdots + x_n * A_{i,n} + 1 * -C_i = 0$
Assim, a i-ésima linha de B é dada por

$$\begin{bmatrix} k * A_{i,1} & k * A_{i,2} & \cdots & k * A_{i,n} & k * -C_i \end{bmatrix}$$

Produzindo como resultado de $B_i \cdot X$ a equação

(b) $x_1 * (k * A_{i,1}) + x_2 * (k * A_{i,2}) + \cdots + x_n * (k * A_{i,n}) + 1 * (k * -C_i) = 0$

Que, por ser equivalente a **(a)** possui a mesma solução que esta, o que mantém o conjunto de soluções do sistema.

3. **Soma de uma linha pelo produto de outra por um escalar:** Seja B obtida de A pela soma da linha i multiplicada pelo escalar k à linha l. Como as outras linhas não sofreram alterações, basta verificar a solução da l-ésima equação produzida por B. Seja

$$\begin{bmatrix} A_{l,1} & A_{l,2} & \cdots & A_{l,n} & -C_l \end{bmatrix}$$

a l-ésima linha de A, a equação resultante de $A_l \cdot X$ é dada por

(a) $x_1 * A_{l,1} + x_2 * A_{l,2} + \cdots + x_n * A_{l,n} + 1 * -C_l = 0$
Assim, a l-ésima linha de B é dada por

$$\begin{bmatrix} A_{l,1} + k * A_{i,1} & A_{l,1} + k * A_{i,2} & \cdots & A_{l,i}k * A_{i,n} & C_l + k * -C_i \end{bmatrix}$$

Produzindo como resultado de $B_l \cdot X$ a equação

(b) $x_1 * (A_{l,1} + k * A_{i,1}) + x_2 * (A_{l,2} + k * A_{i,2}) + \cdots + x_n * (A_{k,n} + k * A_{i,n}) + 1 * (C_l + k * -C_i) = 0$

Que, por manter o espaço de solução, é equivalente a **(a)**, o que mantém o conjunto de soluções do sistema.

□

Conjuntos de equações linearmente independentes, bem como suas operações elementares e propriedades podem ser utilizados para obter os valores das incógnitas das equações que os compõem, ou seja, para *resolver o sistema linear*.

2.1.3 Resolução de Sistemas Lineares

Diversos métodos estão disponíveis para resolução de sistemas lineares, trabalhando de forma direta ou iterativa, com matrizes esparsas (matrizes com a maioria de seus elementos iguais a zero) ou densas (matrizes com a maioria de seus elementos diferentes de zero), cada um apresentando suas particularidades, vantagens e desvantagens Arias-Garcia (2010). Dentre eles, destaca-se aqui o método de Gauss-Jordan:

O Método de Gauss-Jordan

Utilizado simplesmente para inversão de matrizes ou para resolver sistemas lineares, o algoritmo de Gauss-Jordan (GJ) aparece como um dos mais simples, pois utiliza apenas somas e multiplicações em sua execução. Embora existam resultados teóricos computacionalmente mais eficientes que este algoritmo, como Coppersmith and Winograd (1987) e Strassen (1969) com complexidades de $O(n^{2.376})$ e $O(n^{2.807})$, respectivamente, enquanto GJ possui complexidade $O(n^3)$, vantagens práticas relativas à execução com uso de outros algoritmos mais eficientes só são notadas ao se trabalhar com matrizes muito grandes.

Para resolver um sistema linear, GJ organiza as equações em uma matriz de modo que forme uma matriz quadrada estendida, onde a matriz quadrada é dada pelos coeficientes e suas incógnitas e a coluna extra contém os termos independentes. Para inversão de matrizes, GJ estende uma matriz A_n com sua identidade, criando uma matriz AI de dimensões $n \times 2n$, e transformando-a uniformemente até que a parte originalmente contendo A se torne na matriz identidade. Deste modo, como $A \cdot A^{-1} = I$, ao transformar A na matriz identidade teremos IA^{-1} como a nova matriz estendida.

Este algoritmo se divide basicamente em três etapas, após a matriz ser estendida:

- **Triangulação Superior**: Esta fase possui dois passos auxiliares:
  - **Encontrar o Pivô**: Consiste em encontrar o maior elemento de uma coluna j nas i-ésimas linhas tais que $j \leq i \leq n$ (cf. Algoritmo 2.1.1). Este processo evita a propagação de erros numéricos durante a execução do algoritmo.
  - **Troca de Linhas**: Troca a j-ésima linha da j-ésima coluna onde o pivô foi encontrado pela linha que contém o pivô.

  O processamento então inicia-se escolhendo em cada coluna, da esquerda para direita e a partir do elemento da diagonal principal, o pivô para esta coluna. Troca-se então a linha que contém o pivô desta coluna pela linha correspondente ao elemento inicialmente na diagonal principal. É feito então um processo de combinações lineares, utilizando fatores adequados, com a linha do pivô (agora a i-ésima) para que todos os elementos j-ésima coluna que se encontrem abaixo da i-ésima linha sejam iguais a zero. Neste passo do algoritmo, a escolha do pivô, a troca de linhas e o processo de zerar parcialmente uma coluna se alternam (cf.

Algoritmo 2.1.2). Ao final desta execução, a matriz estendida AI é transformada numa matriz estendida $A'I'$, na qual $A'_{n\times n}$ é uma matriz triangular superior.

Algoritmo 2.1.1: (Busca_Pivo)

Entrada: m: matriz inversível estendida de dimensão $n \times 2 * n$;
j: coluna a verificar pivô tal que $0 \geq j < n$;
Saída: k: índice da linha da matriz onde se encontra o pivô

```
1 k = j;
2 for i = j ... n-1 do
3     if |m[i,j]|>|m[k,j]| then
4         k = i
5 return k
```

Algoritmo 2.1.2: Triang_Superior

Entrada: m: matriz estendida de dimensão $n \times 2 * n$.
Saída: Uma matriz estendida cuja primeira matriz quadrada é uma matriz triangular superior.

```
1 for l = 0 ... n-2 do
2     k = Busca_Pivo(m, l);
3     if k ≠ l then
4         intercambia as linhas l e k da matriz ;
5     for i = l+1 ... n-1 do
6         for j = l ... (n*2)-1 do
7             m[i,j] = m[i,j] - m[l,j] * m[i,l] / m[l,l]
8 return m
```

Note que $I'_{n\times n}$ não é mais, necessariamente, a matriz identidade.

- **Normalização**: Cada linha da matriz é dividida pelo pivô, que agora se encontra na posição da diagonal principal da matriz A' de $A'I'$ (cf. Algoritmo 2.1.3). Ao final, $A'I'$ é transformada em $A''I''$, onde A'' é uma matriz triangular superior cuja diagonal principal possui todos seus elementos iguais a um.

Algoritmo 2.1.3: Norm

Entrada: m: matriz estendida de dimensão $n \times 2 * n$.
Saída: a matriz m com sua primeira matriz triangular normalizada.

```
1 for i = 0 ... n-1 do
2     for j = 0 ... (n*2)-1 do
3         m[i,j] = m[i,j] / m[i,i];
4 return m
```

- **Triangulação Inferior**: Iniciando na n-ésima linha e realizando seu processamento até a primeira, um processo similar ao da triangulação superior é efetuado, de modo que todos elementos de uma j-ésima coluna nas i-ésimas linhas tais que $j > i \geq 0$ se tornem iguais a zero, por meio de transformações lineares com a j-ésima linha. Ao final, obtém-se a matriz estendida IA^{-1}.

Algoritmo 2.1.4: Triang_Inferior

Entrada: m: matriz triangular superior não-singular normalizada estendida de dimensão $n \times 2 * n$.

Saída: a matriz m com sua primeira matriz quadrada transformada em matriz triangular inferior.

```
1 for k = n-1 ... 1 do
2     for i = 0 ... n-1 do
3         for j = 0 ... (n*2)-1 do
4             m[i,j] = m[i,j] - m[k,j] * m[i,k];
5 return m
```

Todo processo do algoritmo GJ é apresentado no Algoritmo 2.1.5.

Algoritmo 2.1.5: Algoritmo de Gauss-Jordan

Entrada: m: matriz inversível estendida de dimensão $n \times 2 * n$.

Saída: matriz m transformada em cuja primeira matriz quadrada é uma matriz identidade e a segunda a inversa de m.

```
1 return Triang_Inferior(Norm(Triang_Superior(m)));
```

Mesmo apresentando grande complexidade computacional ($O(n^3)$), GJ permite sua fácil implementação em hardware, graças ao uso de apenas operações simples em sua execução. Isso permite sua inserção em diversos dispositivos de hardware que necessitam de inversão de matrizes, podendo aumentar o desempenho de um sistema com um hardware dedicado para esta operação.

2.2 Arquiteturas Computacionais

Existem diversas técnicas para execução de programas de computadores, cada uma com suas particularidades e aplicações em que melhor se adaptam. Uma coisa porém é comum a todas elas: o objetivo de executar operações de maneira eficaz e eficiente. São apresentados aqui alguns modelos organizacionais de computadores, em especial as arquiteturas de von Neumann e os FPGAs.

2.2.1 Arquiteturas Clássicas

Sendo a mais conhecida e tradicional forma de organização do hardware de um computador, a arquitetura de von Neumann propõe a execução sequencial de programas por

meio do uso de:

1. Uma memória, que irá armazenar tanto os programas a serem executados quanto os dados a serem processados;

2. Uma Unidade de Processamento Central (CPU), responsável por toda parte de aquisição e processamento, que é dividida em:

   (a) Unidade Lógica e Aritmética (ALU): responsável por operações básicas de processamento, como adições, multiplicações, AND, OR, NOT, etc.

   (b) Unidade de Controle (CU): busca ordenadamente da memória as intruções do programa a ser executado, as interpreta e executa;

   (c) Registradores: memórias para armazenamento temporário de dados a processar.

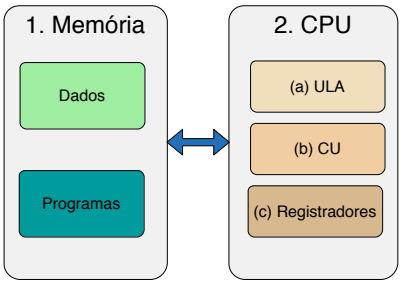

Figura 2.1: Estrutura da Arquitetura de von Neumann

Tal organização (c.f. Figura 2.1) permite que diversos programas sejam executados em um mesmo processador, pois as instruções a serem executadas estão armazenadas em uma memória externa ao módulo que efetivamente faz o processamento. Deste modo, evita-se que um novo processador tenha que ser desenvolvido sempre que a tarefa a ser cumprida se altere.

Outros modelos arquiteturais, como o de Harvard, que se assemelha ao de Von Neumann, exceto por possuir memórias e barramentos diferentes para dados e instruções, arquiteturas VLIW, que executam um grupo de instruções simultaneamente, processadores multicore, que possuem mais de uma CPU e várias outras são utilizadas com objetivo de aumentar o desempenho dos computadores. Uma das abordagens utilizadas para melhorar desempenho é a hierarquias de memórias, que divide a parte de armazenamento de dados e/ou programas geralmente em memórias cache, memória principal e secundária, para melhorar o desempenho na execução de programas. Essas memórias se comunicam entre si, armazenam e trocam dados a serem repassados ao processador. Quanto mais próximas ao processador, as memórias apresentam uma menor capacidade de armazenamento e maior custo de produção, porém sua velocidade de acesso é muito maior.

Porém, mesmo com o uso de hierarquia de memórias, a comunicação entre memória e processadores não consegue ser feita de maneira a aproveitar totalmente o poder de processamento da CPU. Isso ocorre por conta da grande quantidade de acesso à memória para operações de leitura/escrita necessárias quando da execução de programas, causando o chamado Gargalo de von Neumann, pois a execução do processador geralmente possui dependência de dados com essas operações na memória.

Diversas soluções tem sido propostas para eliminar o Gargalo de von Neumann, como uso de processamento paralelo com pipeline e processadores multicore e hardware dedicado para tarefas que demandam grande processamento para atividades específicas. Embora não haja consenso sobre qual a melhor maneira de eliminar o gargalo, diversas pesquisas tem sido desenvolvidas para tal, inclusive combinando mais de uma destas técnicas.

2.2.2 Hardware Dedicado

Com aumento no tamanho e complexidade dos sistemas que são criados hoje, mesmo as técnicas de processamento paralelo e uso de memórias cache não tem sido suficientes para evitar o Gargalo de von Neumann. Embora diversas abordagens, como o processamento paralelo, possam obter melhor desempenho quando a aplicação a ser executada não é conhecida no momento da criação da CPU, hardware dedicados são escolhas mais adequadas quando uma única aplicação específica precisa ser executada com melhor desempenho possível. Isso ocorre porque hardware dedicados evitam o *overhead* de comunicação com a memória para busca de instruções e também o desperdício de elementos da CPU que podem nunca ser usados, mas que são necessários para arquiteturas de propósito geral, como as de von Neumann.

Existem basicamente duas abordagens para criação de hardware. A primeira é o uso de uma arquitetura fixa, que provê um projeto que não poderá ser alterado quando findada sua implementação. A segunda é o uso de uma arquitetura reprogramável, como os FPGA's (utilizados no caso de estudo abordado neste trabalho), que possibilitam o reuso do circuito criado para abrigar novos projetos uma vez que aquele ali inserido inicialmente precise de alteração, seja por ter se tornado obsoleto ou por conter erros não detectados durante seu projeto Scott Hauck and DeHon (2008).

FPGAs

Constituídos basicamente de blocos lógicos, blocos de entrada/saída e interconexões programáveis, os Field Programmable Gate Arrays (FPGAs) (cf. Figura 2.2) são dispositivos de hardware reconfiguráveis de propósito geral que possibilitam sua programação através de Linguagens de Descrição de hardware (HDL). Essa programação pode se dar de maneira comportamental ou estrutural, fornecendo um nível de abstração muito maior ao projetista do que quando manipula-se diretamente conexões e elementos físicos para a criação de seu sistema. Com isso é possível criar um circuito específico para determinada tarefa de maneira otimizada.

A maioria das HDLs estão em um nível de abstração acima das conexões físicas, mas abaixo da linguagem de máquina (Assembler), embora algumas mais recentes, como o SystemC, (uma extensão do C++) propõem extensões de linguagens de alto nível para aumentar a flexibilidade e combinar vantagens de projetos em software e hardware, como facilidade de projeto e velocidade de execução. Especificamente para FPGAs, duas grandes fabricantes, a Altera e a Xilinx, oferecem ferramentas que analisam códigos em C escritos para o processador NiosII e sugerem estruturas em HDL para aceleração. São elas, respectivamente, chamada C2H (C to hardware) e Vivado HLS (High Level Synthesis). Porém, deve-se tomar cuidado ao utilizá-las, pois todas essas propostas apresentam restrições e regras que não permitem uma liberdade total de programação, sob pena de sínteses incorretas.

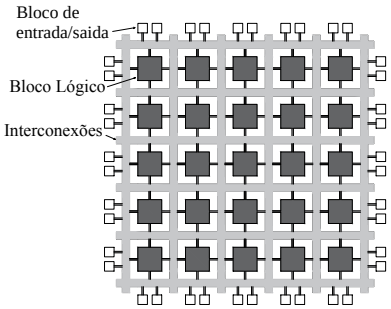

Figura 2.2: Arquitetura típica de um FPGA.

FPGA's apresentam a propriedade de serem reconfiguráveis porque usam memórias em seus blocos lógicos para implementar funções, cujos resultados serão posteriormente consultadas, são as chamadas *Lookup Tables* (LUT's) Vahid (2007). Estas memórias possuem N linhas de endereço e podem portanto implementar funções de N entradas. Assim, os resultados pré-definidos de uma função são armazenados e utilizadas apenas para consulta do resultado dada uma entrada. O exemplo apresentado na Figura 2.3 mostra como são dispostas funções lógicas na memória a ser consultada.

Um FPGA fornece uma velocidade muito maior quando comparado ao mesmo projeto concebido em software e executado em um GPP. Isso dado pelo fato de não possuir necessariamente uma execução sequencial, o que permite a realização de centenas a milhares de operações a cada ciclo Huffmire et al. (2010) e também porque o FPGA pode ser configurado de forma a conter apenas as operações necessárias para a execução do procedimento nele implantado, enquanto um circuito de propósito geral com instruções fixas necessita de todas as possíveis operações para qualquer estrutura de dados em seu projeto Gokhale and Graham (2005). Outras vantagens observadas são o tempo e o custo de produção, que são bem inferiores se comparados aos da criação de um ASIC.

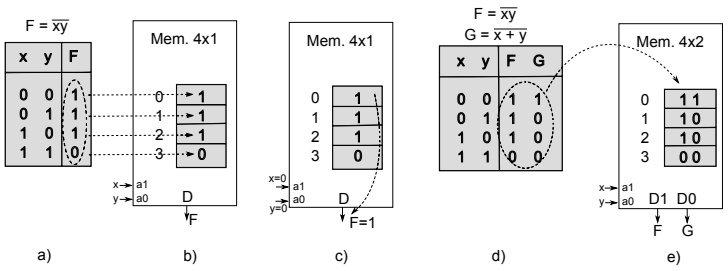

Figura 2.3: Implementação de LUT's: (a) tabela-verdade de uma função de duas entradas (NAND), (b) conteúdos e conexões da memória que implementa a função descrita em (a) (entradas $a0$ e $a1$ e saída D), (c) a saída produzida quando a entrada fornecida é "00", (d) tabela-verdade de duas funções diferentes (NAND e NOR) com as mesmas entradas, (e) conteúdo da memória que implementa as duas funções de (d) (entradas $a0$ e $a1$ e saída D). Adaptada deVahid (2007).

E mesmo apresentando tanto as vantagens de software quanto as de hardware, como facilidade de implementação, desempenho e possibilidade de reuso do dispositivo, os FPGA's também possuem desvantagens, pois criar programas eficientes com tais dispositivos é extremamente mais complexo, além de serem de cinco a 25 vezes inferiores em área do circuito, atrasos e desempenho quando comparados aos ASICs Scott Hauck and DeHon (2008). Sendo assim, seu uso é mais recomendado quando são executadas operações que trabalham com processamento de grande quantidade de dados, como processamento de sinais e atividades afins.

Pode-se ver na Figura 2.4 os passos que segue o projeto de um FPGA. Inicia-se o projeto pela especificação do sistema, ou seja, o levantamento de todas as suas funcionalidades, módulos necessários, etc. A partir disso, o sistema é implementado em uma HDL (ou mesmo um projeto esquemático) e então é testado para analisar seu comportamento, ou seja, se os dados de saída produzidos são os esperados quando fornecidas certas entradas, juntamente com a verificação formal, que por meio de mecanismos lógico-matemáticos fornece um certificado de garantia do funcionamento do circuito. Pode-se então fazer a síntese do sistema para adaptá-lo ao circuito no qual será inserido, outra rodada de testes e simulações é feita para validar esta etapa. Finalmente, é feito o posicionamento e roteamento do sistema no dispositivo, que pode então ser testado quanto ao seu funcionamento.

Vale esclarecer aqui que, embora utilizado com sentido de *teste* no âmbito do desenvolvimento de hardware, neste trabalho o termo *verificação* será utilizado exclusivamente para denotar provas formais.

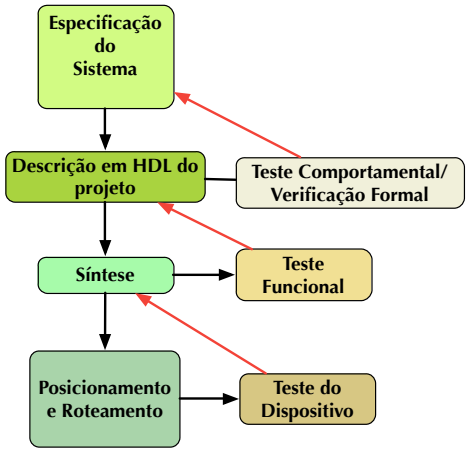

Figura 2.4: Típico Fluxo de Projeto de um FPGA adaptado de Hu (2012).

2.3 Mecanismos de Prova

Mecanismos lógico-matemáticos são utilizados para raciocinar sobre determinadas asserções e inferir resultados a partir delas. Assim, é bastante utilizada, por exemplo, para análise de correção de programas de computadores, verificando se eles executam suas funções corretamente. Nessa seção são apresentados alguns conceitos básicos dos mecanismos lógicos que podem ser utilizados para verificação da correção de programas, o Cálculo de Gentzen e a Lógica de Floyd-Hoare, e também um ambiente computacional para especificação e prova de sistemas, o PVS.

2.3.1 Cálculo de Gentzen

Cálculo de Sequentes (ou Sistema de Gentzen - CS) é um conjunto de formalismos introduzidos em 1935 por Gerhard Gentzen (matemático e lógico alemão) para tratar mais facilmente de raciocínios matemáticos onde a dedução natural se mostrava muito trabalhosa. Embora intuitivamente seja mais complicada que a dedução natural, sua simplicidade prática tem favorecido seu uso em diversos provadores de teoremas. Os elementos básicos deste cálculo serão resumidos aqui, mas as provas de seus resultados fogem ao escopo deste trabalho e podem ser vistos em Troelstra and Schwichtenberg (2000).

Definição 2.3.1 (***Sequente***)**.** Um sequente é uma estrutura de afirmação da forma $\Sigma \vdash_\Gamma \Lambda$, onde Σ é o conjunto de fórmulas antecedentes (lado esquerdo) e Λ o conjunto de fórmulas consequentes (lado direito) do contexto Γ, logo, tem-se que a conjunção das fórmulas em Σ implica na disjunção das fórmulas em Λ.

Definição 2.3.2 (***Prova***)**.** Em CS, uma prova é uma árvore rotulada com apenas uma raiz, que é o objetivo da prova, onde cada nó é um sequente e suas folhas são compostas

apenas de axiomas. Cada nó é conectado com aquele imediatamente sucessor a si de acordo com uma das regras à esquerda, à direita ou regra de corte do cálculo. Cada ramo da árvore representa um caso ou uma subprova (prova parcial) da prova principal. Assim, L$\star$ denota a aplicação da regra representada por $\star$ no lado esquerdo do sequente, e R$\star$ a aplicação desta regra no lado direito do sequente.

As regras do CS são descritas abaixo:

- **Axiomas:**

$$\frac{}{A \vdash_\Gamma A}\ (\mathrm{Ax}) \qquad \frac{}{\bot \vdash_\Gamma}\ (\mathrm{L}\bot) \qquad \frac{}{A = A}\ (=)$$

- **Regra de Substituição**

$$\frac{s = t, \Sigma[\frac{x}{t}] \vdash \Gamma[\frac{x}{t}]}{s = t, \Sigma[\frac{x}{s}] \vdash \Gamma[\frac{x}{s}]}\ (sub)$$

Onde $\Sigma[\frac{y}{r}]$ denota a substituição do termo r pela variável y em Σ.

- **Regras Estruturais de Enfraquecimento (W) e Contração (C):**

$$\frac{\Sigma \vdash_\Gamma \Lambda}{A, \Sigma \vdash_\Gamma \Lambda}\ (\mathrm{LW}) \qquad \frac{\Sigma \vdash_\Gamma \Lambda}{\Sigma \vdash_\Gamma \Lambda, A}\ (\mathrm{RW})$$

$$\frac{A, A, \Sigma \vdash_\Gamma \Lambda}{A, \Sigma \vdash_\Gamma \Lambda}\ (\mathrm{LC}) \qquad \frac{\Sigma \vdash_\Gamma \Lambda, A, A}{\Sigma \vdash_\Gamma \Lambda, A}\ (\mathrm{RC})$$

- **Regras de Operadores Lógicos:**

$$\frac{\Sigma, A \vdash_\Gamma \Lambda \qquad \Sigma, B \vdash_\Gamma \Lambda}{\Sigma, A \vee B \vdash_\Gamma \Lambda}\ (\mathrm{L}\vee) \qquad \frac{\Sigma \vdash_\Gamma \Lambda, A_i}{\Sigma \vdash_\Gamma \Lambda, A_0 \vee A_1}\ (\mathrm{R}\vee)$$

$$\frac{\Sigma, A_i \vdash_\Gamma \Lambda}{\Sigma, A_0 \wedge A_1 \vdash_\Gamma \Lambda}\ (\mathrm{L}\wedge) \qquad \frac{\Sigma \vdash_\Gamma A, \Lambda \qquad \Sigma \vdash_\Gamma B, \Lambda}{\Sigma \vdash_\Gamma A \wedge B, \Lambda}\ (\mathrm{R}\wedge)$$

$$\frac{\Sigma \vdash_\Gamma A, \Lambda \qquad \Sigma, B \vdash_\Gamma \Lambda}{\Sigma, A \to B \vdash_\Gamma \Lambda}\ (\mathrm{L}\to) \qquad \frac{\Sigma, A \vdash_\Gamma B, \Lambda}{\Sigma \vdash_\Gamma A \to B, \Lambda}\ (\mathrm{R}\to)$$

$$\frac{A[x/y]\Sigma \vdash_\Gamma \Lambda}{\forall x A \Sigma \vdash_\Gamma \Lambda}\ (\mathrm{L}\forall) \qquad \frac{\Sigma \vdash_\Gamma \Lambda A[x/y]}{\Sigma \vdash_\Gamma \Lambda, \forall x A}\ (\mathrm{R}\forall)$$

$$\frac{A[x/y], \Sigma \vdash_\Gamma \Lambda}{\exists x A, \Sigma \vdash_\Gamma \Lambda}\ (\mathrm{L}\exists) \qquad \frac{A[x/y], \Sigma \vdash_\Gamma \Lambda}{\exists x A, \Sigma \vdash_\Gamma \Lambda}\ (\mathrm{R}\exists)$$

Para as regras L$\exists$ e R$\forall$, $y \notin free(\Sigma, \Lambda)$ (y não é livre na conclusão). E para as regras L$\wedge$ e R$\vee$, $i = 0$ ou $i = 1$.

Uma regra adicional deste sistema é a regra do corte, que permite, por meio do compartilhamento de contexto de dois sequentes, onde em um deles uma fórmula é antecedente e no outro tal fórmula é consequente, a eliminação desta fórmula sem alteração do resultado:

$$\frac{\Sigma \vdash_\Gamma A, \Lambda \qquad \Sigma', A \vdash_\Gamma \Lambda'}{\Sigma\Sigma' \vdash_\Gamma \Lambda\Lambda'} \text{ (corte)}$$

Cabe mencionar que, desconsiderando a regra de corte e os axiomas, CS trabalha sempre com a inclusão de conectivos lógicos e quantificadores($\neg, \wedge, \vee, \rightarrow, \exists, \forall$) no antecedente ou no consequente do sequente. Um resultado importante obtido por Gentzen (1964) é a eliminação do corte, ou seja, qualquer prova feita com essa regra também tem uma prova que não a utiliza. Porém, a regra de corte facilita muito as deduções, pois provas sem corte se tornam muito mais complexas.

Este cálculo apresenta a particularidade de poder ser mecanizado de maneira simples, pois como contém apenas introdução de conectivos, é fácil inferir qual regra deve ser usada para se obter o que deseja, sendo bastante utilizado em provadores (semi)automáticos de teoremas, como o PVS, apresentado na Seção 3. Assim, pode ser utilizado para provar diversas propriedades matemáticas e sobre programas de maneira sistemática, como visto no Exemplo 2.3.1

Exemplo 2.3.1 (Existência de racional potência de irracionais)**.** Neste exemplo será apresentada a obtenção da Lei do Terceiro Excluído (*Law of Excluded Middle* - LEM) e seu uso na prova sobre existência de racionais que são potências de irracionais, onde $I(x)$ indica que x é um número irracional. Vale lembrar também que $x \rightarrow \bot \equiv \neg x$, que será utilizado na introdução da implicação à direita na prova, e também que $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$, ou seja, é racional, que será utilizado como premissa.

Δ_0 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{I(\sqrt{2}^{\sqrt{2}}) \vdash I(\sqrt{2}^{\sqrt{2}})}\,(Ax)}{I(\sqrt{2}^{\sqrt{2}}) \vdash I(\sqrt{2}^{\sqrt{2}}), \bot}\,(RW)}{\vdash I(\sqrt{2}^{\sqrt{2}}), \neg I(\sqrt{2}^{\sqrt{2}})}\,(R\rightarrow)}{\vdash I(\sqrt{2}^{\sqrt{2}}) \vee I(\sqrt{2}^{\sqrt{2}}), \neg I(\sqrt{2}^{\sqrt{2}})}\,(R\vee)}{\vdash I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}})}\,(R\vee)}{\vdash I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}})}\,(RC)$$

As árvores de prova serão abreviadas como:

$$\begin{array}{c} \Delta_i \\ \vdots \\ \Psi_i \end{array}$$

Indicando que a prova Δ_i conclui Ψ_i. Embora a conclusão Ψ_i já faça parte da prova Δ_i, seu resultado será mantido na abreviação para fins de facilitar a leitura e entendimento ao usar tais abreviações. Assim, a prova apresentada é abreviada por:

$$\begin{array}{c} \Delta_0 \\ \vdots \\ \vdash I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}}) \end{array}$$

Tem-se também como provas parciais:

Δ_1 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg I(\sqrt{2}^{\sqrt{2}}) \vdash \neg I(\sqrt{2}^{\sqrt{2}})}(Ax)}{\neg I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}) \vdash \neg I(\sqrt{2}^{\sqrt{2}})}(LW)}{\neg I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}), I(\sqrt{2}) \vdash \neg I(\sqrt{2}^{\sqrt{2}})}(LW)}{\neg I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}) \vdash I(\sqrt{2}) \to \neg I(\sqrt{2}^{\sqrt{2}})}(R\to)}{\neg I(\sqrt{2}^{\sqrt{2}}) \vdash I(\sqrt{2}) \to I(\sqrt{2}) \to \neg I(\sqrt{2}^{\sqrt{2}})}(R\to)}{\neg I(\sqrt{2}^{\sqrt{2}}) \vdash \exists y(I(\sqrt{2}) \to I(y) \to \neg I(\sqrt{2}^{y}))}(R\exists)}{\neg I(\sqrt{2}^{\sqrt{2}}) \vdash \exists x \exists y(I(x) \to I(y) \to \neg I(x^y))}(R\exists)$$

e

Δ_2 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg I((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})}Premissa}{I(\sqrt{2}^{\sqrt{2}}) \vdash \neg I((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})}(LW)}{I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}^{\sqrt{2}}) \vdash \neg I((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})}(LW)}{I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}) \vdash \neg I((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})}(LW)}{I(\sqrt{2}^{\sqrt{2}}), I(\sqrt{2}^{\sqrt{2}}) \vdash I(\sqrt{2}) \to \neg I((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})}(R\to)}{I(\sqrt{2}^{\sqrt{2}}) \vdash I(\sqrt{2}^{\sqrt{2}}) \to I(\sqrt{2}) \to \neg I((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})}(R\to)}{I(\sqrt{2}^{\sqrt{2}}) \vdash \exists y(I(\sqrt{2}^{\sqrt{2}}) \to I(y) \to \neg I((\sqrt{2}^{\sqrt{2}})^{y}))}(R\exists)}{I(\sqrt{2}^{\sqrt{2}}) \vdash \exists x \exists y(I(x) \to I(y) \to \neg I(x^y))}(R\exists)$$

Produzindo:

Δ_3 :

$$\dfrac{\begin{array}{cc}\Delta_1 & \Delta_2 \\ \vdots & \vdots \\ \neg I(\sqrt{2}^{\sqrt{2}}) \vdash \exists x \exists y (I(x) \to I(y) \to \neg I(x^y)) & I(\sqrt{2}^{\sqrt{2}}) \vdash \exists x \exists y (I(x) \to I(y) \to \neg I(x^y))\end{array}}{\begin{array}{cc} I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}}) \vdash \exists x \exists y (I(x) \to I(y) \to \neg I(x^y)) & (1)\end{array}} (L\vee)$$

E, finalmente:

$$\dfrac{\begin{array}{cc}\Delta_0 & \Delta_3 \\ \vdots & \vdots \\ \vdash I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}}) & I(\sqrt{2}^{\sqrt{2}}) \vee \neg I(\sqrt{2}^{\sqrt{2}}) \vdash \exists x \exists y (I(x) \to I(y) \to \neg I(x^y))\end{array}}{\vdash \exists x \exists y (I(x) \to I(y) \to \neg I(x^y))} (corte)$$

2.3.2 Esquemas Indutivos

Muitas vezes, o Cálculo de Gentzen e outros podem ser associados a esquemas indutivos de prova para complementá-los e permitir provas mais sofisticadas. Assim como ocorre na indução matemática, utilizada para provar uma dada propriedade sobre os números naturais, provas devem ser fornecidas para um caso inicial (Base de Indução - B.I.) e que o processo que leva de um caso anterior ao próximo é válido (Passo Indutivo - PI) se o caso anterior for válido (Hipótese de Indução - H.I.).

A indução estrutural sobre estruturas em que se deseja verificar propriedades é um esquema frequentemente utilizado, onde tais estruturas devem ser definidas de maneira recursiva ou tendo uma ordem definida sobre seus elementos. Quando se pretende provar alguma propriedade $\mathcal{P}$ sobre um estrutura x, deve-se provar que $\mathcal{P}$ vale para todo x minimal, consistindo na base indutiva da prova, e também provar que, assumindo-se que $\mathcal{P}$ vale para todas subestruturas de x, $\mathcal{P}$ vale para x não minimal, ou seja, o passo indutivo.

Exemplo 2.3.2. Tome como exemplo a estrutura *lista* construída usualmente com os construtores lista vazia denotada usualmente como *nil* ou [] e concatenação denotado como *cons* ou $\cdot$. A partir da lista *nil*, o construtor *cons* concatena elementos de um tipo dado T para construir listas de tipo $list[T]$, usualmente abreviado como $c \cdot l$, onde $\cdot$ é o construtor que concatena elementos c de tipo T em listas l compostas de elementos com tipo T. A indução estrutural dessa estrutura segue um esquema do tipo:

$$\begin{cases} \text{Base de Indução: } \mathcal{P}([\,]) \\ \text{Passo Indutivo: Se } \mathcal{P}(l) \text{ então } \mathcal{P}(c \cdot l) \end{cases}$$

Supondo que seja aplicado um algoritmo de ordenação *SortList* sobre uma lista l. Ao verificar se a lista resultante é ordenada através da propriedade *Sorted?(l)*, com CS deve-se obter a prova de que *Sorted?(SortList(l))* da seguinte maneira:

$$\dfrac{\dfrac{\vdots}{l = [],\ \vdash Sorted?(SortList([]))} \quad \dfrac{\vdots}{l = c \cdot l',\ Sorted?(SortList(l')) \vdash Sorted?(SortList(c \cdot l'))}}{\vdash \forall l : Sorted?(SortList(l))}\ \text{(indução)}$$

Como a prova da propriedade *Sorted?* sobre *SortList* depende das definições de ambos e estas fogem ao escopo deste trabalho, ela não será apresentada aqui. A intensão deste exemplo é ilustrar a utilização de indução estrutural em estruturas bem conhecidas associada ao CS.

Existem porém, algoritmos cuja estrutura demanda uso de técnicas auxiliares para sua verificação. Este é o caso dos programas imperativos apresentados a seguir.

2.3.3 Lógica de Floyd-Hoare

Já é, há muito tempo, conhecido que paradigmas de programação imperativos e recursivos possuem mesmo poder de expressão computacional. No entanto, provas relativas à definições recursivas são facilmente derivadas com indução, enquanto definições imperativas necessitam de mecanismos extras para tais raciocínios. A lógica de Floyd-Hoare (cf. Floyd (1967) and Hoare (1969)) vem como mecanismo para facilitar essa tarefa, propondo meios de tratar da correção de programas com laços de repetição.

O aspecto principal dessa tecnologia é o uso de uma tripla $\{\mathcal{P}\}\ \boldsymbol{\mathcal{C}}\ \{\mathcal{Q}\}$ (*Tripla de Hoare*), onde $\{\mathcal{P}\}$ é uma *pré-condição* associada ao *comando* $\mathcal{C}$, e $\{\mathcal{Q}\}$ é sua pós-condição. Percebe-se, assim, que a correção fornecida até aqui é apenas parcial, pois envolve a análise de apenas um comando. Para toda execução de laços de repetição, a correção se dá com a aplicação da regra de *iteração* desta lógica associada à indução. Essa regra, juntamente com as demais para tratar das condições sobre os comandos são dadas como:

- **Axioma de Atribuição:** Qualquer variável E anteriormente verdadeira para o lado direito da atribuição mantém seu predicado após uma atribuição:

$$\overline{\{\mathcal{P}[E/x]\}\ \boldsymbol{x := E}\ \{\mathcal{P}\}}$$

- **Regra de Consequência:** Quando um comando $\boldsymbol{\mathcal{Q}}$ tem uma pré-condição $\{\mathcal{P}\}$ e uma pós-condição $\{\mathcal{R}\}$, e uma condição $\{\mathcal{S}\}$ pode ser inferida de $\{\mathcal{R}\}$, então $\{\mathcal{S}\}$ também é pós-condição de $\mathcal{Q}$:

$$\frac{\{\mathcal{P}\}\ \boldsymbol{\mathcal{Q}}\ ,\ \{\mathcal{R}\}\ ,\ \{\mathcal{S}\} \subset \{\mathcal{R}\}}{\{\mathcal{P}\}\ \boldsymbol{\mathcal{Q}}\ \{\mathcal{S}\}}$$

E também, se a pré-condição $\{\mathcal{P}\}$ pode ser inferida de outra $\{\mathcal{S}\}$, $\{\mathcal{S}\}$ também é pré-condição do comando:

$$\frac{\{\mathcal{P}\}\ \boldsymbol{\mathcal{Q}}\ ,\ \{\mathcal{R}\}\ ,\ \{\mathcal{P}\} \subset \{\mathcal{S}\}}{\{\mathcal{S}\}\ \boldsymbol{\mathcal{Q}}\ \{\mathcal{R}\}}$$

- **Regra de Composição:** Quando dois comandos $\mathcal{S}$ e $\mathcal{T}$ são executados sequencialmente, e nesta ordem $(S;T)$, ao se ter uma condição intermediária $\{\mathcal{Q}\}$ que é pós-condição de $\mathcal{S}$ e pré-condição de $\mathcal{T}$, as pré-condições $\{\mathcal{P}\}$ de $\mathcal{S}$ e pós-condições $\{\mathcal{R}\}$ de $\mathcal{T}$ se mantêm após a execução de ambos:

$$\frac{\{\mathcal{P}\}\ \boldsymbol{\mathcal{S}}\ ,\ \boldsymbol{\mathcal{T}}\{\mathcal{R}\}}{\{\mathcal{P}\}\ \boldsymbol{\mathcal{S};\mathcal{T}}\ \{\mathcal{R}\}}$$

- **Regra de Iteração:** Ao tratar de iterações, uma condição, chamada *invariante iterativo* ou *loop invariant*, deve valer antes, durante, e também após a execução de um laço de repetição. Além do invariante, a condição de execução do laço deve valer antes e durante a execução, porém precisa pará-la quando assumir valor falso:

$$\frac{\{\mathcal{P}\} \wedge B\ \boldsymbol{\mathcal{S}}\ \{\mathcal{P}\}}{\{\mathcal{P}\}\ (\text{while } \boldsymbol{B} \text{ do } \boldsymbol{\mathcal{S}})\ \neg B \wedge \{\mathcal{P}\}}$$

  Essa regra é muito importante e permite a comparação e tradução de definições imperativas em recursivas, como será abordado durante o próximo capítulo para uso no assistente de provas PVS.

Para ilustrar o uso da regra de iteração, tome como exemplo o Algoritmo 2.3.1 para cálculo de fatorial:

Algoritmo 2.3.1: Fatorial com laço `while`

Entrada: n: natural
Saída: i: natural

```
1 natural result = 1;
2 natural i = n;
3 while i≠ 0 do
4 |   result = result * i;
5 |   i = i − 1;
6 return result;
```

Este algoritmo é simples e, como as condições a serem verificadas previamente ao laço são apenas referentes a comandos de atribuição, o invariante da regra de iteração contempla totalmente a correção do algoritmo. Para simplificar a notação, cada execução parcial do algoritmo será analisado pelo invariante dado por $\mathcal{I}(k, v)$, onde k representa a k-ésima iteração do laço e n é o parâmetro para análise do invariante. A condição final será dada por $\mathcal{P}(e, n)$, sendo e a última iteração do laço.

Vale notar que, nem sempre os parâmetros para última iteração e de análise do invariante serão os mesmos, nesse algoritmo, porém, $\mathcal{I}(n,n) = \mathcal{P}(n,n)$. Assim, basta provar o invariante:

$$\mathcal{I}(k,n) \quad \equiv \quad i = n-k \quad \wedge \quad result = \frac{n!}{(n-k)!}$$

A correção do algoritmo é dada de forma indutiva sobre o parâmetro de iteração do invariante, ou seja, por meio do uso da regra de iteração e deve-se provar a base indutiva:

$$\mathcal{I}(0,n) \quad \equiv \quad i = n-0 \quad \wedge \quad result = \frac{n!}{(n-0)!}$$

Que é trivial, e o passo indutivo:

Se para todo $k|k < n\ \mathcal{I}(k,n)$ vale, então $\mathcal{I}(k+1,n)$ vale

Como $\mathcal{I}(k+1,n) \quad \equiv \quad i = n-(k+1) \quad \wedge \quad result = \frac{n!}{(n-(k+1))!}$, tem-se trivialmente a primeira parte da conjunção e, como $result = result * i$ e por H.I. $i = n-k$ e $result = \frac{n!}{(n-k)!}$, obtem-se

$$result = \frac{n!}{(n-k)!} * (n-k) = \frac{n!}{(n-k-1)! * (n-k)} * (n-k) = \frac{n!}{(n-k-1)!}$$

A prova da condição $\mathcal{P}(n,n)$ é consequência direta dessa prova, pois $i = n-n = 0$ e $result = \frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$, provando assim a correção do algoritmo.

Capítulo 3

O Assistente PVS

Dentre diversos ambientes de especificação e formalização, foi escolhido o PVS. Neste estão incluídas uma linguagem de especificação funcional com um sistema de tipos elaborados que permite polimorfismo, subtipagem e tipos dependentes, e uma linguagem de prova baseada no cálculo de sequentes de Gentzen. Seu sistema dedutivo utiliza a abordagem de ordem superior, o que permite uma semântica mais forte, como o uso de funções como parâmetros para outras funções. Também estão presentes neste assistente um analisador léxico, um checador de tipos, bibliotecas de especificação e várias ferramentas de navegação, fornecendo um ambiente integrado para especificação e desenvolvimento de provas formais Owre et al. (2001).

Este capítulo explora algumas particularidades desse assistente, como suas linguagens de especificação e prova. Além disso, é apresentada uma maneira de lidar recursivamente com definições imperativas e suas provas.

3.1 Linguagem de Especificação

A linguagem de especificação do sistema Owre et al. (1999), mesmo não possuindo diversos recursos das linguagens de programação mais comuns, como laços de repetição e estruturas de dados sofisticadas, possibilita expressar sistemas baseados em funções de forma robusta e simplificada por ser uma linguagem funcional que possui diversos elementos para tal.

Quanto aos elementos da linguagem, além dos tipos primitivos já presentes na linguagem, como inteiros, naturais, booleanos, reais e etc, é possível a declaração de tipos abstratos e mais complexos, através de parametrizações e subtipagem, o que fornece uma grande flexibilidade para especificar os mais diversos tipos que podem ser abstraídos de definições matemáticas, das linguagens de programação ou de descrição de hardware. Deste modo, é possível criar tipos que apresentam restrições em sua estrutura e podem assim representar mais fielmente elementos implementados em hardware. Esses tipos mantém essas características e impedem o uso de argumentos incorretos durante a execução. Como mostrado no Exemplo 3.1.1:

Exemplo 3.1.1 (Restrição de tipos para denominador de frações)**.** Ao se trabalhar com frações, é importante restringir o denominador para que este não seja igual a zero. Assim, ao trabalhar com funções que tenham operações do tipo:

$$\frac{a}{a-b}$$

supondo que a seja um natural, pode-se definir o tipo do elemento b como

$$\texttt{b : TYPE} = \{\texttt{x : nat}|\texttt{x/ = a}\}$$

evitando assim uma divisão por 0, que é indefinida. O tipo b então é dependente do tipo a, que pode ser um natural qualquer.

Sobre os tipos primitivos da linguagem, como naturais, naturais positivos e também sobre tipos definidos no prelúdio do PVS, já existe uma ampla gama de lemas com suas respectivas provas acerca de suas propriedades algébricas e estruturais (também no prelúdio do PVS), como os relacionados à divisão de naturais (divisão inteira e resto), propriedades de funções e inequações, concatenação, eliminação e inclusão de elementos de listas e sequências, etc. Tais lemas facilitam muito a formalização de problemas que utilizam tais estruturas, pois não é necessário refazer as provas, apenas utilizá-las.

A linguagem também dispõe de um construtor para funções, recursivas ou não, que devem receber argumentos de determinado tipo e seu resultado se dará do tipo especificado para a função. Quando definidas funções recursivas, é compulsório o uso da função `MEASURE` ao seu final sobre seus parâmetros, sendo utilizada então para demonstrar a terminação da função. Ao usar uma função de inserção, como dada no Exemplo 3.1.2 para que se possa verificar a ordenação de uma lista, como visto no Exemplo 2.3.2, a medida que se deve verificar (o tamanho da lista) precisa ser decrementada a cada chamada recursiva.

Também dessas funções, pode-se utilizar as expressões suportadas pela linguagem, sendo elas booleanas, condicionais do tipo `if-then-else` ou `case`, registros, conjuntos, tuplas, expressões numéricas e outras. Logo, esta linguagem é bastante expressiva e adequada para especificação de sistemas de hardware. Quando se tratando de condicionais, embora tais comandos não estejam presentes na lógica clássica, estes estão presentes na linguagem de especificação do PVS. Assim, sua semântica deve ser observada cuidadosamente, pois essas estruturas são analisadas e utilizadas frequentemente durante as provas. Um condicional do tipo `if-then-else` é interpretado como:

$$\text{if } A \text{ then } B \text{ else } C \qquad \equiv \qquad A \to B \wedge \neg A \to C$$

Onde $\phi \equiv \psi$ denota que ϕ e ψ são *logicamente equivalentes*, ou seja, que qualquer interpretação é verdadeira em ϕ se e somente se é verdadeira em ψ. O uso desse condicionais também pode ser vista no Exemplo 3.1.2.

Exemplo 3.1.2 (Ordenação de listas por inserção)**.**

```
insert (x, l): RECURSIVE list[nat] =
IF null?(l) THEN
cons(x,null)
ELSIF x<= car(l) THEN
cons(x,l)
ELSE cons(car(l), insert(x,cdr(l)))
ENDIF
MEASURE length(l)


insertion_sort(l): RECURSIVE list[nat] =
IF null?(l) THEN
null
ELSE
insert(car(l), insertion_sort(cdr(l)))
ENDIF
MEASURE length(l)
```

O conjunto de declarações de tipos, funções, constantes, variáveis e lemas a serem provados constitui uma teoria, que pode também importar outras teorias, provenientes do prelúdio do PVS ou de diversas bibliotecas já criadas e disponibilizadas com diversos resultados acerca de diferentes estruturas a fim de utilizar seus elementos e/ou lemas nela provados. Uma ou mais teorias relacionadas formam então uma especificação.

Quando terminada a especificação desejada, é feita uma checagem de tipos antes de se iniciarem efetivamente as provas. Tal checagem é necessária para verificar se os parâmetros passados na especificação atendem ao tipo que ele deve ter para que as provas se completem. São geradas então Condições de Checagem de Tipo (TCC's), que tem sua maioria automaticamente verificados pelo sistema do PVS, outros necessitam de prova explícita, que deve ser feita como qualquer outra prova de lema a ser verificado. Pode-se então passar para a fase prova dos lemas especificados.

3.2 Linguagem de Prova - As Regras e os Comandos do PVS

PVS apresenta um sequente como uma conjunção de fórmulas antecedentes (ou premissas) precedidas de uma numeração negativa inferindo a disjunção das fórmulas consequentes (ou conclusões), que aparecem precedidas de numeração positiva. Vale salientar que cada fórmula antecedente pode ser lida como a negação desta em uma fórmula consequente e vice-versa. O Exemplo 3.2.1 mostra o formato como um sequente se apresenta:

Exemplo 3.2.1 (Um sequente em PVS)**.**

$$
\begin{array}{c}
[\text{-1}]\ x \geq 1 \\
[\text{-2}]\ y \geq 2 \\
\vdots \\
\text{|- - - - - - - - - -} \\
[1]\ x + y \geq 3 \\
\vdots
\end{array}
$$

As provas se iniciam com o sequente do resultado final a ser provado, aplicando comandos de prova equivalentes às regras do CS com objetivo de chegar a um sequente onde se possa aplicar uma regra axiomática. Isso ocorre em cinco tipos de sequentes:

1. Uma fórmula antecedente coincide com uma fórmula consequente, como em:

$$
\begin{array}{c}
[\text{-1}]\ x = 1 + 2 \\
\vdots \\
\text{|- - - - - - - - - -} \\
[1]\ x = 3 \\
\vdots
\end{array}
$$

2. Uma premissa é trivialmente falsa, como em:

$$
\begin{array}{c}
[\text{-1}]\ 1 > 3 \\
\vdots \\
\text{|- - - - - - - - - -} \\
\vdots
\end{array}
$$

3. Duas ou mais premissas se contradizem, como em:

$$
\begin{array}{c}
[\text{-1}]\ x > 3 \\
[\text{-2}]\ x < 1 \\
\vdots \\
\text{|- - - - - - - - - -} \\
\vdots
\end{array}
$$

4. Uma conclusão é trivialmente verdadeira, como em:

$$
\begin{array}{c}
\vdots \\
\text{|- - - - - - - - - -} \\
[1]\ 1+2>2 \\
\vdots
\end{array}
$$

5. Um subconjunto de fórmulas consequentes engloba todas as possibilidades para um dado elemento, como em:

$$
\begin{array}{c}
\vdots \\
\text{|- - - - - - - - - -} \\
[1]\ x \geq 3 \\
[2]\ x < 3 \\
\vdots
\end{array}
$$

Como são utilizadas regras do cálculo de sequentes, o PVS possui essas regras com nomes específicos Shankar et al. (1999), bem como particularizações destes e alguns outros comandos mais fortes que agrupam várias regras em apenas uma. As árvores de prova produzidas são geradas de maneira inversa daquelas no CS. Assim, é como se as regras fossem aplicadas "de baixo para cima", e sua leitura deve ser feita ao contrário.

Seguem os principais comandos utilizados, sua utilidade e relação com as regras do CS:

- `typepred` - Regra de abstração de tipo que insere uma fórmula antecedente determinando o tipo de um elemento no contexto em que a prova se insere. Esta regra não aparece no CS, sempre que a linguagem da lógica clássica é monossórtida. Mas na linguagem de especificação de PVS, temos objetos com diferentes tipos. Assim, em qualquer momento/ponto de uma prova, pode ser invocado o tipo de uma expressão utilizada nos sequentes.

  Exemplo 3.2.2. Considere novamente o exemplo de ordenação 2.3.2. Na rama da prova por indução estrutural, pode ser incluído o tipo das listas como segue, supondo o tipo dos objetos nas listas é T:

$$
\frac{l = c \cdot l',\ Sorted?(SortList(l')) \vdash Sorted?(SortList(c \cdot l'))}{l : List[T],\ l = c \cdot l',\ Sorted?(SortList(l')) \vdash Sorted?(SortList(c \cdot l'))}\ (typepred\ \ l)
$$

- `induct` e `MEASURE-induct` - Iniciam uma prova por indução natural para o primeiro e por indução estrutural no segundo, fornecendo a hipótese, no termo indicado. Esse último permite que seja escolhido o termo em que uma medida deve ser aplicada, além de gerar ramos extras que representam TCC's a serem verificados. Nem sempre a prova se divide automaticamente em caso base e passo indutivo, sendo necessário o comando `case`, visto adiante, para fornecer essa divisão.

  Exemplo 3.2.3. Ainda considerando o Exemplo 2.3.2, o esquema indutivo de prova em PVS é dado pela H.I.:

  ```
  FORALL (l'):
   length(l') < length(l) IMPLIES
   Sorted?(SortList(l'))
  ```

  Ou seja:

  $$\frac{\vdash \forall(l) : Sorted?(SortList(l))}{\forall(l') : legth(l') < legth(l) \rightarrow Sorted?(SortList(l')) \vdash Sorted?SortList(l)} *$$

  Onde $*$ é o comando `MEASURE-induct "length(l)"l`

- `lift-if` - Às vezes resultados de condicionais são utilizados em atribuições, comparações, etc. Quando isso ocorre, é necessário que todas essas manipulações extras com o resultado do condicional fiquem internas a ele, para que a semântica apresentada na Seção anterior possa ser aplicada. Este comando faz exatamente isso, possibilitando que a condição verificada fique mais externa na fórmula tratada. O Exemplo ilustra sua aplicação.

  Exemplo 3.2.4 (Apicação do comando `lift-if`)**.** Ao verificar o incremento em 1 do tamanho de uma lista ao usar a função `insert` do Exemplo 3.1.2, ou seja:

  ```
  length(insert(x, x!1)) = 1 + length(x!1)
  ```
  Esta função é expandida, ou seja, sua definição substitui seu nome, dando:

  ```
  IF null?(x!1) THEN
  length(cons(x, null))
  ELSE
  IF x <= car(x!1) THEN
  length(cons(x, x!1))
      ELSE
          length(cons(car(x!1), insert(x, cdr(x!1))))
      ENDIF
  ENDIF
  = 1 + length(x!1)
  ```

Ao utilizar o comando `lift-if`, essa fórmula passa a ter a atribuição final interna ao condicional:

```
IF null?(x!1) THEN
length(cons(x, null)) = 1 + length(x!1)
ELSE
IF x <= car(x!1) THEN
length(cons(x, x!1)) = 1 + length(x!1)
    ELSE
     length(cons(car(x!1), insert(x, cdr(x!1)))) = 1 + length(x!1)
    ENDIF
ENDIF
```

- `hide` - Oculta fórmulas antecedentes desnecessárias para a prova do consequente desejado, bastando passar a numeração referente às fórmulas que não se deseja utilizar. É equivalente às regras de enfraquecimento do cálculo de Gentzen (LW, RW) observando-se a inversão da leitura mencionada anteriormente, e é dada por:

$$\frac{\Gamma \vdash \Lambda}{\Gamma' \vdash \Lambda'} \ (\texttt{hide}) \ \text{ onde } \Gamma' \subseteq \Gamma \text{ e/ou } \Lambda' \subseteq \Lambda$$

- `case` - Regra que divide a prova em dois casos, um em que alguma suposição, exemplificada no exemplo abaixo como A vale e outro em que esta não vale. Deste modo, esse comando está relacionado à regra `corte` de CS, pois esta parte de dois sequentes com mesmo consequente, um onde uma fórmula do antecedente vale e outro em que esta não vale, para concluir um novo sequente sem essa fórmula no antecedente. Para este comando tem-se:

$$\frac{\Sigma \vdash_\Gamma \Lambda}{\Sigma, A \vdash_\Gamma \Lambda \qquad \Sigma, \neg A \vdash_\Gamma \Lambda} \ (\texttt{case})$$

ou ainda

$$\frac{\Sigma \vdash_\Gamma \Lambda}{\Sigma, A \vdash_\Gamma \Lambda \qquad \Sigma \vdash_\Gamma A, \Lambda} \ (\texttt{case})$$

Ou seja, se acrescentada uma suposição que, tanto verdadeira como falsa, não altera o resultado, tal suposição pode ser eliminada.

Exemplo 3.2.5. Tomando a H.I. obtida pelo comando `MEASURE-induct` no Exemplo 3.1.2, a divisão em caso base e passo indutivo se dá ao introduzir o comando `case` para verificar listas vazias, ou seja:

$$\frac{\forall(l') : legth(l') < legth(l) \rightarrow Sorted?(SortList(l')) \vdash Sorted?SortList(l)}{\Delta_1 \qquad\qquad \Delta_2} \ *$$

Onde

$$\begin{cases} * : \texttt{case}\ \text{“}l = nil\text{”} \\ \Delta_1 : \quad l = [], \forall(l') : legth(l') < legth(l) \rightarrow Sorted?(SortList(l')) \vdash Sorted?SortList(l) \\ \Delta_2 : \quad l = c \cdot l', \forall(l') : legth(l') < legth(l) \rightarrow Sorted?(SortList(l')) \vdash Sorted?SortList(l) \end{cases}$$

- `lemma` - Permite a chamada de lemas já provados para utilizá-los como antecedentes através de sua instanciação com os parâmetros corretos. Este comando se assemelha ao `case`, também se relaciona com a regra de `corte` de Gentzen. A diferença entre eles está no fato de o lema invocado já ter sido provado, ou seja, o segundo ramo gerado pelo `case` não é mais necessário.

- `split` - Como comandos `if-then-else` são semanticamente interpretados como descrito anteriormente, é necessário que duas provas diferentes sejam produzidas quando estes estão presentes no consequente, já que se trata de uma conjunção de resultados, como na regra $R\wedge$ de CS. Outra situação em que duas provas também são necessárias é quando há uma disjunção no antecedente, pois a prova da conclusão se dará apenas se ambos elementos das premissas são verdadeiros, como na regra $L\vee$. Este comando faz exatamente essa divisão da prova em seus devidos casos, e como a implicação é equivalente ao uso da negação e disjunção ($A \rightarrow B \ \equiv \ \neg A \vee B$), esta regra também é utilizada quando há uma implicação no antecedente, sendo comparada à regra $L \rightarrow$ em CS. Assim, este comando é aplicado da seguinte maneira:

$$\frac{\Sigma \vdash \text{ if } A \text{ then } B \text{ else } C\Lambda}{\Sigma \vdash A \rightarrow B \qquad\qquad \vdash \neg A \rightarrow C\Lambda} \ (\texttt{split})$$

ou

$$\frac{\Sigma \ \vdash \ A \wedge B\Lambda}{\Sigma \ \vdash \ A\Lambda \qquad\qquad \Sigma \vdash B\Lambda} \ (\texttt{split})$$

ou

$$\frac{\Sigma, A \vee B \ \vdash \ C\Lambda}{\Sigma, A \ \vdash \ C\Lambda \qquad\qquad \Sigma, B \ \vdash \ C\Lambda} \ (\texttt{split})$$

ou

$$\frac{\Sigma, A \rightarrow B \vdash_\Gamma \Lambda}{\Sigma \vdash_\Gamma A, \Lambda \qquad \Sigma, B \vdash_\Gamma \Lambda} \ (\texttt{split})$$

- `flatten` - Quando há uma conjunção no antecedente, a prova pode ser simplificada tal que os dois elementos dessa conjunção se tornem antecedentes independentes, pois para a conjunção ser verdadeira, ambos deverão também ser verdadeiros. Isso é equivalente à regra $L\wedge$ de CS. De maneira semelhante, quando há

uma disjunção no sucedente, como é necessário provar como verdadeira apenas uma das fórmulas do sucedente e para a conjunção ser verdadeira basta que um de seus elementos o seja, tais elementos podem ser colocados como conclusões separadas, como é dado na regra $R\vee$ de CS. Além disso, de maneira similar ao que ocorre com implicações vistas para o comando `split`, estas também podem ser vistas aqui como disjunções com negações, e quando no sucedente também se relacionam com a regra $R\vdash$. A aplicação deste comando então se dá assim:

$$\frac{\Sigma,\ \text{if } A \text{ then } B \text{ else } C \vdash \Lambda}{\Sigma, A \to B, A \to C \vdash \Lambda}\ (\texttt{flatten})$$

ou

$$\frac{\Sigma, A \wedge B\ \vdash \Lambda}{\Sigma, A, B \vdash \Lambda}\ (\texttt{flatten})$$

ou

$$\frac{\Sigma \vdash A \vee B \Lambda}{\Sigma \vdash A, B\ \Lambda}\ (\texttt{flatten})$$

ou

$$\frac{\Sigma \vdash_\Gamma A \to B, \Lambda}{\Sigma, A \vdash_\Gamma B, \Lambda}\ (\texttt{flatten})$$

- `prop` - Este comando é utilizado para simplificações proposicionais, tais como aquelas realizadas pelos comandos `split` e `flatten`, porém, é um comando muito mais forte, que tenta fazer todas as simplificações possíveis de uma única vez;

- `skosimp` - Realiza o processo de skolemização, ou seja, elimina quantificadores universais (quando no consequente) e existenciais (quando no antecedente) fornecendo uma testemunha (variável) da existencialidade ou da universalidade de uma variável arbitrária, e renomeando-a para evitar colisões e capturas indesejadas. Sua variante, o `skeep`, mantém o mesmo nome das variáveis quando possível. Estes comandos são equivalente às regras de introdução do quantificador universal à direita e do quantificador existencial à esquerda do sistema de Gentzen:

$$\frac{\Sigma \vdash_\Gamma \Lambda, \forall x A}{\Sigma \vdash_\Gamma \Lambda, A[x/y]}\ (\texttt{skeep -})$$

ou

$$\frac{\exists x A, \Sigma \vdash_\Gamma \Lambda}{A[x/y], \Sigma \vdash_\Gamma \Lambda}\ (\texttt{skeep +})$$

- `inst` - Realiza a instanciação de quantificadores universais (quando no antecedente) e existenciais (quando no consequente) para sua eliminação, onde podemos então trabalhar com o valor instanciado. É equivalente às regras de introdução do quantificador universal à esquerda e do existencial à direita do sistema de Gentzen:

$$\frac{\forall x A, \Sigma \vdash_\Gamma \Lambda}{A[x/y], \Sigma \vdash_\Gamma \Lambda} \ (\texttt{inst +})$$

ou

$$\frac{\exists x A, \Sigma \vdash_\Gamma \Lambda}{A[x/y], \Sigma \vdash_\Gamma \Lambda} \ (\texttt{inst -})$$

- `replace` - Substitui igualdades entre as fórmulas, ou seja, encontra as ocorrências de determinado elemento e substitui pela sua igualdade apresentada em fórmula antecedente. É equivalente à regra de substituição:

$$\frac{s \equiv t, \Sigma[\frac{x}{s}] \vdash \Gamma[\frac{x}{s}]}{s \equiv t, \Sigma[\frac{x}{t}] \vdash \Gamma[\frac{x}{t}]} \ (replace)$$

- `assert` - Este comando, relacionado às regras axiomáticas de CS, realiza automaticamente procedimentos proposicionais e aritméticos a fim de fechar uma prova ao se dar conta de casos básicos, ou seja, um dos cinco casos apresentados inicialmente nesta Seção.

Juntamente com suas linguagens de especificação e prova, PVS também possui um ambiente para representação gráfica das provas geradas. Essas são ilustradas como árvores, onde cada sequente é um vértice representado por $\vdash$ e os comandos de prova rotulam as arestas que ligam um sequente anterior ao próximo.

Essas árvores permitem um melhor entendimento das provas durante sua execução, pois a visualização dos comandos e sequentes possibilitam uma visão mais intuitiva das provas. Veja por exemplo, a utilização do PVS com a representação gráfica da árvore de prova para o Exemplo 2.3.1 no Exemplo 3.2.6.

Exemplo 3.2.6 (Exisência de racional potência de irracionais em PVS)**.** Em PVS, a prova para o Exemplo 2.3.1 se segue a partir do sequente inicial que indica a propriedade final a ser provada, ou seja:

```
rat_pot_irrat: LEMMA
EXISTS (x,y: real):
NOT rational?(x) IMPLIES
NOT rational?(y) IMPLIES
rational?(expt(x,y))
```

Assim, como a última regra aplicada para a prova com CS foi a regra de corte, o comando referente a esta regra, ou seja `case`, deve ser o primeiro a ser utilizado, e dois ramos então serão gerados, como pode ser visto na Figura 3.1.

Após aplicar o comando `(case "rational?(expt(SQRT(2),SQRT(2))) or not rational?(expt(SQRT(2),SQRT(2)))")`, onde `expt(x,y)` denota x^y, o primeiro ramo é iniciado pelo sequente [1], resultado da subprova Δ_3 do referido exemplo:

```
{-1}  rational?(expt(SQRT(2), SQRT(2))) OR
NOT rational?(expt(SQRT(2), SQRT(2)))
  |-------
[1]   EXISTS (x, y: real):
        NOT rational?(x) IMPLIES
         NOT rational?(y) IMPLIES rational?(expt(x, y))
```

Assim, como a última regra utilizada nesta subprova foi L $\vee$, o comando de prova a ser utilizado neste momento é o `split`, que dividirá este ramo em dois outros, sendo o primeiro deles iniciado pelo sequente [1.1] descrito abaixo e equivale à subprova Δ_1:

```
{-1}  rational?(expt(SQRT(2), expt(2)))
  |-------
[1]   EXISTS (x, y: real):
        NOT rational?(x) IMPLIES
         NOT rational?(y) IMPLIES rational?(expt(x, y))
```

Para esta subprova, deve-se então fornecer as testemunhas que garantem a existencialidade dos x e y da conclusão. A instanciação é feita com o comando `(inst 1 "SQRT(2)SQRT(2))")`, sendo equivalente à R$\exists$ e gerando o sequente:

```
[-1]  rational?(expt(SQRT(2), SQRT(2)))
  |-------
{1}  NOT rational?(SQRT(2)) IMPLIES
       NOT rational?(SQRT(2)) IMPLIES
       rational?(expt(SQRT(2), SQRT(2)))
```

Este sequente então possui implicação no consequente, e ao utilizar o comando `flatten`, equivalente à regra R$\rightarrow$, a premissa da implicação será adicionada às regras do antecedente, como visto anteriormente. Assim, a fórmula [-1] será igual à fórmula {1}, gerando um sequente axiomático do tipo 1 visto anteriormente e completando a prova deste ramo.

O ramo [1.2] é referente à subprova Δ_2, sendo iniciado pelo sequente:

```
  |-------
{1}   rational?(expt(SQRT(2),SQRT(2)))
```

```
[2]   EXISTS (x, y: real):
        NOT rational?(x) IMPLIES
         NOT rational?(y) IMPLIES
         rational?(expt(x, y))
```

Esta subprova utiliza como premissa que `expt(expt(SQRT(2), SQRT(2)), SQRT(2)) = 2`, então é chamado um axioma auxiliar que indica isso para dar prosseguimento à prova, produzindo o sequente:

```
{-1}  expt(expt(SQRT(2), SQRT(2)), SQRT(2)) = 2
  |-------
[1]   rational?(SQRT(2))
[2]   EXISTS (x, y: real):
        NOT rational?(x) IMPLIES
         NOT rational?(y) IMPLIES
         rational?(expt(x, y))
```

Assim como no ramo [1.1], é necessário fornecer as testemunhas de existencialidade para a fórmula 2. Para isso se usa o comando `(inst 2 "expt(SQRT(2), SQRT(2)) SQRT(2)")`, produzindo:

```
[-1]  expt(expt(SQRT(2), SQRT(2)), SQRT(2)) = 2
  |-------
[1]   rational?(expt(SQRT(2), SQRT(2)))
{2}   NOT rational?(expt(SQRT(2), SQRT(2))) IMPLIES
       NOT rational?(SQRT(2)) IMPLIES
        rational?(expt(expt(SQRT(2), SQRT(2)), SQRT(2)))
```

Novamente, a presença de implicação na fórmula 2 do consequente possibilita o uso do comando `flatten`, que produz o sequente:

```
[-1]  expt(expt(SQRT(2), SQRT(2)), SQRT(2)) = 2
  |-------
[1]   rational?(expt(SQRT(2),SQRT(2)))
{2}   rational?(expt(SQRT(2), SQRT(2)))
{3}   rational?(SQRT(2))
{4}   rational?(expt(expt(SQRT(2), SQRT(2)), SQRT(2)))
```

Note que a fórmula {4} é um predicado que verifica se seu conteúdo, que é igual à primeira parte da igualdade da fórmula 1, é um número racional. Assim, a regra `hide` é utilizada para ocultar as fórmulas [1], {2} e {3}, desnecessárias à prova, atuando equivalentemente à regra RW e produzindo:

```
[-1]  expt(expt(SQRT(2), SQRT(2)), SQRT(2)) = 2
  |-------
[1]   rational?(expt(expt(SQRT(2), SQRT(2)), SQRT(2)))
```

Ao utilizar a regra `replace -1 1`, o predicado da fórmula [1] passará a verificar se 2 é um número racional:

```
  |-------
{1}   rational?(2)
```

E como a definição do predicado `rational?` produz resultado trivialmente verdadeiro quanto aplicado ao número 2, o comando `grind` encerra esta subprova. Assim, também é completada a prova do ramo [1].

O ramo [2] representa a subprova Δ_0 e é dada pelo sequente:

```
  |-------
{1}   rational?(expt(SQRT(2), SQRT(2))) OR
NOT rational?(expt(SQRT(2), SQRT(2)))
[2]   EXISTS (x, y: real):
        NOT rational?(x) IMPLIES
         NOT rational?(y) IMPLIES rational?(expt(x, y))
```

Note que o resultado principal, provado no ramo iniciado pelo sequente [1] é repetido na fórmula [2], porém, como já foi provado, é desnecessário à prova deste ramo, sendo então omitido com o comando `hide`. Isso gera o sequente:

```
  |-------
[1]   rational?(expt(SQRT(2), SQRT(2))) OR
NOT rational?(expt(SQRT(2), SQRT(2)))
```

Que possui uma disjunção no consequente e ao ser aplicada a regra `flatten` são geradas duas fórmulas consequentes que englobam todas as possibilidades para o resultado do predicado que verifica se a raiz quadrada de 2 é um número racional. Isso completa a prova, pois gera o caso axiomático 5. Este ramo também pode ser provado pela aplicação explícita da regra LEM (provada em Δ_0), já presente no PVS, dada pelo lema `excluded_middle` , que dá o sequente:

```
{-1}  FORALL (A: bool): A OR NOT A
  |-------
[1]  rational?(expt(SQRT(2), SQRT(2))) OR
NOT rational?(expt(SQRT(2), SQRT(2)))
```

E sua instanciação com `inst -1 ‘‘expt(rational?(SQRT(2), SQRT(2)’’)` completa a prova. Assim, a prova do resultado inicial desejado se completa.

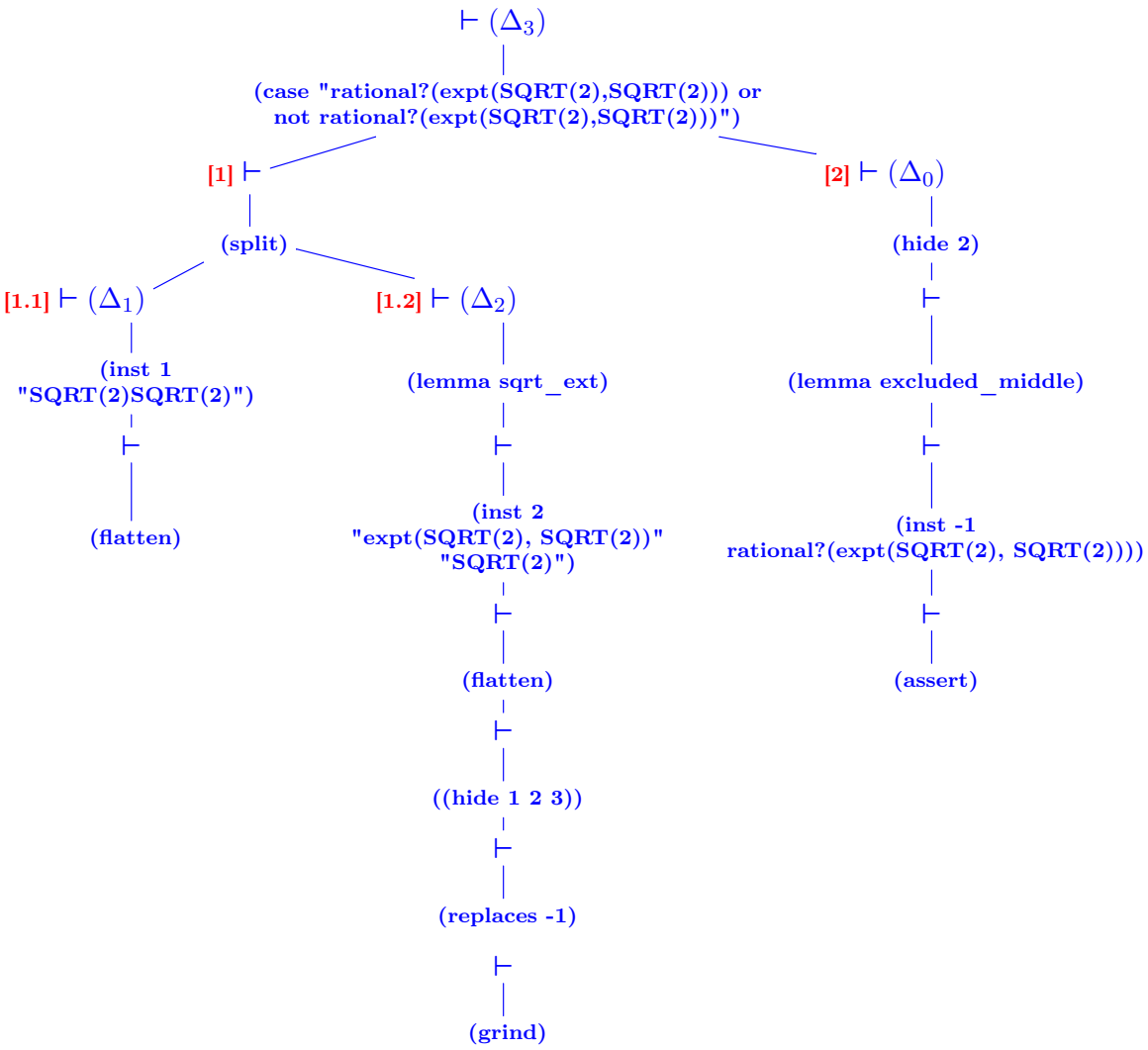

Figura 3.1: Árvore de prova da existência de racional potência de irracionais em PVS

3.3 Definições Imperativas Versus Definições Recursivas no PVS

Diversos paradigmas podem ser utilizados para modelar e implementar sistemas nos mais diversos níveis de abstração. Tecnologias de descrição de hardware, como as vistas na Seção 2.2.2 geralmente usam linguagens imperativas com comandos tais como `for`, `while`, `repeat until`, etc para representar comportamentos repetitivos. No entanto, linguagens de especificação e verificação comumente usam linguagens funcionais recursivas, como a linguagem do PVS, pois esquemas indutivos são inerentes a este tipo de definição. Assim, quando se trabalha com especificação e verificação formal de

implementações de hardware, é essencial que a tradução de um paradigma imperativo para um recursivo seja feita de maneira conservativa.

Provas indutivas podem ser feitas sobre comandos de laços de repetição imperativos quando estes são simulados por implementações recursivas. No entando, isso aumenta a complexidade das provas a serem obtidas. Veja abaixo os comandos imperativos e sua equivalente definição recursiva em PVS.

3.3.1 O Comando FOR

O comando `for`, que pode ser abstraído em sua versão imperativa como:

Algoritmo 3.3.1: Estrutura do comando `for`.

```
1 v_{b-1} = valores_iniciais;
2 for k = b ... e do
3     v_k = f(k, v_{k-1});
```

Onde v representa um conjunto de variáveis modificadas em cada iteração do comando `for`, cujo corpo é abstraído como a função f, que atua sobre as variáveis v_{k-1} (com valores inicialmente v_{b-1}) na iteração k, provendo novos valores para estas variáveis em cada uma das $e - b + 1$ iterações, nomeadamente: $v_b = f(b, v_{b-1}); v_{b+1} = f(b+1, f(b, v_{b-1})); \ldots, v_e = f(e, v_{e-1}))$. Para provar a correção de comandos deste tipo, usa-se a propriedade de invariante iterativo visto na Seção 2.3.3.

Utilizando tal mecanismo, suponha que se queira provar que determinada propriedade $\mathcal{P}$ vale após a execução do comando `for`. Para fins de compreensão da aplicação desta técnica com PVS, essa propriedade como um todo, ou seja, após a execução completa do laço, será dada por $\mathcal{P}(e, v_e)$, e seu invariante, dado por $\mathcal{I}$, indicará tal propriedade parcialmente, ou seja, após cada iteração do laço, e é dada tal que:

$$\mathcal{I}(k, v_k) \text{ vale}, \forall k : b - 1 \leq k \leq e, \text{e}$$
$$\mathcal{I}(e, v_e) \text{ implica } \mathcal{P}(e, v_e).$$

Note que a condição **B** é dada aqui pela pertinência ao intervalo $k | b - 1 \leq k \leq e$, e os comandos $\mathcal{S}$ são abstraídos pela função f. Além disso, como a execução destes laços se dá de maneira diferente, embora para solução do mesmo problema, as propriedades a serem verificadas são diferentes daquelas da Seção 2.3.3. Deste modo, o invariante verifica se os resultados produzidos pelo corpo do laço em uma dada iteração possuem a propriedade esperada.

Usando as possibilidades do uso de lógica de ordem superior providas pelo PVS, o comando `for` pode ser facilmente implementado. Inicialmente, define-se uma função

genérica de ordem superior para esse comando, como dado no Algoritmo 3.3.2.

Algoritmo 3.3.2: Definição recursiva para o comando `for`

```
1 for_rec(b, k, e : int, v : T, f : int × T → T) : T ;
2 if k > e then
3 |  v
4 else
5 |  for_rec(b, k + 1, e, f(k, v), f)
```

Nessa função, a medida que sempre decrementa e garante sua terminação é $mx(e - k + 1, 0)$. Os elementos do `for` imperativo são abstraídos como:

- Os parâmetros b, e e k são, respectivamente, a primeira iteração, a última e o contador do laço;
- v representa o valor a ser processado pelo comando `for`, podendo ser dado como uma tupla de parâmetros de tipo T;
- f representa o corpo da função que modifica esses valores em cada iteração, dando novos valores de mesmo tipo como resposta.

Supondo que a seguinte chamada seja executada: $\texttt{for\_rec}(b, b, e, v_0, f)$. Quando $b > e$ a chamada retorna v_0; se $b = e$, o valor retornado é $f(b, v_0)$; se $b = e - 1$, retorna $f(b + 1, f(b, v_0))$ e; em geral, quando $(e - b) + 1 \in \mathbb{N}$, retorna $f(e, f(e - 1, \cdots f(b + 1, f(b, v_0)) \cdots))$, que é exatamente a ação do `for` com valores iniciais v_0.

Para verificar matematicamente que a tradução entre o `for` iterativo e o `for_rec` recursivo é feita corretamente, a saída das duas abordagens deve prover exatamente as mesmas respostas, ou seja:

Para todo $b, e : int$, $v_{b-1} : T$ e $f : int \times T \to T$:

$$\begin{cases} \text{Se } b > e \text{ então } \texttt{for\_rec}(b, b, e, v_{b-1}, f) = v_{b-1}; \\ \text{senão, para todo } k,\ b \leq k \leq e: \\ \qquad \texttt{for\_rec}(b, b, k, v_{b-1}, f) = v_k, \\ \text{onde os } v_i\text{'s são computados pelo } \texttt{for} \text{ imperativo} \end{cases}$$

Adicionalmente, assumindo que a equivalência esteja provada, um invariante $\mathcal{I}$ e a propriedade $\mathcal{P}$ devem ser dados e verificados. Provar que alguma propriedade vale em definições imperativas corresponde a provar indutivamente que tais propriedades valem em uma definição recursiva. Então, deve ser provado que:

$\mathcal{I}(b - 1, v_{b-1})$ e
$\forall k : b \leq k \leq e, (\texttt{for\_rec}(b, b, k, v_{b-1}, f) = v_k$ e
$\texttt{for\_rec}(b, k, k, v_{k-1}, f) = \texttt{for\_rec}(b, k + 1, k, f(k, v_{k-1}), f) = v_k)$ e
$\forall k : b \leq k < e,$
$(\mathcal{I}(k, \texttt{for\_rec}(b, b, k, v_{b-1}, f))$ implica $\mathcal{I}(k + 1, \texttt{for\_rec}(b, k + 1, k + 1, v_k, f)))$ e
$\mathcal{I}(e, \texttt{for\_rec}(b, b, e, v_{b-1}, f))$ implica $\mathcal{P}(e, \texttt{for\_rec}(b, b, e, v_{b-1}, f))$.

Algoritmo 3.3.3: Definição imperativa da função fatorial com comando `for`

```
1 v = 1;
2 for k=1 ... e do
3 |  v = k * v
```

Para ilustrar essa transformação, considere o código para fatorial abaixo:
A tradução dá a chamada

$$\texttt{for\_rec}(1, 1, \texttt{e}, 1, *)$$

onde $*$ é a função de multiplicação com tipos $* : int \times nat \rightarrow nat$. O invariante e a propriedade a serem provados são dados como $\mathcal{I}(m, n)$ iff $m! = n$ e, $\mathcal{P}(m, n)$ iff $m! = n$.

A prova de que a tradução é feita de maneira conservativa é facilmente dada se a expressão $\texttt{for\_rec}(1, 1, e, 1, *)$ retorna 1 para $1 > e$. Caso contrário, a mesma expressão $\texttt{for\_rec}(1, 1, k, 1, *)$ retorna o valor v_k (para todo k tal que $1 \leq k \leq e$), que é computado com k iterações do `for` imperativo (o que é provado por indução em k). Para isso, primeiro é provado que o código imperativo computa $v_k = k!$. Assim, supondo que $\texttt{for\_rec}(1, 1, k, 1, *)$ retorna $k!$ (para $k < e$), prova-se que $\texttt{for\_rec}(1, 1, k, 1, *) = \texttt{for\_rec}(1, k + 1, k + 1, k!, *) = \texttt{for\_rec}(1, k + 2, k + 1, (k + 1) * k!, *) = (k + 1)!$.

A prova de que o invariante e a propriedade valem é dada de maneira similar:

- B.I.: Note que $\mathcal{I}(0, 1)$ vale, pois $0! = 1$;
- P.I.: assumindo que para todo k tal que $1 \leq k < e$, $\mathcal{I}(k, \texttt{for\_rec}\ (1, 1, k, 1, *))$ vale, ou seja, $\texttt{for\_rec}(1, 1, k, 1, *) = k!$, tem-se que $\mathcal{I}(k+1, \texttt{for\_rec}\ (1, k+1, k+1, k!, *)))$ também vale, já que $\texttt{for\_rec}(1, k+1, k+1, k!, *)$ chama recursivamente $\texttt{for\_rec}(1, k + 2, k + 1, (k + 1) * k!, *)$, que retorna $(k + 1) * k! = (k + 1)!$;
- A propriedade é consequência trivial do invariante para a última iteração: $\mathcal{I}(e, \texttt{for\_rec}(1, 1, e, 1, *))$ se e somente se $e! = \texttt{for\_rec}(1, 1, e, 1, *)$ se e somente se $\mathcal{P}(e, \texttt{for\_rec}(1, 1, e, 1, *))$.

A simulação da estrutura de repetição imperativa `for` é feita em PVS com uso da teoria `for_iterate`, disponível na biblioteca NASA PVS `structures` NASA (2013), que contém não apenas as definições recursivas correspondentes aos comandos, mas também formalizações relacionadas com a aplicação indutiva de invariantes. Logo, é uma forma viável de formalizar projetos de hardware, fornecendo um mecanismo rápido de adaptar suas definições imperativas naquelas recursivas utilizadas pelo assistente de prova.

3.3.2 Os Comandos WHILE e REPEAT

Laços de repetição `while` e `repeat` não possuem variável contadora para monitorar a parada de sua execução, mas a verificação de determinada condição sobre um dado

elemento (variável ou não), podendo ser abstraídos em sua versão imperativa como:

Algoritmo 3.3.4: Estrutura Geral do comando `while`.

```
1 s = elemento_controle_inicial;
2 v_0 = valores_iniciais;
3 t =< s, v_0 >;
4 while condicao(t'1) do
5 |  t = f(t);
```

Algoritmo 3.3.5: Estrutura Geral do comando `repeat`.

```
1 s = elemento_controle_inicial;
2 v_0 = valores_iniciais;
3 t =< s, v_0 >;
4 repeat
5 |  t = f(t);
6 until condicao(s);
```

Onde s abstrai um elemento inicial a ser verificado sob determinada condição para controle de execução do laço. v_0 abstrai os valores iniciais das variáveis a serem alteradas pela execução dos laços. O elemento de condição e as variáveis são então inseridos em uma tupla t para que apenas um parâmetro seja passado à função f, que abstrai o corpo do laço, simplificando a notação.

Em geral, linguagens de programação permitem que o elemento de controle seja modificado ou não pela função f, o que compromete a terminalidade do laço e sua transcrição recursiva em PVS, já que a garantia de terminalidade é exigida pelo uso da função `MEASURE`. Os exemplos descritos abaixo não pode ser corretamente definido de maneira recursiva:

Algoritmo 3.3.6: Exemplo de laço infinito

```
1 n = 0;
2 while 1 = 1 do
3 |  n = n+1
```

Algoritmo 3.3.7: Exemplo de laço com elemento de controle dependente

```
1 n = outra_funcao(x, y);
2 k = 0;
3 repeat
4 |  k = k+1
5 until n = true;
```

Isso porque, no Algoritmo 3.3.6 a condição de controle nunca será falsa, ou seja, não possui um caso base de execução e nunca termina, e no Algoritmo 3.3.7 o elemento de

controle é dependente de outra função, que pode produzir uma execução cuja terminação não pode ser determinada. Tais laços, embora tenham sua importância, (como em ambientes multithread) não podem ser corretamente definidos de maneira recursiva.

Quando laços `while` e `repeat` se comportam de maneira similar ao comando `for`, ou seja, possuem alteração em seu elemento de controle no corpo do laço a cada iteração, pode-se descrevê-las como comandos `for`, permitindo então sua tradução em definição recursiva. Ou seja, quando s é modificado pela função f de modo a fazer falsa ao fim de um número finito de iterações, isto é, s é um contador.

Capítulo 4

Abordagem Utilizada

Este capítulo apresenta a verificação formal de hardware, sua importância e esforços realizados neste sentido. Também é descrita a abordagem utilizada para verificação formal de hardware ilustrada no caso de estudo do Capítulo 5.

4.1 Verificação Formal de hardware

A complexidade dos circuitos criados, sejam ASIC's ou FPGAs, tem aumentado a cada dia. Nesse sentido, a tarefa de verificação do funcionamento correto destes sistemas tem tomado cada vez mais importância, chegando a representar 80% do custo do desenvolvimento de um sistema Drechsler (2004). Assim sendo, as técnicas de verificação formal tem se tornado uma opção bastante interessante, já que podem garantir corretude funcional de sistemas, o que não se pode alcançar com a tradicional análise dos resultados via testes e simulações Perry and Foster (2005), Kropf (1999).

A verificação formal de sistemas busca, por meio da aplicação de técnicas baseadas em formalismos matemáticos e lógicos, certificar que determinadas funcionalidades e propriedades dos sistemas e dados processados por estes sejam atendidas em qualquer situação. A formalização se dá por meio da especificação do sistema e de provas de correção da sua funcionalidade. É um trabalho que apresenta certa complexidade e, por ser bastante minucioso e exaustivo, demanda muito tempo. Porém, seus resultados o tornam justificável e viável Harrison (2006). A fim de facilitar tal tarefa e evitar que sejam introduzidos erros oriundos de falha humana, foram criados os provadores de teorema, ou assistentes de prova, dos quais se destacam ACL2, Coq, HOL e PVS Harrison (2006).

Equivalência Funcional

Existem várias maneiras de verificar formalmente um sistema, sendo as mais comuns a equivalência funcional (ou checagem de equivalência -*Equivalence Checking* - EC) e a checagem de propriedades (*Property Checking* - PC). Com o método PC são identificadas e especificadas propriedades referentes ao circuito pretendido que são diretamente

demonstradas ou testadas via técnicas como *model checking*. Tais propriedades então devem ser provadas satisfeitas em todas as possíveis circunstâncias do sistema Li and Thornton (2010).

Já a verificação por EC é uma técnica mais simples, que objetiva garantir a equivalência entre dois circuitos, que podem estar descritos de maneiras e níveis diferentes de abstração. Para isso, um dos circuitos a ser comparado deve ter sido previamente provado correto (por PC, por exemplo). Neste caso, deve-se estabelecer uma correspondência entre os dois e verificar então sua equivalência Drechsler (2004), Li and Thornton (2010).

Equivalência funcional é um processo bastante utilizado para verificar equivalência entre diferentes níveis de abstração e representação de um hardware, como a descrição em linguagem de hardware e uma *netlist* criada a partir dele, como no processo descrito na Seção 4.3.2. Neste caso, são comparadas as duas descrições para garantir que a *netlist* criada para inserção no hardware corresponde à descrição inicial. Essa técnica será utilizada neste trabalho de maneira mais geral, comparando o comportamento de um circuito com seu operador matemático.

4.2 Trabalhos Correlatos

No âmbito na verificação formal aplicada à hardware, há abordagens que apenas usam mecanismos lógicos diretamente para modelar e verificar sistemas, como Siegl et al. (2011), que propõe um modelo formal de testes para sistemas automotivos embarcados, especificando os requerimentos do projeto formalmente de maneira única durante a criação do modelo utilizado para testes, que passam a ser sistemáticos e mais confiáveis. Em Zhang and Duan (2011) é utilizada lógica temporal de projeção proposicional (*Propositional Projection Temporal Logic*) em um caso de estudo para provar a correção de um somador completo, a fim de mostrar a adequabilidade de um sistema axiomático para verificação de hardware.

A verificação formal de uma memória RAM utilizando *model checking* e provadores de teoremas é apresentada em Hu (2012), no contexto da aplicação de verificação formal em conjunto com simulações nos estágios iniciais de desenvolvimento de sistemas críticos, como em armas nucleares. Em Sutton (2010) é proposta toda uma metodologia para criação de sistemas FPGA altamente confiáveis, explorando o uso de padronizações, prototipação, simulações e outras abordagens. A verificação formal é aplicada ao final do processo de desenvolvimento do circuito.

Em outro trabalho, Singh and Lillieroth (1999), é apresentada uma abordagem para verificação em núcleos de hardware reconfiguráveis, que depende de uma especificação em alto nível da funcionalidade desejada do núcleo. Tal metodologia requer a decomposição do núcleo em partes menores, permitindo uma verificação *bottom up*. São essas partes: a especificação do núcleo (feita de modo comportamental em alguma HDL), a geração de uma *netlist* da implementação para a criação de fórmulas lógicas e a efetiva utilização de um provador, que permite determinar então se a implementação atende ou não a especificação desejada.

Outras propostas utilizam assistentes de provas para validar seus sistemas, como em Pitchumani and Stabler (1982), onde mecanismos da lógica de Floy-Hoare apresentados na Seção 2.3.3 são propostos como complemento à simulação ou mesmo a sua substituição em sistemas de hardware síncronos descritos com uma linguagem de transferência de registro através de duas fases distintas: um gerador de verificação de condição, que aceita um programa com asserções fixadas e gera todas as condições de verificação, e um provador de teoremas, que é alimentado com essas condições juntamente com os axiomas necessários.

Referente ao uso de provadores de teoremas, a ferramenta Spectool, que usa o provador Clio (sistema para verificar propriedades de programas escritos na linguagem funcional Caliban) que através de normalização provê a verificação de hardware, como microprocessadores e circuitos com pipeline, é proposta em Srivas (1991). As propriedades a serem verificadas são expressas como fórmulas da lógica de predicados de primeira ordem e a Spectool gera a especificação do hardware em Caliban. Outra abordagem é proposta em Rajan et al. (1997) com uso do PVS para verificação formal de hardware integrando *model-checking* baseado em μ-calculus, uma lógica modal proposicional, com a prova de teoremas para verificar complexos projetos de hardware em níveis de arquitetura, registradores e portas, bem como a verificação de *testbench* utilizados para verificação de hardware, como somadores e multiplicadores, visando a aplicação de testes mais confiáveis.

A utilização do PVS em implementações com FPGA em diversos níveis de abstração também é explorada em Deng (2011), que foca em meios de provar a correção de sua funcionalidade. São apresentadas especificações de registradores, máquinas de estados finitos e outras, além da prova de correção de um somador que possui 18 entradas e soma apenas os 16 maiores valores. Também abordando o PVS para verificação de hardware temos o trabalho de Owre et al. (1994), que apresenta exemplo de verificação de um microprocessador com pipeline e também de um somador completo.

Uma metodologia é proposta em Ayala-Rincón and Sant'Ana (2006) para verificação de reescrita de especificações em ELAN, um ambiente para especificação e prototipagem de sistemas dedutivos, onde podem ser feitos testes e simulações, traduzindo essas especificações para PVS para geração de pares críticos (um par de termos com condições formado pela sobreposição de seus lados esquerdos) e propriedades adicionais a serem verificadas também em PVS, gerando então um certificado de correção da especificação ou detectando um erro que deve ser corrigido na especificação.

Ao tratar transformações conservativas entre definições de especificação e de hardware, descrições recursivas em ordem superior especificadas com HOL4 são traduzidas para código de hardware sintetizável correto por projeto na linguagema Verilog em Gordon et al. (2006). Os trabalhos de Morra et al. (2008); Morra (2010) propõe o uso de lógica de reescrita com especificações em ELAN e Maude para gerar, a partir de descrições textuais, código sintetizável em VHDL.

Trabalhos envolvendo equivalência funcional têm sido propostos para verificar projetos de hardware utilizando abordagens com lógica proposicional e lógica de predicados, como em Emmer et al. (2010) e Khasidashvili et al. (2009), provendo certificados para

blocos de hardware industriais e seu respectivo modelo esquemático e também projetos de memórias. Sistemas de reescrita e técnicas indutivas também são utilizadas para verificar circuitos aritméticos, como somadores, multiplicadores e divisores, especificados com RRL (um provador de teoremas que envolve especificações com sistemas de reescrita) Kapur and Subramaniam (2000).

Esforços vem sendo empregados para não apenas verificar, mas também apontar diretamente o erro e propor soluções para auto-correção de erros encontrados durante o processo de verificação. Uma abordagem para tal é utilizada para circuitos aritméticos em Haghbayan et al. (2014), onde uma comparação é feita entre uma implementação em nível de registradores e uma descrição em alto nível. Este trabalho utiliza técnicas de abstração capazes de modelar em baixo nível descrições em alto nível, chamados *somadores funcionais a nível de bit* e *somador lógico a nível de bit*. Tais técnicas, embora capazes de modelar circuitos aritméticos simples, como somadores e divisores, não são suficientemente flexíveis para modelar sistemas mais complexos, como a abordagem aqui proposta e ilustrada no caso de estudo do próximo Capítulo.

4.3 A Abordagem deste Trabalho

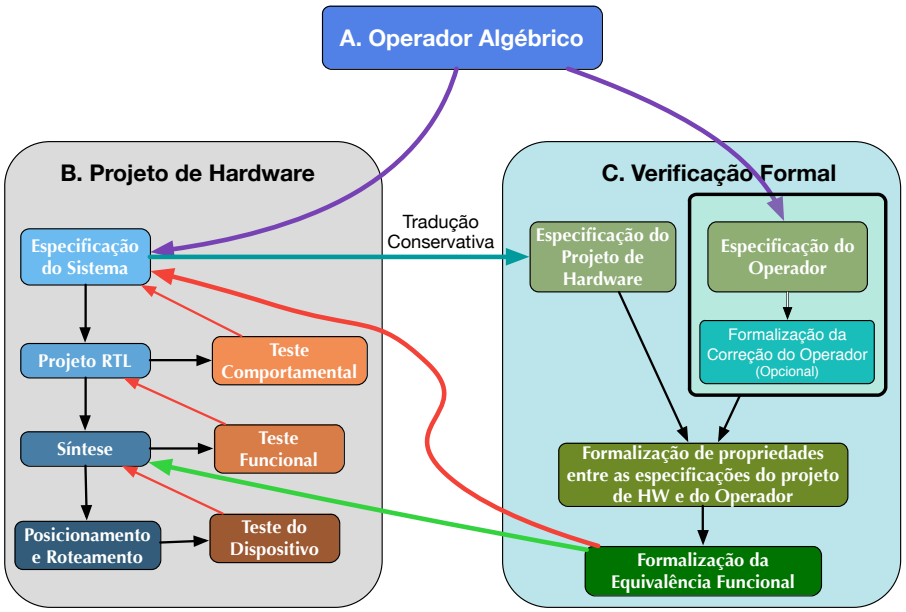

Figura 4.1: Metodologia a ser utilizada neste trabalho

A proposta da abordagem utilizada é obter um operador algébrico (ou algoritmo) correto que depois passará pelo processo de desenvolvimento de um hardware correto a ser certificado formalmente por meio de verificação de sua ***Equivalência Funcional*** com o operador que implementa. Esta abordagem então se divide em três etapas:

4.3.1 Operador Algébrico

Representada pelo diagrama **A.** da Figura 4.1, nesta etapa é definida uma operação matemática (ou algoritmo) a ser desenvolvida em hardware, chamada aqui de operador. Para garantir a correção do projeto desenvolvido, este operador deve ser logicamente correto, já que um projeto que implementa uma solução incorreta é um produto incorreto.

O operador a ser implementado pode ser assumido correto ou assim demonstrado por meio de verificação formal. Quando assumido correto, todo o processo seguinte poderá ser desenvolvido normalmente, porém deve-se levar em conta que a total correção do sistema se dará apenas ao obter-se a prova de que o operador é correto.

A melhor maneira de se definir um operador alvo correto, de modo a evitar erros de especificação e, consequentemente de prova, é especificando-o em um assistente de provas. Deste modo, propriedades do operador podem ser definidas e verificadas, garantindo sua correção. Passa-se então para sua implementação em hardware.

4.3.2 Projeto de hardware

Aqui se encontram as fases de concepção, planejamento e criação de um hardware, segundo o diagrama **B.** da Figura 4.1, seja ele uma abordagem reconfigurável ou não. Nesta etapa seguem-se os passos:

Especificação do Sistema

Após a escolha do operador, define-se a solução para implementá-lo em um circuito alvo, devendo ser capaz de abstrair o operador desejado de modo que sua implementação em hardware se torne viável.

Projeto RTL

Uma vez definida a solução a se implementar, pode-se então modelá-la com uso de uma HDL. Esta deve ser escolhida de acordo com as habilidades e necessidades do projetista, e é gerada então uma descrição a nível de registradores (RTL) do operador.

Teste Comportamental

Esse projeto em RTL é então testado de maneira comportamental por meio de simulações, ou seja, são verificados os dados de saída produzidos com os resultados esperados quando fornecidos dados de entrada específicos. Caso algum erro seja detectado, volta-se à fase de definição da solução para que devidos reparos e ajustes sejam efetuados.

Síntese

Quando produzidos resultados satisfatórios na fase de testes comportamentais, o projeto passa pela fase de síntese, que gera uma *netlist*. Essa *netlist* é uma descrição também

em HDL do projeto, porém em um nível mais baixo e específica para o dispositivo alvo em que o projeto será inserido.

Teste Funcional

Testes funcionais são feitos então para determinar se a *netlist* criada atende à descrição inicial e se o dispositivo de destino do projeto é capaz de suportá-lo. Mais uma vez o projeto passa por testes que validam a síntese, voltando à fase anterior de projeto caso algum erro seja detectado.

Posicionamento e Roteamento

Finalmente, o projeto pode ser posicionado no circuito para o qual foi desenhado, com devido roteamento e colocação de seus componentes para seu pleno funcionamento.

Teste do Dispositivo

Assim, o dispositivo final pode ser analisado e testado, verificando se seus resultados produzidos condizem com o esperado. Caso seja utilizada uma arquitetura fixa, ao findar esta fase de projeto, nenhuma alteração é possível no dispositivo. Assim, a etapa de verificação formal descrita a seguir toma uma importância ainda maior, pois pode certificar melhor um sistema do que apenas os testes e simulações presentes nesta etapa.

4.3.3 Verificação Formal

Neste estágio, representado na Figura 4.1 pelo diagrama **C.**, estão presentes as especificações tanto do operador quanto do projeto de hardware. Para auxiliar o processo de especificação e prova, é desejável o uso de um assistente de prova, que reduz potenciais erros introduzidos pela manipulação humana das especificações e provas. Essa etapa segue os passos:

Especificação do Operador

A especificação formal deve ser capaz de modelar diferentes níveis de abstração presentes em ambas definições. A definição matemática do operador pode ser feita diretamente, pois a descrição correta do operador é facilmente obtida.

Formalização da Correção do Operador

Além de sua especificação, pode ser feita, opcionalmente, a formalização da correção do operador escolhido. Como alguns operadores já são bem conhecidos e amplamente utilizados, sua correção pode ser assumida, reduzindo o tempo de formalização de correção do hardware.

Especificação do Projeto de Hardware

A definição do hardware, pode ser especificada a partir do momento que os projetistas consigam entregar um modelo do projeto, com o fluxo de dados e das funções e módulos do projeto, ou de sua descrição em HDL. Porém, deve ter seu comportamento abstraído e traduzido para modelá-lo corretamente, o que é feito por meio de um mecanismo de engenharia reversa, que deve produzir uma especificação conservativa do hardware em uma linguagem de especificação, tomando-se os devidos cuidados ao lidar com laços de repetição iterativos, como observado na Seção3.3.

Formalização de Propriedades entre as Especificações do Projeto do hardware e do Operador

A partir das especificações, operações extras podem ser necessárias para que a correspondência entre os elementos manipulados entre ambas seja obtida. Isso se dá por conta dos diferentes níveis de abstração e modelagem dos dados utilizados nas definições de hardware. Funções extras também podem ser necessárias para garantir a sincronização do comportamento de ambos, ou seja, que exatamente os mesmos dados são processados por funções equivalentes a todo instante.

Pode-se então especificar lemas e teoremas para modelar a equivalência funcional entre as especificações, ou seja, que quando providos os mesmos dados de entrada, as mesmas saídas são produzidas por ambas descrições.

Formalização da Equivalência Funcional

Assim, quando provados esses lemas e teoremas, pode-se fornecer um certificado de correção do projeto de hardware, caso a equivalência se verifique. Caso contrário, será possível identificar onde o projeto não se comporta de maneira adequada, fornecendo ao projetista informações da localização do erro e possível correção.

Como as especificações frequentemente modelam estruturas de repetição, a aplicação da lógica de Floyd-Hoare é utilizada para verificá-los de maneira relativamente fácil. Um problema se apresenta, porém, ao relacionar funções que utilizam tais estruturas e devem ser comparadas para atestar sua equivalência funcional. Esse mecanismo de prova fornece invariantes que permitem a verificação de alguma propriedade para um laço, mas não é possível expressar em um mesmo invariante uma propriedade que relaciona dois laços distintos. Também não é possível relacionar dois invariantes para atestar que os resultados produzidos pelos laços sejam equivalentes. A solução adaptada aqui é a criação de um novo laço que execute o corpo dos laços a serem comparados de maneira sincronizada. Assim, é possível estabelecer como invariante que a cada passo da execução, os resultados obtidos são equivalentes.

Note que, nessa abordagem a verificação formal pode ser feita em paralelo ao desenvolvimento do circuito, mas nada impede que também seja aplicada posteriormente. Embora neste trabalho esteja sendo aplicada apenas à solução para modelagem do hardware e seu comportamento, também pode ser feita a nível da *netlist* criada na fase de

síntese. Porém, é considerada apenas a lógica do circuito, ou seja, seu comportamento. Quaisquer defeitos apresentados devido a restrições físicas de um circuito específico devem ser analisadas por meio de outros mecanismos.

Deixar a verificação formal como estágio separado do desenvolvimento do dispositivo evita atrasos, pois este processo demanda tempo e minúcia para que as provas sejam produzidas de maneira adequada. Mesmo assim, prejuízos maiores podem ser evitados caso erros sejam detectados nas fases de testes, pois pode-se obter certificados de correção ou detecção de erros antes do processo de posicionamento do hardware.

Capítulo 5

Caso de Estudo

A fim de ilustrar o uso da abordagem apresentada no Capítulo 4, é apresentado aqui um caso de estudo utilizando-a. Foi escolhida uma implementação eficiente desenvolvida em Arias-Garcia et al. (2012a) e Arias-Garcia et al. (2012b), que utiliza FPGAs para criação de um circuito para inversão de matrizes utilizando o algoritmo de Gauss-Jordan apresentado na Seção 2.1.3.

5.1 O Operador e o Projeto de hardware

O operador escolhido foi o algoritmo de Gauss-Jordan apresentado na Seção 2.1.3. Diversos estudos já foram feitos acerca deste algoritmo, portanto, sua correção será assumida para fins de verificação da arquitetura que o implementa.

O projeto de implementação em hardware deste algoritmo foi feito utilizando um FPGA Virtex-5 XC5VLX110T e bibliotecas aritméticas de ponto flutuante. A arquitetura se organiza em dois grandes circuitos: o circuito de controle, e o sistema completo.

As matrizes a serem invertidas são armazenadas em memórias RAM controladas pela unidade de acesso à memória (c.f. Figura 5.1). Cada RAM corresponde a uma coluna da matriz, e seus elementos são inicialmente inseridos linha por linha, um divisor de sinal então conduz cada dado para uma memória RAM respectiva a sua coluna. Um registro adicional armazena a ordem das linhas da matriz, sendo assim, a ordem dos elementos nas memórias não é tão importante, desde que elementos de mesma linha ocupem o mesmo endereço em cada RAM. Ao solicitar uma linha, selecionando o respectivo endereço no registro, um agrupador de sinais seleciona cada elemento deste endereço nas memórias, os une e entrega então uma linha inteira para processamento. O uso do registro de endereços reduz a quantidade de operações de escrita na memória, melhorando a performance do circuito.

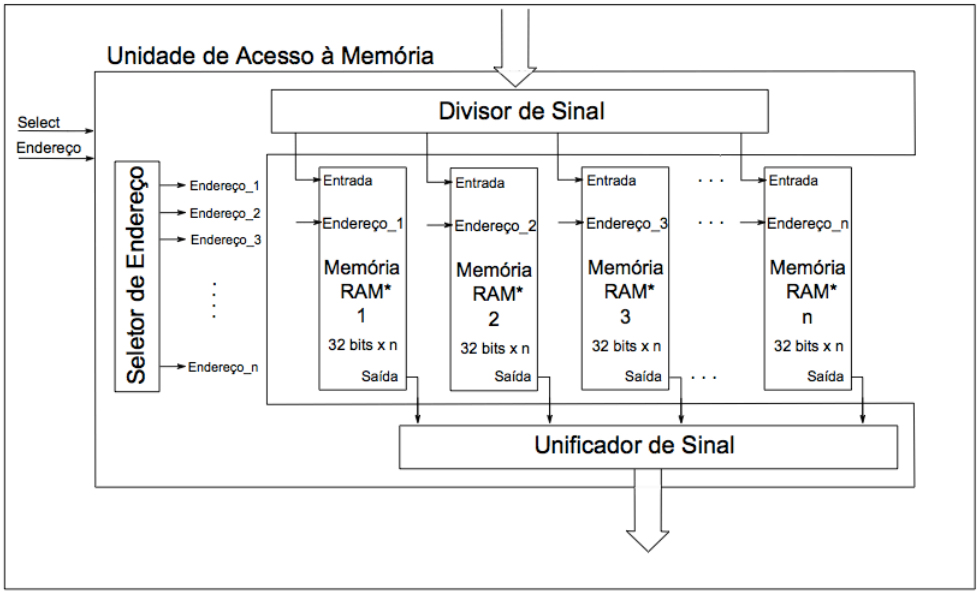

Figura 5.1: Unidade de acesso à memória da arquitetura, adaptada de Arias-Garcia et al. (2012b).

Esses dados são então solicitados e acessados pelo circuito de controle, que é composto por 5 máquinas de estados finitos (FSM) (c.f. Figura 5.2). A FSM principal é uma unidade de controle para a execução do algoritmo e determina o fluxo dos dados entre a unidade de acesso à memória, as unidades aritméticas de ponto flutuante para multiplicação, divisão e soma e as outras 4 máquinas.

Inicialmente, a matriz armazenada é analisada pela FSM responsável pela busca do pivô na primeira coluna. Os dados necessários para sua execução são fornecidos pelo módulo principal, o que também acontece com as outras FSM. O índice da linha é então selecionado segundo o registro de linhas da memória e este dado é entregue à unidade de controle. Este dado então é repassado à FSM que troca linhas, juntamente com o índice da linha atual, que executa sua função através da troca dos valores dos índices armazenados no registro.

Após a troca, a unidade principal aciona a FSM que realiza os processos de triangulação, mandando um bit de controle para indicar que deve ser realizada a triangulação superior, e novamente os dados são buscados na unidade de acesso à memória e fornecidos a esta FSM pela unidade de controle. Ao receber o resultado da triangulação, esta aciona a FSM de normalização, passando a ela os dados necessários a sua execução e recebendo seu resultado para então proceder novamente à unidade de triangulação para a triangulação inferior, que finaliza o processamento.

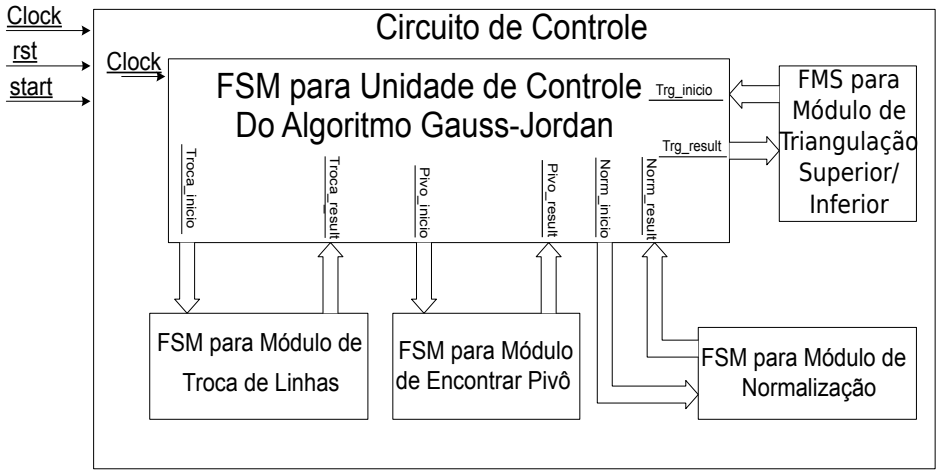

Figura 5.2: Circuito de controle da arquitetura a ser verificada, adaptada de Arias-Garcia et al. (2012b).

A arquitetura geral é então composta da unidade de acesso à memória, da unidade de controle e dos módulos aritméticos, como sumarizado na Figura 5.3.

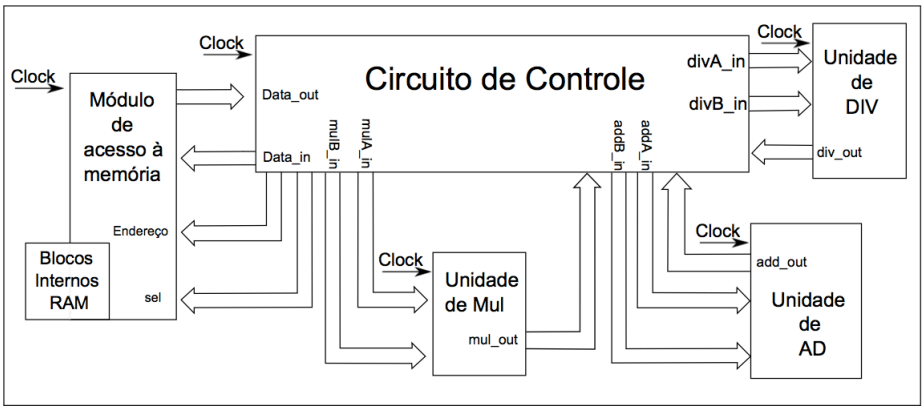

Figura 5.3: A Arquitetura a ser verificada, adaptada de Arias-Garcia et al. (2012b).

5.2 Modelando os Tipos

Para provar a equivalência funcional entre o algoritmo e a arquitetura implementada, o comportamento de ambos deve ser abstraído e modelado utilizando a linguagem de especificação de um assistente de provas. Assim, são necessárias duas especificações distintas, uma para o algoritmo e outra para a arquitetura, onde os tipos de elementos a manipular devem ser definidos.

Como foi utilizado o assistente PVS, uma matriz `M` de dimensões $n \times n$, que é o objeto manipulado na formalização, foi especificado como uma função de tipo `M : [below[`n`], below[`n`]]` $\rightarrow \mathbb{R}$, onde para $n \in \mathbb{N}$, `below[`n`]` indica o tipo dos naturais k onde $k < n$. Assim, para $i, j < n$, `M(`i, j`)` é bem definida como a imagem em $\mathbb{R}$ da função, ou seja, o elemento da linha i e coluna j da matriz.

A escolha pela modelagem das matrizes como funções é que deste modo é possível o uso de funções lambda, ou seja, funções anônimas, para que modificações lineares nas linhas da matriz possam ser aplicadas de maneira direta em apenas um passo, sem prejuízo para a formalização proposta.

Assim, seja `mm_size` o tamanho fixado da matriz especificada, o tipo `matrix`, para modelar matrizes quadradas pode ser especificado como `[below[mm_size], below[mm_size]]` $\rightarrow \mathbb{R}$. Para matrizes estendidas com outra matriz quadrada é definido o tipo `mmatrix`, como sendo `[below[mm_size], below[mm_size*2]]` $\rightarrow \mathbb{R}$.

Além disso, a forma com que os dados são armazenados, transmitidos e processados na arquitetura devem ser analisados e abstraídos. Como a matriz a ser invertida é armazenada na unidade de acesso à memória armazenando cada coluna em uma RAM e o acesso a cada linha é feito por meio de um registro de endereços, a troca de linhas não é feita de maneira física entre os elementos armazenados na memória, ao contrário do que aconteceria na definição algorítmica padrão. Deste modo, esse comportamento do armazenamento dos dados, seu acesso e manipulação também são modelados.

Para modelar matrizes quadradas para a arquitetura, duas estruturas são necessárias: uma para a matriz de fato armazenada em RAMs e outra para o registro de endereços. A matriz arquitetural é então representada como uma tupla da forma $\langle m, r \rangle$, onde m é uma matriz de tipo `matrix` e r é uma função bijetiva de tipo `below[mm_size]` $\rightarrow$ `below[mm_size]`. Esse tipo é então chamado `mat_reg` e cada elemento de uma matriz `M` com este tipo na posição i, j, para $0 \leq i, j <$ `mm_size`, é dado por `M‘m(M‘r(`i`), `j`)`. O fato de o registro ser modelado como uma função bijetiva garante um mapeamento único para cada linha física da matriz.

O mesmo raciocínio é usado para modelar matrizes arquiteturais estendidas, dadas como o tipo `mmat_reg`, sendo tuplas $\langle m, r \rangle$, onde o primeiro elemento é do tipo `mmatrix` e o segundo do tipo `below[mm_size]` $\rightarrow$ `below[mm_size]`. Seus elementos são então dados por `M‘m(M‘r(`i`), `j`)`, onde $0 \leq i <$ `mm_size` e $0 \leq j <$ `mm_size*2`.

Dados os tipos a serem utilizados, a especificação foi dividida em três partes principais, uma para a definição algorítmica, outra para a arquitetura e uma terceira para as funções e lemas necessários à prova de equivalência funcional das duas primeiras. Foi utilizada a técnica de tradução de definições imperativas em recursivas apresentada na

Seção 3.3 para ambas especificações dos laços de repetição do algoritmo GJ, dadas a seguir.

5.3 Especificação da Definição Algorítmica

São utilizadas variáveis dos tipos definidos na Seção 5.2 para ambas especificações como parâmetros de funções que modelam o comportamento de cada etapa de GJ. Inicialmente, a entrada do algoritmo é uma matriz estendida cuja primeira matriz quadrada é assumida como não singular e a segunda sua identidade. Neste trabalho é assumido também que sempre haverá um pivô a ser escolhido cujo valor não é nulo, ou seja, que a matriz é inversível, pois sempre trabalha-se com matrizes não singulares.

A função para o passo de escolha do pivô é dada por `find_pivot`, definida abaixo:

Algoritmo 5.3.1: Função auxiliar para escolha do pivô

```
find_pivot(A: mmatrix, i: below[mm_size],
           (j : below[mm_size] | j <= i),
           (k:  below[mm_size] | k >= j)) :
  RECURSIVE below[mm_size] =
   IF i < mm_size -1 THEN
     IF A(k,j) < A(i,j) THEN
          find_pivot(A, i+1, j, i)
     ELSE find_pivot(A, i+1, j, k)
     ENDIF
  ELSIF A(k,j) < A(i,j) THEN i
  ELSE k
  ENDIF
MEASURE mm_size - i
```

Para encontrar a linha que contém o pivô da `j`-ésima coluna de uma matriz `A`, a chamada dessa função é dada como `find_pivot(A, j, j, j)` pela chamada da função auxiliar `f_p(A, j)`. Esse artifício é utilizado apenas para simplificar a leitura da função que utiliza a escolha do pivô em seu processamento.

Algumas características interessantes do assistente PVS podem ser observadas nessa função, que é definida recursivamente, sendo elas:

- O uso de *tipos dependentes*, permitindo restringir o valor dos parâmetros `i, j` e `k` de acordo com o espaço de busca das linhas de elementos da matriz `A`, ou seja, o conjunto de índices de linhas para os elementos na `j`-ésima coluna a partir do elemento `A(j,j)` da diagonal até o último elemento desta coluna. Note que `j` permanece inalterado como índice de coluna na posição diagonal, enquanto `k` armazena o índice da linha atual candidata a conter o pivô e `i` é incrementado para procurar o índice da linha que contenha um pivô possivelmente maior. Deste modo, a dependência de tipos dá a relação $\texttt{j} \leq \texttt{k} \leq \texttt{i} < \texttt{mm\_size}$ entre os parâmetros, e argumentos que não obedecem tal dependência não são permitidos.

- O uso de uma medida que garanta a terminalidade da função, como já mencionado também na Seção 3.3. Essa medida, dada por `MEASURE` é uma função em naturais definida para o tipo dos parâmetros da função e é utilizada para demonstrar que a esta termina e está assim bem definida sobre argumentos do tipo dos parâmetros, devendo decrementar a cada chamada recursiva, permitindo provas indutivas sobre essas definições. Para a função `find_pivot`, o parâmetro `i` é incrementado em cada chamada recursiva, enquanto a dimensão da matriz permanece inalterada, decrementando a medida `mm_size - i`.

O índice dado como resultado dessa função para uma dada coluna é usado na troca de linhas dada pela função auxiliar `partial_pivoting`, que recebe também o índice da outra linha e a matriz onde será efeituada a troca:

Algoritmo 5.3.2: Função para troca de linhas

```
partial_pivoting(A: mmatrix, i,j:below[mm_size]): mmatrix =
LAMBDA (k: below[mm_size], l: below[mm_size*2]):
             IF (k=i) THEN
      A(j,l)
     ELSIF (k=j) THEN
      A(i,l)
     ELSE
A(k,l)
     ENDIF
```

Pode-se então transformar os elementos de uma coluna posicionados em índices de linhas maiores que zero, através da função `elim_col_below`. Inicialmente é selecionada a linha com o pivô esta é trocada pela linha atual `i` e realiza as devidas transformações lineares utilizando a linha que contém o pivô e aquelas abaixo dela.

Algoritmo 5.3.3: Função parcial para zerar uma coluna abaixo da diagonal principal

```
elim_col_below(i: below[mm_size], A : mmatrix ): mmatrix =
 LAMBDA(m: below[mm_size], n:below[mm_size*2]) :
  LET Ap =  partial_pivoting(A, f_p(A,i), i) IN
    IF m <= i THEN
      Ap(m,n)
    ELSE
      Ap(m,n) - Ap(i,n) * Ap(m,i)  / Ap(i,i)
    ENDIF
```

Note que essa função utliza uma função `LAMBDA` sobre os parâmetros `m` e `n`, encontrando a linha do pivô e trocando-a com a `i`-ésima linha através da função `partial_pivoting`, produzindo então uma matriz cujos valores abaixo do elemento da diagonal principal na posição `i, i` são transformados em zero.

Em uma abordagem imperativa, a função `elim_col_below` pode ser aplicada com

comandos de repetição com índices `i` de zero até `mm_size` -1 para computar a triangulação superior de uma matriz estendida `A`. Em PVS é usada a transformação dada na Seção 3.3, passando como argumento para o `for_rec` a função `elim_col_below` como seu corpo. O `for` utilizado abaixo se encarrega de ajustar os parâmetros de início e o contador inicialmente com os mesmos valores ao chamar `for_rec`, também escrito assim para simplificar a leitura.

Algoritmo 5.3.4: Função triangulação superior

```
up_trg(A : mmatrix): mmatrix =
 for(0, mm_size - 1, A, elim_col_below)
```

O resultado dessa função é uma matriz estendida cuja primeira matriz quadrada é triangular superior. Tais matrizes são modeladas como do tipo `up_trg_mmatrix` dado por `a: mmatrix | up_triang?(a)`. Embora a função não garanta por si só tal resultado, ele é obtido por um lema auxiliar, a fim de garantir a correção do algoritmo. Matrizes desse tipo são então aceitas para o passo de normalização, feito pela função `normalization`, outra função anônima responsável por dividir todo elemento de cada linha por seu pivô, isto é, o elemento da diagonal principal.

Algoritmo 5.3.5: Função para normalização

```
normalization(A: up_trg_mmatrix) : mmatrix =
LAMBDA(i: below[mm_size], j: below[mm_size*2]):
A(i,j)/A(i,i)
```

A normalização resulta em uma matriz estendida cuja primeira matriz quadrada é triangular superior e os elementos de sua diagonal são iguais a um. Semelhante ao que ocorre com a triangulação superior, um tipo mais restrito de matriz, chamado `normalized?` é modelado como `a: mmatrix | FORALL (i : below[mm_size]) : a(i,i) = 1 AND up_triang?(a)` e um lema auxiliar verifica que a matriz produzida por `normalization` é deste tipo, que será então utilizada no processo de triangulação inferior.

Essa função é análoga à de triangulação superior, ou seja, utiliza uma função auxiliar como seu corpo e uma tradução recursiva do comando `for`. Aqui, porém, outra definição deste laço, chamada `for_down` é utilizada, pois a triangulação inferior se inicia na última linha da matriz e termina na primeira. Ela simplesmente se encarrega de fazer a passagem dos parâmetros corretamente quando se deseja um decremento ao invés de incremento do contador do laço. É responsável também pela passagem dos devidos parâmetros ao corpo da função, que neste caso é dada por `elim_col_above`, similar a `elim_col_below`, porém sem a necessidade de escolha de pivô e troca de linhas e com operações elementares mais simples.

Ao combinar essas funções que representam as fases de execução do método de GJ, a especificação do algoritmo é dada pela função `Gauss_Jordan_math` abaixo. Tal função recebe como parâmetro apenas uma matriz quadrada, que é então estendida

Algoritmo 5.3.6: Função para zerar uma coluna acima da diagonal

```
elim_col_above(i : below[mm_size], A :norm_mmatrix): norm_mmatrix =
  LAMBDA(m: below[mm_size],n:below[mm_size*2]) :
    IF m >= i THEN A(m,n)
    ELSE A(m,n) - A(i,n) * A(m,i)
    ENDIF
```

Algoritmo 5.3.7: Função para triangulação inferior

```
low_trg(A : norm_mmatrix): norm_mmatrix =
for_down[norm_mmatrix](mm_size-1, 0, A, elim_col_above)
```

com sua identidade pela função `matrix_to_mmatrix`, e produz como resultado uma matriz quadrada que é a inversa da matriz original, selecionada da matriz estendida por `second_mmatrix`.

Algoritmo 5.3.8: Função que representa o algoritmo GJ

```
gauss_Jordan_math (a: matrix): matrix =
   second_mmatrix(low_trg(normalization(
   up_trg(matrix_to_mmatrix(a)))))
```

5.4 Especificação da Arquitetura

A especificação dada em PVS da arquitetura proposta como solução para o algoritmo GJ se dá de maneira similar à especificação matemática. Porém, é necessário modelar o registro de endereços das linhas da matriz e tomar o devido cuidado ao utilizá-lo, pois expressa a posição virtual das linhas, não podendo então sempre acessá-las de maneira sequencial. Para correto acesso aos elementos da matriz, é necessária uma função auxiliar que encontre a real linha física da matriz, armazenada em memória, associada ao seu índice virtual. Tal função é dada por `real_pivot` no Algoritmo 5.4.1,

que é chamada com os argumentos `r` e `i` pela função `r_p` para facilitar a leitura.

Algoritmo 5.4.1: Função para encontrar o índice físico de linha da matriz

```
real_pivot(r:reg_mem, i,j:below[mm_size]):
RECURSIVE below[mm_size + 1] =
 IF r(j) = i THEN
  j
 ELSIF j = mm_size - 1 THEN
  mm_size
 ELSE
  real_pivot(r, i, j+1)
 ENDIF
MEASURE mm_size - j
```

Uma propriedade importante de se observar sobre esta função é que, ao selecionar o conteúdo do índice de linha no registro e procurar seu índice real, o resultado será ele mesmo. Esta propriedade é bastante importante para as provas posteriores de equivalência e é dada por:

Lema 5.4.1 (Relação entre índices)**.**

```
idx_relation : LEMMA
FORALL  (r: reg_mem, i:below[mm_size], (k:below[mm_size] | k <=i)):
     i =  real_pivot(r, r(i), k)
```

Demonstração.
Note que a função auxiliar que retorna o real índice começa sua busca a partir de um valor qualquer dentro do intervalo permitido. Porém, a chamada da função principal garante que a busca é feita em todo registro, pois chama a função auxiliar com o valor 0 para o parâmetro de primeiro índice a ser analisado.
Esta prova se dá por indução na diferença entre o índice procurado e o início da busca, tendo como B.I. o caso em que o índice procurado é o próprio índice de início, o que é trivial. O P.I. é dado ao se procurar o índice em um intervalo menor de índices disponíveis, podendo usar assim a H.I. e o fato de que os registros são dados por funções bijetivas. Assim, a instanciação correta das definições leva ao resultado desejado. □

Deve-se sempre ser considerado o registro ao selecionar os elementos da matriz, para que seja escolhida a linha correta para cada operação. Assim, a escolha do pivô (c.f. Algoritmo 5.4.2) analisa a matriz segundo linhas armazenadas no registro a cada passo. Inicia-se a pesquisa com a linha referente ao índice do registro do início da busca. Se o elemento da linha indicada pelo registro com o índice atual for um candidato a pivô melhor do que aquele com que é comparado, este índice é escolhido como novo índice de linha que contém o pivô. A busca então segue através do incremento do índice do registro e comparação dos elementos com o indicado pelo índice do pivô.

Algoritmo 5.4.2: Função para encontrar o pivô

```
find_pivot_arch(A:mmat_reg, i: below[mm_size],
 (j : below[mm_size] | j <= i) ,
             (k:  below[mm_size] | k >= j)):
             RECURSIVE below[mm_size] =
  IF i < mm_size -1 THEN
       IF A‘m(A‘r(k),j) < A‘m(A‘r(i),j) THEN
        find_pivot_arch(A, i+1, j, i)
       ELSE
        find_pivot_arch(A, i+1, j, k)
       ENDIF
  ELSIF  A‘m(A‘r(k),j) < A‘m(A‘r(i),j) THEN
   i
  ELSE
   k
  ENDIF
MEASURE mm_size -i
```

A função `find_pivot_arch` é utilizada pela função `f_p_arch`, que a chama com os devidos parâmetros, bem como acontece na especificação do operador dada na seção anterior, simplificando a leitura.

Como é retornado apenas o índice do registro referente à linha que contém o pivô, a troca de linhas é feita apenas alterando os índices internos a este registro, tal como no Algoritmo 5.4.3.

Algoritmo 5.4.3: Função para troca de linhas

```
partial_pivoting_arch(r:reg_mem,
i: below[mm_size], j:below[mm_size]):
reg_mem =
LAMBDA (k: below[mm_size]):
             IF (k=i) THEN
      r(j)
     ELSIF (k=j) THEN
      r(i)
     ELSE
      r(k)
     ENDIF
```

Cada passo da triangulação superior, dado pela função `elim_col_below_arch` conforme o Algoritmo 5.4.4, é feito analisando se na coluna em questão o índice real da linha analisada indica uma linha que deve ser alterada ou não. Dado o tipo como as matrizes foram modeladas para a arquitetura, tal função (bem como todas as outras que modelam a arquitetura) deve retornar dois valores, uma nova matriz e um novo registro. Inicialmente, então, é necessário que o registro da nova matriz seja alterado pela troca da linha atual com aquela que contém o pivô da coluna analisada. Caso nenhuma alteração precise ser feita, o elemento se mantém, caso contrário as devidas operações são feitas tomando cuidado para verificar no registro a linha que contém o pivô para que as transformações lineares sejam feitas corretamente.

Algoritmo 5.4.4: Função para zerar uma coluna abaixo da diagonal

```
elim_col_below_arch(k:below[mm_size], A: mmat_reg): mmat_reg =
LET reg = partial_pivoting_arch(A‘r,k,f_p_arch(A,k)) IN
(#r := reg,
   m := LAMBDA(i: below[mm_size], j: below[mm_size*2]):
  IF r_p(reg, i) <= k THEN
   A‘m(i,j)
  ELSE
   A‘m(i,j) -  A‘m(reg(k),j)* A‘m(i,k)/ A‘m(reg(k),k)
  ENDIF #)
```

A triangulação superior é feita aplicando-se recursivamente a função indicada para zerar apenas uma coluna em toda matriz. Isso também é feito utilizando a função para um passo como corpo de um `for` recursivo, como visto no Algoritmo 5.4.5.

Algoritmo 5.4.5: Função para triangulação superior

```
up_trg_arch(A : mmat_reg): mmat_reg =
     for[mmat_reg](0, mm_size - 1, A, elim_col_below_arch)
```

Esta função produz também uma matriz triangular superior, resultado dado por

um lema auxiliar. Tais matrizes serão utilizadas como entrada para o processo de normalização e são dadas como o tipo `up_trg_mmatrix_arch` definido por `a: mmat_reg | up_triang_arch?(a)`. A normalização, dada pelo Algoritmo 5.4.6 como `normalization_arch`, também utiliza os índices reais e virtuais de maneira correta para produzir uma nova matriz estendida e um novo registro.

Algoritmo 5.4.6: Função para normalização

```
normalization_arch( A: up_trg_mmatrix_arch) :up_trg_mmatrix_arch =
(#r := A‘r,
  m := LAMBDA(i: below[mm_size], j: below[mm_size*2]):
A‘m(i ,j)/A‘m(i,r_p(A‘r, i))#)
```

Seu resultado são matrizes estendidas cuja primeira matriz quadrada é triangular superior e tem os elementos da diagonal principal iguais a um, e o registro da matriz resultante não se altera. Ambos resultados são dados também por lemas auxiliares. Outro tipo de matriz restrita a estas condições é definido por `normalized_arch` como `a:mmat_reg | FORALL (i : below[mm_size]) : a‘m(a‘r(i),i) = 1) AND up_trg_arch(a)`. Estas são então usadas para o processo de triangulação inferior, dada por `lw_trg_arch` no Algoritmo 5.4.8, definida pelo `for_down` recursivo já mencionado anteriormente e utilizando como função de corpo a função `elim_col_above` dada no Algoritmo 5.4.7, responsável por deixar os elementos acima do pivô iguais a zero de maneira similar ao passo para zerar uma coluna para a triangulação superior.

Algoritmo 5.4.7: Função para zerar uma coluna acima da diagonal

```
elim_col_above_arch(k : below[mm_size],
A : norm_mmatrix_arch): norm_mmatrix_arch =
   (#r := A‘r,
   m := LAMBDA(i: below[mm_size],
   j: below[mm_size*2]):
  IF r_p(A‘r, i) >= k THEN
   A‘m(i,j)
  ELSE
   A‘m(i,j) - A‘m(A‘r(k),j)*A‘m(i,k)
  ENDIF #)
```

Algoritmo 5.4.8: Função para triangulação inferior

```
lw_trg_arch(A : mmat_reg): mmat_reg =
LET new_function =
    LAMBDA (k: subrange(0, mm_size - 1),
    a: norm_mmatrix_arch):
    elim_col_above_arch(-1 - k + mm_size, a) IN
     for_it(0, 0, mm_size - 1, A, new_function)
```

Toda arquitetura para inversão de matrizes com o algoritmo GJ é então abstraída pela composição das funções responsáveis por suas fases de processamento, dada pela função abaixo, que recebe uma única matriz quadrada e seu registro, estendendo-a com sua identidade e selecionando a segunda matriz do resultado, que é a inversa.

Algoritmo 5.4.9: Função que representa o algoritmo GJ

```
Gauss_Jordan_arch (A: mat_reg): mat_reg =
 second_mmatrix_arch(low_trg_arch(
   normalization_arch(up_trg_arch(
                   matrix_to_mmatrix_arch(A)))))
```

5.5 Formalização da Equivalência entre as Definições Matemática e Arquitetural

Assumindo que a definição matemática/algorítmica é correta, o circuito que o implementa é certificado provando-se a equivalência funcional entre ambas especificações. A formalização em PVS expressa propriedades de ambos sistemas como lemas sobre as funções envolvidas e provê, como corolário principal, a verificação da equivalência entre as funções `Gauss_Jordan_math` e `Gauss_Jordan_arch`. Este corolário deve indicar que duas matrizes equivalentes produzem o mesmo resultado quando aplicadas as duas definições, ou seja, para todas `mmatrix A` e `mmat_reg B`, `A` $\equiv$ `B` implica `Gauss_Jordan_math(A)` $\equiv$ `Gauss_Jordan_math(B)`.

Como os dados manipulados por ambas especificações não é o mesmo, é preciso especificar também predicados que relacionam os dois tipos de matrizes. Por exemplo, um para verificar se matrizes, com e sem registro, são equivalentes. Isso é dado por:

```
eq_mat_arq?(A : mmatrix, B : mmat_reg) : bool =
  FORALL (i : below[mm_size], j : below[2* mm_size]) :
       A(i,j) = B‘m(B‘r(i),j)
```

Como ambas especificações possuem laços definidos recursivamente de maneira diferente para executar os passos de triangulação, é necessário fazer um ajuste para que a equivalência possa ser provada. Isso porque o invariante a ser dado para cada um seria diferente, tornando impossível relacionar os resultados parciais de cada iteração se aplicados separadamente. Assim, funções auxiliares são criadas para organizar e sincronizar as execuções de ambas abordagens, matemática e arquitetural. Isto foi feito de maneira que o invariante a ser escolhido é dado como a equivalência entre matrizes após cada passo de execução, garantindo assim a pós-condição, pois estabelece que após cada passo sincronizado realizado sobre entradas equivalentes, o resultado deve ser matrizes também equivalentes.

Para este ajuste, alguns tipos e funções adicionais são necessários para assegurar a sincronização. Para os tipos de matrizes estendidas iniciais, é criado o tipo `sync_math_arch_mmatrix` definido como uma tupla de elementos com tipos `[mmatrix, mmat_reg]`, cujos elementos são pares da forma `(# A, B #)`, onde `A : mmatrix` e `B : mmat_reg`. Este tipo é então usado como parâmetro da função `sync_elim_col_below` descrita abaixo, que sincroniza as funções `elim_col_below` e `elim_col_below_arch`. A seleção do primeiro e do segundo componente (as matrizes) de um elemento deste tipo, `AB`, é dada por `AB‘1` e `AB‘2`.

Algoritmo 5.5.1: Função que sincroniza um passo da triangulação superior

```
sync_elim_col_below(i: below[mm_size], AB :  sync_math_arch_mmatrix):
 sync_math_arch_mmatrix =  (# elim_col_below(i, AB‘1),
           elim_col_below_arch(i, AB‘2) #)
```

É aplicada então a função correta respectiva ao elemento da tupla, passando-o como parâmetro juntamente com um índice para sua atuação. Esta função é usada então como corpo de um novo `for` recursivo, visando executar todos os passos envolvidos em ambas definições ao mesmo tempo, de maneira sincronizada.

Algoritmo 5.5.2: Função que sincroniza a triangulação superior

```
sync_up_trg(AB :  sync_math_arch_mmatrix):  sync_math_arch_mmatrix =
  for_it(0,0, mm_size - 1, AB, sync_elim_col_below)
```

O processo de formalização inicia-se pela prova da equivalência das funções mais básicas, ou seja, que seu comportamento é o mesmo sobre matrizes equivalentes, produzindo o mesmo resultado. Note, em particular, que a prova indutiva da equivalência funcional para função anterior é dada pela prova do invariante $\mathcal{I}$`(k, AB`$_k$`)` se e somente

se $\texttt{AB}_k\texttt{`1} \equiv \texttt{AB}_k\texttt{`2}$, isto é, após cada passo sincronizado de triangulação superior, os componentes da tupla manipulada continuam equivalentes. Isso implicará que os resultados de triangulação superior são matrizes equivalentes de tipos correspondentes.

Aqui são apresentados os esquemas de prova utilizados para verificar os principais lemas deste trabalho. Parte dos sequentes utilizados e de suas árvores de prova serão introduzidos, porém serão omitidos alguns detalhes, principalmente referentes a checagem de tipos e provas de ramos com casos similares. Sendo assim, tem-se:

5.5.1 Formalização da Equivalência da Escolha do Pivô

Como o primeiro passo a ser executado no algoritmo é a escolha do pivô, o lema que relaciona ambas funções para esta tarefa deve ser inicialmente provado para poder ser utilizado como parte da correção de outros lemas. Prova-se inicialmente a equivalência das funções auxiliares, seguindo-se a prova das funções principais que as chamam, assim sendo, tem-se:

Lema 5.5.1 (O Pivô escolhido é o mesmo em matrizes equivalentes)**.**

```
find_pivot_coincide : LEMMA
  FORALL (A: mmatrix, B:mmat_reg, j: below[mm_size],
            (i : below[mm_size] | i >= j),
            (k:  below[mm_size] | k >= j)):
              eq_mat_arq?(A,B) IMPLIES
               find_pivot(A,i,j,k) = find_pivot_arch(B,i,j,k)

f_p_is_the_same : LEMMA
    FORALL (A: mmatrix, B:mmat_reg,
            j: below[mm_size]):
              eq_mat_arq?(A,B) IMPLIES
                f_p(A,j) = f_p_arch(B,j)
```

Demonstração.

O esquema das provas é sintetizado nas Figuras 5.4 e 5.5, onde a prova para `f_p_is_the_same` é apenas a aplicação direta do lema auxiliar `find_pivot_coincide` ajustando-se os parâmetros usados em sua chamada.

A prova do lema auxiliar é dada por indução, que segundo as medidas envolvidas nas definições das funções `find_pivot` e `find_pivot_arch`, deve ser guiada por `mm_size - j` e então aninhada com uma indução em `mm_size - i`. Isso porque se deseja uma indução no parâmetro i, que é dependente de j, que precisa então ser uma variável livre para que ocorra a indução em i. Obtém-se assim a H.I. de que quando procurado em um índice maior, ou seja, quando o intervalo de índices para busca do pivô é menor, os pivôs encontrados coincidem quando procurados em linhas de índice maior que da linha inicial.

É necessário verificar a estrutura das duas funções na conclusão, para poder aplicar a H.I, que quando expandidas nos levam a casos do tipo [1] da Figura 5.5, que tem como conclusão que o pivô encontrado é o mesmo, esteja ele possivelmente na primeira linha procurada ou não.

Os outros casos do tipo [2] são referentes à equivalência das matrizes, e tratam de questionamentos tais como tendo premissa de que determinado elemento é (ou não) candidato a pivô na definição matemática e a conclusão deve mostrar que o da definição arquitetural também o é, ou vice-versa. Essas provas são triviais, já que por hipótese tem-se que a matriz matemática e a arquitetural são equivalentes.

□

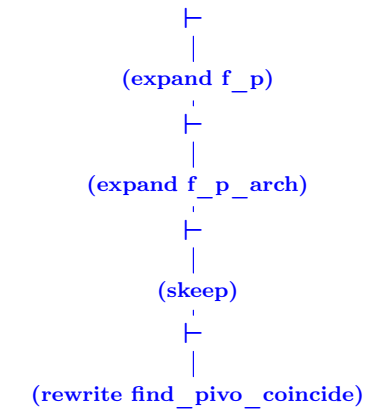

Figura 5.4: Árvore de prova para `f_p_is_the_same`

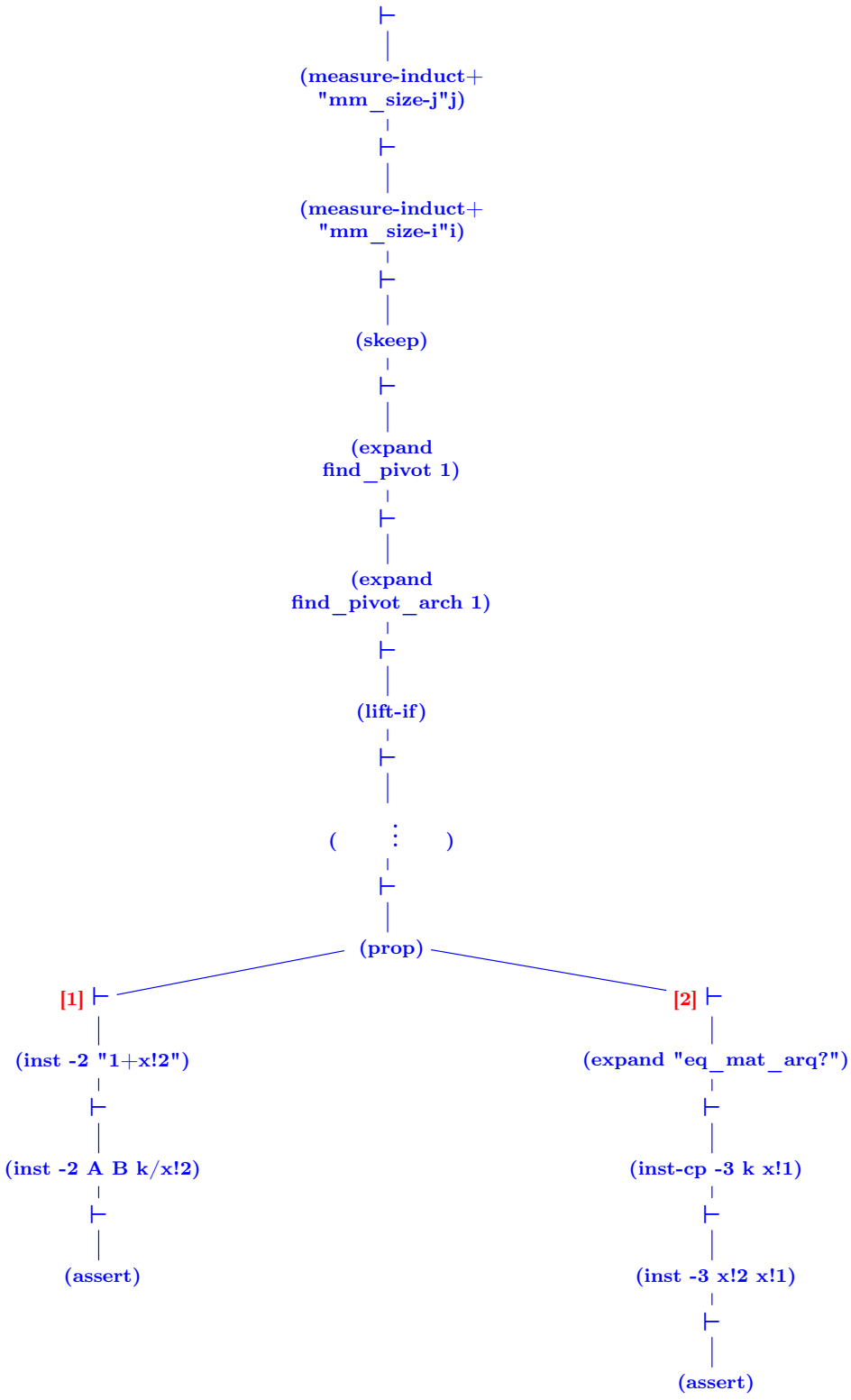

Figura 5.5: Árvore de prova de `find_pivot_coincide`

5.5.2 Formalização da Equivalência quando Troca-se Linhas

O próximo passo é a troca da linha corrente com a que contém o pivô. A prova de que ambas funções para esta tarefa são equivalentes é mais geral, englobando a troca de quaisquer duas linhas em matrizes equivalentes e é dada por:

Lema 5.5.2 (Trocar linhas mantém a equivalência)**.**

```
change_rows_preserves_equivalence : LEMMA
FORALL (A: mmatrix, B:mmat_reg, i,j: below[mm_size]):
eq_mat_arq?(A,B) IMPLIES
LET Bn: mmat_reg = (# r:= partial_pivoting_arch(B`r,i,j), m := B`m#) IN
eq_mat_arq?(partial_pivoting(A,i,j), Bn)
```

Demonstração.

Como essas funções são dadas em termos de funções lambda, observando suas definições e apenas com a instanciação dos elementos corretos de modo que a troca das linhas ocorra é suficiente para completar a prova, como sumarizado na Figura 5.6.

□

5.5.3 Formalização da Equivalência de cada Passo da Triangulação superior

Como apresentado nas Seções 5.3 e 5.4, o passo de triangulação superior foi dividido em duas etapas: uma para eliminação de uma coluna apenas, e outra recursiva para aplicar o procedimento em toda matriz.

A prova para triangulação superior como um todo será feita posteriormente utilizando o ajuste apresentado no início dessa Seção e a prova para cada passo da triangulação fornecida aqui. Deve-se mostrar que, eliminando a mesma coluna abaixo do elemento da diagonal principal em matrizes equivalentes, o resultado é equivalente. Sendo assim, tem-se:

Lema 5.5.3 (Os passos de Triangulação Superior Coincidem)**.**

```
arch_eliminates_col_below : LEMMA
FORALL (A: mmatrix, B:mmat_reg, i:below[mm_size]) :
       eq_mat_arq?(A, B) IMPLIES
       eq_mat_arq?(elim_col_below(i,A),elim_col_below_arch(i,B))
```

Demonstração.
A prova é baseada apenas na estrutura das funções e correta manipulação dos elementos presentes nelas e dos índices envolvidos. São expandidas as definições de `eq_mat_arq?`,

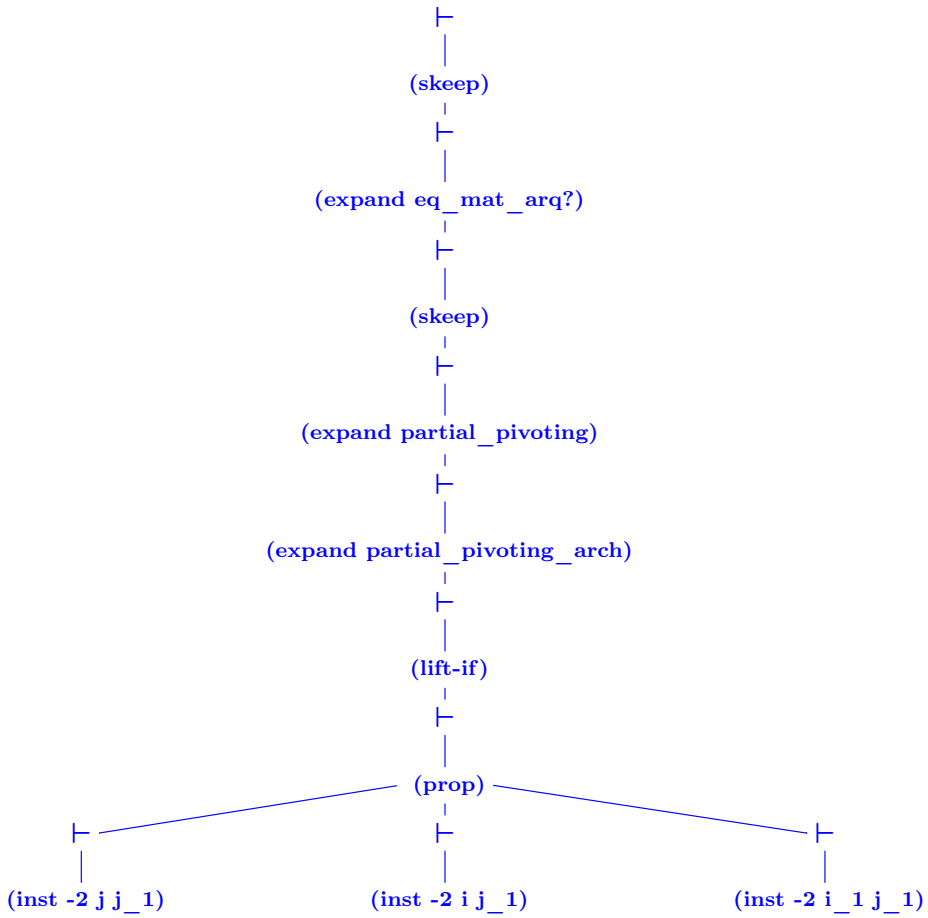

Figura 5.6: Árvore de prova de `change_rows_preserves_equivalence`

`elim_col_below` e `elim_col_below_arch`, mantendo-se uma cópia de `eq_mat_arq?` para fins de utilização dos lemas anteriores, já que essa premissa é essencial para aplicá-los.

Assim, são verificados os casos em que operações devem ou não ser feitas nos elementos das linhas. Em ambos os casos, basta aplicar os resultados referentes à equivalência da troca de linhas e de escolha do pivô já obtidos, bem como lemas sobre a comutatividade da troca de linhas em matrizes equivalentes.

□

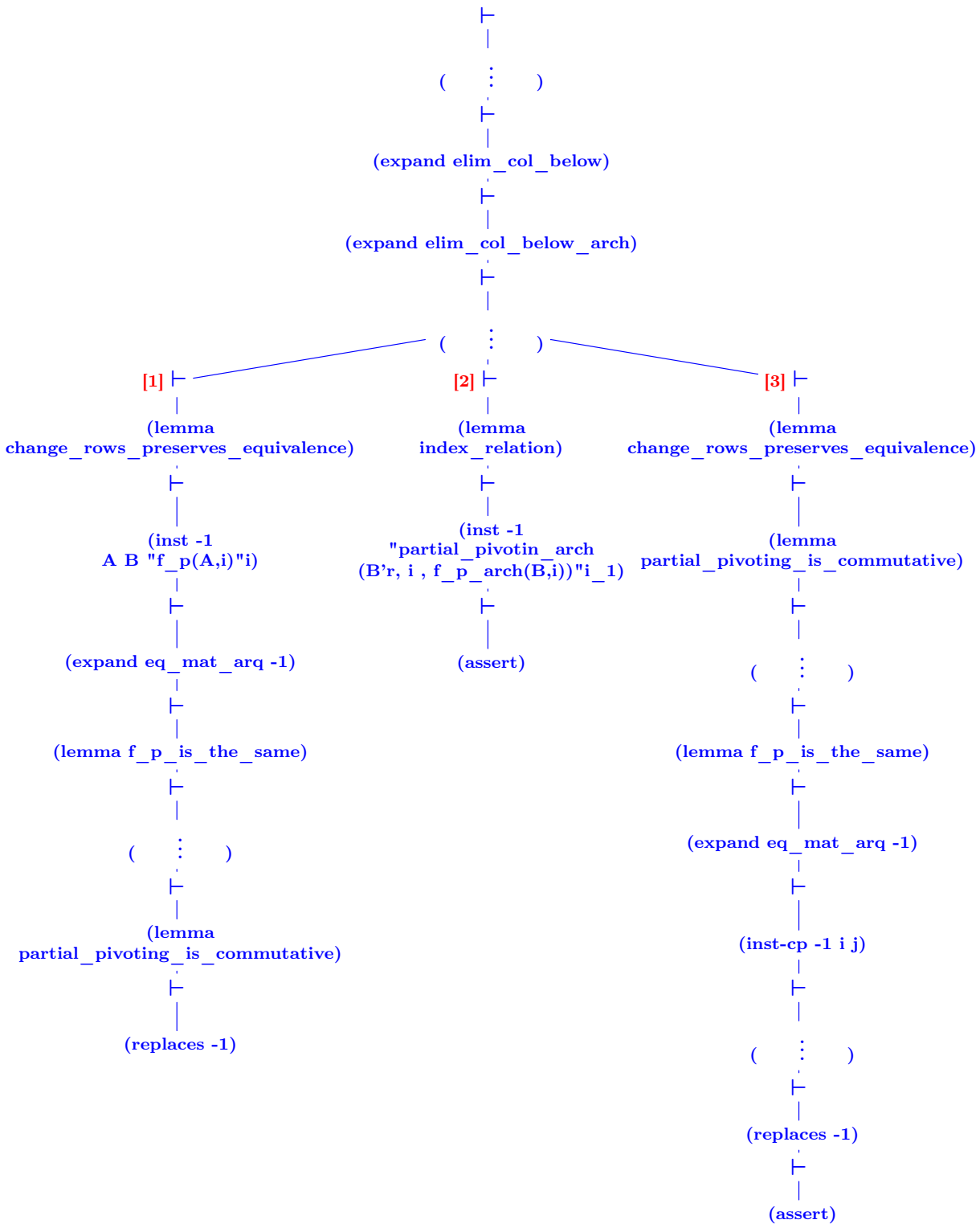

Figura 5.7: Árvore de prova para `arch_eliminates_col_below`

É possível então provar que os componentes da triangulação superior sincronizada previamente discutida são equivalentes.

5.5.4 Formalização da Equivalência da Triangulação Superior Sincronizada

A sincronização é feita por meio de uma tupla para agrupar os elementos de entrada que serão manipulados recursivamente por uma função de sincronização parcial. Deste modo, prova-se que quando estes elementos da tupla são equivalentes, o resultado da função também o é. Isso é dado pelo lema:

Lema 5.5.4 (Triangulação Superior produz resultados equivalentes.)**.**

```
sync_up_trg_eq_result: LEMMA
FORALL (AB: sync_math_arch_mmatrix):
       eq_mat_arq?(AB‘1,AB‘2) IMPLIES
       eq_mat_arq?(sync_up_trg(AB)‘1, sync_up_trg(AB)‘2)
```

Demonstração.

Como a sincronização da triangulação superior usa a função `sync_elim_col_below` para garantir que as operações de eliminação de apenas uma coluna ocorram de maneira uniforme como corpo de um `for` recursivo, a prova de sua correção é dada por meio de um invariantes, dado por:

```
Lambda (k : upto[mm_size], MM : sync_math_arch_mmatrix ) :
eq_mat_arq?(MM‘1, MM‘2)
```

Deste modo, a indução fornecida visa verificar se após cada iteração do `for`, os dados produzidos são os mesmos. Como o corpo do laço é dado por `elim_col_below`, basta utilizar o lema que atesta a equivalência dessa função em matrizes equivalentes.A árvore de prova resultante é mostrada na Figura 5.8.

□

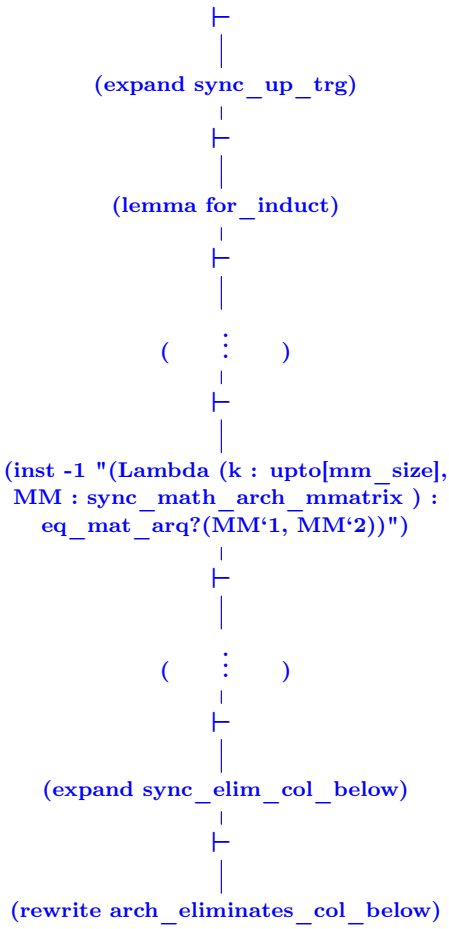

Figura 5.8: Árvore de prova para `sync_up_trg_eq_result`

5.5.5 Formalização da Equivalência da Normalização

Provando que para matrizes de entrada equivalentes `normalization` e `normalization_arch` computam matrizes equivalentes é muito mais simples e não requer tipos sincronizados. Isso porque, assim como no Lema 5.5.3, não há qualquer dependência de dados entre os passos realizados. O lema é dado por

Lema 5.5.5 (Normalização produz resultados equivalentes)**.**

```
arch_does_normalization : LEMMA
FORALL (A: up_trg_mmatrix, B: up_trg_mmatrix_arch) :
        eq_mat_arq?(A,B) IMPLIES
eq_mat_arq?(normalization(A),  normalization_arch(B))
```

Demonstração.

A equivalência da normalização é provada de maneira ainda mais simples, pois não há uso de quaisquer outras funções em seu corpo, dependendo apenas da expansão de suas definições e instanciações do predicado `eq_mat_arq?`, além do Lema 5.4.1 para verificação do real índice da matriz arquitetural para correta instanciação. □

5.5.6 Formalização da Equivalência de cada Passo da Triangulação Inferior

Similarmente à prova para os passos da triangulação superior, deve-se mostrar que, eliminando a mesma coluna abaixo do elemento da diagonal principal em matrizes equivalentes, o resultado é equivalente. Sendo assim, tem-se:

Lema 5.5.6 (Os passos da triangulação inferior coincidem)**.**

```
arch_eliminates_col_above : LEMMA
FORALL (A: norm_mmatrix, B:norm_mmatrix_arch, i:below[mm_size]):
     eq_mat_arq?(A, B) IMPLIES
     eq_mat_arq?(elim_col_above(i,A),elim_col_above_arch(i,B))
```

Demonstração.
Sua prova, embora análoga, é mais simples que aquela para os passos da triangulação superior, pois não há busca por pivô ou troca de linhas, sendo necessária apenas a expansão da definição da função e sua correta instanciação para obter o resultado desejado. □

5.5.7 Formalização da Equivalência da Triangulação Inferior Sincronizada

Assim como na formalização da triangulação superior sincronizada, prova-se que quando os elementos da tupla da sincronização são equivalentes, o resultado da função também o é. Isso é dado pelo lema:

Lema 5.5.7 (Triangulação inferior sincronizada produz resultados equivalentes)**.**

```
sync_lw_trg_eq_result: LEMMA
FORALL (AB: sync_mat_arch_norm):
       eq_mat_arq?(AB`1,AB`2) IMPLIES
       eq_mat_arq?(sync_lw_trg(AB)`1, sync_lw_trg(AB)`2)
```

Demonstração.
Também análoga à prova para a triangulação superior, o invariante fornecido apenas se difere quanto ao tipo de matriz recebido pela função, que passa a ser `sync_mat_arch_norm`, dando como invariante:

```
Lambda (k : upto[mm_size], MM : sync_mat_arch_norm ):
eq_mat_arq?(MM‘1, MM‘2)
```

Que produz um passo indutivo semelhante, porém referente às funções `elim_col_above` e `elim_col_above_arch`, que é provado reescrevendo o Lema 5.5.6.

□

5.5.8 Formalização do Teorema Principal de Equivalência para o Algoritmo e Arquitetura de GJ

De posse de todas as provas preliminares já fornecidas para os passos do algoritmo GJ, pode-se então verificar o teorema principal que o formaliza. Devem ser verificadas tanto propriedades sobre a equivalência entre os elementos da tupla quando aplicado a função de sincronização quanto sobre o resultado desta sincronização em si.

Essas últimas foram parcialmente provadas neste trabalho, porém foram deixadas como axiomas pois faltam ainda resultados referente a aplicação de mais um passo de um laço de repetição ao resultado deste laço. Esse resultado será melhor entendido durante a demonstração parcial do Lema 5.5.8, que exemplifica esse tipo de propriedade:

Lema 5.5.8 (A triangulação superior matemática é igual à primeira parte da triangulação superior sincronizada.)**.**

```
sync_up_trg_first: AXIOM
FORALL (AB: sync_math_arch_mmatrix): eq_mat?(sync_up_trg(AB)‘1, up_trg(AB‘1))
```

Demonstração.

Esta prova se dá também por meio de invariante, que é fornecido como:
`Lambda(i: below[mm_size+1], MM : sync_math_arch_mmatrix): eq_mat?(MM‘1, for[mmatrix](0, i - 1, AB‘1, elim_col_below))`

Após sua instanciação, são dados os casos de B.I. ([1]) e P.I. ([2]), mostrados na Figura 5.9. O primeiro é trivial e se completa ao expandir as definições dos laços. Já o passo indutivo necessita da instanciação correta da hipótese, de modo a indicar que exatamente o mesmo passo estão sendo feitos na definição sincronizada e na original, produzindo como resultado matrizes com a mesma coluna zerada. A partir do uso disso, é necessário um outro resultado que consiga relacionar a aplicação do corpo de um laço a um outro laço de mesmo corpo. Este resultado é axiomatizado por:

Axioma 5.5.1 (Relação entre quantidade de iteração de um `for`)**.** Determina a equivalência entre aplicar o corpo de um `for` ao resultado deste com $n-1$ iterações equivale a executá-lo com n iterações.

```
one_step_for : AXIOM
FORALL (m: nat, (n : int | n>=m-1),
        (l:int | l>n), init : T, f:ForBody(m,l)) :
  for(m,n+1,init,f)  = f(n+1,for(m,n,init,f))
```

Tal axiomatização se dá pelo fato de não ter se conseguido uma prova indutiva para validá-lo, mesmo sendo um resultado conhecido. A dificuldade desta prova se insere na manipulação de índices n e $n+1$ nas chamadas recursivas, o que não permite garantir a terminação da prova.

Assim, utilizando corretamente estes resultados com suas devidas instanciações. completa-se a prova do ramo [2] e, consequentemente, deste lema. A árvore deste esboço pode ser visto na Figura 5.9.

□

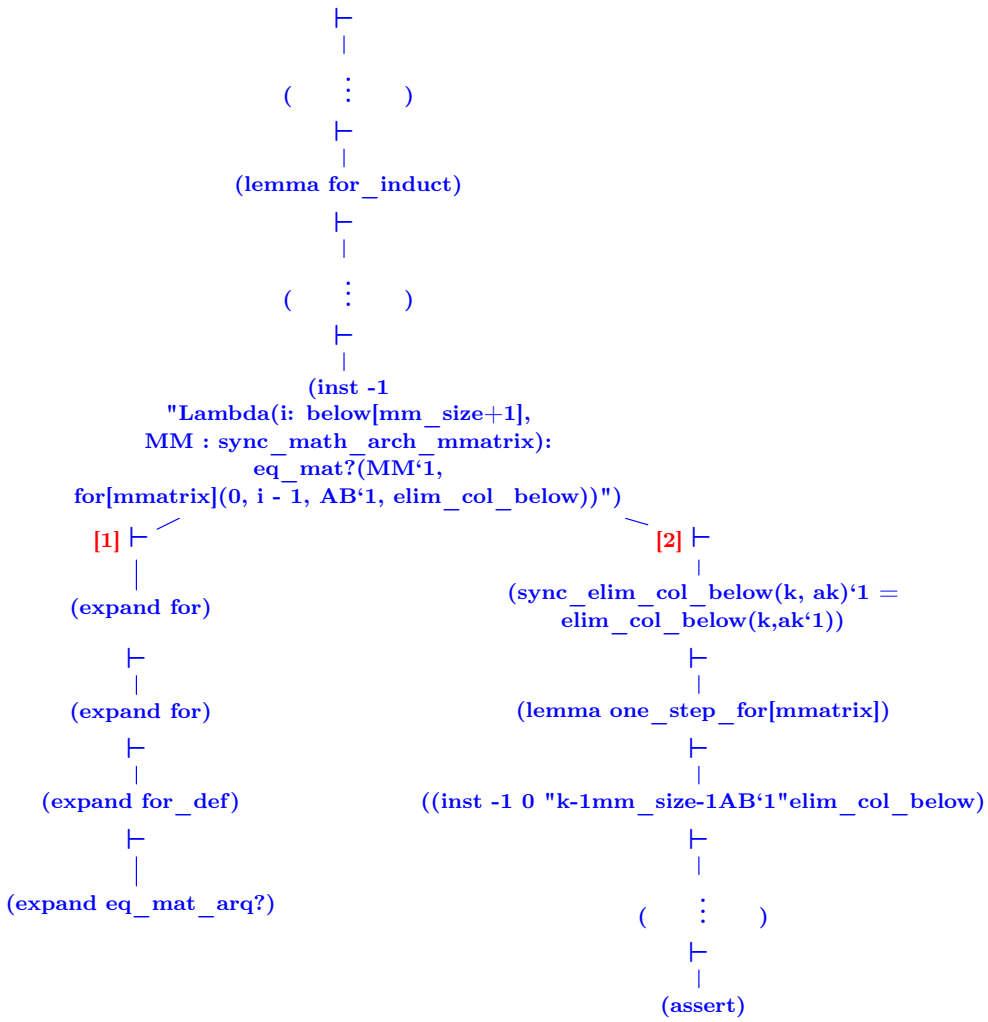

Figura 5.9: Árvore de esboço da prova para `sync_up_trg_first`

Outros resultados similares, relativos a equivalência da função sincronizada com aquelas originais que apresentam a mesma problemática, também foram axiomatizados, sendo eles:

Axioma 5.5.2 (A triangulação superior arquitetural é igual à segunda parte da triangulação superior sincronizada.)**.**

```
sync_up_trg_second: AXIOM
FORALL (AB: sync_math_arch_mmatrix):
eq_arch?(sync_up_trg(AB)‘2, up_trg_arch(AB‘2))
```

Axioma 5.5.3 (A triangulação inferior matemática é igual à primeira parte da triangulação inferior sincronizada.)**.**

```
sync_lw_trg_first: AXIOM
FORALL (AB: sync_mat_arch_norm):
eq_mat?(sync_lw_trg(AB)‘1, low_trg(AB‘1))
```

Axioma 5.5.4 (A triangulação inferior arquitetural é igual à segunda parte da triangulação inferior sincronizada.)**.**

```
sync_lw_trg_second: AXIOM
FORALL (AB: sync_mat_arch_norm):
eq_arch?(sync_lw_trg(AB)‘2, low_trg_arch(AB‘2))
```

Finalmente, pode-se formalizar como um todo a equivalência funcional entre a definição matemática e a arquitetural. No geral, deve-se analisar se a função para GJ matemática é equivalente à segunda parte da matriz estendida do primeiro elemento da tupla quando aplicada a triangulação inferior à normalização da triangulação superior sincronizada, se a função para GJ arquitetural é equivalente à segunda parte da matriz estendida do segundo elemento da tupla dessa mesma aplicação e, finalmente, que os dois elementos do resultado da tupla dessa aplicação são equivalentes entre si. Isto é:

Teorema 5.5.1 (Equivalência Funcional de GJ)**.**

```
sync_arch_does_gauss_jordan : THEOREM
FORALL (A: matrix, (B: mat_reg | id_reg?(B‘r))):
       LET
        AB: sync_math_arch_mmatrix =
        (matrix_to_mmatrix(A), matrix_to_mmatrix_arch(B))
       IN
       LET
        ABproc =
        sync_lw_trg(normalization(sync_up_trg(AB)‘1),
        normalization_arch(sync_up_trg(AB)‘2))
       IN
          eq_single_mat_arq?(A,B) IMPLIES
            (eq_mat?(second_mmatrix(ABproc‘1),
            math_gauss_jordan(first_mmatrix(AB‘1)))
```

```
AND
   eq_arch?(second_mmatrix_arch(ABproc‘2),
   gauss_jordan_arch(first_mmatrix_arch(AB‘2)))
AND
   eq_mat_arq?(ABproc‘1, ABproc‘2))
```

Demonstração.

Note que, como as entradas dos algoritmos `math_gauss_jordan` e `gauss_jordan_arch` são matrizes não estendidas, é necessário estendê-las antes de iniciar os processos de triangulações sincronizadas. A partir disso, basta a utilização das premissas que indicam que matrizes equivalentes são dadas como entrada juntamente com a instanciação correta dos lemas provados anteriormente (relativos ao resultado das triangulações sincronizadas, normalização e sobre sincronização e si), para completar a prova.

□

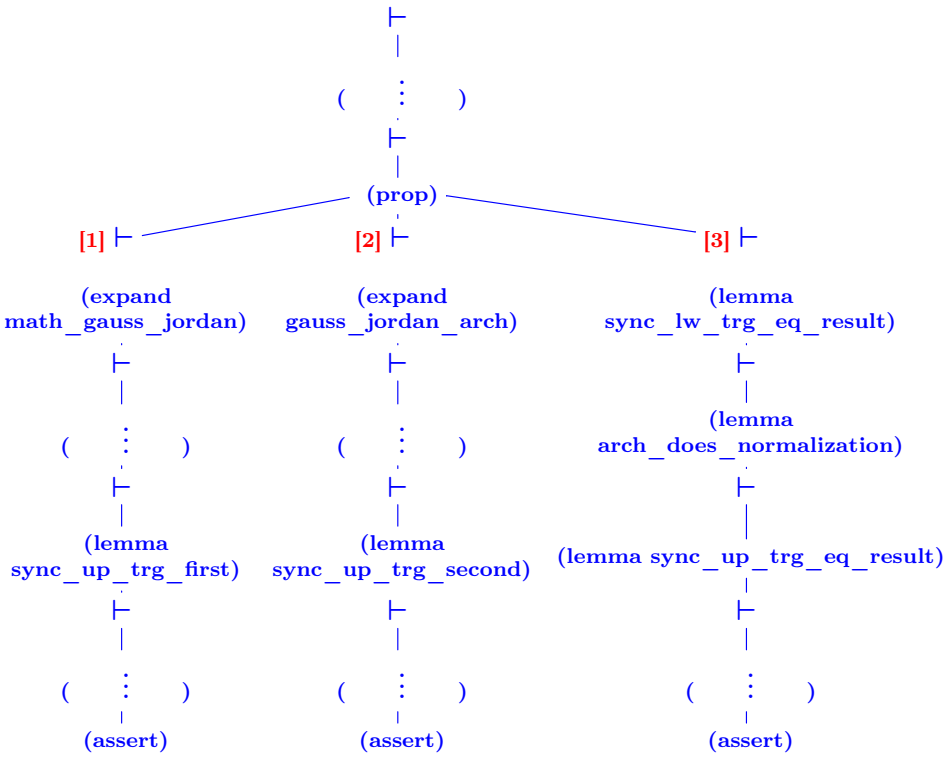

Figura 5.10: Árvore de prova para `sync_arch_does_gauss_jordan`

Deste modo, a prova da equivalência funcional entre a definição matemática de GJ e a arquitetura que a implementa é concluída como corolário através da aplicação direta do lema anterior.

Capítulo 6

Discussões e Trabalho Futuro

É apresentada uma metodologia para verificação formal de operadores algébricos implementados em hardware por meio de provas da equivalência funcional entre suas especificações, visando garantir a funcionalidade destes circuitos. Nesta abordagem é proposta a aplicação da verificação formal como passo paralelo ao desenvolvimento do circuito, minimizando possíveis atrasos no projeto, mas também sendo possível sua aplicação após completa produção do circuito.

Para isso, a especificação da arquitetura deve modelar seu comportamento, sendo capaz de traduzir conservativamente para a linguagem recursiva dos assistentes de prova seus elementos imperativos. Utiliza-se então a lógica de Floyd-Hoare para prover invariantes que verificam esses elementos.

É proposta também a formalização opcional do operador implementado, permitindo que se assuma sua correção para fins de verificação da arquitetura produzida. Essa metodologia então pode ser utilizada para validar circuitos, fornecendo certificados de seu funcionamento e garantindo sua confiabilidade.

Um caso de estudo que analisa uma implementação em FPGA do algoritmo GJ para inversão de matrizes é apresentado. Tal escolha é justificada pelo amplo uso de inversões de matrizes, especialmente em aplicações de engenharia, e pela grande demanda de tempo necessária para realização dessas operações. Assim, sua implementação em hardware é desejável, bem como um certificado de garantia de seu funcionamento.

A formalização desenvolvida garante a equivalência funcional entre o circuito analisado e seu operador. Tal equivalência se dá mesmo trabalhando com fluxos, armazenamentos e manipulações de dados de maneira diferente, pois é fornecida uma relação entre os dados e sua manipulação.

Essa formalização envolve diferentes elementos e mecanismos de prova, como o CS, que é a base do assistente PVS utilizado, lógica de Floyd-Hoare e esquemas indutivos. Também foram necessários dois tipos de traduções, uma entre o comportamento do hardware e a linguagem de especificação, e outra entre definições imperativas e recursivas. Ambas foram manualmente, estando assim sujeitas a falhas e abrem caminhos para trabalho futuro no sentido de automatizá-las, a fim de evitar tais erros. Os resultados apresentados foram reportados em Almeida et al. (2014)

6.1 Dificuldades do Trabalho

Uma das principais dificuldades apresentadas no decorrer do desenvolvimento deste trabalho é a junção de duas áreas tão amplas e complexas: a prototipagem de hardware e a verificação de sistemas.

Além disso, há certa confusão ao tratar de verificação formal no âmbito de hardware. Isso porque o termo *verificação* já é amplamente utilizado no sentido de *teste*, como nas ferramentas de verificação de hardware como SystemC e SystemVerilog, que embora sejam muito utilizadas para validar implementações de hardware via testes, não fornecem provas matemáticas de sua correção.

PVS por sua vez, é uma linguagem de especificação e prova que permite abstrair o comportamento e a estrutura de sistemas de modo a verificá-los. Ainda assim, problemas se apresentam ao tratar de definições imperativas nessa linguagem, pois às vezes são necessárias provas quanto à aplicação do corpo de um laço de repetição em um resultado deste mesmo laço. Tais provas não são triviais, pois as definições recursivas oferecidas como suporte àquelas imperativas se apresentam de modo a dificultar esse tipo de prova, já que os índices e casos base que controlam a recursão não convergem, ou seja, podem não chegar a um valor comum.

Ainda em relação às definições imperativas/recursivas, há a necessidade de escolher cuidadosamente os invariantes que modelam as propriedades a serem verificadas. Estes devem ser capazes de abstrair todas as características das variáveis modificadas durante a execução do laço, e como estas devem se apresentar não só após seu término, mas também antes e durante cada iteração. Em PVS o invariante deve ser um predicado cujos parâmetros são um elemento inicial de um dado tipo e um natural referente à iteração corrente da verificação desse elemento. Deste modo, deve ser analisado internamente a este predicado as pré e pós-condições necessárias.

Também é necessário realizar ajustes ao tratar de equivalência entre recursões, pois o resultado da execução destas deve ser analisado de acordo com a iteração em questão. Para isso, cria-se funções auxiliares, modelando internamente a elas a sincronia necessária à verificação de cada passo da recursão. Provas adicionais também são geradas, garantindo que esta sincronia produz os mesmos resultados que quando cada função é aplicada individualmente.

6.2 Trabalho Futuro

Espera-se inicialmente, como trabalho futuro, a formalização de todas as propriedades da álgebra linear envolvidas no caso de estudo apresentado, como aquelas relacionadas às matrizes não singulares. Isso permitirá fornecer um certificado completo ao prover a correção do operador. Já existem formalizações de diversas propriedades necessárias a este trabalho em PVS nas bibliotecas da NASA, como em Herencia-Zapana et al. (2012), porém as matrizes são definidas como vetores de vetores, diferente da abordagem de funções anônimas utilizadas neste trabalho. Pode-se então estabelecer provas de isomorfismo entre as duas estruturas de dados utilizadas e reutilizar tais provas.

Formalizações adicionais também são necessárias ao tratar de traduções imperativas em recursivas. Isso porque a teoria `for_iterate` utilizada não possui estruturas para modelar laços `while-do` e `repeat-until`, também bastante utilizados HDLs. Faltam ainda nessa teoria lemas para lidar com a aplicação de um passo extra de execução de um laço, como dado no Axioma 5.5.1. Outra carência importante é relacionada ao tipo de invariante fornecido para as provas indutivas, pois estes se relacionam apenas com uma função, sendo possível provar propriedades inerentes a ela, porém dificultam a verificação de propriedades que relacionam duas funções, como a equivalência funcional utilizada neste trabalho.

Outra proposta mais ambiciosa para dar continuidade a este trabalho, é a geração de uma ferramenta para traduzir conservativamente HDLs em PVS de maneira automática. Isso eliminará a possibilidade de que erros de especificação oriundos da má interpretação do funcionamento de circuitos sejam introduzidos. Também deve ser possível gerar código HDL sintetizável a partir de sistemas e operadores já formalizados em PVS. As traduções entre essas duas linguagens precisam ter a garantia de conservatividade, que deve ser fornecida por meio de provas de equivalência entre os elementos de ambas.

Referências Bibliográficas

Almeida, A. A., Arias-Garcia, J., Ayala-Rincon, M., and Llanos, C. (2014). In *A aparecer em Proceedings SBCCI 2014*.

Arias-Garcia, J. (2010). Implementação em FPGA de uma Biblioteca Parametrizável para Inversão de Matrizes Baseada no Algoritmo de Gauss-Jordan, Usando Representação em Ponto Flutuanta. Master's thesis, Universidade de Brasília.

Arias-Garcia, J., Llanos, C., Ayala-Rincon, M., and Jacobi, R. (2012a). A fast and low cost architecture developed in FPGAs for solving systems of linear equations. In *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on*, pages 1–4.

Arias-Garcia, J., Llanos, C., Ayala-Rincon, M., and Jacobi, R. (2012b). FPGA implementation of large-scale matrix inversion using single, double and custom floating-point precision. In *Programmable Logic (SPL), 2012 VIII Southern Conference on*, pages 1–6.

Ayala-Rincón, M. and Sant'Ana, T. M. (2006). Saeptum: verification of elan hardware specifications using the proof assistant pvs. In *Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 125–130. ACM.

Boldrini, J. (1986). *Algebra linear*. HARBRA.

Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6. ACM.

Deng, H. (2011). Formal Verification of FPGA based systems. Master's thesis, McMaster University.

Drechsler, R., editor (2004). *Advanced Formal Verification*. Falk Symposium Series. Springer.

Emmer, M., Khasidashvili, Z., Korovin, K., and Voronkov, A. (2010). Encoding industrial hardware verification problems into effectively propositional logic. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 137–144.

Floyd, R. (1967). Assigning meaning to programs. In Schwartz, J. T., editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society.

Gentzen, G. (1964). Investigations into Logical Deduction. *American Philosophical Quarterly*, 1.

Gokhale, M. and Graham, P. (2005). *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer.

Gordon, M., Iyoda, J., Owens, S., and Slind, K. (2006). Automatic formal synthesis of hardware from higher order logic. *Electron. Notes Theor. Comput. Sci.*, 145:27–43.

Haghbayan, M., Alizadeh, B., Behnam, P., and Safari, S. (2014). Formal verification and debugging of array dividers with auto-correction mechanism. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pages 80–85.

Harrison, J. (2006). Floating-point verification using theorem proving. In *Proceedings of the 6th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems*, SFM'06, pages 211–242. Springer-Verlag.

Herencia-Zapana, H., Jobredeaux, R., Owre, S., Garoche, P.-L., Feron, E., Perez, G., and Ascariz, P. (2012). PVS linear algebra libraries for verification of control software algorithms in c/acsl. In *NASA Formal Methods'12*, pages 147–161.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Hu, Y. (2012). Exploring Formal Verification Methodology for FPGA-Based Digital Systems. Technical Report SAND2012-7926, Sandia National Laboratories.

Huffmire, T., Nguyen, C. I. T. D., Levin, T., Kastner, R., and Sherwood, T. (2010). *Handbook of FPGA Design Security*. Springer.

Kapur, D. and Subramaniam, M. (2000). Using an induction prover for verifying arithmetic circuits. *International Journal on Software Tools for Technology Transfer*, 3(1):32–65.

Khasidashvili, Z., Kinanah, M., and Voronkov, A. (2009). Verifying equivalence of memories using a first order logic theorem prover. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 128–135. IEEE.

Kropf, T. (1999). *Introduction to Formal Hardware Verification*. Springer.

Li, L. and Thornton, M. (2010). *Digital System Verification: A Combined Formal Methods and Simulation Framework*. Synthesis Lectures on Digital Circuits and Systems Series. Morgan & Claypool.

Morra, C. (2010). *A Flexible Framework for Hardware/Software Design Space Exploration using Rewriting Logic*. PhD thesis, Karlsruher Institut für Technologie.

Morra, C., Bispo, J. a., Cardoso, J. a. M. P., and Becker, J. (2008). Combining rewriting-logic, architecture generation, and simulation to exploit coarse-grained reconfigurable architectures. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 320–321. IEEE Computer Society.

NASA (2013). NASA LaRC PVS libraries. Avaliable at `shemesh.larc.nasa.gov /fm/ftp/larc/PVS-library/`.

Owre, S., Rushby, J. M., Shankar, N., and Srivas, M. K. (1994). A tutorial on using PVS for hardware verification. In Kumar, R. and Kropf, T., editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *Lecture Notes in Computer Science*, pages 258–279, Bad Herrenalb, Germany. Springer-Verlag.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. (1999). *PVS Language Reference*. Computer Science Laboratory, SRI International.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. (2001). *PVS System Guide*. Computer Science Laboratory, SRI International.

Perry, D. and Foster, H. (2005). *Applied Formal Verification: For Digital Circuit Design*. McGraw-Hill Electronic engineering. McGraw-Hill Companies,Incorporated.

Pitchumani, V. and Stabler, E. (1982). A formal method for computer design verification. In *Design Automation, 1982. 19th Conference on*, pages 809–814.

Rajan, S., Shankar, N., and Srivas, M. (1997). Industrial strength formal verification techniques for hardware designs. In *VLSI Design, 1997. Proceedings., Tenth International Conference on*, pages 208–212.

Scott Hauck, A. and DeHon, A. (2008). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Systems on Silicon Series. Morgan Kaufmann.

Shankar, N., Owre, S., Rushby, J. M., and Stringer-Calvert, D. W. J. (1999). *PVS Prover Guide*. Computer Science Laboratory, SRI International.

Siegl, S., Hielscher, K.-S., German, R., and Berger, C. (2011). Formal specification and systematic model-driven testing of embedded automotive systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6.

Singh, S. and Lillieroth, C. (1999). Formal verification of reconfigurable cores. In *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on*, pages 25–32.

Srivas, M. (1991). Bridging the formal methods gap: a computer-aided verification tool for hardware designs. In *Compcon Spring '91. Digest of Papers*, pages 456–461.

Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356.

Sutton, A. (2010). No Room for Error: Creating Highly Reliable, High-Availability FPGA Designs. White Paper.

Troelstra, A. S. and Schwichtenberg, H. (2000). *Basic proof theory (2nd ed.)*. Cambridge University Press, New York, NY, USA.

Vahid, F. (2007). *Sistemas Digitais*. Bookman.

Zhang, N. and Duan, Z. (2011). Verification of hardware designs: A case study. In *Computers, Networks, Systems and Industrial Engineering (CNSI), 2011 First ACIS/JNU International Conference on*, pages 198–203.