Bruno Rafael dos Santos

Formalização do Algoritmo Ressol em Coq

Brasil

6 de novembro de 2024

Bruno Rafael dos Santos

Formalização do Algoritmo Ressol em Coq

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Universidade do Estado de Santa Catarina – UDESC

Bacharelado em Ciência da Computação

Orientador: Karina Girardi Roggia

Coorientador: Paulo Henrique Torrens

Brasil

6 de novembro de 2024

Bruno Rafael dos Santos
Formalização do Algoritmo Ressol em Coq/ Bruno Rafael dos Santos. – Brasil, 6 de novembro de 2024-
99p. : il. (algumas color.) ; 30 cm.

Orientador: Karina Girardi Roggia

Trabalho de Conclusão de Curso – Universidade do Estado de Santa Catarina – UDESC
Bacharelado em Ciência da Computação , 6 de novembro de 2024.

1. Algoritmo *RESSOL*. 2. Algoritmo Tonelli-Shanks. 2. Lei de Reciprocidade Quadrática. I. Karina Girardi Roggia. II. Universidade do Estado de Santa Catarina. III. Faculdade de Ciência da Computação. IV. Formalização do Algoritmo *RESSOL* utilizando Coq.

Errata

Elemento opcional.

Bruno Rafael dos Santos

Formalização do Algoritmo Ressol em Coq

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Brasil, 24 de novembro de 2012:

Karina Girardi Roggia
Orientadora (Doutora)

Cristiano Damiani Vasconcelos
Doutor

Rafael Castro Gonçalves
Mestre

Paulo Henrique Torrens
Co-orientador

Brasil
6 de novembro de 2024

*Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.*

Agradecimentos

Incialmente agradeço a Laurent Théry pela provas disponibilizadas em seu GitHub (principalmente as mencionadas no Capítulo 4), além de todas as suas contribuições feitas tanto para a biblioteca Mathematical Components quanto para outros temas de grande relevância acadêmica. Agradeço a Laurent também pela atenção em responder aos e-mails de um garoto (eu) aleatório de outro canto deste mundo.

"Em cima desse rato
tinha uma pulga...
Será possível?
Uma pulga acordada,
em cima de um rato dormitando,
em cima de um gato ressonando,
em cima de um cachorro cochilando,
em cima de um menino sonhando,
em cima de uma avó roncando,
numa cama aconchegante,
numa casa sonolenta,
onde todos viviam dormindo."
(WOOD, 1999).

Resumo

O ramo da matemática conhecido como Teoria dos Números tem grande influência nos campos de estudo da Ciência da Computação, apresentando diversos algoritmos e teoremas relacionados principalmente à criptografia. Não isoladamente, como em todos os ramos da matemática, as formalizações e provas de conceitos desta área são essenciais para o seu desenvolvimento. Para isso, o presente trabalho busca contribuir com esses itens por meio de métodos formais utilizando o assistente de provas *Coq* e estabelecendo, como objeto de implementação, a função conhecida pelo nome de *símbolo de Legendre* e parte de suas propriedades. Além disso, se pretende utilizar nesta implementação, a biblioteca Mathematical Components, a fim de que o resultado deste trabalho possa servir como contribuição para a mesma.

Palavras-chave: criptografia, Teoria dos Números, *símbolo de Legendre*, Algoritmo de Tonelli-Shanks, Algoritmo RESSOL, Coq, *Lei de Reciprocidade Quadrática.*

Abstract

The math field known as Number Theory has a great influence in the study fields from Computer Science, presenting a series of algorithms and theorems mainly related to cryptography. Not alone, as all the math fields, formalizations and proofs for concepts in this area are essencial for it's development. For that, the following work seeks to contribute for these items by means of formal methods, using the proof assistant *Coq* and establishing, as implementation objects, the function known by the name *Legendre symbol* and part of it's properties. Furthermore, it's pretended to be used in these implementations, the library Mathematical Components, in order to make this work's result to serve as a contribution for the same.

Keywords: cryptography, Number Theory, *Legendre Symbol*, Tonelli-Shanks algorithm, RESSOL algorithm, Coq, *Quadratic Reciprocity Law.*

Lista de ilustrações

Figura 1 – Grafo do módulo *ssreflect* 31

Lista de abreviaturas e siglas

RESSOL	Residue Solver
RSA	Rivest-Shamir-Adleman

Índice de algoritmos

1 Euclides 46

2 Fatorar-Potência-de-Dois 85

3 Obter-Resíduo-Não-Quadrático 85

4 Repetir-Quadrados 86

5 Ressol 86

Sumário

1 INTRODUÇÃO 25
1.1 Objetivo Geral 28
1.2 Objetivos Específicos 28
1.3 Estrutura do Trabalho 28

2 BIBLIOTECA MATHEMATICAL COMPONENTS 31
2.1 Módulos 31
2.2 Igualdades 32
2.3 Structures e Records 33
2.3.1 Comando Canonical 34
2.3.2 Comando Coercion 37
2.3.3 Exemplo de Implementação de grupos 39
2.3.4 Mantendo Informações de um Record ou Structure 41

3 BASE TEÓRICA 45
3.1 Máximo Divisor Comum 45
3.2 Algoritmo de Euclides 46
3.3 Teorema Bachet-Bézout 47
3.4 Propriedades de Congruência 48
3.5 Anel de Inteiros Módulo n 49
3.6 Função φ de Euler 54
3.7 Congruência de Grau 2 e Símbolo de Legendre 56

4 IMPLEMENTAÇÃO 65
4.1 Implementações Externas de Maior Relevância 65
4.2 Formalização do Símbolo de Legendre 77

5 CONCLUSÃO 79

REFERÊNCIAS 81

APÊNDICES 83

APÊNDICE A – ALGORITMO DE TONELLI-SHANKS (OU RESSOL) 85
A.1 Descrição do Algoritmo 85

A.2 Prova Manual 87

APÊNDICE B – RECIPROCIDADE QUADRÁTICA 91

1 Introdução

Durante os cursos de Ciência da Computação, são vistas estruturas matemáticas muito diferentes daquelas as quais alunos de ensino médio estão habituados. No geral, grande parte destas estruturas são abstratas por não parecerem uma representação de um objeto real ou por, apesar de parecer, a razão de sua formulação não ser bem motivada de início. A exemplo de tais estruturas temos vetores, matrizes, filas e grafos, utilizados na modelagem de diversos problemas. Apesar destas ferramentas serem extremamente úteis, há um tipo de objeto matemático sempre presente na maioria dos problemas e que muitas vezes são considerados limitados e apenas objetos auxiliares demasiadamente utilizados: estes são os números inteiros. O conjunto dos números inteiros, apesar de ser formado por objetos (números) vistos como simples, possui diversas endorrelações que levam a muitas conclusões e invenções de grande importância, principalmente para o campo da criptografia. Dentre estas relações, duas delas são pilares fundamentais para tais conclusões e invenções mencionadas: a relação de divisibilidade e de congruência. A primeira é definida da seguinte forma (BROCHERO et al., 2013):

Definição 1 *$\forall d, a \in \mathbb{Z}$, d* ***divide*** *a (ou em outras palavras: a é um múltiplo de d) se e somente se a seguinte proposição é verdadeira:*

$$\exists q \in \mathbb{Z}, a = d \cdot q$$

assim, se tal proposição é verdadeira e portanto d divide a, tem-se a seguinte notação que representa tal afirmação:

$$d \mid a$$

caso contrário, a negação de tal afirmação (d não divide a) é representada por:

$$d \nmid a$$

Se introduz também aqui o conceito de resto da divisão, para o qual deve-se lembrar da divisão euclidiana, também conhecida como divisão com resto. Todo algoritmo equivalente a tal divisão tem como resultados um quociente q e um resto r, de forma que a seguinte proposição é verdadeira:

$$\forall a, b \in \mathbb{Z}, \exists q, r \in \mathbb{Z}, (a = b \cdot q + r \wedge 0 \leq r < |b|)$$

Define-se então o que se chama de congruência (BROCHERO et al., 2013):

Definição 2 *Para todo $a, b, n \in \mathbb{Z}$, a é congruente a b módulo n se e somente se, pela divisão euclidiana a/n e b/n (onde $0 \leq r_a < |n|$ e $0 \leq r_b < |n|$) tem-se*

$$a = n \cdot q_a + r_a$$

e

$$b = n \cdot q_b + r_b$$

com $r_a = r_b$, o que também equivale a dizer que:

$$n \mid a - b$$

tal relação entre os inteiros a, b e n é representada por:

$$a \equiv b \pmod{n}$$

Tais definições levam a uma série de teoremas como os relacionados à função φ de Euler, muito utilizados em criptografia, e além disso, a criação de estruturas mais complexas a partir do conjunto dos números inteiros, como os anéis e grupos de unidades (BROCHERO et al., 2013).

Um conteúdo que carece de formalizações e provas, e possui relação com o conteúdo a ser apresentado neste trabalho, é o algoritmo de Tonelli-Shanks, também conhecido como Algoritmo RESSOL (HUYNH, 2021), acrônimo este que significa *Residue Solver* de acordo com (NIVEN; ZUCKERMAN, 1991). Esse método resolve congruências quadráticas, isto é, equações da seguinte forma:

$$r^2 \equiv n \pmod{p} \tag{1.1}$$

em que $r, n, p \in \mathbb{Z}$, onde p é um número primo, n é um valor conhecido e r é o valor a ser computado. Este método foi proposto em (SHANKS, 1972 apud MAHESWARI; DURAIRAJ, 2017), sendo uma versão aprimorada do que foi apresentado em (TONELLI, 1891). Como motivação ao leitor, uma das utilidades deste algoritmo está relacionada ao *Rabin Cryptosystem*, visto que esse sistema tem relações com resíduos quadráticos (HUYNH, 2021). No entanto esse não é único contexto em que aparecem equações com resíduos quadráticos, por isso, pode-se dizer que existe uma vasta quantidade de aplicações do Algoritmo RESSOL. Um exemplo adicional são os sistemas de criptografia que utilizam curvas elípticas, conforme mencionado em (SARKAR, 2024), (KUMAR, 2020) e (LI; DONG; CAO, 2014).

Essas considerações (sobre utilidades) valem portanto para qualquer algoritmo que resolve congruências quadráticas. Além disso, há uma função que possui relação com Algoritmo RESSOL (e possivelmente com outros métodos que possuem o mesmo objetivo), por indicar a existência ou não de uma solução para um instância da Equação 1.1. Esta é chamada de *símbolo de Legendre*, o principal tema deste trabalho.

Tais conceitos matemáticos explorados até o momento e quaisquer outros de áreas diversas sempre necessitam de alguma formalização. Especificamente quando se trata de algoritmos e teoremas, estes requerem provas para que sejam úteis (válidos). Nesse contexto, a matemática por muito tempo sempre se baseou na verificação de provas manualmente, isto é, por outros matemáticos, devido às limitações tecnológicas no passado. Tal dependência na verificação manual permitiu erros que fizeram com que muitas provas incorretas fossem tomadas como válidas, até que alguém notasse algum erro. A exemplo disso tem-se o teorema tratado em (NEEMAN, 2002), onde se apresenta um contra-exemplo para o mesmo.

Solucionando o risco das provas manuais, atualmente, muito se emprega o uso de auxiliadores de prova: programas que verificam se um prova está correta, inutilizando a necessidade de verificação manual e sendo também uma forma muito mais confiável de verificação (pois se trata de um processo mecânico). Se pretende neste trabalho utilizar o assistente de provas Coq, no entanto existem diversos outros, como Lean e Idris. Especificamente o assistente Coq é baseado em um formalismo chamado de Cálculo de Construções Indutivas (PAULIN-MOHRING, 2015), e a confiança em tal programa se deve a simplicidade de sua construção, no sentido de que tal programa pode ser verificado manualmente com facilidade.

Tendo em mente as informações mencionadas sobre formalizações e o assistente Coq, deve-se apresentar aqui a biblioteca disponível em tal assistente, cujo presente trabalho pretende contribuir: a biblioteca Mathematical Components, que está disponível em repositório no site Github[1]. Este projeto teve início com e contém a sustenção da prova do Teorema da Ordem Ímpar e do Teorema das 4 Cores (MAHBOUBI; TASSI, 2022), este último o qual é muito famoso na área de assistentes de prova, visto que foi proposto (porém não provado) em 1852 por Francis Guthrie, de acordo com (GONTHIER, 2023). A então conjectura só veio a ser provada em 1976 por (APPEL; HAKEN, 1976), no entanto a prova apresentada foi alvo de críticas, das quais parte se devem ao fato de que a prova envolvia uma análise manual de 10000 casos em que pequenos erros foram descobertos (GONTHIER, 2023). Devido ao ceticismo quanto a prova apresentada em 1976, foi então desenvolvida e publicada por (GONTHIER, 2023) uma nova versão da prova, feita em Coq, no ano de 2005.

A biblioteca Mathematical Components, apesar de vasta, obviamente não apresenta todos os teoremas conhecidos. Sendo assim, a decisão de se tratar sobre o *símbolo de Legendre* neste trabalho, se sustenta pelas seguintes justificativas:

1. Esta função não está implementada e/ou formalizada na biblioteca Mathematical Components.

[1] https://github.com/math-comp/math-comp

2. A base de teoremas, lemas e definições necessários para a formalização deste conteúdo inclui diversos itens, dos quais, parte não se encontram na biblioteca Mathematical Components.

3. A formalização deste tema possibilita futuros trabalhos sobre outros conteúdos relevantes ainda não implementados na biblioteca como o já mencionado Algoritmo RESSOL (sobre o qual não há implementação em *Coq*) e a *Lei de Reciprocidade Quadrática*.

1.1 Objetivo Geral

O objetivo geral do presente trabalho é realizar uma implementação do *símbolo de Legendre* voltada para a biblioteca Mathematical Components.

1.2 Objetivos Específicos

Seguindo as necessidades para realização do objetivo geral, os objetivos específicos deste trabalho são:

1. Obter conhecimentos avançados sobre o assistente de provas *Coq*.

2. Realizar o estudo sobre as principais documentações da biblioteca Mathematical Components.

3. Desenvolver a capacidade de realizar provas em *Coq* utilizando as táticas da linguagem de provas *SSReflect*.

4. Estudar conteúdos de Teoria dos Números relacionados ao *símbolo de Legendre*.

5. Implementar uma função que compute o valor do *símbolo de Legendre* e provar que a corretude da mesma utilizando-se da biblioteca Mathematical Components.

6. Provar teoremas úteis para manipulação de expressões envolvendo o *símbolo de Legendre* utilizando-se da biblioteca Mathematical Components.

1.3 Estrutura do Trabalho

O presente trabalho está dividido da seguinte maneira: o Capítulo 2 trata sobre conhecimentos básicos em relação a biblioteca Mathematical Components e ferramentas em *Coq* utilizadas nesta; o Capítulo 3 traz conteúdos de Teoria dos Números com objetivo de introduzir o conceito de *símbolo de Legendre*; no Capítulo 4 são apresentadas a formalização e as provas relacionadas ao *símbolo de Legendre*; o Capítulo 5 traz por fim as conclusões sobre este trabalho e finalmente os apêndices A e B trazem conteúdos que podem vir a ser implementados na biblioteca Mathematical Components em trabalhos futuros: o Algoritmo Ressol e a *Lei de Reciprocidade Quadrática* respectivamente.

2 Biblioteca Mathematical Components

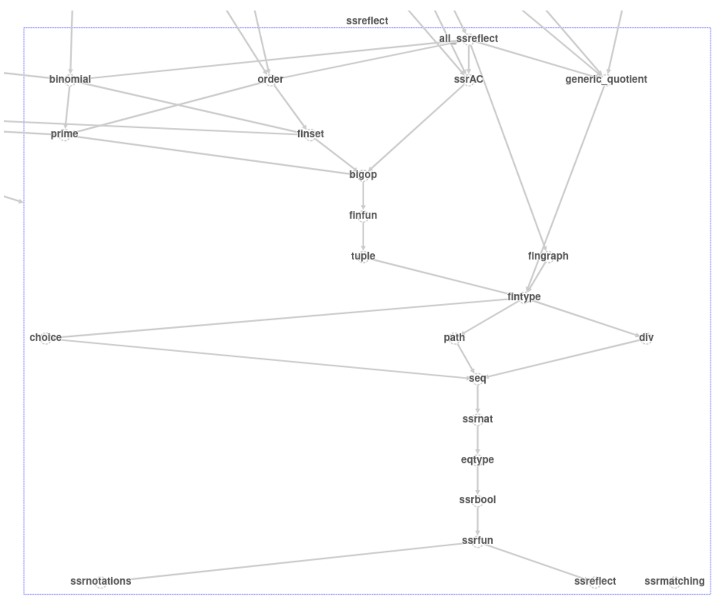
Nos capítulos seguintes será usada uma série de itens disponíveis na biblioteca Mathematical Components e outros implementados com uso da biblioteca. Todavia é necessário, por parte do leitor, um conhecimento básico sobre essa biblioteca, e para isso, neste capítulo serão explicados elementos que foram considerados mais essenciais de acordo com o tema definido. Todo o conteúdo a seguir se baseia em (MAHBOUBI; TASSI, 2022).

Ademais, é importante ressaltar que, na apresentação dos conteúdos deste capítulo, se assume que o leitor possui um conhecimento básico sobre *Coq*, e em caso contrário recomenda-se que o leitor acesse o material disponível em: <https://softwarefoundations.cis.upenn.edu/lf-current/index.html>

2.1 Módulos

A biblioteca Mathematical Components é divida em módulos, nos quais alguns são simplesmente a união de outros menores relacionados entre si. No site oficial da biblioteca[1] está disponível, além do livro utilizado como referência neste trabalho (MAHBOUBI; TASSI, 2022), um grafo de tais módulos. A Figura 1 apresenta uma parte desse grafo:

Figura 1 – Grafo do módulo *ssreflect*



Disponível em: <<https://math-comp.github.io/htmldoc_2_2_0/libgraph.html>>. Acesso em: 18 de maio de 2024.

[1] <https://math-comp.github.io/>

Os módulos principais para o desenvolvimento da prova sobre o algoritmo Tonelli-Shanks (com base no conteúdo de Teoria dos Números que sustenta a lógica do mesmo) são: *all_ssreflect* (que contém diversos outros módulos), *ring_quotient*, *zmodp* e *intdiv*.

2.2 Igualdades

Na maior parte dos teoremas das bibliotecas nativas de *Coq*, usa-se uma definição indutiva de igualdade, que é por sua vez equivalente a igualdade de Leibniz (isto pode ser provado). Por sua vez, essa definição de igualdade é dada pela seguinte proposição indutiva (de acordo com a documentação[2] (TEAM, 2024)):

```
Inductive eq {A : Type} (x : A) : A → Prop := eq_refl : eq x x.
```

De maneira distinta, a biblioteca Mathematical Components, em suas definições e teoremas, utiliza com frequência predicados booleanos (MAHBOUBI; TASSI, 2022), que são basicamente funções cujo tipo de retorno é `bool`, para então representar proposições da forma `x = true`, onde `x` é uma expressão cujo retorno é do tipo `bool`. Tais proposições são construídas pela função `is_true`.No entanto, através do comando `Coercion` (que será explicado mais detalhadamente adiante neste documento) e por questões de legibilidade, tal função é omitida e o sistema de tipos de *Coq* é capaz de inferir quando uma expressão deve ter tipo `bool` ou tipo `Prop` (e então há uma aplicação de `is_true` omitida).

Semelhantes aos teoremas existentes para proposição comuns, estão disponíveis diversos teoremas para proposições geradas com o uso de `is_true`, como por exemplo o lema `contraLR`. Esse é uma versão da contraposição utilizando predicados booleanos (junto à função `is_true`) e sua definição é dada por:

```
Lemma contraLR (c b : bool) : (~~ c → ~~ b) → b → c.
```

onde ~~ é a operação de negação definida na biblioteca Mathematical Components.

Outra informação relevante ao se tratar do conteúdo da biblioteca é o tipo `eqType`: para que seja construído qualquer habitante desse tipo é necessário um elemento `T` de tipo `Type`, uma função `eq_op` de tipo `T → T → bool` e um elemento `eqP` cujo tipo é um teorema relacionando a igualdade de Leibniz com `T` e `eq_op`. Este último possui, na biblioteca, uma notação de nome `eq_axiom`, e sua descrição é:

```
Definition eq_axiom: forall (T : Type) (e : rel T), forall x x0 : T,
   reflect (x = x0) (e x x0)
```

onde `rel T` é equivalente ao tipo `T → T → bool`.

2 <https://coq.inria.fr/doc/V8.18.0/refman/proofs/writing-proofs/equality.html>

Portanto, para que um tipo `A` pertença ao primeiro campo mencionado, é necessário que se tenha uma prova da proposição `eq_axiom A`, isto é, a definição `eq_axiom` com `T` igual a `A`. Tal teorema indica que a igualdade sobre `A` é decidível, o que fica claro pelo seguinte lema:

```
Lemma decP: forall (P : Prop) (b : bool), reflect P b → decidable P
```

A utilização de um tipo como `eqType` facilita que se provem teoremas genéricos, no sentido de que servem para diferentes tipos (pertencentes a `Type`) desde que estes possuam uma relação de equivalência decidível. Existem outros tipos semelhantes a `eqType`, no sentido de que servem como interfaces. Em grande parte, esses são implementados por meio do açúcar sintático `Record`, qual será explicado na sessão seguinte.

2.3 Structures e Records

`Structure` e `Record` são comandos sinônimos para geração de tipos indutivos que possuem somente um construtor e cujo os campos são dependentemente tipados, isto é, o tipo de cada campo pode depender dos valores de campos anteriores, assim como nas definições indutivas (MAHBOUBI; TASSI, 2022). A vantagem do uso desses comandos é que por meio desses são geradas automaticamente funções para extrair valores dos argumentos do construtor do tipo declarado.

Estes comandos são frequentemente utilizados na biblioteca Mathematical Components para definir interfaces (como o `eqType`) e subtipos (ex.: tipo em que os habitantes são todos os números naturais menores que 8). Para melhor entendimento do leitor, tem-se a seguir um exemplo semelhante ao tipo `eqType` definido na biblioteca, apresentado em (MAHBOUBI; TASSI, 2022):

```
Record eqType : Type := Pack
{
    sort : Type;
    eq_op : sort → sort → bool;
    axiom : eq_axiom eq_op
}.
```

Como explicado acima, essa declaração é equivalente a se fazer as seguintes declarações:

```
Inductive eqType : Type :=
    | Pack (sort : Type) (eq_op : sort → sort → bool) (axiom : eq_axiom eq_op).

Definition sort (e : eqType) : Type :=
    match e with
    | Pack t _ _ ⇒ t
```

```
    end.
Definition eq_op (e : eqType) : (sort e → sort e → bool) :=
    match e with
    | Pack _ f _ ⇒ f
    end.
Definition axiom (e : eqType) : (eq_axiom (eq_op e)) :=
    | Pack _ _ a ⇒ a
    end.
```

Observe que o uso de tipos dependentes ocorre nos campos `eq_op` e `axiom`. No primeiro, o tipo do campo depende do valor do campo `sort` e no segundo o tipo do campo depende do valor do campo `eq_op` e portanto também do campo `sort`.

2.3.1 Comando Canonical

Assim como apresentado em (MAHBOUBI; TASSI, 2022), para instanciar um habitante de `eqType` com campo `sort` igual a `nat`, deve-se provar o seguinte teorema:

```
Theorem axiom_nat: eq_axiom eqn.
```

onde `eqn` é uma operação de comparação booleana entre números naturais. Tendo esta prova, podemos instanciar tal habitante da seguinte forma:

```
Definition natEqtype := Pack nat eqn axiom_nat.
```

Note que, agora, pode-se comparar dois números naturais da seguinte maneira:

```
Compute (@eq_op natEqType 2 2).
```

o que nesse caso equivale a:

```
Compute (eqn 2 2).
```

Entretanto, o objetivo de criar o tipo `eqType` não é estabelecer essa possibilidade de computação para relações de comparação, mas sim construir definições, funções e provas genéricas para todos os tipos que pertencem ao campo `sort` de algum habitante de `eqType` e estabelecer *overloading* de notações. Para exemplo de como alcançar este último objetivo, se define uma notação da seguinte forma:

```
Notation "x == y" := (@eq_op _ x y).
```

porém havendo apenas esta definição, caso executado o comando `Check (3 == 2)` tem-se um falha. Ao se executar:

```
Fail Check (3 == 2).
```

a seguinte mensagem é apresentada:

```
The command has indeed failed with message:
The term "3" has type "nat" while it is
expected to have type "sort ?e"
```

Isto ocorre pois o *Coq* não é capaz de inferir o argumento implícito[3] (_).

Note que é mencionada uma variável `?e`. Essa representa um elemento a ser inferido de modo que o tipo `sort ?e` seja igual ao tipo `nat`. Como exposto em (MAHBOUBI; TASSI, 2013) o algoritmo de inferência do *Coq* não é capaz de descobrir o valor de tal variável por meio das regras de inferência que possui. Para resolver este problema o *Coq* permite que se adicione regras de inferência de tipo por meio do comando `Canonical` (que recebe um construtor de algum `Record` ou `Structure` aplicado aos seus argumentos). Assim, resolver esse problema é possível por meio do seguinte código:

```
Canonical natEqType.
```

Com isto, de maneira semelhante ao exemplo exposto em (MAHBOUBI; TASSI, 2013), é adicionada a seguinte regra de inferência ao algoritmo presente em *Coq*:

$$\frac{\texttt{nat} \sim \texttt{sort natEqType} \qquad \texttt{?e} \sim \texttt{natEqType}}{\texttt{nat} \sim \texttt{sort ?e}}$$

em que a notação $\sim$ representa uma chamada do algoritmo de unificação (MAHBOUBI; TASSI, 2013) (que é o nome dado ao algoritmo de comparação de tipos chamado nas rotinas de inferência de tipos). Neste momento o leitor pode se perguntar como tal regra de inferência leva o algoritmo a chegar em um resultado final, e a resposta de acordo (MAHBOUBI; TASSI, 2013) está em na existência de outras regras de inferência, como *eq* e *assign*:

$$\frac{}{\texttt{t} \sim \texttt{t}}\,eq \qquad\qquad \frac{}{\texttt{?x} \sim \texttt{t}}\,assign$$

Retornando ao comando `Check`, se agora esse for executado da mesma maneira feita anteriormente (porém sem o `Fail`):

```
Check (3 == 2).
```

tem-se o seguinte resultado:

```
3 == 2
   : bool
```

[3] Note que o comando `Fail Check (3 == 2)` é equivalente a `Fail Check (@eq_op _ 3 2)`.

Como mencionado previamente o tipo `eqType` serve para diversas generalizações. Para exemplo disso, adiante se apresenta a definição de uma comparação entre valores do tipo `option`. Antes desse exemplo, vale aqui relembrar o leitor da definição deste tipo:

```
Inductive option (A : Type) : Type :=
  | None : option A
  | Some : A → option A.
```

A função de comparação a ser declarada irá considerar que o argumento `A` pertence ao campo `sort` de algum habitante de `eqType`. Tem-se então a definição dessa função:

```
Definition cmp_option (e : eqType) (o1 o2 : option (sort e)) :=
  match o1, o2 with
  | Some e1, Some e2 ⇒ op e e1 e2
  | None, Some _ ⇒ false
  | Some _, None ⇒ false
  | None, None ⇒ true
  end.
```

Agora, para criar um habitante de `eqType` para todo tipo da forma `option A` (note que para todo `A` diferente tem-se um tipo diferente) em que o tipo `A` segue o que foi considerado na função, deve-se provar o seguinte teorema:

```
Theorem axiom_option:
  forall e : eqType, eq_axiom (cmp_option e).
```

que para facilidade de entendimento do leitor, pode ser escrito como:

```
Theorem axiom_option:
  forall e : eqType, forall x y : option (sort e), reflect (x = y) (@cmp_option e x y).
```

Com esta prova pode-se construir a seguinte definição:

```
Definition optionEqType (e : eqType) :=
  Pack (option (sort e)) (cmp_option e) (axiom_option e).
```

Visto que ainda não foi executado o comando `Canonical` com esta definição, se executado o comando `Check (Some 1 == Some 2)`, esse irá falhar, logo, ao se executar:

```
Fail Check (Some 1 == Some 2).
```

É apresentada a seguinte mensagem:

```
The command has indeed failed with message:
The term "Some 1" has type "option nat"
while it is expected to have type "sort ?e".
```

Semelhante ao que foi feito anteriormente, para resolver este problema deve-se executar:

```
Canonical optionEqType.
```

e com isso se adiciona a seguinte regra de inferência:

$$\frac{\texttt{t} \sim \texttt{sort ?x} \qquad \texttt{?e} \sim \texttt{optionEqType ?x}}{\texttt{option t} \sim \texttt{sort ?e}}$$

Assim note que no problema de inferência acima, ocorre a seguinte (sub)sequência de aplicação de regras de inferência para se determinar o valor de ?e:

$$\frac{\dfrac{\texttt{nat} \sim \texttt{sort natEqType} \qquad \texttt{?x} \sim \texttt{natEqType}}{\texttt{nat} \sim \texttt{sort ?x}} \qquad \texttt{?e} \sim \texttt{optionEqType ?x}}{\texttt{option nat} \sim \texttt{sort ?e}}$$

Agora, com o comando:

```
Check (Some 1 == Some 2).
```

tem-se a mensagem:

```
Some 1 == Some 2
    : bool
```

2.3.2 Comando Coercion

Em provas manuais costuma-se utilizar notações iguais para operações sobre diferentes tipos. Como exemplo, há o uso do símbolo + para operação de soma sobre os conjuntos númericos $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, como operador lógico *ou* e também como operação binária qualquer que forma um monoide genérico. Nessa aplicação cotidiana de *overloading* de notações, as informações sobre tipos são inferidas pelo cérebro humano conforme o contexto em que se encontram (MAHBOUBI; TASSI, 2013). Tomando isso em consideração e tendo em mente o conteúdo abordado na subseção anterior, pode-se dizer que o comando `Canonical` auxilia os usuários, tornando a escrita em *Coq* mais semelhante a que se faz manualmente.

Outro mecanismo relacionado a tipos em *Coq*, e que de certa forma serve para esse mesmo propósito, é provido pelo comando `Coercion`. Como motivação para o uso deste, suponha a declaração em *Coq* de um tipo que contenha todos os números naturais múltiplos de um número n. Utilizando `Record`, temos então:

```
Record multiple (n : nat) : Type := Build
{
    x : nat;
```

```
    axiom : (n %| x)
}.
```

Observe que para construir um habitante deste tipo é dado um elemento `n` de tipo `nat` e é necessário um elemento `x` de mesmo tipo, e além disso, uma prova[4] de que `x` é divisível por `n`. Agora, com tal declaração, visto que o objetivo da mesma é representar qualquer conjunto de números divisíveis por um determinado natural n, o usuário de *Coq* provavelmente desejará que se possa escrever algo como:

```
forall (n : nat) (a : multiple n), a + 0 = a.
```

sem que o uso da operação de adição leve a um erro pela razão dessa possuir tipo `nat` → `nat` → `nat` enquanto o argumento `a` tem tipo `multiple n`, quando a intenção do usuário é de que este último represente um número natural. Se for utilizado o comando `Check` na proposição acima:

```
Fail Check (forall (n : nat) (a : multiple n), a + 0 = a).
```

tem-se a mensagem:

```
The command has indeed failed with message:
In environment
n : nat
a : multiple n
The term "a" has type "multiple n" while it is
expected to have type "nat".
```

Buscando solucionar este tipo de problema, sem que tenha que se escrever:

```
forall (n : nat) (a : multiple n), (x n a) + 0 = (x n a).
```

o que por sua vez não geraria erro algum pois `(x n a)` tem tipo `nat`[5], defini-se uma função que retira o campo `x` de um elemento como `a`:

```
Definition multiple_nat (n : nat) (e : multiple n) : nat :=
    let t := (x n e) in t.
```

e agora, para que o *Coq* aplique esta função de maneira implícita, de modo a evitar erros de tipo, usa-se o comando `Coercion` da seguinte maneira:

```
Coercion multiple_nat : multiple ↦ nat.
```

[4] Há de maneira implícita a aplicação da função `is_true` no tipo do campo `axiom`, portanto o que foi declarado como tipo deste campo, isto é, `n %| x`, é equivalente a `n %| x = true`.

[5] Lembre-se que `x`, no contexto externo a declaração do seu respectivo `Record`, é uma função que extrai o campo `x` de um elemento do tipo `multiple n` (para qualquer `n`) e não o valor do campo em si. Portanto a seguinte definção poderia ser dada simplesmente como `x`.

Agora, realizando o comando Check como anteriormente:

```
Check (forall n (a : multiple n), a + 0 = 0).
```

é gerada a mensagem:

```
forall (n : nat) (a : multiple n), a + 0 = 0
   : Prop
```

2.3.3 Exemplo de Implementação de grupos

Um uso semelhante do mecanismo *coercion*, junto ao *canonical*, pode ser proposto com um tipo que representa grupos. Um grupo é uma estrutura algébrica dada por $(G, \otimes)$ onde:

1. $\otimes$ é uma operação binária sobre G, isto é, $\otimes$ é uma função tal que $\otimes : (G \times G) \to G$.
2. $\otimes$ é associativa, ou seja, $\forall a, b \in G, (a \otimes b) \otimes c = a \otimes (b \otimes c)$.
3. Existe um elemento neutro e, o que significa: $\exists e \in G(\forall a \in G, e \otimes a = a \otimes e = a)$.
4. Para todo elemento em G existe um elemento inverso, isto é, $\forall x \in G(\exists\, \overline{x} \in G, x \ \otimes \ \overline{x} = \overline{x} \ \otimes \ x = e)$

Em *Coq* um grupo pode ser representado pelo seguinte Record:

```
Record Group : Type := group
{
   sort :> Type;
   bin_op : sort → sort → sort;
   associative_axiom : associative bin_op;
   e : sort;
   neutral_left : left_id e bin_op;
   neutral_right : right_id e bin_op;
   inverse_left : ∀ x : sort, ∃ y : sort, bin_op y x = e;
   inverse_right : ∀ x : sort, ∃ y : sort, bin_op x y = e
}.
```

Observe que, diferente dos exemplos anteriores, o campo sort é seguido de :>. Esse operador além de atribuir o tipo de sort como Type define a função sort como uma *coercion* para todo habitante do tipo Group. Assim, suponha a definição do seguinte habitante:

```
Definition int_group :=
   Group int addz addzA 0 add0z addz0 inverse_left_int inverse_right_int.
```

Esse habitante tem como campo `sort` o tipo `int` (que representa os números inteiros) e devido a *coercion*, se for feita uma declaração com uma variável de tipo `int_group` em que se aplica uma função de tipo `int` → `int` sobre esta variável, o *Coq* irá automaticamente tratar a variável como tendo tipo `int` (ou mais especificamente, irá tratá-la como se fosse o valor de seu campo `sort`, aplicando de maneira implícita a função `sort` sobre a mesma).

Para fins de demonstrar uma implentação completa de grupos em *Coq*, define-se agora uma notação para as operações binárias e uma para os elementos neutros que formam grupos quaisquer, da seguinte forma:

```
Notation "x ⊗ y" := (@bin_op _ x y) (at level 10).
Notation "0" := (@e _).
```

Usa-se então o comando `Canonical` para que o *Coq* seja capaz de inferir os argumentos implícitos presentes nas descrições destas notações (para o caso de `x` e `y` possuirem tipo `int`):

```
Canonical int_group.
```

Com este conjunto de configurações passa a ser mais fácil a escrita e leitura de declarações relacionadas a grupos. Assim, torna-se então possível a formalização compacta de teoremas genéricos sobre quaisquer tipos presentes no campo `sort` de algum habitante de `group`. Como exemplo, tem-se o seguinte teorema e sua respectiva prova:

```
Theorem exemplo_sobre_grupos:
    ∀ G : group, ∀ a b : G, (a ⊗ b) ⊗ 0 = (a ⊗ 0) ⊗ b.
Proof.
    intros. rewrite (neutral_right G). rewrite (neutral_right G).
    reflexivity.
Qed.
```

Como este teorema serve para qualquer grupo `G`, esse pode então ser usado na seguinte prova:

```
Theorem exemplo_sobre_int:
    ∀ a b : int, (a + b) + 0 = (a + 0) + b.
Proof.
    apply exemplo_sobre_grupos.
Qed.
```

Tal prova exemplifica como o uso dos mecanismos em *Coq*, que foram apresentados neste capítulo, podem ser utilizados para que seja mais fácil trabalhar com um vasta quantidade de tipos que apresentam propriedades em comum (como é o caso dos grupos).

2.3.4 Mantendo Informações de um Record ou Structure

Em meio as provas que envolvem tipos de `Record` ou `Structure`, o usuário de *Coq* pode se deparar com situações em que, ao se aplicar uma determinada função sobre uma variável relacionada a um desses comandos, que portanto apresenta um determinado conjunto de propriedades, o resultado da computação dessa função irá retornar um dado do tipo definido pela *coercion*. Em algumas dessas ocasiões, no entanto, a função aplicada retornará um dado com o qual se poderia construir uma nova variável do mesmo tipo de `Record` (ou `Structure`) do elemento do qual o argumento da função foi extraído. Manter o tipo do resultado como o mesmo `Record` ou `Structure` pode ser útil em algumas provas, e fazer isso é possível através do comando `Canonical`. A exemplo disso, retomando ao tipo `multiple`, note que é possível provar os dois seguintes teoremas:

```
Theorem exemplo_multiple_axiom {n} :
    ∀ (a : multiple n), n %| a.
Theorem exemplo_aplicacao_f_mul {n} :
    ∀ (f : nat → nat) (a : multiple n), n %| (f a) ∗ n.
```

onde %% é a operação de resto da divisão. Agora, imagine que se queira provar o seguinte:

```
Example exemplo_a_provar {n} :
    ∀ (a : multiple n), (n %| ((fun x ⇒ x + 2) a) ∗ n).
```

Note que, se tratando `multiple n` como um conjunto de números naturais (apesar desse não ser precisamente isso) faria sentido poder utilizar o teorema `exemplo_multiple_axiom` para reescrever o lado esquerdo da equação como `true` (o operador %| tem tipo `nat` → `nat` → `bool` portanto há uma *coercion* `is_true` na declaração do exemplo), ao invés de ter que utilizar um teorema mais específico como `exemplo_aplicacao_f_mul`. Dado que se trata de uma aplicação de função em que após a aplicação o resultado é multiplicado por `n`, é óbvio que o resultado da expressão:

```
(((fun x ⇒ x + 2) a) ∗ n)
```

é divisível por `n`. Entretanto, a reescrita desejada não é possível pois ocorre um problema de unificação ao tentar se usar a tática `rewrite exemplo_multiple_axiom`. Para obter-se uma melhor noção sobre este problema é possível utilizar o seguinte código fornecido por (MAHBOUBI; TASSI, 2022) (sobre o qual a explicação de seu funcionamento vai além do escopo do presente trabalho):

```
Notation "X (*...*)" :=
    (let x := X in let y := _ in x) (at level 100, format "X (*...*)").

Notation "[LHS 'of' equation ]" :=
    (let LHS := _ in let _infer_LHS := equation : LHS = _ in LHS) (at level 4).
```

```
Notation "[unify X 'with' Y ]" :=
    (let unification := erefl _ : X = Y in True).
```

Com isso, pode-se executar o seguinte comando:

```
Fail Check (forall n (a : multiple n) (f : nat → nat),
    let LHS := [LHS of exemplo_multiple_axiom _] in
    let RDX := (n %| (f a) * n) in
    [unify LHS with RDX]).
```

Este comando irá apresentar uma mensagem de confirmação da falha, que em parte, apresenta o seguinte conteúdo:

```
The command has indeed failed with message:
In environment
n :
nat
a : multiple n
f : nat → nat
LHS := [LHS of exemplo_multiple_axiom ?a] : bool
RDX := n %| f (x n a) * n : bool
The term "erefl LHS" has type "LHS = LHS"
while it is expected to have type "LHS = RDX"
```

Com isto, pode-se verificar que o problema de unificação encontrado é descobrir qual o valor da variável ?a. De modo mais específico, o problema está em encontrar um valor que torne equivalentes as seguintes expressões:

```
n %| (x n ?a)
```

e

```
n %| ((f (x n a)) * n)
```

Para resolver este problema usa-se o comando `Canonical` junto ao teorema `exemplo_aplicacao_f_mul`, através do seguinte código:

```
Canonical f_mul_multiple {n : nat} (f : nat → nat) (a : multiple n) :=
    (@Build n ((f a) * n) (@exemplo_aplicacao_f_mul n f a)).
```

Retornando então a prova de motivação para introdução ao problema discutido (`exemplo_a_provar`) e utilizando a tática:

```
intros. simpl.
```

O *goal* da prova se torna:

```
n %| (a + 2) * n
```

Agora, para que se possa aplicar novamente a tática `rewrite` `exemplo_smaller_axiom`, é necessário deixar o *goal* escrito de modo que a expressão mais interna:

```
(a + 2)
```

fique na forma de uma função aplicada sobre um elemento múltiplo de `n`. Isso é necessário para que o *Coq* possa inferir um elemento do tipo `multiple n` dado pela definição `f_mul_multiple`, permitindo assim o uso do teorema `exemplo_multiple_axiom`. Como 2 é de tipo `nat`, o *Coq* não consegue construir um habitante do tipo `multiple n` que resolva o problema de unificação. O que se pode então fazer é inverter a ordem da função de soma, realizando a tática `rewrite` `addnC`, em que `addnC` é um teorema de comutativa da soma. Assim, o *goal* resultante será:

```
n %| ((2 + a) * n)
```

Agora, utilizando novamente a tática `rewrite` `exemplo_multiple_axiom`, tem-se:

```
true
```

Com isso a prova pode ser finalizada com o uso de `reflexivity`.

3 Base Teórica

Para que se realizem as implementações serão necessários diversos teoremas, lemas e funções, dos quais, parte, já estão implementados na biblioteca Mathematical Components. Sendo assim, o presente capítulo busca trazer a descrição da maioria destes itens, colocando também suas respectivas implementações disponíveis na biblioteca (se houver).

A maior parte do conteúdo deste capítulo se baseia no livro (BROCHERO et al., 2013), que foi amplamente estudado para realização deste trabalho. Sendo assim, as provas não apresentadas aqui se encontram nesse livro.

3.1 Máximo Divisor Comum

Tendo sido apresentado os conceitos de módulo e divisibilidade, outro pilar fundamental da Teoria dos Números é o conceito de mdc (máximo divisor comum). A princípio, a definição seria auto-explicativa, mas parte de diversos teoremas importantes a utiliza e portanto é interessante que se tenha uma definição equivalente específica para facilitar provas futuras. Para se obter tal definição alternativa, observe que, formalmente, a definição de mdc é:

$$\forall a, b, n \in \mathbb{Z}, \mathrm{mdc}(a, b) = n \Leftrightarrow (n \mid a) \wedge (n \mid b) \wedge (\forall k \in \mathbb{Z}, (k \mid a) \wedge (k \mid b) \rightarrow k \leq n)$$

Analisando a proposição:

$$\forall k \in \mathbb{Z}, (k \mid a) \wedge (k \mid b) \rightarrow k \leq n$$

Pode-se verificar os seguintes casos:

1. $|k| = |n|$, isto é, $k = n$ ou $k = -n$. Em ambos estes casos $k \mid n$.

2. $|k| < |n|$, assim, como $n \mid a$ e $n \mid b$, então, $\exists q_a, q_b \in \mathbb{Z}, (a = q_a \cdot n \wedge b = q_b \cdot n)$, onde $\mathrm{mdc}(q_a, q_b) = 1$ (caso contrário n não seria $\mathrm{mdc}(a, b)$, pois haveria o divisor $\mathrm{mdc}(q_a, q_b) \cdot n$ que é maior que n, se $n > 1$). Portanto, como $k \mid a$ e $k \mid b$, então $k \mid q_a \cdot n$ e $k \mid q_b \cdot n$, mas sabe-se que $k \nmid q_a$ e $k \nmid q_b$, logo, $k \mid n$.

Olhando o que acontece quando $k \mid n$, é fácil notar que $|k| \leq |n|$, logo, no contexto em que $k \mid a$ e $k \mid b$, as afirmações $k \leq n$ e $k \mid n$ são equivalentes. Sendo assim tem-se a seguinte definição alternativa:

$$\forall k \in \mathbb{Z}, (k \mid a) \wedge (k \mid b) \rightarrow k \mid n$$

3.2 Algoritmo de Euclides

O algoritmo de Euclides é um método de computar o mdc entre dois números inteiros a e b, tendo a seguinte descrição (Algoritmo 1):

Algoritmo 1: EUCLIDES

```
Entrada: a, b ∈ ℤ
Saída: inteiro n.
se a = 0 então
  | retorna b
senão
  | retorna EUCLIDES(b, a mod b)
```

Este algoritmo se baseia no seguinte lema:

Lema 1 *$\forall a, b \in \mathbb{Z}$, seja $a = b \cdot q + r$, onde $0 \leq r < |b|$, então:*

$$\mathrm{mdc}(a, b) = \mathrm{mdc}(b, r)$$

Demonstração: inicialmente deve-se notar que, provar tal lema equivale a demonstrar:

$$\forall n \in \mathbb{Z}, (\mathrm{mdc}(a, b) = n \leftrightarrow \mathrm{mdc}(b, r) = n) \tag{3.1}$$

Além disso, deve-se considerar o seguinte lema trivial que versa sobre combinações lineares:

Lema 2 (*Divisibilidade e combinações lineares*)

$$\forall a, b, n \in \mathbb{Z}, (n \mid a) \wedge (n \mid b) \rightarrow \forall c_1, c_2 \in \mathbb{Z}, n \mid (c_1 \cdot a + c_2 \cdot b)$$

Dados esses adendos, prova-se inicialmente a volta da bi-implicação 3.1. Para isso, observe que, se $x = \mathrm{mdc}(b, r)$ então $x \mid b$ e $x \mid r$, e como $r = a - b \cdot q$ então $x \mid (a - b \cdot q)$. Pode-se então fazer uma combinação linear escolhendo $c_1 = q$ e $c_2 = 1$, donde se chega em:

$$x \mid q \cdot b + 1 \cdot (a - b \cdot q)$$

$$x \mid q \cdot b + a - b \cdot q$$

$$x \mid a$$

Resta então provar que:

$$\forall y \in \mathbb{Z}, (y \mid a) \wedge (y \mid b) \rightarrow (y \mid x)$$

Da hipótese tem-se que

$$\forall y \in \mathbb{Z}, (y \mid b) \wedge (y \mid r) \rightarrow (y \mid x)$$

Se $y \mid a$ e $y \mid b$, como $a = b \cdot q + r$ então $y \mid (b \cdot q + r)$. Utilizando o teorema sobre combinação linear novamente, com $c_1 = -q$ e $c_2 = 1$:

$$y \mid -q \cdot b + 1 \cdot (b \cdot q + r)$$

$$y \mid -b \cdot q + b \cdot q + r$$

$$y \mid r$$

Como $y \mid b$ e $y \mid r$, da hipótese temos que $y \mid x$ portanto se provou o que restava. Para a ida da bi-implicação a prova é semelhante. ■

O algoritmo EUCLIDES é implementado da seguinte forma na biblioteca Mathematical Components (em sua versão para números naturais):

```
Fixpoint gcdn m n :=
  let n' := n %% m in if n' is 0 then m else
  if m - n'.-1 is m'.+1 then gcdn (m' %% n') n' else n'.
```

A versão para números inteiros é basicamente `gcdn` aplicada sobre o valor absoluto dos números inteiros de entrada. Sobre a notação %% essa representa a operação de resto da divisão, que, para números naturais é definida por:

```
Definition modn_rec d :=
  fix loop m := if m - d is m'.+1 then loop m' else m.
Definition modn m d := if d > 0 then modn_rec d.-1 m else m.
Notation "m %% d" := (modn m d) : nat_scope.
```

E utilizando a função `divz` para divisão entre números inteiros, é implementada da seguinte forma (para inteiros):

```
Definition modz (m d : int) : int := m - divz m d * d.
Infix "%%" := modz : int_scope.
```

3.3 Teorema Bachet-Bézout

Uma das consequências teóricas da definição de mdc é o Teorma Bachet-Bézout. Este por sua vez traz uma aplicação do conceito de mdc na resolução de equações. O seu enunciado é dado por:

Teorema 1 (*Bachet-Bézout*) $\forall a, b \in \mathbb{Z}, \exists x, y \in \mathbb{Z}$ *tal que*

$$a \cdot x + b \cdot y = \mathrm{mdc}(a, b)$$

Como exemplo de consequências deste teorema têm-se:

1. $\forall c \in \mathbb{Z}, c \mid a \wedge c \mid b \Rightarrow c \mid \mathrm{mdc}(a,b)$, ou seja, caso c divida tanto a quanto b então c divide $\mathrm{mdc}(a,b)$.

2. $\forall c \in \mathbb{Z}, (\exists x, y \in \mathbb{Z}, a \cdot x + b \cdot y = c) \iff \mathrm{mdc}(a,b) \mid c$, ou seja, a equação $a \cdot x + b \cdot y = c$ tem solução se e somente se $\mathrm{mdc}(a,b)$ divide c.

A prova manual do Teorema 1 e das consequências mencionadas se encontra em (BROCHERO et al., 2013).

Em relação à biblioteca Mathematical Components e se tratando do Teorema 1, essa possui a implementação de um algoritmo que encontra os coeficientes x e y e o valor de $\mathrm{mdc}(a,b)$. Este algoritmo possui duas versões, sendo identificado na biblioteca como `egcdn` para naturais e `egcdz` para inteiros. Além disso, existem os lemas `egcdn_spec` e `egcdz_spec`, tratando da corretude desses algoritmos (respectivamente) e para o caso dos números inteiros, há o lema `Bezoutz` que é equivalente ao Teorema 1.

3.4 Propriedades de Congruência

As propriedade da relação de congruência serão usadas com muita frequência (e de maneira implícita) no decorrer desse documento. Por essa razão, nesta seção serão listadas essas propriedades. Recomenda-se então ao leitor consultar o conteúdo aqui apresentado em caso de dúvidas no desenvolvimento de equações modulares.

Seguindo para as propriedades, para todo $a, b, c, d, n \in \mathbb{Z}$ têm-se:

1. (*Reflexividade*) $a \equiv a \pmod{n}$

2. (*Simetria*) $a \equiv b \pmod{n} \implies b \equiv a \pmod{n}$

3. (*Transitividade*) $a \equiv b \pmod{n} \wedge b \equiv c \pmod{n} \implies a \equiv c \pmod{n}$

4. (*Compatibilidade com a soma*)
$$a \equiv b \pmod{n} \wedge c \equiv d \pmod{n} \implies a + c \equiv b + d \pmod{n}$$

5. (*Compatibilidade com a diferença*)
$$a \equiv b \pmod{n} \wedge c \equiv d \pmod{n} \implies a - c \equiv b - d \pmod{n}$$

6. (*Compatibilidade com o produto*)
$$a \equiv b \pmod{n} \wedge c \equiv d \pmod{n} \implies a \cdot c \equiv b \cdot d \pmod{n}$$

   A partir dessa propriedade, note que, para todo $k \in \mathbb{N}$:
$$a \equiv b \pmod{n} \implies a^k \equiv b^k \pmod{n}$$

7. (*Cancelamento*) $\mathrm{mdc}(c, n) = 1 \Longrightarrow (a \cdot c \equiv b \cdot c \pmod{n} \Longleftrightarrow a \equiv b \pmod{n})$

Na biblioteca Mathematical Components, as relações de módulo são definidas pelas seguintes notações:

```
Notation "m = n %[mod d ]" := (modz m d = modz n d) : int_scope.
Notation "m == n %[mod d ]" := (modz m d == modz n d) : int_scope.
Notation "m <> n %[mod d ]" := (modz m d <> modz n d) : int_scope.
Notation "m != n %[mod d ]" := (modz m d != modz n d) : int_scope.
```

Deve-se observar que existem duas igualdades, sendo a primeira a de Leibniz e a segunda uma função booleana. O mesmo ocorre com as desigualdades (na mesma ordem).

Tais definições são equivalentes a definição apresentada por (BROCHERO et al., 2013), conforme é mostrado pelo lema `eqz_mod_dvd` implementado na biblioteca [1]:

```
Lemma eqz_mod_dvd d m n : (m == n %[mod d])%Z = (d %| m - n)%Z.
```

Quanto a implementação das propriedades apresentadas nesta seção, a biblioteca não as implementa, apesar de fazer isso para lemas semelhantes a algumas destas propriedades. A exemplo tem-se:

```
Lemma modzDm m n q : ((m %% q)%Z + (n %% q)%Z = m + n %[mod q])%Z.
```

Este lema se assemelha a propriedade de *compatibilidade com a soma*, no sentido de que, considerando a existência do seguinte lema:

```
Lemma modz_mod m d : ((m %% d)%Z = m %[mod d])%Z.
```

possuem a mesma utilidade.

3.5 Anel de Inteiros Módulo n

Uma estrutura que será utilizada no presente trabalho, e que pode ser implementada usando elementos disponíveis na biblioteca Mathcomp, são os anéis de inteiros módulo n. De acordo com (BROCHERO et al., 2013), dada a relação $\sim$ sobre um conjunto X, se esta relação é uma relação de equivalência, isto é, possui as seguintes propriedades:

1. ***reflexividade***: $\forall x \in X, x \sim x$

2. ***transitividade***: $\forall x, y, z \in X, x \sim y \wedge y \sim z \rightarrow x \sim z$

3. ***simetria***: $\forall xy \in X, x \sim y \leftrightarrow y \sim x$

[1] O operador %Z serve para indicar o escopo da operação dentro do parênteses

Como exposto em (BROCHERO et al., 2013), estabelecer uma relação de equivalência sobre um conjunto X é o mesmo que definir um partição sobre o mesmo, isto é, dividir X em subconjuntos, em que, sendo cada subconjunto identificado como X_λ onde λ pertence a um conjunto Λ, então

$$X = \bigcup_{\lambda \in \Lambda} X_\lambda$$

Particionando X por meio da relação $\sim$, tem-se que, dados $x, y \in X$, então $x, y \in X_\lambda$ se e somente se $x \sim y$. Além disso pode-se definir a *classe de equivalência* $\overline{x}$ em que:

$$\overline{x} = \{y \in X \mid y \sim x\}$$

O conjunto de classes de equivalência $\{\overline{x} \mid x \in X\}$ é denominado quociente de X por $\sim$ e é representado por $X/\sim$.

Particionando $\mathbb{Z}$ por meio da relação $\equiv \mod n$ para algum $n \in \mathbb{Z} - \{0\}$, tem-se um conjunto de *classes de equivalência* denominado *anel de inteiros módulo* n, que costuma ser representado por $\mathbb{Z}/(n)$, onde então:

$$\mathbb{Z}/(n) = \{\overline{0}, ..., \overline{n-1}\}$$

Note que, no entanto, as classes de equivalência podem ser denotadas por diferentes números, desde que tenha o mesmo resto na divisão inteira por n. A exemplo disso observe que:

$$\overline{0} = \overline{n}$$

pois

$$0 \equiv n \pmod{n} \tag{3.2}$$

Portanto, para quaisquer $a, b \in \mathbb{Z}$, se $\overline{a}, \overline{b} \in \mathbb{Z}/(n)$ tem-se:

$$\overline{a} = \overline{b} \iff a \equiv b \pmod{n}$$

Com isso, dada que as seguintes propriedades são válidas para as relações de congruência, com quaisquer $a, b, c, d \in \mathbb{Z}$:

- $a \equiv b \mod n \wedge c \equiv d \pmod{n} \Rightarrow a + c \equiv b + d \pmod{n}$
- $a \equiv b \mod n \wedge c \equiv d \pmod{n} \Rightarrow a - c \equiv b - d \pmod{n}$
- $a \equiv b \mod n \wedge c \equiv d \pmod{n} \Rightarrow a \cdot c \equiv b \cdot d \pmod{n}$

Se define então as operações de soma, subtração e multiplicação em $\mathbb{Z}/(n)$, das seguintes formas para todo $\overline{a}, \overline{b} \in \mathbb{Z}/(n)$

- $\overline{a} + \overline{b} = \overline{a + b}$

- $\overline{a} - \overline{b} = \overline{a - b}$
- $\overline{a} \cdot \overline{b} = \overline{a \cdot b}$

Como explicado em (BROCHERO et al., 2013) e devido a estas operações, o nome *anel de inteiros módulo n* é justificado pela definição de *anel*: qualquer conjunto A com duas operações binárias $+$ e $\cdot$, de modo que A satisfaz as seguintes propriedades:

- $(A, +)$ é um *grupo abeliano*, isto é, um grupo que possui a propriedade de *comutatividade* por meio da operação binária $+$, ou seja:

$$\forall a, b \in A, a + b = b + a$$

  com elemento neutro 0 (neste caso 0 é um elemento quaisquer, e não necessariamente o número 0).

- $(A, \cdot)$ é um monoide com elemento neutro 1.

Com esta definição de *anel* chega-se em outras também importantes:

- se $\forall a, b \in A, a \cdot b = b \cdot a$ então A é um *anel comutativo*.
- se A é um *anel comutativo* em que os elementos neutros deste são diferentes $(0 \neq 1)$ e $\forall a, b \in A, a \cdot b = 0 \Rightarrow a = 0 \vee b = 0$ então A é um *domínio*.
- se A é um *anel comutativo* em que os elementos neutros deste são diferentes $(0 \neq 1)$ e todo elemento diferente de 0 em A possui inverso na operação $\cdot$, isto é, $(A - \{0\}, \cdot)$ é um grupo então A é um *corpo*, o que em inglês é conhecido como *field*, e em geral se representa como $\mathbb{F}_n$ ao invés de simplesmente $\mathbb{Z}/(n)$.

Havendo conhecimento da definição de *corpos*, tem-se o seguinte importante Lema apresentado em (BROCHERO et al., 2013) (cuja a prova depende fortemente do Lema 5):

Lema 3 *$\forall n \in \mathbb{Z}, \mathbb{Z}/(n)$ é um corpo se e somente se n é primo.*

Note que, pelo Lema 4, em um *anel de inteiros módulo n*, só há inverso multiplicativo para um determinada *classe de equivalência* $\overline{a}$ se $\mathrm{mdc}(a, n) = 1$, pois só assim existirá outra *classe de equivalência* $\overline{b}$ tal que $\overline{a} \cdot \overline{b} = \overline{a \cdot b} = \overline{1}$ (onde $\overline{1}$ é o elemento neutro da operação $\cdot$).

Outro conceito importante originado a partir da definição de *anéis de inteiros módulo n* é a definição de *grupo de unidades*, denotado por $(\mathbb{Z}/(n))^{\times}$. Esse é um subconjunto formado pelas *classes de equivalência* invertíveis de $\mathbb{Z}/(n)$, ou seja:

$$(\mathbb{Z}/(n))^{\times} = \{\overline{a} \in \mathbb{Z}/(n) |\, \mathrm{mdc}(a, n) = 1\} \tag{3.3}$$

Sobre a implementação de *anéis de inteiros módulo n* na biblioteca Mathematical Components, conforme é apresentado em (MAHBOUBI; TASSI, 2022), essa implementação envolve o tipo `ordinal`. Este é declarado da seguinte forma (junto de sua notação e *coercion* para `nat`):

```
Inductive ordinal n := Ordinal m of m < n.
Notation "'I_' n" := (ordinal n).
Coercion nat_of_ord n (i : 'I_n) := let: @Ordinal _ m _ := i in m.
```

Com isso, para implementar operações sobre *inteiros módulo n*, utiliza-se o seguinte código:

```
Variable p' : nat.
Local Notation p := p'.+1.
Implicit Types x y z : 'I_p.
Definition inZp i := @Ordinal p (i %% p) (ltn_pmod i (ltn0Sn p')).
```

em que o comando `Variable` declara uma variável no contexto de quaisquer declarações a partir daquela linha, o que é equivalente a utilizar nessas ∀ (p' : `nat`) (e é o que é considerado nas declarações fora da `Section` em que foi declarada a variável). Sobre o comando `Local Notation`, esse cria um notação válida apenas para o módulo em que ela é declarada (assim, importar o módulo não importará a notação), e quanto ao comando `Implicit Types`, esse faz com que nas declarações a seguir, se forem utilizadas variáveis com tipos implícitos e de nome `x`, `y` ou `z`, o *Coq* infira como tipo dessas `'I_p`.

Em relação a definição `InZp`, esta serve como maneira para converter números naturais em elementos do tipo `'I_p`, e portanto, é útil para definir operações como soma e multiplicação sobre `'I_p`. Tal definição utiliza 2 lemas, cujas proposições são:

```
Lemma ltn_pmod m d : 0 < d → m %% d < d.
Lemma ltn0Sn n : 0 < n.+1.
```

Portanto note que a expressão `ltn_pmod i (ltn0Sn p')` constrói uma prova de que `i %% p < p`, assim construindo um objeto do tipo `'I_p`. Por isso a definição de local da variável `p` força que esta seja maior ou igual a 1 (não é possível construir um objeto do tipo `'I_0`).

São então definidas as classes de equivalência $\overline{0}$ e $\overline{1}$ e operações para instâncias de `'I_p` da seguinte maneira:

```
Definition Zp0 : 'I_p := ord0.
Definition Zp1 := inZp 1.
Definition Zp_opp x := inZp (p − x).
Definition Zp_add x y := inZp (x + y).
Definition Zp_mul x y := inZp (x ∗ y).
Definition Zp_inv x := if coprime p x then inZp (egcdn x p).1 else x.
```

onde `ord0` tem a seguinte definição:

```
Definition ord0 := Ordinal (ltn0Sn n').
```

ou seja, `ord0` é sempre o elemento de `'I_p` construído com `m` sendo 0 (em que `p` é um valor inferido pelo *Coq*).

A partir disso podem ser provados os lemas que tornam o conjunto `'I_p` dotado das operações definidas com `inZp` em um *grupo abeliano* (conforme a definição mais em (BROCHERO et al., 2013)):

```
Lemma Zp_add0z : left_id Zp0 Zp_add.
Lemma Zp_addNz : left_inverse Zp0 Zp_opp Zp_add.
Lemma Zp_addA : associative Zp_add.
Lemma Zp_addC : commutative Zp_add.
```

e também podem ser então provados os lemas que farão de `'I_p` um *anel comutativo* (de acordo com (BROCHERO et al., 2013)), considereando as operações binárilas `Zp_add` e `Zp_mul`:

```
Lemma Zp_mulz1 : right_id Zp1 Zp_mul. (* Elemento neutro do produto a direita *)
Lemma Zp_mulC : commutative Zp_mul. (* Comutatividade do produto *)
Lemma Zp_mul1z : left_id Zp1 Zp_mul. (* Elemento neutro do produto a esquerda *)
Lemma Zp_mulA : associative Zp_mul. (* Associatividade do produto *)
Lemma Zp_mul_addr : right_distributive Zp_mul Zp_add. (* Distributividade a
    direita *)
Lemma Zp_mul_addl : left_distributive Zp_mul Zp_add. (* Distributividade a
    esquerda *)
```

Por fim, dada a condição para que um *anel comutativo* seja um *domínio* e para que seja *corpo*, as seguintes notações abragem os conjuntos `'I_p` que se encaixam nas respectivas categorias (desconsiderando a necessidade dos lemas sobre tais notações):

```
Definition Zp_trunc p := p.-2.

Notation "''Z_' p" := 'I_(Zp_trunc p).+2
  (at level 8, p at level 2, format "''Z_' p") : type_scope.

Notation "''F_' p" := 'Z_(pdiv p)
  (at level 8, p at level 2, format "''F_' p") : type_scope.
```

onde a notação `'Z_p` sempre retorna um tipo `'I_p` tal que $2 \leq p$, portanto é sempre um *domínio*, e a notação `'F_p` retorna um tipo `'I_p` tal que $2 \leq p$ e p é primo (pois a função `pdiv` retorna o primeiro primo divisor do número passado como argumento), portanto é sempre um *corpo*.

3.6 Função φ de Euler

Uma função muito presente em grande parte dos conteúdos de teoria dos números é a função φ de Euler. Essa também é conhecida como função totiente de Euler e conforme (BROCHERO et al., 2013), para quaisquer n inteiro positivo, é definida como:

$$\varphi(n) = |(\mathbb{Z}/(n))^{\times}| \tag{3.4}$$

e essa possui algumas propriedades importantes a serem destacadas:

1. $\varphi(1) = \varphi(2) = 1$
2. $\forall n, n > 2 \Rightarrow 1 < \varphi(n) < n$
3. $\forall p$, se p é primo então $\forall k \in \mathbb{N} - \{0\}, \varphi(p^k) = p^k - p^{k-1}$, portanto, $\varphi(p) = p - 1$
4. $\forall n, m \in \mathbb{N} - \{0\}, \text{mdc}(n, m) = 1 \Rightarrow \varphi(n \cdot m) = \varphi(n) \cdot \varphi(m)$
5. $\forall n \in \mathbb{N} - \{0\}$, se a fatoração de n em potências de primos distintos é dada por $n = p_1^{\alpha_1} \cdot ... \cdot p_k^{\alpha_k}$, então:

$$\varphi(n) = \prod_{1 \leq i \leq k} \varphi(p_i^{\alpha_i}) = \prod_{1 \leq i \leq k} p_i^{\alpha_i} - p_i^{\alpha_i - 1} = n \cdot \prod_{1 \leq i \leq k} \left(1 - \frac{1}{p_i}\right) \tag{3.5}$$

Além dessas propriedade existem dois teoremas apresentados em (BROCHERO et al., 2013) que devem ser notados, que são eles:

Teorema 2 (***Teorema de Euler-Fermat***) *$\forall a, m \in \mathbb{Z}$, se $m > 0$ e $\text{mdc}(a, m) = 1$ então:*

$$a^{\varphi(m)} \equiv 1 \pmod m$$

Demonstração: seja $R = \{r_1, r_2, ..., r_{\varphi(m)}\}$ o conjunto de valores no intervalo $[1, m-1]$ em que $\text{mdc}(r_i, m) = 1$ para $i \in [1, \varphi(m)]$ (por isso $|R| = |\varphi(m)|$), observe que o conjunto $A = \{a \cdot r_1, a \cdot r_2, ..., a \cdot r_{\varphi(m)}\}$ é composto apenas de valores tais que para $i \in [1, \varphi(m)]$, $\text{mdc}(a \cdot r_i, m) = 1$. Além disso, note que cada elemento do conjunto A, assim como cada um do conjunto R, é único, pois se $a \cdot r_i \equiv a \cdot r_j \pmod m$, então pelo Item 7, $r_i \equiv r_j \pmod m$, logo como $r_i, r_j \in [1, m-1]$, $r_i = r_j$ (ou seja, $i = j$). Como A possui a mesma quantidade de elementos que R, sendo todos distintos módulo m, então para cada elemento $a \cdot r_i \in A$ existe um elemento em $r_j \in R$ tal que $a \cdot r_i \equiv r_j \pmod m$. Essa última afirmação pode ser provada por absurdo, pois supondo que não exista tal r_j, então $a \cdot r_i \equiv r \pmod m$, tal que $r \notin R \wedge r \in [1, m-1] \wedge \text{mdc}(r, m) = 1$ (pelo Lema 1), mas se esse for o caso então R não é o conjunto de valores no intervalo $[1, m-1]$ em que $\text{mdc}(r_i, m) = 1$ para

$i \in [1, \varphi(m)]$, como definido inicialmente, logo, tem-se um absurdo. Por meio dos pares congruentes módulo m, de forma $a \cdot r_i \equiv r_j \pmod{m}$, tem-se pelo Item 6:

$$\prod_{i=1}^{\varphi(m)} a \cdot r_i \equiv \prod_{i=1}^{\varphi(m)} r_i \pmod{m}$$

manipulando a equação:

$$\begin{aligned} \prod_{i=1}^{\varphi(m)} a \cdot r_i &\equiv \prod_{i=1}^{\varphi(m)} r_i \pmod{m} \\ \iff a^{\varphi(m)} \cdot \prod_{i=1}^{\varphi(m)} r_i &\equiv \prod_{i=1}^{\varphi(m)} r_i \pmod{m} \end{aligned}$$

e como:

$$\operatorname{mdc}\left(\prod_{i=1}^{\varphi(m)} r_i, m\right) = 1$$

pelo Item 7 tem-se:

$$\begin{aligned} a^{\varphi(m)} \cdot \prod_{i=1}^{\varphi(m)} r_i &\equiv \prod_{i=1}^{\varphi(m)} r_i \pmod{m} \\ \iff a^{\varphi(m)} &\equiv 1 \pmod{m} \end{aligned}$$

■

Teorema 3 (***Pequeno Teorema de Fermat***) *$\forall a \in \mathbb{N} - \{0\}$, dado um número primo p, tem-se que:*

$$a^p \equiv a \pmod{p}$$

Demonstração: pelo Teorema 2 tem-se que:

$$a^{\varphi(p)} \equiv 1 \pmod{p}$$

e pelo Item 3:

$$\begin{aligned} a^{\varphi(p)} &\equiv 1 \pmod{p} \\ \iff a^{p-1} &\equiv 1 \pmod{p} \end{aligned}$$

assim utilizando-se da propriedade descrita no Item 6 tem-se:

$$\begin{aligned} a^{p-1} &\equiv 1 \pmod{p} \\ \iff a^{p} &\equiv a \pmod{p} \end{aligned}$$

■

Na biblioteca Mathematical Components, a função φ de Euler é implementada da seguinte maneira:

```
Definition totient n :=
  foldr add_totient_factor (n > 0) (prime_decomp n).
```

Em que por meio de uma *coercion* de `bool` para `nat` o valor retornado por `n` >0 é convertido para 0 (se for `false`) ou 1 (se for `true`) e a função `add_totient_factor` é definida como:

```
Definition add_totient_factor f m :=
  let: (p, e) := f in p.-1 * p ^ e.-1 * m.
```

e `prime_decomp` é uma função que recebe um número n qualquer e retorna uma lista de tuplas k da forma (p_i, e_i) em que:

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k}$$

ou seja, retorna a fatoração de n em primos, o que é garantido pelo seguinte lema `prime_decomp_correct` disponível na biblioteca (cuja proposição não será apresentada aqui devido a sua extensão).

Além disso, tem-se na biblioteca, lemas sobre algumas das propriedades aqui expostas. Iniciando pela propriedade 3, tem-se:

```
Lemma totient_pfactor p e :
 prime p -> e > 0 -> totient (p ^ e) = p.-1 * p ^ e.-1.
```

Também há um lema equivalente a propriedade 4 (em que a condição de maior divisor comum igual a 1 é dada por `coprime`, que é por sua vez uma função booleana que recebe dois números e retorna `true` se o mdc desses for 1 e `false` caso contrário):

```
Lemma totient_coprime m n :
  coprime m n -> totient (m * n) = totient m * totient n.
```

Por último, há também um lema que estabelece a equivalência entre a definição da função φ de Euler exposta em (BROCHERO et al., 2013) (Definição 3.4) e a definição da biblioteca baseada na Equação 3.5[2]

```
  Lemma totient_count_coprime n :
    totient n = \sum_(0 <= d < n) coprime n d.
```

3.7 Congruência de Grau 2 e Símbolo de Legendre

Sendo um técnica muito eficiente para verificar se um número é um resíduo quadrático em relação a um outro número primo, os símbolos de Legendre, além de serem

[2] Vale aqui observar que novamente há uma *coercion* de `bool` para `nat` sobre o retorna da função `coprime` (a *coercion* converte `false` para 0 e `true` para 1), dado que o somatório requer valores numéricos.

um objetivo de implementação deste trabalho, são diretamente utilizados no algoritmo RESSOL. Entretanto para se explicar o que são esses, é necessário uma breve introdução sobre congruências de grau 2 (ou quadráticas). Como motivação para se tratar deste assunto, note que, sendo $p > 2$ um número primo e $a, b, c \in \mathbb{Z}$, em que a não é divisível por p, suponha que se deseje resolver a seguinte equação:

$$a \cdot x^2 + b \cdot x + c \equiv 0 \pmod{p} \tag{3.6}$$

Manipulando essa equação com objetivo de obter um resultado semelhante ao da fórmula de Bhãskara, tem-se (multiplicando ambos os lados por 4):

$$4 \cdot a^2 \cdot x^2 + 4 \cdot a \cdot b \cdot x + 4 \cdot a \cdot c \equiv 0 \pmod{p}$$

e como

$$b^2 - 4 \cdot a \cdot c \equiv b^2 - 4 \cdot a \cdot c \pmod{p}$$

pode-se adicionar esses valor em ambos os lados:

$$4 \cdot a^2 \cdot x^2 + 4 \cdot a \cdot b \cdot x + b^2 \equiv b^2 - 4 \cdot a \cdot c \pmod{p}$$

assim, finalmente se chega ao resultado desejado:

$$(2 \cdot a \cdot x + b)^2 \equiv b^2 - 4 \cdot a \cdot c \pmod{p} \tag{3.7}$$

Pode se verificar que resolver a Equação 3.6 é equivalente a resolver 3.7. Rescrevendo com $X = 2 \cdot a \cdot x + b$ e $d = b^2 - 4 \cdot a \cdot c$, obtêm-se:

$$X^2 \equiv d \pmod{p} \tag{3.8}$$

Com isso, note que um problema mais complexo (Equação 3.6) foi transformado em um problema mais simples (Equação 3.8). Sobre esse último, se possui solução, isto é, d é um quadrado perfeito em $\mathbb{Z}/(p)$, então, se diz que d é um *resíduo quadrático* módulo p. Além disso, conforme (BROCHERO et al., 2013), existem precisamente $\frac{p+1}{2}$ resíduos quadráticos módulo p (valores de d menores que p para os quais 3.8 tem solução), que são neste caso:

$$0^2 \bmod p, 1^2 \bmod p, 2^2 \bmod p, 3^2 \bmod p, ..., \left(\frac{p-1}{2}\right)^2 \bmod p \tag{3.9}$$

O motivo desse fato é que para todo $x \in \mathbb{Z}$ existe algum i no intervalo $\left[0, \frac{p-1}{2}\right]$ tal que $x \equiv i \pmod{p}$ ou $x \equiv -i \pmod{p}$. Tal afirmação pode ser inferida facilmente, visto que tem-se todos os restos até $\frac{p-1}{2}$ com i, e para qualquer resto $r > \frac{p-1}{2}$ basta escolher $i = p - r$ (o que está obviamente dentro do intervalo de i), pois:

$$\begin{aligned}(p - r) \equiv i \pmod{p} &\implies -(p - r) \equiv -i \pmod{p} \\ &\implies r - p \equiv -i \pmod{p} \\ &\implies r \equiv -i \pmod{p}\end{aligned}$$

Logo, x^2 é congruente à um dos números da lista 3.9, pois dado $y = \pm i$, ou seja, $y \in \left[-\frac{p-1}{2}, \frac{p-1}{2}\right]$:

$$\begin{aligned} x \equiv y \pmod p &\implies x^2 \equiv y^2 \pmod p \\ &\implies x^2 \equiv i^2 \pmod p \end{aligned}$$

e i^2 está na Lista 3.9.

Outro fato interessante em relação a lista 3.9 é que todos este números são distintos em módulo p, haja vista, para quaisquer $i, j \in \left[0, \frac{p-1}{2}\right]$:

$$\begin{aligned} i^2 \equiv j^2 \pmod p &\iff p \mid (i^2 - j^2) && (3.10) \\ &\iff p \mid (i-j) \cdot (i+j) && (3.11) \\ &\iff p \mid (i-j) \vee p \mid (i+j) && (3.12) \end{aligned}$$

Dado que $i, j \in \left[0, \frac{p-1}{2}\right]$ então $0 \leq i + j \leq p - 1$, logo, existem as seguintes possibilidades:

1. $i = j = 0$ e portanto $i \equiv j \pmod p$.
2. $0 < i + j \leq p - 1$ (visto que $0 < i, j \leq \frac{p-1}{2}$) e portanto p não divide $i + j$ (pois essa soma resulta em um valor menor que p e maior que 0), e então pela disjunção em 3.12 resta apenas a possibilidade de $p \mid (i - j)$, o que equivale a $i \equiv j \pmod p$, ou seja, i é igual j módulo p se e somente se seus quadrados também são.

A partir destas conclusões expostas aqui é importante estabelecer o seguinte lema a ser utilizado futuramente:

Lema 4 *Seja $p > 2$ um número primo, existem exatamente $\frac{p+1}{2}$ resíduos quadráticos módulo p e $\frac{p-1}{2}$ resíduos não quadráticos módulo p.*

Demonstração: note que a seguinte lista contém todos os resíduos quadráticos módulo p

$$0^2 \bmod p, 1^2 \bmod p, 2^2 \bmod p, 3^2 \bmod p, ..., (p-1)^2 \bmod p$$

No entanto, essa lista contém valores repetidos, pois

$$\begin{aligned} (p-x)^2 \equiv (p-x)^2 \pmod p &\iff (p-x)^2 \equiv p^2 - 2 \cdot p \cdot x + x^2 \pmod p && (3.13) \\ &\iff (p-x)^2 \equiv x^2 \pmod p && (3.14) \end{aligned}$$

Assim, retirando os valores repetidos da lista (isto é, remover valores de modo que não hajam pares como em 3.14), tem-se:

$$0^2 \bmod p, 1^2 \bmod p, 2^2 \bmod p, 3^2 \bmod p, ..., \left(\frac{p-1}{2}\right)^2 \bmod p$$

Cada um desses valores é um número em $[1, p-1]$ (são restos), porém existem apenas $\frac{p+1}{2}$ desses valores, logo existem números no mesmo intervalo que não são resíduos quadráticos, e quantidade desses é $p - \frac{p+1}{2} = \frac{2 \cdot p - p - 1}{2} = \frac{p-1}{2}$. ∎

Apresentados estes conceitos sobre congruências quadráticas, dado um número primo $p > 2$ e $a \in \mathbb{Z}$, se define o *símbolo de Legendre* por:

$$\left(\frac{a}{p}\right) = \begin{cases} 1, \text{ se } p \nmid a \text{ e } a \text{ é um resíduo quadrático módulo } p \\ 0, \text{ se } p \mid a \\ -1, \text{ caso contrário } (a \text{ não é um resíduo quadrático}) \end{cases}$$

Essa definição, por si só, não traz qualquer utilidade, no entanto há o então chamado *Critério de Euler*, que apresenta uma maneira eficiente para computar o valor de um símbolo de Legendre. Esse critério afirma o seguinte:

Teorema 4 (*Critério de Euler*) *$\forall a \in \mathbb{Z}$, seja $p > 2$ um número primo, então:*

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

Logo, para se computar um símbolo de Legendre basta verificar se o resto da divisão inteira de $a^{\frac{p-1}{2}}$ por p é igual a 1, 0 ou -1 mod p.

Para se realizar a demonstração do *Critério de Euler*, antes é necessário apresentar o conceito de inverso multiplicativo módulo n e alguns lemas e teoremas envolvidos:

Definição 3 (*Inverso multiplicativo módulo* n) *Dados $a, m, n \in \mathbb{Z}$, se $a \cdot m \equiv 1 \pmod{n}$, se diz que m é um inverso de a módulo n, e pode ser denotado por a^{-1}.*

Lema 5 *Para todo $a, n \in \mathbb{Z}$, se $n > 0$, então, existe $b \in \mathbb{Z}$ tal que $a \cdot b \equiv 1 \pmod{n}$ se, e somente se, $\mathrm{mdc}(a, n) = 1$.*

Demonstração: note que

$$\begin{aligned} a \cdot b \equiv 1 \pmod{n} &\iff n \mid a \cdot b - 1 \\ &\iff \exists q \in \mathbb{Z}, n \cdot q = a \cdot b - 1 \\ &\iff \exists q \in \mathbb{Z}, a \cdot b - n \cdot 1 = 1 \end{aligned}$$

Por consequência do Teorema 1, só existe tal q se $\mathrm{mdc}(a, n) = 1$, e portanto b existe se e somente se isto ocorre.

Lema 6 (*Unicidade de inverso multiplicativo módulo* p) *Dado um número primo p, seja $a \in [1, p-1]$, existe $k \in [1, p-1]$ tal que $a \cdot k \equiv 1 \pmod{p}$ e k é portanto o único inverso multiplicativo de módulo p de a no intervalo $[1, p-1]$.*

Demonstração: primeiramente, sabe-se que k existe pelo Lema 5 (pois p é primo, logo $\text{mdc}(a, p) = 1$). Assim, dado que $a \cdot k \equiv 1 \pmod{p}$, suponha que existe $k' \in [1, p-1]$ tal que $a \cdot k' \equiv 1 \pmod{p}$, então:

$$\begin{aligned} a \cdot k \equiv a \cdot k' \pmod{p} &\iff p \mid a \cdot k - a \cdot k' \\ &\iff p \mid a \cdot (k - k') \\ &\iff p \mid a \cdot (k - k') \\ &\iff p \mid a \ \vee \ p \mid (k - k') \end{aligned}$$

Como $a \in [1, p-1]$, para que a disjunção seja válida deve ser o caso que $p|(k - k')$, e como $|k - k'| < p - 1$, a única maneira disto ocorrer é se $k - k' = 0$, ou seja, $k = k'$. ■

Lema 7 *Seja $a \in [1, p-1]$ em que p é um número primo maior que 2, se $x^2 \equiv a \pmod{p}$ não tem solução, então para todo $h \in [1, p-1]$ existe $k \in [1, p-1]$, tal que:*

$$h \neq k \wedge h \cdot k \equiv a \pmod{p}$$

Demonstração: pelo Lema 6, sabe-se que existe $h^{-1} \in [1, p-1]$ tal que:

$$h^{-1} \cdot h \equiv 1 \pmod{p}$$

e têm-se:

$$a \equiv a \pmod{p} \Rightarrow h^{-1} \cdot a \equiv h^{-1} \cdot a \pmod{p}$$

Neste momento é importante notar que independentemente de ser o caso de $h^{-1} \cdot a > p - 1$ ou não, existe algum $r \in [1, p-1]$ tal que:

$$r \equiv a \cdot h^{-1} \pmod{p}$$

Seguindo então, pode-se obter o seguinte:

$$a \cdot (h^{-1} \cdot h) \equiv a \pmod{p}$$

pois $h^{-1} \cdot h \equiv 1 \pmod{p}$. Manipulando essa equação, se chega em:

$$(a \cdot h^{-1}) \cdot h \equiv a \pmod{p} \iff r \cdot h \equiv a \pmod{p}$$

Como $r \in [1, p-1]$, resta apenas provar que $r \neq h$, o que é válido pela hipótese de que $x^2 \equiv a \pmod{p}$ não tem solução (se $r = h$ haveria solução e portanto se teria uma contradição). ■

Lema 8 *Seja $a, h, k, k' \in [1, p-1]$, se $k \cdot h \equiv a \pmod{p}$ e $k' \cdot h \equiv a \pmod{p}$ então $k = k'$ (k é único).*

Demonstração: observe que, seguindo da hipótese:

$$\begin{aligned} k \cdot h \equiv k' \cdot h \pmod p &\iff p \mid k \cdot h - k' \cdot h \\ &\iff p \mid h \cdot (k - k') \\ &\iff p \mid h \vee p \mid k - k' \end{aligned}$$

Como $p \nmid h$ (pois $|h| < p$) só pode ser o caso de que $p \mid k - k'$, porém $|k - k'| < p$, o que implica que $k - k' = 0$ e portanto $k = k'$. ■

Lema 9 *Seja $p > 2$ um número primo, para todo $a \in \mathbb{Z}$, se $\mathrm{mdc}(a, p) = 1$ e $x^2 \equiv a \pmod p$ não tem solução então:*

$$(p-1)! \equiv a^{\frac{p-1}{2}} \pmod p$$

Demonstração: pelos lemas 6, 7 e 8, pode-se escolher $\frac{p-1}{2}$ pares, utilizando todos os números no intervalo $[1, p-1]$, sem que qualquer número esteja em mais de um par (ou seja, se repita) e de modo que para cada par (x_i, y_i), $x_i \cdot y_i \equiv a \pmod p$, logo:

$$(x_1 \cdot y_1) \cdot (x_2 \cdot y_2) \cdot \ldots \cdot \left(x_{\frac{p-1}{2}} \cdot y_{\frac{p-1}{2}}\right) \equiv a^{\frac{p-1}{2}} \pmod p$$

Note que o lado esquerdo da equação é uma multiplicação entre todos os valores no intervalo $[1, p-1]$ (sem repetição), o que é igual a $(p-1)!$, portanto:

$$(p-1)! \equiv a^{\frac{p-1}{2}} \pmod p$$

■

Lema 10 *Seja p um número primo, então para quaisquer soluções de $x^2 \equiv 1 \pmod p$ têm-se que $x \equiv 1 \pmod p$ ou $x \equiv -1 \pmod p$. Portanto para qualquer outro valor y que não é uma solução, $y \not\equiv y^{-1} \pmod p$.*

Demonstração: se x é uma solução então

$$\begin{aligned} x^2 \equiv 1 \pmod p &\iff p \mid (x^2 - 1) \\ &\iff p \mid (x-1) \cdot (x+1) \\ &\iff p \mid (x-1) \vee p \mid (x+1) \\ &\iff x \equiv 1 \pmod p \vee x \equiv -1 \pmod p \end{aligned}$$

Portanto, para qualquer valor y tal que $y^2 \not\equiv 1 \pmod p$ tem-se que $y \not\equiv y^{-1} \pmod p$ (caso contrário y seria uma solução). ■

Teorema 5 (***Teorema de Wilson***) *Seja número composto um número que pode ser escrito como a multiplicação de dois outros números menores então, dado $n > 1$:*

$$(n-1)! \equiv \begin{cases} -1 \pmod{n} \text{ se } n \text{ é primo} \\ 0 \pmod{n} \text{ se } n \text{ é composto e } n \neq 4 \end{cases}$$

Demonstração: têm-se os seguintes casos:

1. Se n é composto mas não é quadrado de um número primo, pode-se escrever $n = a \cdot b$ em que $1 < a < b < n$, então a e b são fatores de $(n-1)!$, portanto $n \mid (n-1)!$, ou seja, $(n-1)! \equiv 0 \pmod{n}$.

2. Se $n = p^2$ onde p é um número primo maior que 2 então p e $2 \cdot p$ são fatores de $(n-1)!$, portanto, novamente $n \mid (n-1)!$, ou seja, $(n-1)! \equiv 0 \pmod{n}$.

3. Se n é primo, como $n - 1 \equiv -1 \pmod{n}$ partindo de $(n-2)! \equiv (n-2)! \pmod{p}$ se obtém que $(n-1)! \equiv -(n-2)! \pmod{n}$, e agora, observe que do lado direito da equação, pelos lemas 5, 7 e 10, pode-se manipular a expressão de modo a organizá-la em $\frac{n-3}{2}$ pares $(x \cdot y)$ onde $x, y \in [2, n-2]$ e $x \cdot y \equiv 1 \pmod{n}$, portanto $(n-2)! \equiv 1 \pmod{p}$ e então $-(n-2)! \equiv -1 \pmod{p}$, logo $(n-1)! \equiv -1 \pmod{n}$. ■

Agora será então apresentada a demonstração do *Critério de Euler*.
Demonstração: tem-se os seguintes casos

1. se $a \equiv 0 \pmod{p}$, ou seja, $p \mid a$, pelas propriedades de módulo pode-se elevar ambos os lados por $\frac{p-1}{2}$, e então se chega em $a^{\frac{p-1}{2}} \equiv 0 \pmod{p}$

2. se $p \nmid a$, então pelo Teorema 2 tem-se

$$\begin{aligned} a^{\varphi(p)} \equiv 1 \pmod{p} \iff a^{p-1} &\equiv 1 \pmod{p} \\ a^{\frac{p-1}{2}} \cdot a^{\frac{p-1}{2}} &\equiv 1 \pmod{p} \end{aligned}$$

subtraindo 1 de ambos os lados:

$$\begin{aligned} a^{\varphi(p)} \equiv 1 \pmod{p} &\iff a^{\frac{p-1}{2}} \cdot a^{\frac{p-1}{2}} - 1 \equiv 0 \pmod{p} \\ &\iff (a^{\frac{p-1}{2}} + 1) \cdot (a^{\frac{p-1}{2}} - 1) \equiv 0 \pmod{p} \\ &\iff p \mid (a^{\frac{p-1}{2}} + 1)(a^{\frac{p-1}{2}} - 1) \\ &\iff p \mid (a^{\frac{p-1}{2}} + 1) \ \vee \ p \mid (a^{\frac{p-1}{2}} - 1) \\ &\iff a^{\frac{p-1}{2}} \equiv -1 \pmod{p} \ \vee \ a^{\frac{p-1}{2}} \equiv 1 \pmod{p} \end{aligned}$$

Agora deve-se mostrar que o lado direito da disjunção é válido se e somente (bi-implicação) a é um resíduo quadrático módulo p. Para isso, provando a volta da bi-implicação, suponha que a é um resíduo quadrático, e portanto existe algum i tal que $a \equiv i^2 \pmod p$. Podemos elevar ambos os lados por $\frac{p-1}{2}$, donde se obtêm:

$$a^{\frac{p-1}{2}} \equiv i^{p-1} \pmod p$$

pelo Teorema 2 (e pela transitividade da relação de módulo), ocorre o seguinte:

$$i^{p-1} \equiv 1 \pmod p \implies a^{\frac{p-1}{2}} \equiv 1 \pmod p$$

Assim está provada a volta. Agora, para provar a ida:

$$a^{\frac{p-1}{2}} \equiv 1 \pmod p \implies \exists i, a \equiv i^2 \pmod p$$

por contraposição, é equivalente provar que:

$$\forall i, a \not\equiv i^2 \pmod p \implies a^{\frac{p-1}{2}} \not\equiv 1 \pmod p \tag{3.15}$$

Pela hipótese e pelo Lema 9 tem-se que $a^{\frac{p-1}{2}} \equiv (p-1)! \pmod p$. Usando o Teorema 5, por transitividade, $a^{\frac{p-1}{2}} \equiv -1 \pmod p$, então de fato $a^{\frac{p-1}{2}} \not\equiv 1 \pmod p$. ■

Quanto aos conceitos apresentados nessa sessão, a grande parte (se não todos) não estão implementados na biblioteca Mathematical Components. Sendo assim, apresentam um desafio considerável para que se complete o objetivo proposto neste trabalho, a ser apresentado com maiores detalhes no Capítulo 4.

4 Implementação

Neste capítulo será tratada a implementação do *símbolo de Legendre*, trazendo um conteúdo dividido em duas partes: discussão sobre implementações de implementações externas relacionadas a este trabalho (consideradas fundamentais para realização do objetivo proposto) e as implementações realizadas no presente trabalho.

Anteriormente vale ressaltar que existem implementações sobre *símbolo de Legendre* fora da biblioteca Mathematical Components. Como exemplo de tais implementações tem-se, utilizando o auxiliador de provas *Lean*, na biblioteca Mathlib[1], tanto a implementação de *símbolo de Legendre* quanto da *Lei de Reciprocidade Quadrática* (tema este discutido no Apêndice B). Há também a implementação de ambos os conteúdos mencionados em *Coq* (porém sem utilização da biblioteca Mathematical Components) públicada em repositório[2] de Nathanaëlle Courant disponível no GitHub.

4.1 Implementações Externas de Maior Relevância

Dentre os teoremas e lemas utilizados neste trabalho (dos quais a grande maioria são lemas simples da biblioteca Mathematical Components), existe um lema e um teorema que merecem destaque devido a complexidade para a prova dos mesmos. Este empecilho em suas respecitvas provas ocorre pela dificuldade em se fazer uma formalização com base nas ideias de prova manual para os mesmos (ambas apresentadas neste trabalho).

O primeiro lema a ser mencionado trata do caso 3 do Teorema 5 e tem na biblioteca o seguinte enunciado:

```
Theorem Wilson p : p > 1 → prime p = (p %| ((p.-1)`!).+1).
```

É importante notar, que, a igualdade entre as proposições booleanas `prime p` e `p %| ((p.-1)`!).+1` equivale a uma bi-implicação caso estas proposições não fossem booleanas. Além disso, vale aqui lembrar que, pela definição de equivalência em módulo p, há a seguinte bi-implicação:

$$p \mid (p-1)! + 1 \iff (p-1)! \equiv -1 \pmod{p} \tag{4.1}$$

Quanto ao segundo item a ser mencionado, este trata do Lema 9 e foi provado por Laurent Théry durante o período de realiazação deste trabalho. A prova deste está disponível em: https://github.com/thery/mathcomp-extra/blob/master/euler.v e tem a seguinte declaração:

[1] Disponível em: https://github.com/leanprover-community/mathlib4
[2] Disponível em: https://github.com/Ekdohibs/coq-proofs/tree/master/Reciprocity

```
Lemma fact_sqr_exp a p :
        prime p → ~~ (p %| a) → ~~ res_quad p a → (p.−1`!) = a ^ p.−1./2 %[mod p].
```

onde `res_quad` tem a seguinte definição:

```
Definition res_quad p a := has (fun i ⇒ i ∗ i == a %[mod p]) (iota 0 p).
```

em que, a função `has` recebe um predicado booleano e uma lista, e então retorna `true` se há algum elemento da lista que satisfaz o predicado e `false` caso contrário. Já a função `iota` recebe dois números naturais m e n e retorna uma lista crescente de todos os naturais de m até $n-1+m$. Sendo assim, a função `res_quad` é um método exaustivo para verificar se há um valor r tal que $r^2 \equiv a \pmod{p}$.

As dificuldades relacionadas as provas destas declarações em *Coq*, se baseando nas ideias de provais manuais apresentadas neste trabalho, deve-se ao fato de que tais ideias sugerem a reorganização de todos os termos de produtórios com tamanho arbitrário. Tais provas além de dificeís não podem ser evitadas ao se buscar a implementação de *símbolo de Legendre* em *Coq* dado que tratam de conteúdos diretamente relacionados (principalmente ao se considerar o Teorema 4).

Devido a essa dificuldade, que pode vir a aparecer em diversas situações onde tem-se uma ideia de prova semelhante a ser formalizada, é de interesse se ter conhecimento de como as provas do Teorema 5 e do Lema 9 foram feitas em *Coq*. Para satisfazer tal necessidade será aqui explicada em detalhe (com excessão da maioria dos *subgoals* e outras partes triviais) o conteúdo da prova do Lema 9 realizada por Laurent Théry. O código da prova pode-ser divido nas seguintes etapas:

i. Inicialmente tem-se o seguinte *goal*:

```
prime p → ~~ (p %| a) → ~~ res_quad p a → (p.−1`!) = a ^ p.−1./2 %[mod p]
```

Introduz-se então todas as hipóteses e se reescreve $(p-1)!$:

```
1 move⇒ pP pNDa aR.
2 have → : p.−1`! = \prod_(i in 'F_p | i != 0%R) i.
3        (* ... prova do subgoal... *)
```

Explicando os comandos feitos em linhas específicas têm-se:

(1) Introdução dos precedentes do *goal* para hipóteses com os nomes `pP`, `pNDa` e `aR` respectivamente.

(2) Reescrita de $(p-1)!$ como $\prod_{i\in(\mathbb{F}_p-\{0\})} i$ (o que por sua vez abre um *subgoal*, que é por sua vez trivial).

ii. Tendo agora o seguinte *goal*:

```
\prod_(i in 'F_p | i != 0%R) i = a ^ p.-1./2 %[mod p]
```

faz-se então uma série de declarações de variáveis e funções junto à introdução de novas hipóteses:

```
4  pose a' : 'F_p := inZp a.
5  have a'E : a' = a %% p :> nat
6          by rewrite /= Fp_cast.
7  have a'_neq0 : a' != 0%R.
8          (*... prova do subgoal...*)
9  rewrite -modnXm -a'E.
10 pose f (i : 'F_p) : 'F_p := (a' / i)%R.
11 have f_eq0 : f 0%R = 0%R by rewrite /f GRing.invr0 GRing.mulr0.
12 have fM (i : 'F_p) : i != ord0 -> (f i * i = a')%R.
13         (*... prova do subgoal...*)
14 have fI (i : 'F_p) : f (f i) = i.
15         (*... prova do subgoal...*)
16 have fI_neq0 (i : 'F_p) : i != 0%R -> f i != i.
17         (*... prova do subgoal...*)
18 have fB : {on [pred i |  i != ord0],  bijective f}.
19         (*... prova do subgoal...*)
20 pose can (i : 'F_p) :=  if i < (f i) then i else f i.
```

Explicando os comandos feitos em linhas específicas têm-se:

(4) Declaração da variável $a' \in \mathbb{F}_p$ que por sua vez depende do valor de a.

(5) Introdução da hipótese `a'E` de que $a' = a \bmod p$.

(7) Introdução da hipótese `a'_neq0` de que $a' \neq 0$.

(9) Reescrita (no *goal*) de $a^{\frac{p-1}{2}}$ como $(a')^{\frac{p-1}{2}}$.

(10) Declaração de uma função $f : \mathbb{F}_p \to \mathbb{F}_p$ tal que $f(i) = (a') \cdot i^{-1}$ (onde i^{-1} é o inverso de i em $\mathbb{F}_p$ e a notação `"x / y"` é definida como `x * y^-1`).

(11) Introducão (e prova na mesma linha a partir do comando `by`) da hipótese `f_eq0` de que $f(0) = 0$.

(12) Introducão da hipótese `fM` de que $\forall i \in \mathbb{F}_p, (i \neq 0 \to f(i) \cdot i = a')$;

(14) Introducão da hipótese `fI` de que $\forall i \in \mathbb{F}_p, f(f(i)) = i$, isto é, f é involutiva.

(16) Introducão da hipótese `fI_neq0` de que $\forall i \in \mathbb{F}_p, (i \neq 0 \to f(i) \neq i)$.

(18) Introducão da hipótese `fB` de existe uma função $f : (\mathbb{F}_p - \{0\}) \to (\mathbb{F}_p - \{0\})$ que é bijetora (o que é óbvio já que a função f declarada anteriormente é involutiva).

(20) Declaração de uma função $can : \mathbb{F}_p \to \mathbb{F}_p$ tal que $can(i)$ retorna o valor mínimo entre i e $f(i)$.

iii. Tendo agora o seguinte *goal*:

```
\prod_(i in 'F_p | i != 0%R) i = a' ^ p.-1./2 %[mod p]
```

faz-se a seguinte reescrita:

```
21 have -> : \prod_(i in 'F_p | i != 0%R) i =
22              \prod_(j in 'F_p | (j < f j))
23                     \prod_(i in 'F_p | (i != 0%R) && (can i == j)) i.
24      (*... prova do subgoal...*)
```

Explicando os comandos feitos em linhas específicas têm-se:

(21) Substituição (no *goal*) de $\prod_{i \in (\mathbb{F}_p - \{0\})} i$ por:

$$\prod_{j \in \mathbb{F}_p | j < f(j)} \left(\prod_{i \in (\mathbb{F}_p - \{0\}) | can(i) = j} i \right)$$

(24) Código da prova do *subgoal* gerado na linha 21 - este código não será apresentado aqui, no entanto, pode-se dizer que tal subprova se baseia na utilização do lema `partition_big`, que tem a seguinte declaração:

```
Lemma partition_big {R : Type} {idx : R} {op : Monoid.com_law idx}
{I : Type} {s : seq I} {J : finType} {P : pred I} (p : I -> J) (Q : pred J)
{F : I -> R} :
(∀ i : I, P i -> Q (p i)) ->
       \big[op/idx]_(i <- s | P i) F i =
       \big[op/idx]_(j | Q j) \big[op/idx]_(i <- s | P i && (p i == j)) F i
```

onde `\big[op/idx]` é parte comum de uma série de notações para uso da função `bigop`, que, conforme apresentado em (MAHBOUBI; TASSI, 2022), pode ser definida como:

```
Definition bigop R I idx op r (P : pred I) (F : I -> R) : R :=
       foldr (fun i x => if P i then op (F i) x else x) idx r.
```

e dentre as notações para o uso desta função tem-se:

```
Notation "\big [ op / idx ]_ ( i <- r | P ) F" :=
       (bigop idx op r (fun i => P%B) (fun i => F)) : big_scope.
```

Assim, pode-se perceber que a notação de um produtório é um açucar sintático para a uso do operador `bigop` com o argumento `op` sendo a operação de multiplicação e o argumento `idx` sendo 1.

Nesta subprova usa-se então o lema `partition_big` com `P i` sendo a condição (`i` != 0), `p` sendo a função `can` e `Q j` sendo a condição (`j` < `f(j)`). Após o uso de tal lema é então necessário provar mais um subgoal:

```
forall i :
 fintype_ordinal__canonical__fintype_Finite (Zp_trunc (pdiv p)).+2,
        (i  \in 'F_p) && (i != 0%R) -> (can i \in 'F_p) && (can i < f (can i))
```

onde a expressão:

```
fintype_ordinal__canonical__fintype_Finite (Zp_trunc (pdiv p)).+2
```

equivale à `'F_p`, pois basta realizar uma computação sobre a expressão para que ela se torne `'F_p`, o que pode ser feito utilizando nesse caso a tática (utilizando casamento de padrões):

```
rewrite [X in forall i : X, _]/=.
```

iv. Após a resolução dos *subgoals* gerados anteriormente, o *goal* atual se torna o seguinte:

```
\prod_(j in 'F_p | j < f j)
       \prod_(i in 'F_p | (i != 0%R) && (can i == j)) i
                                    = a' ^ p.-1./2  %[mod p]
```

Usa-se então a seguinte tática:

```
25 apply: etrans (_ : \prod_(j in 'F_p | j < f j) (j * f j) = _ %[mod p]).
26        (*... prova do goal 1...*)
                    ⋮
35        (*... prova do goal 2...*)
                    ⋮
```

Esta tática então aplicada na linha 25 utiliza da transitividade da igualdade por meio do lema `etrans` junto a um argumento que permite que o *Coq* infira o termo médio desta transitividade. Este argumento é uma prova de igualdade, qual por não existir nas hipóteses irá gerar 2 *goals* no lugar do *goal* atual:

iv.(a) Primeiro *goal*:

```
\prod_(j in 'F_p | j < f j) \prod_(i in 'F_p | (i != 0%R) && (can i == j)) i
      = \prod_(j in 'F_p | j < f j) (j * f j)  %[mod p]
```

Para este *goal* são usadas as seguintes táticas:

```
26 congr (_ %% _).
27 apply: eq_bigr => j /andP[jF jLfj].
```

```
28 rewrite (bigD1 j); last first.
29        (*... prova do subgoal...*)
30 rewrite [LHS]/=.
31 rewrite (bigD1 (f j)); last first.
32        (*... prova do subgoal...*)
33 rewrite big1 /= ?muln1 // => i.
34        (*... prova do subgoal...*)
```

Explicando os comandos feitos em linhas específicas têm-se:

(26) Retirada da aplicação de mod p em ambos os lados da equação.

(27) Aplicação do lema `eq_bigr` junto à eliminação do ∀ e introdução da hipótese gerada por tal aplicação; é interessante ao leitor não acostumado com as táticas da linguagem *SSReflect* notar que, sabendo que o lema `eq_bigr` tem o seguinte enunciado:

```
Lemma eq_bigr r (P : pred I) F1 F2 : (forall i, P i -> F1 i = F2 i) ->
       \big[op/idx]_(i <- r | P i) F1 i = \big[op/idx]_(i <- r | P i) F2 i.
```

após a aplicação do lema e eliminação do ∀, o *goal* se torna:

```
(j  \in 'F_p) && (j < f j) ->
       \prod_(i in 'F_p | (i != 0%R) && (can i == j)) i = j * f j
```

mas nesse caso também é feita a introdução do precedente com `/andP[jF jLfj]`, que torna, em tal hipótese, a expressão com operador && (conjunção booleana) em uma expressão equivalente com ∧ (conjunção proposicional) que é então separada nas hipóteses `jF` e `jLfj`.

Pode-se notar portanto que a partir da aplicação do lema `eq_bigr` o *goal* consiste em provar que o termo geral dos produtórios é igual.

(28) Reescrita por meio do lema `bigD1` com o *goal* sendo (devido as alterações feitas nas linhas anteriores):

```
\prod_(i in 'F_p | (i != 0%R) && (can i == j)) i = j * f j
```

Este lema basicamente retira um elemento de dentro da aplicação do operador `bigop` (nesse caso o elemento `j` pois foi este o argumento passado ao lema), no entanto deve-se demonstrar que tal elemento realmente ocorre em alguma das iterações, por isto é então gerado um *subgoal*. Esse *subgoal* normalmente deveria ser resolvido após a resolução do *goal* principal, mas devido ao uso de `last first` tal *subgoal* deve ser resolvido antes do *goal* principal.

(30) Comando adicionado apenas para tornar o *goal* mais legível (não é algo que estava na prova original feita por Laurent), e nesse caso, após a execução dos comandos da linha 3 e deste comando, o *goal* se torna:

```
j * \prod_(i < (Zp_trunc (pdiv p)).+2 | (i != 0%R) && (can i == j) && (
    i != j)) i
        = j * f j
```

em que (`Zp_trunc (pdiv p)).+2` é igual a `p`, o que é provado pelo lema `Fp_cast`.

(31) Reescrita análoga a feita na linha 3, porém retirando o elemento `f j` do produtório, de modo que o *goal* se torne então:

```
j * ssrnat_muln__canonical__SemiGroup_ComLaw (f j)
        (\big[ssrnat_muln__canonical__SemiGroup_ComLaw/1]_(i | (i !=
            0%R)
                && (can i == j) && (i != j) && (i != f j)) i)
                        = j * f j
```

onde `ssrnat_muln__canonical__SemiGroup_ComLaw` é a operação de multiplicação entre naturais.

(33) Reescrita do produtório ainda presente no *goal* como 1 (elemento neutro) por meio do lema `big1`, que por sua vez tem o seguinte enunciado:

```
Lemma big1 {R : Type} {idx : R} {op : Monoid.law idx} I r (P : pred I) F :
        (forall i : I, P i -> F i = idx) ->
                \big[op/idx]_(i <- r | P i) F i = idx.
```

seguindo de computação (dada por /=), reescrita da multiplicação por 1 (omissão), resolução do goal por meio de // (semelhante ao comando `simplify`) e eliminação do ∀ (com "=> i") do *subgoal* aberto pelo uso do lema `big1`.

Este *subgoal* aberto consiste em provar que: o termo geral é sempre igual a 1 ou que as condições do produtório nunca serão atendidas (que é o caso desta prova) e portanto o produtório retorna apenas 1.

Após a prova deste *subgoal* resta então apenas o segundo *goal* gerado na aplicação da tática `etrans`.

iv.(b) Segundo *goal*:

```
\prod_(j in 'F_p | j < f j) (j * f j)   = a' ^ p.-1./2  %[mod p]
```

Para este *goal* é inicialmente utilizada a aplicação do lema `etrans` novamente:

```
apply: etrans (_ : \prod_(j in 'F_p | j < f j) a' = _ %[mod p]).
        (*... prova do goal 1...*)
                ⋮
        (*... prova do goal 2...*)
                ⋮
```

e assim, como feito em item anterior, têm-se agora dois *goals* a serem resolvidos:

(b.1) Primeiro *goal*:

```
\prod_(j in 'F_p | j < f j) (j * f j)
        = \prod_(j in 'F_p | j < f j) a'  %[mod p]
```

Para este *goal* são inicialmente aplicadas as seguintes táticas:

```
36 rewrite -modn_prodm.
37 congr (_ %% _).
38 apply: eq_bigr => i /andP[_ iLfi].
39        (*... restante da prova do goal atual...*)
40        (*... prova do goal seguinte...*)
```

Explicando os comandos feitos em linhas específicas têm-se:

(36) Reescrita do termo geral do produtório do lado esquerdo como ((`j * f j`) `%% p`).

(37) Remoção da aplicação de mod p em ambos os lados da equação.

(38) aplicação do lema `eq_bigr`, que por sua vez tem o seguinte enunciado:

```
Lemma eq_bigr r (P : pred I) F1 F2 : (∀ i, P i -> F1 i = F2 i) ->
        \big[op/idx]_(i <- r | P i) F1 i = \big[op/idx]_(i <- r | P i)
            F2 i.
```

junto a eliminação do ∀ e introducão das hipóteses do *goal* alterado após a aplicação de `eq_bigr`. Com isto o *goal* se torna:

```
(i * f i) %% p = a'
```

o que é trivial considerando a hipótese `fM` introduzida anteriormente.

(b.2) Segundo *goal*:

```
\prod_(j in 'F_p | j < f j) a'  = a' ^ p.-1./2  %[mod p]
```

Para este *goal* são inicialmente utilizadas as seguintes táticas:

```
40 congr (_ %% _).
41 rewrite prod_nat_const.
42 rewrite [X in fun i : X => _]/=.
43 congr (_ ^ _).
44 rewrite -[p in RHS](card_Fp pP).
45 rewrite [in RHS](cardD1 0%R).
46 rewrite inE add1n -pred_Sn.
47 set A := [predD1 'F_p & 0%R].
48 pose B := [pred i |  (i : 'F_p) < f i].
49 rewrite -(cardID B A).
```

```
50 have <- : #|image f [predI A & B]| = #|[predD A & B]|.
         (*... prova do subgoal...*)
63 rewrite card_image; last by move=> i j fiEfj; rewrite -[i]fI fiEfj fI.
64 rewrite addnn (half_bit_double _ false).
65 apply: eq_card => i.
66 rewrite !inE.
```

Explicando os comandos feitos em linhas específicas têm-se:

(40) Remoção da aplicação de mod p em ambos os lados da equação.

(41) Reescrita utilizando `prod_nat_const`, lema este que possui o seguinte enunciado:

```
Lemma prod_nat_const n : \prod_(i in A) n = n ^ #|A|.
```

em que #|`A`| denota a cardinalidade de um conjunto `A`. Assim o lado direito do goal se torna:

```
a' ^ #|(fun i : fintype_ordinal__canonical__fintype_Finite
        (Zp_trunc (pdiv p)).+2 => eqn (i.+1 - f i) 0)|
```

onde, como já explicado anteriormente, a expressão:

```
fintype_ordinal__canonical__fintype_Finite
        (Zp_trunc (pdiv p)).+2
```

é equivalente à `'F_p`, e portanto ao se executar uma computação sobre tal expressão ela resulta em `'F_p`. Esta computação é então realizada na linha 42 (qual não estava no código de Laurent), e assim, tem-se no *goal*:

```
a' ^ #|(fun i : 'F_p => eqn (i.+1 - f i) 0)| = a' ^ p.-1./2
```

em que a expressão:

```
eqn (i.+1 - f i) 0
```

é a operação < (a notação está "unfolded"). Pode-se notar aqui que o conjunto não é definido por uma lista, mas sim por um predicado booleano, cujo domínio é um tipo finito (`'F_p` neste caso).

(43) Tranformação do *goal* em uma igualdade entre os expoentes, dado que em cada lado da equação do *goal* haviam exponenciações de mesma base. Com isto o *goal* se torna:

```
#|(fun i : 'F_p => eqn (i.+1 - f i) 0)| = p.-1./2
```

(44) Reescrita de `p` como #|`'F_p`| no lado direito da equação.

(45) Reescrita de #|'F_p| como (0%R \in 'F_p) + #|[predD1 'F_p & 0%R]| em que [predD1 'F_p & 0%R] é uma notação para o seguinte:

```
[fun x => (x != 0%R) && (x \in 'F_p)]
```

ou seja, [predD1 'F_p & 0%R] é o conjunto $\mathbb{F}_p - \{0\}$.

(46) Reescrita de (0%R \in 'F_p) como `true`, por meio da tupla `inE` (`inE` é uma tupla de lemas e portanto quando usada para reescrita o *Coq* tenta realizar a reescrita usando cada um dos membros da tupla). O resultado desta reescrita é então convertido para 1 (devido a *coercion* de `bool` para `nat`); reescrita da soma de 1 à esquerda na forma da notação _.+1 (pelo lema `add1n`); cancelamento da adição de 1 com a subtração de 1 (pelo lema `pred_Sn`, onde o uso de − antes do nome do lema indica a direção contrária a usual da rescrita, isto é, do padrão do lado direito para o padrão do lado esquerdo). Com estas reescritas o *goal* se torna:

```
#|(fun i : 'F_p => eqn (i.+1 - f i) 0)| = #|[predD1 'F_p & 0%R]|./2
```

(47) Introducão do *alias* `A` para o conjunto [predD1 'F_p & 0%R].

(48) Introducão do *alias* `B` para o conjunto [pred i | (i : 'F_p) < f i] (a diferença do uso de `set` e `pose` é que o primeiro realiza substituição da expressão pelo *alias* no *goal*).

(49) Reescrita da cardinalidade de `A` como a cardinalidade da intersecção entre `A` e `B` somada a cardinalidade de `A` menos `B` (por meio do lema `cardID`); em termos das notações mais comuns usadas em Teoria dos Conjunto faz-se a substituição de $|A|$ por $|A \bigcap B| + |A - B|$.

(50) Substituição de #|[predD A & B]| por #|image f [predI A & B]| (que é printada como #|[seq f x | x in [predI A & B]]) em que está ultima expressão indica a imagem de `f` tendo #|[predD A & B]| como domínio.

Tal substituição abre um *subgoal* que é resolvido em seguida, conforme indicado no comentário da linha 12 e com isso o *goal* se torna:

```
#|(fun i : 'F_p => eqn (i.+1 - f i) 0)| =
        (#|[predI A & B]| + #|[seq f x | x in [predI A & B]]|)./2
```

Por não ser trivial, é útil se analisar a prova do *subgoal* mencionado. Tal prova consiste no uso das seguintes táticas:

```
50 apply: eq_card => i.
51 rewrite !inE.
52 rewrite -[in LHS](fI i).
53 rewrite mem_map; last first.
54     by move=> i1 j1 fiEfj; rewrite -[i1]fI fiEfj fI.
55 rewrite mem_enum.
```

```
56 rewrite !inE fI !andbT.
57 case: (i =P 0%R) => [-> |];
58         first by rewrite f_eq0.
59 case: (f i =P 0%R) => [fi0|/eqP fi_neq0 /eqP i_neq0].
60         by case; rewrite -(fI i) fi0 f_eq0.
61 case: ltngtP => // /eqP/val_eqP fiEi.
62         by have := fI_neq0 i i_neq0; rewrite fiEi eqxx.
```

Explicando determinadas linhas (com o item de cada uma destas linhas) da subprova têm-se:

(50) Transformação do *goal* de uma igualdade entre cardinalidade em uma igualdade de conjunto, isto é, em:

```
(i \in [seq f x | x in [predI A & B]]) = (i \in [predD A & B])
```

(51) Simplificação de expressões como `x \in 'F_p` (que aparecem na expressão "unfolded") em `true`. Com isso o *goal* se torna:

```
(i \in [seq f x | x in [predI A & B]])
        = [&& ~~ (i < f i), i != 0%R & true]
```

(52) Reescrita de `i` como `f (f i)` no lado esquerdo do *goal*.

(53) Reescrita de:

```
(f (f i) \in [seq f x | x in [predI A & B]])
```

como:

```
((f i) \in enum [predI A & B])
```

o que é possível pois `f` é injetora (*subgoal* aberto por este `rewrite` e provado na linha 54). Com isto o *goal* se torna:

```
(f i \in enum [predI A & B]) = [&& ~~ (i < f i), i != 0%R & true]
```

onde `enum` é uma função que retorna os elementos do tipo finito que atende a proposição booleana passada como argumento (o tipo finito é inferido pelo tipo do argumento de tais proposições), conforme consta na documentação da biblioteca[3].

(55) Reescrita de `(f i \in enum [predI A & B])` como `(f i \in [predI A & B])`.

(56) Reescritas atráves da tupla `inE`, reescrita de `f (f i)` com `i` e por fim reescritas de conjunções booleanas com um dos elementos sendo `true`. Assim obtêm-se o seguinte *goal*:

[3] <https://math-comp.github.io/htmldoc_2_2_0/mathcomp.ssreflect.fintype.html>

```
(f i != 0%R) && (f i < i) = ~~ (i < f i) && (i != 0%R)
```

O *goal* é agora considerávelmente trivial e pode ser resolvido por análise de caso, o que é feito nas linhas seguintes.

(57) Análise dos dois seguintes casos: `i = 0%R` e `i <> 0%R`, feito por meio do comando `case` sobre a expressão `i =P 0%R`. É resolvido o primeiro caso e então `i <> 0%R` é uma hipótese no restante da resolução.

(59) Análise dos dois seguintes casos análoga a linha anterior porém sobre o termo comando `f i` ao invés de comando `i`.

(61) Análise dos casos: `i < f i`, `i > f i` e por último `i = f i`. Os dois primeiros casos são resolvidos com a simplificação `//` na mesma linha do `case`, por isso se introduz a hipótese `fiEi` apenas. Esta hipótese ao ser introduzida é reescrita por meio de `/eqP` e `/val_eqP`, o que retira a *coercion* para `nat` aplicada sobre `f i` e sobre `i`. A partir disso o *goal* é resolvido na linha 62 obtêndo a partir da hipótese `fI_neq0` (introduzida no perto do começo do código) e `i_neq0` (de que `i != 0`, introduzida na linha 59) para então se obter uma contradição junto a hipótese `fiEi`. Com isto a prova do *subgoal* se encerra.

(63) Reescrita (e resolução do *subgoal* aberto por tal tatíca) de:

```
#|[seq f x  | x in [predI A & B]]|
```

como `#|[predI A & B]|`. Em termos de notações comumente usadas em Teoria dos Conjuntos, substitui-se $|Im(f, A \bigcap B)|$ por $|A \bigcap B|$ (o que é possível por f é injetora). Com isso o *goal* se torna:

```
#|(fun i : 'F_p => eqn (i.+1 - f i) 0)| =
        (#|[predI A & B]| + #|[predI A & B]|)./2
```

(64) Reescrita da expressão:

```
(#|[predI A & B]| + #|[predI A & B]|)./2
```

como `|[predI A & B]|`.

(65) Reescrita da igualdade entre 2 conjuntos como uma igualdade entre expressões booleanas de pertencimento de um elemento (instânciado após a eliminação do ∀ na mesma linha) a cada conjunto. Assim o *goal* se torna:

```
(i \in (fun i0 : 'F_p => eqn (i0.+1 - f i0) 0)) = (i \in [predI A & B])
```

(66) Ambas as expressões da igualdade podem ser simplificadas com o uso da tupla `inE` diversas vezes, por isso para realizar o máximo de reescritas possíveis é utilizado "!" antes do nome da tupla. Obtêm-se então a seguinte expressão no *goal*:

```
(i < f i) = (i != 0%R) && true && (i < f i)
```

que é por sua vez trivial considerando as hipóteses `f_eq0` e `fI_neq0` introduzidas no início. Após isso a prova se encerra.

4.2 Formalização do Símbolo de Legendre

Seguindo a ideia utilizada (e recomendada) por Laurent Théry, em se utilizar uma função exaustiva para verificar se um determinado número é um resíduo quadrático módulo p (em que p é um número primo), e portanto, ter-se uma função que é pelo menos computável (apesar de sua ineficácia), fez-se neste trabalho a seguinte definição para o *símbolo de Legendre*:

```
Definition legendre_symb {p : int} (pL2 : (2 < p)%R) (pP : primez.primez p)
        (a : int) :=
        if (p %| a)%Z then 0%Z
        else if has (fun i => ((i * i)%:Z == a %[mod p])%Z) (iota 0 `|p|)
        then 1%Z
        else (-1)%Z.
```

Com esta definição, o *símbolo de Legendre* é portanto uma função que recebe um número inteiro p, uma prova de que $2 < p$, uma prova de que p é um número primo e por último um número inteiro a. Note que os argumentos que constituem provas podem ser gerados automaticamente pelo usuário dado que as funções $<$ e `primez` são computáveis. Como exemplo de uso da função tem-se portanto:

```
Compute (legendre_symb (_ : 2 < 7)%R (_ : primez.primez 7) 2).
```

que neste caso irá retornar 1.

Havendo a definição de *símbolo de Legendre* resta então provar as suas principais propriedades, que podem vir a ser úteis em trabalhos futuros. Se baseando em (BROCHERO et al., 2013), as principais propriedades do *símbolo de Legendre*, considerando um número primo $p > 2$, são, para todo $a, b \in \mathbb{Z}$:

1. $a \equiv b \pmod{p} \rightarrow \left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$, cuja declaração dada em *Coq* é:

```
Lemma legendre_symbE (p a b : int) (pL2 : (2 < p)%R)
(pP : primez.primez p):
(a == b %[mod p])%Z ->
```

```
((legendre_symb pL2 pP a) == (legendre_symb pL2 pP b)).
```

2. $p \nmid a \to \left(\frac{a^2}{p}\right)$, cuja declaração dada em *Coq* é:

```
Lemma legendre_symb_Ndvd (p a b : int) (pL2 : (2 < p)%R)
(pP : primez.primez p):
~~(p %| a)%Z → (legendre_symb pL2 pP (a^2)) == 1.
```

3. $\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}} = 1 \leftrightarrow p \equiv 1 \pmod p$, cuja declaração dada em *Coq* é:

```
Lemma legendre_symb_Neg1 (p : int) (pL2 : (2 < p)%R)
(pP : primez.primez p):
((legendre_symb pL2 pP (−1)) == 1) = (p == 1 %[mod 4])%Z.
```

4. $\left(\frac{a \cdot b}{p}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right)$, cuja declaração dada em *Coq* é:

```
Lemma legendre_symb_Neg1 (p : int) (pL2 : (2 < p)%R)
(pP : primez.primez p):
((legendre_symb pL2 pP (−1)) == 1) = (p == 1 %[mod 4])%Z.
```

A formalização (isto é, prova) de todos estes lemas está disponível no seguinte repositório.

5 Conclusão

O *símbolo de Legendre* junto aos conteúdos relacionados e necessários para sua formalização são elementos que possuem aplicações relevantes, pois são o caminho para o desenvolvimente de trabalhos sobre temas como o algoritmo RESSOL e a *Lei de Reciprocidade Quadrática*. A exemplo estes dois conteúdos estão relacionados a aplicações como curvas elípticas que são amplamente utilizadas em criptografia. Além disso, esses não são conteúdos isolados, no sentido de que são parte do caminho para outros.

No presente trabalho foram implementados diversos teoremas

Apesar desses fatos, esta área especifica de Teoria dos Números não chegou (até o momento) a ser explorada na biblioteca Mathematical Components, o que é de se esperar que aconteça com diversos temas, dado que abranger toda a matemática já desenvolvida é algo difícil, se não impossível. Tal vácuo de conteúdo com relação à área citada é um problema que, após esse trabalho, se torna muito mais viável de ser resolvido em trabalhos futuros, tendo o conhecimento disponibilizado neste trabalho como base. Esta viabilidade muito se dá à possibilidade de que iniciantes no uso da biblioteca Mathematical Components tenham uma curva de aprendizado mais suave, dado que muitos detalhes muitas vezes não discutidos na documentação da biblioteca são, neste trabalho, discutidos.

Com relação especificimente ao tema sobre a *Lei de Reciprocidade Quadrática*, uma implentação deste tema voltada para a bibloteca Mathematical Components ganha com este trabalho uma maior possibilidade de ser realizada. Isto ocorre pois neste trabalho é dada a implentação do *símbolo de Legendre* em *Coq* e a explicação sobre uma prova (na Seção 4.1) que envolve (principalmente no que consta a Teoria de Conjuntos) elementos úteis (da biblioteca Mathematical Components) para a prova da *Lei de Reciprocidade Quadrática* (haja vista o que se tem na prova manual apresentada em B).

Referências

APPEL, K.; HAKEN, W. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, American Mathematical Society, v. 82, n. 5, p. 711–712, 1976. Citado na página 27.

BROCHERO, F. E. et al. *Teoria dos Números: um passeio com primos e outros números familiares pelo mundo inteiro*. 3. ed. Rio de Janeiro : IMPA: IMPA, 2013. (Projeto Euclides). ISBN 978-85-2444-0312-5. Citado 13 vezes nas páginas 25, 26, 45, 48, 49, 50, 51, 53, 54, 56, 57, 77 e 91.

COOK, J. D. *Quadratic reciprocity algorithm*. 2023. Disponível em: <https://www.johndcook.com/blog/2023/01/01/quadratic-reciprocity-algorithm/>. Acesso em: 06 de jun. de 2024. Citado na página 91.

GONTHIER, G. *A computer-checked proof of the Four Color Theorem*. [S.l.], 2023. Disponível em: <https://inria.hal.science/hal-04034866/file/FINALA%20computer-checked%20proof%20of%20the%20four%20color%20theorem%20-%20HAL.pdf>. Citado na página 27.

HUYNH, E. *Rabin's Cryptosystem*. 39 p. Monografia (Bachelor) — Linnaeus University, Department of Mathematics, Suécia, 2021. Citado 2 vezes nas páginas 26 e 85.

KUMAR, R. *An algorithm for finding square root modulo p*. 2020. Disponível em: <https://doi.org/10.48550/arXiv.2008.11814>. Acesso em: 15 de maio de 2024. Citado na página 26.

LI, Z.; DONG, X.; CAO, Z. Generalized cipolla-lehmer root computation in finite fields. In: ICINS 2014 - 2014 INTERNATIONAL CONFERENCE ON INFORMATION AND NETWORK SECURITY, CP657. *ICINS 2014 - 2014 International Conference on Information and Network Security*. Pequim, China, 2014. p. 163–168. Citado na página 26.

MAHBOUBI, A.; TASSI, E. Canonical structures for the working coq user. In: BLAZY, S.; PAULIN-MOHRING, C.; PICHARDIE, D. (Ed.). *Interactive Theorem Proving*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 19–34. ISBN 978-3-642-39634-2. Citado 2 vezes nas páginas 35 e 37.

MAHBOUBI, A.; TASSI, E. *Mathematical Components*. Zenodo, 2022. Disponível em: <https://doi.org/10.5281/zenodo.7118596>. Citado 8 vezes nas páginas 27, 31, 32, 33, 34, 41, 52 e 68.

MAHESWARI, A. U.; DURAIRAJ, P. An algorithm to find square roots of quadratic residues modulo p (p being an odd prime), $p \equiv 1 \pmod 4$. *Global Journal of Pure and Applied Mathematics*, v. 13, n. 4, p. 1223–1239, 2017. Citado na página 26.

NEEMAN, A. A counterexample to a 1961 "theorem" in homological algebra. *Inventiones Mathematicae*, v. 148, n. 2, p. 397–420, maio 2002. Disponível em: <http://dx.doi.org/10.1007/s002220100197>. Acesso em: 15 de jun. de 2024. Citado na página 27.

NIVEN, I.; ZUCKERMAN, H. S. *An introduction to the theory of numbers*. Estados Unidos da América: John Wiley & Sons, Inc, 1991. Citado na página 26.

PAULIN-MOHRING, C. Introduction to the calculus of inductive constructions. In: PALEO, B. W.; DELAHAYE, D. (Ed.). *All about Proofs, Proofs for All*. College Publications, 2015, (Studies in Logic (Mathematical logic and foundations), v. 55). Disponível em: <https://inria.hal.science/hal-01094195>. Citado na página 27.

PENN, M. *Quadratic Reciprocity proof – Number Theory 23*. 2021. Youtube. Disponível em: <https://youtu.be/2UlqaUZiyZ8?si=iMU-yaCjPHWxIdQ8>. Citado na página 91.

SARKAR, P. Computing square roots faster than the tonelli-shanks/bernstein algorithm. *Advances in Mathematics of Communications*, v. 18, n. 1, p. 141–162, 2024. Disponível em: <https://www.aimsciences.org/article/id/6212ee892d80b75aa4a24c21>. Citado na página 26.

SHANKS, D. Five number theoretical algorithms. In: MANITOBA CONFERENCE ON NUMERICAL MATHEMATICS, 2. *Proceedings of the Second Manitoba Conference on Numerical Mathematics*. Winnipeg: Utilitas Mathematica Pub., 1972. Citado na página 26.

TEAM, T. C. D. *The Coq Reference Manual*. France, 2024. Citado na página 32.

TONELLI, A. Bemerkung über die auflösung quadratischer congruenzen. *Nachrichten von der Königl. Gesellschaft der Wissenschaften und der Georg-Augusts-Universität zu Göttingen*, v. 30, n. 1, p. 344–346, 1891. Disponível em: <http://eudml.org/doc/180329>. Citado na página 26.

WOOD, A. *A Casa Sonolenta*. Original. Estados Unidos da América: Editora Ática, 1999. (Abracadabra). ISBN 9788508032761. Citado na página 11.

WRIGHT, S. Four interesting applications of quadratic reciprocity. In: ______. *Quadratic Residues and Non-Residues: Selected Topics*. Suíça: Springer Cham, 2016. p. 79–118. ISBN 978-3-319-45955-4. Disponível em: <https://doi.org/10.1007/978-3-319-45955-4_4>. Citado na página 91.

Apêndices

APÊNDICE A – Algoritmo de Tonelli-Shanks (ou RESSOL)

Neste Capítulo será apresentado um conteúdo sobre o qual trabalhos futuros podem se desdobrar, tendo como base o presente trabalho. Essa apresentação será separada em duas partes: descrição do algoritmo e prova manual de corretude e terminação. Ambas se baseiam em (HUYNH, 2021).

A.1 Descrição do Algoritmo

Para apresentação do pseudocódigo do algoritmo RESSOL, é conveniente que se definam algoritmos auxiliares, de modo a evitar que o pseudocódigo principal fique demasiadamente extenso para o leitor.

Dentre este algoritmos auxiliares, o primeiro a ser apresentado recebe um inteiro n e retorna um valor s tal que $\exists q \in \mathbb{Z}, q \cdot 2^s = n$. Seu pseudocódigo é dado a seguir:

Algoritmo 2: FATORAR-POTÊNCIA-DE-DOIS

Entrada: $n \in \mathbb{Z}$
Saída: $s \in \mathbb{Z}$.

$q \leftarrow n$
$s \leftarrow 0$
enquanto $2 \mid q \land q \neq 0$ **faça**
  $s \leftarrow s + 1$
  $q \leftarrow \frac{q}{2}$
retorna s

O segundo algoritmo auxiliar tem como objetivo receber um inteiro p e então retornar um inteiro z tal que z não é um resíduo quadrático módulo p:

Algoritmo 3: OBTER-RESÍDUO-NÃO-QUADRÁTICO

Entrada: $p \in \mathbb{Z}$
Saída: $z \in \mathbb{Z}$

$z \leftarrow 2$ **enquanto** $z^{\frac{p-1}{2}} \not\equiv -1 \pmod{p}$ **faça**
  $z \leftarrow z + 1$
retorna z

O terceiro e último algoritmo auxiliar a ser apresentado recebe dois inteiros n e p e computa um valor i tal que $n^{2^i} \equiv 1 \bmod p$:

Algoritmo 4: REPETIR-QUADRADOS

Entrada: $n, p \in \mathbb{Z}$
Saída: $i \in \mathbb{Z}$

$i \leftarrow 0$ $t \leftarrow n$ **enquanto** $t \neq 1$ **faça**
  $i \leftarrow i + 1$
  $t \leftarrow t^2 \bmod p$
retorna i

Vistos estes algoritmos auxiliares, o pseudocódigo do algoritmo RESSOL, é apresentado a seguir:

Algoritmo 5: RESSOL

Entrada: $a, p \in \mathbb{Z}$
Saída: inteiro r ou *erro*.

se p não é primo **então**
  retorna *erro*
se $a \equiv 1 \pmod p$ **então**
  retorna 1
se $a^{\left(\frac{p-1}{2}\right)} \not\equiv 1 \pmod p$ **então**
  se $a^{\left(\frac{p-1}{2}\right)} \equiv 0 \pmod p$ **então**
    retorna 0
  retorna *erro*
$s \leftarrow$ FATORAR-POTÊNCIA-DE-DOIS$(p-1)$
$q \leftarrow \frac{p-1}{2^s}$
$z \leftarrow$ OBTER-RESÍDUO-NÃO-QUADRÁTICO(p)
$m \leftarrow s$
$c \leftarrow z^q \bmod p$
$t \leftarrow a^q \bmod p$
$r \leftarrow a^{\frac{q+1}{2}} \bmod p$
enquanto $t \neq 1$ **faça**
  $i \leftarrow$ REPETIR-QUADRADOS(t, p)
  $b \leftarrow c^{2^{m-i-1}} \bmod p$
  $m \leftarrow i$
  $c \leftarrow b^2 \bmod p$
  $t \leftarrow t \cdot b^2 \bmod p$
  $r \leftarrow r \cdot b \bmod p$
retorna r

A.2 Prova Manual

A prova do algoritmo RESSOL consiste nas seguintes partes:

1. Provar que as funções FATORAR-POTÊNCIA-DE-DOIS e OBTER-RESÍDUO-NÃO-QUADRÁTICO terminam e retornam o resultado correto: quanto a primeira, sobre a condição do *loop*, é trivial notar que eventualmente $2 \nmid q$ ou $q = 0$, portanto esta função termina (e o seu resultado também é trivial). Quanto a segunda função, pelo Lema 4, há algum resíduo não quadrático módulo p, portanto essa função eventualmente irá terminar e irá retornar o resultado correto.

2. Provar as seguintes invariantes do *loop* no algoritmo RESSOL, considerando (da primeira parte da prova) que $p = q \cdot 2^s$ e $z^{\frac{p-1}{2}} \equiv -1 \pmod{p}$:

   Lema 11 *No algoritmo* RESSOL*, em toda iteração do loop, para as variáveis a, c, m, t e r são válidas as seguintes equações:*

   - $c^{2^{m-1}} \equiv -1 \pmod{p}$
   - $t^{2^{m-1}} \equiv 1 \pmod{p}$
   - $r^2 \equiv t \cdot a \pmod{p}$

   Demonstração: utilizando indução sobre o número de iterações k do *loop* tem-se [1]:

   - *Caso base* (0-ésima iteração): pela inicialização das variáveis (antes do *loop*), note que:
     - ▷ $c^{2^{m-1}} \equiv (z^q)^{2^{s-1}} \equiv (z^{\frac{p-1}{2}}) \pmod{p}$, pois $p - 1 = q \cdot 2^s$, e como z é um resíduo não quadrático módulo p, então, $c^{2^{m-1}} \equiv -1 \pmod{p}$.
     - ▷ $t^{2^{m-1}} \equiv (a^q)^{2^{s-1}} \equiv (a^{\frac{p-1}{2}}) \pmod{p}$, pois $p - 1 = q \cdot 2^s$, e como a é um resíduo quadrático módulo p, $t^{2^{m-1}} \equiv 1 \pmod{1}$.
     - ▷ $r^2 \equiv (a^{\frac{q+1}{2}})^2 \equiv a^{q+1} \equiv a^q \cdot a \equiv t \cdot a \pmod{p}$.
   - *Hipótese de indução*: para $j, k \in \mathbb{N}$, para todo $0 \leq j \leq k$, na j-ésima iteração têm-se[2]:
     - ▷ $c_j^{2^{(m_j-1)}} \equiv -1 \pmod{p}$
     - ▷ $t_j^{2^{(m_j-1)}} \equiv 1 \pmod{p}$

[1] Uma observação a se fazer ao leitor antes do início desta prova é que muitas substituições nas manipulações algébricas são feitas com base nas atribuições que ocorrem no pseudocódigo.

[2] A variável i não será enumerada pela iteração pois seu valor é calculado sempre no inicio dessa, tornando isso desnecessário (nas equações ocorre apenas o uso do valor de i na iteração atual).

▷ $r_j^2 \equiv t_j \cdot a \pmod{p}$

- *Passo*: realizando a próxima iteração ($k+1$-ésima iteração) têm-se:

▷ quanto a função REPETIR-QUADRADOS, como o *loop* dessa termina ao encontrar um valor i tal que $t_k^{2^i} \equiv 1 \pmod{p}$ e pela *hipótese de indução* $t_k^{2^{m_k-1}} \equiv 1 \pmod{p}$, o *loop* desta função irá terminar em no máximo $m-1$ iterações.

▷ tem-se que $b_{k+1} \equiv c_k^{2^{m_k-i-1}} \pmod{p}$.

▷ tem-se que $m_{k+1} = i$.

▷ $c_{k+1}^{2^{(m_{k+1}-1)}} \equiv c_{k+1}^{2^{i-1}} \equiv (b_{k+1}^2)^{2^{i-1}} \equiv b_{k+1}^{2^i} \equiv (c_k^{2^{(m_k-i-1)}})^{2^i} \equiv c_k^{2^{(m_k-1)}} \pmod{p}$, e pela *hipótese de indução* $c_k^{2^{(m_k-1)}} \equiv -1 \pmod{p}$, portanto tem-se que $c_{k+1}^{2^{(m_{k+1}-1)}} \equiv -1 \pmod{p}$.

▷ $t_{k+1}^{2^{(m_{k+1}-1)}} \equiv (t_k \cdot b_{k+1}^2)^{2^{(m_{k+1}-1)}} \equiv (t_k \cdot b_{k+1}^2)^{2^{(i-1)}} \equiv t_k^{2^{(i-1)}} \cdot b_{k+1}^{2^i} \pmod{p}$, nesta situação, note que i é sempre o menor inteiro tal que $t_k^{2^i} \equiv 1 \pmod{p}$, assim, tem-se os seguintes casos:

► se $i = 0$, então $t_k^{2^0} \equiv t_k \equiv 1 \pmod{p}$, mas note que, pelas atribuições feitas no pseucódigo, $0 < t \leq p-1$, portanto só pode ser o caso de que $t_k = 1$, mas se isso ocorre então o *loop* teria terminado na k-ésima iteração, com (pela hipótese de indução) $r_k^2 \equiv t_k \cdot a \equiv a \pmod{p}$, portanto r_k é a solução de $r_k^2 \equiv a \pmod{p}$, e o algoritmo além de ter terminado retornou o resultado correto.

► para qualquer $i > 0$, note que, se $t_k^{2^i} \equiv 1 \pmod{p}$ então $(t_k^{2^{i-1}})^2 \equiv (1)^2 \pmod{p}$ e:

$$\begin{aligned}
(t_k^{2^{i-1}})^2 \equiv (1)^2 \pmod{p} &\iff p \mid (t_k^{2^{i-1}})^2 - (1)^2 \\
&\iff p \mid (t_k^{2^{i-1}} - 1) \cdot (t_k^{2^{i-1}} + 1) \\
&\iff p \mid (t_k^{2^{i-1}} - 1) \vee p \mid (t_k^{2^{i-1}} + 1) \\
&\iff t_k^{2^{i-1}} \equiv 1 \pmod{p} \vee t_k^{2^{i-1}} \equiv -1 \pmod{p}
\end{aligned}$$

porém, sabe-se que i é o menor natural tal que $t_k^{2^i} \equiv 1 \pmod{p}$ (e $i-1 < i$), portanto só pode ser o caso de que $t_k^{2^{i-1}} \equiv -1 \pmod{p}$, assim, note que

$$\begin{aligned}
t_{k+1}^{2^{m_{k+1}-1}} \equiv t_{k+1}^{2^{m_{k+1}-1}} \pmod p &\iff t_{k+1}^{2^{m_{k+1}-1}} \equiv (t_k \cdot b_{k+1}^2)^{2^{m_{k+1}-1}} \pmod p \\
&\iff t_{k+1}^{2^{m_{k+1}-1}} \equiv (t_k)^{2^{m_{k+1}-1}} \cdot b_{k+1}^{2^{m_{k+1}}} \pmod p \\
&\iff t_{k+1}^{2^{m_{k+1}-1}} \equiv (t_k)^{2^{i-1}} \cdot b_{k+1}^{2^{m_{k+1}}} \pmod p \\
&\iff t_{k+1}^{2^{m_{k+1}-1}} \equiv (t_k)^{2^{i-1}} \cdot b_{k+1}^{2^{i}} \pmod p \\
&\iff t_{k+1}^{2^{m_{k+1}-1}} \equiv (t_k)^{2^{i-1}} \cdot (c_k^{2^{m_k-i-1}})^{2^i} \pmod p \\
&\iff t_{k+1}^{2^{m_{k+1}-1}} \equiv (t_k)^{2^{i-1}} \cdot (c_k^{2^{m_k-1}}) \pmod p
\end{aligned}$$

Como $(t_k)^{2^{i-1}} \equiv -1 \pmod p$ e pela hipótese de indução $(c_k^{2^{m_k-1}}) \equiv -1 \pmod p$ tem-se que $t_{k+1}^{2^{m_{k+1}-1}} \equiv (-1) \cdot (-1) \equiv 1 \pmod p$.

▷ por último, para r_{k+1} temos que:

$$\begin{aligned}
r_{k+1}^2 \equiv r_{k+1}^2 \pmod p &\iff r_{k+1}^2 \equiv (r_k \cdot b_{k+1})^2 \pmod p \\
&\iff r_{k+1}^2 \equiv r_k^2 \cdot b_{k+1}^2 \pmod p \\
&\iff r_{k+1}^2 \equiv t_k \cdot a \cdot b_{k+1}^2 \pmod p \\
&\iff r_{k+1}^2 \equiv a \cdot (t_k \cdot b_{k+1}^2) \pmod p
\end{aligned}$$

e pela atribuição feita à t_{k+1}:

$$\begin{aligned}
r_{k+1}^2 \equiv r_{k+1}^2 \pmod p &\iff r_{k+1}^2 \equiv a \cdot t_{k+1} \pmod p \\
&\iff r_{k+1}^2 \equiv t_{k+1} \cdot a \pmod p
\end{aligned}$$

■

3. Tendo provado as invariante do *loop*, resta provar os teoremas de terminação e corretude:

Teorema 6 (*Terminação do algoritmo RESSOL*) *O algoritmo* Ressol *executa sempre um número finito de iterações.*

Demonstração: tendo provado as invariante do *loop*, observe que, a cada iteração do *loop*, como i é o menor número natural para o qual $t^{2^i} \equiv 1 \pmod p$ e $i \leq m-1$, pois $t^{2^{m-1}} \equiv 1 \pmod p$, ao atualizar o valor de m fazendo $m \leftarrow i$, o valor de m irá diminuir. Assim, eventualmente se terá que $m = 1$ e portanto, para o novo valor de t, se terá pela invariante do *loop*, que $t \equiv 2^{m-1} \equiv 2^{1-1} \equiv 1 \pmod p$, ou seja, $t = 1$ pois $0 \leq t \leq p-1$. Observe que não ocorre $m = 0$ dentro do *loop* pois $0 \leq t \leq p-1$, logo para que ocorre-se isso seria necessário $t = 1$, mas então o algoritmo teria parado antes de alterar o valor de m. Assim está provado que o algoritmo sempre termina (quanto a parte externa ao *loop*, a demonstração de que essa termina foi feita em 2 se mostrando que os algoritmos auxiliares executados nessa terminam). ■

Teorema 7 (*Corretude do algoritmo RESSOL*) *O algoritmo* Ressol *ao receber como argumentos a e p tem como retorno:*

- *erro se p não é primo ou se a não é um resíduo quadrático.*
- 0 *se* $p \mid a$.
- r *tal que* $r^2 \equiv a \pmod{p}$.

Demonstração: Quanto ao valor retornado pelo algoritmo, fora do *loop* a conclusão sobre sua corretude é trivial, pois:

- Se p não é primo, por ser quebrada a hipótese em que se baseia o algoritmo (p ser primo) se retorna *erro*;
- se $a \equiv 1 \pmod{p}$ basta retornar 1 pois $a \equiv 1 \equiv 1^2 \pmod{p}$;
- se $a^{\frac{p-1}{2}} \equiv 0 \pmod{p}$ então, pelo teorema 4, $p \mid a$, portanto $a \equiv 0 \equiv 0^2 \pmod{p}$, por isso se retorna 0;
- se $a^{\frac{p-1}{2}} \not\equiv 1 \pmod{p}$ e $a^{\frac{p-1}{2}} \not\equiv 0 \pmod{p}$ então, de acordo com o teorema 4 só pode ser o caso de $a^{\frac{p-1}{2}} \equiv -1 \pmod{p}$, situação em que não existe r tal que $a \equiv r^2 \pmod{p}$, e por isso se retorna *erro*.

Caso o retorno só ocorra após iniciar o *loop*, note que, esse só encerra quando $t = 1$, e pela invariante tem-se $r^2 \equiv t \cdot a \equiv a \pmod{p}$ e r é o valor retornado. ■

APÊNDICE B – Reciprocidade Quadrática

Outro conteúdo sobre o qual trabalhos futuros podem tratar, tendo como base o presente trabalho, será apresentado neste Capítulo a Lei de Reciprocidade Quadrática. Como motivação, esta lei permite tornar mais eficiente o algoritmo *RESSOL* (COOK, 2023) e também possui outras aplicações como *zero-knowledge proofs* (WRIGHT, 2016). O conteúdo apresentado nesse Capítulo é baseado em (BROCHERO et al., 2013) e em (PENN, 2021).

O Teorema conhecido como Lei de Reciprocidade Quadrática possui a seguinte descrição:

Teorema 8 (*Reciprocidade Quadrática*) *Sejam p e q primos ímpares (maiores que 2) distintos, então:*

1. $\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2} \cdot \frac{q-1}{2}}$
2. $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 \ se \ p \equiv \pm 1 \pmod 8 \\ -1 \ se \ p \equiv \pm 3 \pmod 8 \end{cases}$

Para demostração deste teorema antes deve-se demonstrar um lema necessário para tal, que é o seguinte:

Lema 12 (*Gauß*) *Seja $p > 2$ um número primo e $a \in \mathbb{Z}$ um número coprimo de p, isto é, $\mathrm{mdc}(a, p) = 1$, sendo s o número de elementos do conjunto*

$$\left\{x \in \mathbb{Z} | x \in \left\{a, 2 \cdot a, 3 \cdot a, ..., \frac{p-1}{2} \cdot a\right\} \wedge x \bmod p > \frac{p-1}{2}\right\}$$

então

$$\left(\frac{a}{p}\right) = (-1)^s$$

Demonstração: dado o seguinte conjunto $\left\{\pm 1, \pm 2, \pm 3, ..., \pm \frac{p-1}{2}\right\}$, observe que todos os elementos desse conjunto possuem *inverso módulo* p de acordo com o Lema 5, e, para todo $j \in \left[1, \frac{p-1}{2}\right]$ é possível escolher $\epsilon_j \in \{-1, 1\}$ e $m_j \in \left[1, 2, ..., \frac{p-1}{2}\right]$ tal que $a \cdot j \equiv \epsilon_j \cdot m_j \pmod p$ (pois com tais valores de ϵ_j e m_j pode-se obter um número equivalente em módulo p a qualquer outro), e além disso, para $i, k \in \left[1, \frac{p-1}{2}\right]$, se $i \neq k$ então $m_i \neq m_k$, pois há uma combinação única $\epsilon_j \cdot m_j$ para cada resto possível. Tal afirmação, significa que, caso $m_i = m_k$ então:

- $a \cdot i \equiv a \cdot j \pmod{p}$ é o único caso possível, em que pelo Item 7, $i \equiv j \pmod{p}$, e portanto $i = j$ (devido ao intervalo que os valores i e j pertencem);
- $a \cdot i \equiv -a \cdot j \pmod{p}$ é um caso impossível, pois pelo Item 7, $i \equiv -j \pmod{p}$, mas isso é impossível dado o intervalo que esses valores i e j se encontram.

Com isso, tem-se que:

$$(a \cdot 1) \cdot (a \cdot 2) \cdot \ldots \cdot (a \cdot \frac{p-1}{2}) \equiv \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \cdot \ldots \cdot \epsilon_{\frac{p-1}{2}} \cdot \ldots \cdot m_1 \cdot m_2 \cdot \ldots \cdot m_{\frac{p-1}{2}} \pmod{p}$$
$$\iff a^{\frac{p-1}{2}} \cdot \left(\frac{p-1}{2}\right)! \equiv \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \cdot \ldots \cdot \epsilon_{\frac{p-1}{2}} \cdot \left(\frac{p-1}{2}\right)! \pmod{p}$$

como $\mathrm{mdc}(\left(\frac{p-1}{2}\right)!, p) = 1$, pelo Item 7 e pelo Teorema 4:

$$a^{\frac{p-1}{2}} \cdot \left(\frac{p-1}{2}\right)! \equiv \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \cdot \ldots \cdot \epsilon_{\frac{p-1}{2}} \cdot \left(\frac{p-1}{2}\right)! \pmod{p}$$
$$\iff a^{\frac{p-1}{2}} \equiv \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \cdot \ldots \cdot \epsilon_{\frac{p-1}{2}} \pmod{p}$$
$$\iff \left(\frac{a}{p}\right) \equiv \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \cdot \ldots \cdot \epsilon_{\frac{p-1}{2}} \pmod{p}$$

E como ambos os valores pertencem ao conjunto $\{-1, 1\}$, então:

$$\left(\frac{a}{p}\right) = \epsilon_1 \cdot \epsilon_2 \cdot \epsilon_3 \cdot \ldots \cdot \epsilon_{\frac{p-1}{2}}$$

Dado que para todo m_j tem-se que $m_j \cdot \epsilon_j \bmod p > \frac{p-1}{2}$ se e somente se $\epsilon_j < 0$, então:

$$\left(\frac{a}{p}\right) = (-1)^s$$

■

Outro teorema que deve-se provar é o seguinte:

Teorema 9 *Seja mp e q números primos ímpares, então:*

$$\frac{p-1}{2} \cdot \frac{q-1}{2} = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{q \cdot i}{p} \right\rfloor + \sum_{j=1}^{\frac{q-1}{2}} \left\lfloor \frac{p \cdot j}{q} \right\rfloor \tag{B.1}$$

Demonstração: inicialmente define-se o seguinte conjunto:

$$S = \left\{ (x, y) \in \mathbb{Z} \times \mathbb{Z} \;\middle|\; 1 \leq x \leq \frac{p-1}{2} \wedge 1 \leq y \leq \frac{q-1}{2} \right\}$$

Separando este conjunto em dois novos conjuntos S_1 e S_2 tal que:

$$S_1 = \left\{ (x, y) \in \mathbb{Z} \times \mathbb{Z} \;\middle|\; p \cdot y < q \cdot x \wedge (x, y) \in S \right\}$$
$$S_2 = \left\{ (x, y) \in \mathbb{Z} \times \mathbb{Z} \;\middle|\; p \cdot y > q \cdot x \wedge (x, y) \in S \right\}$$

É fácil notar que estes conjuntos são disjuntos pois suas restrições são excludentes. Quanto a $S = S_1 \cup S_2$, note que, como $p \neq q$ não existem pontos em S tais que $p \cdot y = q \cdot x$, pois caso existissem teriam-se números iguais com fatoração em primos diferentes (devido aos intervalos em que x e y estão), o que é impossível. Assim, como todo par (x, y) satisfaz uma das restrições, então $S = S_1 \cup S_2$.

Agora, reescrevendo as restrições dos conjuntos, começando por S_1, tem-se:

$$p \cdot y < q \cdot x \Longleftrightarrow y < \frac{q \cdot x}{p}$$

Sabe-se que $y \in \mathbb{Z}$ e que $p \nmid q \cdot x$ (pois $x < p$ e $p \nmid q$ visto que q é um primo diferente de p), então

$$y < \frac{q \cdot x}{p} \Longleftrightarrow y \leq \left\lfloor \frac{q \cdot x}{p} \right\rfloor$$

Além disso, note que, como o valor máximo de x é $\frac{p-1}{2}$ e $\frac{p-1}{p} < 1$ obtêm-se o seguinte:

$$\frac{q \cdot x}{p} \leq \frac{q \cdot (p-1)}{2 \cdot p} < \frac{q}{2}$$

portanto

$$\left\lfloor \frac{q \cdot x}{p} \right\rfloor \leq \frac{q-1}{2}$$

Assim pode-se alterar a definição do conjunto S_1 para:

$$S_1 = \left\{ (x, y) \in \mathbb{Z} \times \mathbb{Z} \;\middle|\; 1 \leq x \leq \frac{p-1}{2} \wedge 1 \leq y \leq \left\lfloor \frac{q \cdot x}{p} \right\rfloor \right\}$$

Quanto a restrição de S_2, de maneira semelhante, tem-se:

$$p \cdot y > q \cdot x \Longleftrightarrow x < \frac{p \cdot y}{q}$$

Sabe-se que $x \in \mathbb{Z}$ e que $q \nmid p \cdot y$ (pois $y < q$ e $q \nmid p$ visto que p é um primo diferente de q), então

$$x < \frac{p \cdot x}{q} \Longleftrightarrow x < \left\lfloor \frac{p \cdot y}{q} \right\rfloor$$

E como o valor máximo de y é $\frac{q-1}{2}$ e $\frac{q-1}{q} < 1$, obtêm-se

$$\frac{p \cdot y}{q} \leq \frac{p \cdot (q-1)}{2 \cdot q} < \frac{p}{2}$$

portanto

$$\left\lfloor \frac{p \cdot y}{q} \right\rfloor \leq \frac{p-1}{2}$$

Assim pode-se alterar a definição do conjunto S_2 para:

$$S_2 = \left\{ (x, y) \in \mathbb{Z} \times \mathbb{Z} \;\middle|\; 1 \leq y \leq \frac{q-1}{2} \wedge 1 \leq x \leq \left\lfloor \frac{p \cdot y}{q} \right\rfloor \right\}$$

Neste momento da demonstração, note que $|S| = \frac{p-1}{2} \cdot \frac{q-1}{2}$ (devido aos valores possíveis de escolha para x e y na montagem de um par). Para o conjunto S_1 note que o número de escolhas possíveis do valor de y depende do valor de x, ou seja, para $x = i$ tem-se $\left\lfloor \frac{q \cdot i}{p} \right\rfloor$ pares possíveis, logo:

$$|S_1| = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{q \cdot i}{p} \right\rfloor$$

De modo similar, para o conjunto S_2 observe que o número de escolhas possíveis do valor de x depende do valor y, isto é, para $y = j$ tem-se $\left\lfloor \frac{p \cdot j}{q} \right\rfloor$ pares possíveis, logo:

$$|S_2| = \sum_{j=1}^{\frac{q-1}{2}} \left\lfloor \frac{p \cdot j}{q} \right\rfloor$$

Como $|S| = |S_1| + |S_2|$, então:

$$\frac{p-1}{2} \cdot \frac{q-1}{2} = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{q \cdot i}{p} \right\rfloor + \sum_{j=1}^{\frac{q-1}{2}} \left\lfloor \frac{p \cdot j}{q} \right\rfloor$$

■

Por último, deve-se provar o seguinte teorema:

Teorema 10 *Seja p um número primo ímpar, então, para $a \in \mathbb{Z}$, se* $\mathrm{mdc}(a, 2 \cdot p) = 1$ *(a é ímpar) e sendo*

$$t = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{a \cdot i}{p} \right\rfloor$$

então

$$\left(\frac{a}{p} \right) = (-1)^t$$

Demonstração: dado o conjunto de restos $A = \{a \bmod p, 2 \cdot a \bmod p, ..., \frac{p-1}{2} \cdot a \bmod p\}$, define-se $R = \{r_1, r_2, ..., r_m\}$ tal que esse é o conjunto de restos em A menores ou iguais a $\frac{p-1}{2}$ e $S = \{s_1, s_2, ..., s_n\}$ o conjunto de restos em A maiores que $\frac{p-1}{2}$. Observe que para qualquer $i \in [1, \frac{p-1}{2}]$ existe $r \in R$ tal que

$$i \cdot a = \left\lfloor \frac{i \cdot a}{p} \right\rfloor \cdot p + r$$

ou existe $s \in S$ tal que:

$$i \cdot a = \left\lfloor \frac{i \cdot a}{p} \right\rfloor \cdot p + s$$

portanto

$$\sum_{i=1}^{\frac{p-1}{2}} i \cdot a = p \cdot \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{i \cdot a}{p} \right\rfloor + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k$$

manipulando essa equação, tem-se:

$$\sum_{i=1}^{\frac{p-1}{2}} i \cdot a = p \cdot \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{i \cdot a}{p} \right\rfloor + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k \tag{B.2}$$

$$\iff a \cdot \sum_{i=1}^{\frac{p-1}{2}} i = p \cdot \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{i \cdot a}{p} \right\rfloor + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k \tag{B.3}$$

mas note que, de maneira similar ao que se teve na demonstração do Lema 12, os conjuntos R e S não possuem valores repetidos, pois, para $i_1, i_2 \in [1, \frac{p-1}{2}]$, se $i_1 \cdot a \equiv i_2 \cdot a \pmod{p}$, então pelo Item 7 (como $\text{mdc}(a, p) = 1$), tem-se que $i_1 \equiv i_2 \pmod{p}$, e portanto dado que $i_1, i_2 \in [1, \frac{p-1}{2}]$, se obtêm $i_1 = i_2$. Além disso, note que não é possível para $i_1, i_2 \in [1, \frac{p-1}{2}]$ que $i_1 \cdot a \equiv p - i_2 \cdot a \pmod{p}$, pois teria-se então $i_1 \cdot a \equiv -i_2 \cdot a \pmod{p}$ e por conseguinte (novamente pelo Item 7) $i_1 \equiv -i_2 \pmod{p}$, o que é impossível dado que $i_1, i_2 \in [1, \frac{p-1}{2}]$. Portanto, sendo $S' = \{s \in S \mid p - s\}$, tem-se que:

$$\left\{1, 2, ..., \frac{p-1}{2}\right\} = R \cup S'$$

logo

$$\sum_{i=1}^{\frac{p-1}{2}} i = \sum_{k=1}^{n} (p - s_k) + \sum_{j=1}^{m} r_j \tag{B.4}$$

e então multiplicando ambos os lados por a:

$$a \cdot \sum_{i=1}^{\frac{p-1}{2}} i = a \cdot \left(\sum_{k=1}^{n} (p - s_k) + \sum_{j=1}^{m} r_j \right) \tag{B.5}$$

Realizando então a substituição $t = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{i \cdot a}{p} \right\rfloor$ e B.5 em B.3, obtêm-se:

$$a \cdot \left(\sum_{k=1}^{n} (p - s_k) + \sum_{j=1}^{m} r_j \right) = p \cdot t + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k$$

e realizando manipulações:

$$a \cdot \left(\sum_{k=1}^{n} (p - s_k) + \sum_{j=1}^{m} r_j \right) = p \cdot t + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k$$

$$\iff a \cdot \left(n \cdot p - \sum_{k=1}^{n} s_k + \sum_{j=1}^{m} r_j \right) = p \cdot t + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k$$

somando $\sum_{k=1}^{n} s_k$ e subtraindo $\sum_{j=1}^{m} r_j$ de ambos os lados

$$\begin{aligned}
& a \cdot \left(n \cdot p - \sum_{k=1}^{n} s_k + \sum_{j=1}^{m} r_j \right) = p \cdot t + \sum_{j=1}^{m} r_j + \sum_{k=1}^{n} s_k \\
\iff & a \cdot n \cdot p + (a-1) \cdot \left(- \sum_{k=1}^{n} s_k + \sum_{j=1}^{m} r_j \right) = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & n \cdot p + (a-1) \cdot n \cdot p + (a-1) \cdot \left(- \sum_{k=1}^{n} s_k + \sum_{j=1}^{m} r_j \right) = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & n \cdot p + (a-1) \cdot \left(n \cdot p - \sum_{k=1}^{n} s_k + \sum_{j=1}^{m} r_j \right) = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & n \cdot p + (a-1) \cdot \left(\sum_{k=1}^{n} (p - s_k) + \sum_{j=1}^{m} r_j \right) = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k
\end{aligned}$$

e pela Equação B.4, tem-se:

$$\begin{aligned}
& n \cdot p + (a-1) \cdot \left(\sum_{k=1}^{n} (p - s_k) + \sum_{j=1}^{m} r_j \right) = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & n \cdot p + (a-1) \cdot \sum_{i=1}^{\frac{p-1}{2}} i = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k
\end{aligned}$$

Então, pela fórmula da Soma de Gauss:

$$\begin{aligned}
& n \cdot p + (a-1) \cdot \sum_{i=1}^{\frac{p-1}{2}} i = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & n \cdot p + (a-1) \cdot \frac{1}{2} \cdot \frac{p-1}{2} \cdot \frac{p+1}{2} = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & n \cdot p + (a-1) \cdot \frac{p^2-1}{8} = p \cdot t + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & (a-1) \cdot \frac{p^2-1}{8} = p \cdot (t-n) + 2 \cdot \sum_{k=1}^{n} s_k
\end{aligned}$$

Agora, trabalhando com congruência módulo 2, tem-se:

$$\begin{aligned}
& (a-1) \cdot \frac{p^2-1}{8} = p \cdot (t-n) + 2 \cdot \sum_{k=1}^{n} s_k \\
\iff & (a-1) \cdot \frac{p^2-1}{8} \equiv p \cdot (t-n) + 2 \cdot \sum_{k=1}^{n} s_k \pmod{2}
\end{aligned}$$

Dado que $2 \cdot \sum_{k=1}^{n} s_k$ é par e $p \equiv 1 \pmod{2}$ (o que implica que $1 \cdot (t-n) \equiv p \cdot (t-n) \pmod{2}$, utilizando o Item 6), então:

$$\begin{aligned}
& (a-1) \cdot \frac{p^2-1}{8} \equiv p \cdot (t-n) + 2 \cdot \sum_{k=1}^{n} s_k \pmod{2} \\
\iff & (a-1) \cdot \frac{p^2-1}{8} \equiv (t-n) \pmod{2}
\end{aligned}$$

mas como a é ímpar, chega-se em:

$$0 \equiv (t - n) \pmod 2$$

portanto:

$$n \equiv t \pmod 2$$

Por fim, observe que, aplicando o Lema 12 com p e a, o valor s deste teorema é igual ao valor n (pois ambos são o número de restos em A maiores que $\frac{p-1}{2}$), ou seja:

$$\left(\frac{a}{p}\right) = (-1)^s = (-1)^n = (-1)^t$$

■

Finalmente pode então se iniciar a demonstração do Teorema 8:

Demonstração: inciando pela prova do Item 1, sendo:

$$t_1 = \sum_{j=1}^{\frac{q-1}{2}} \left\lfloor \frac{p \cdot j}{q} \right\rfloor$$

e

$$t_2 = \sum_{i=1}^{\frac{p-1}{2}} \left\lfloor \frac{q \cdot i}{p} \right\rfloor$$

então usando o Teorema 10:

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{t_1} \cdot (-1)^{t_2} = (-1)^{t_1+t_2}$$

e pelo Teorema 9:

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2} \cdot \frac{q-1}{2}}$$

Portanto está provado o Item 1. Quanto ao Item 2, note que para um número primo p ímpar, em relação ao módulo 4 existem apenas as duas seguintes possibilidades:

1. $p \equiv 1 \pmod 4$
2. $p \equiv 3 \pmod 4$

Se $p \equiv 1 \pmod 4$ então existe $k \in \mathbb{Z}$ tal que $p = 4 \cdot k + 1$, logo, $\frac{p-1}{2} = 2 \cdot k$. Sendo assim, há, primeiramente, a possibilidade de que $p \equiv 1 \pmod 8$, então, existe $j \in \mathbb{Z}$ tal que:

$$\begin{aligned} p \equiv 1 \pmod 8 &\iff p = 8 \cdot j + 1 \\ &\iff 4 \cdot k + 1 = 8 \cdot j + 1 \\ &\iff 4 \cdot k = 8 \cdot j \\ &\iff k = 2 \cdot j \end{aligned}$$

Portanto, como k é par, $(-1)^k = 1$, e pelo Lema 12 com $a = 2$, note que k é igual ao valor s, pois para $1 \leq i \leq k = \frac{p-1}{4}$ tem-se $2 \leq 2 \cdot i \leq \frac{p-1}{2}$ (isto é, $k = \frac{p-1}{4}$ restos menores ou iguais a $\frac{p-1}{2}$) e para $k+1 = \frac{p-1}{4}+1 \leq i \leq \frac{p-1}{2} = 2 \cdot k$ tem-se $\frac{p-1}{2} - \left(\frac{p-1}{4}+1\right)+1 = \frac{p-1}{4} = 2 \cdot k - (k+1)+1 = k$, (ou seja, há $k = \frac{p-1}{4}$ restos maiores que $\frac{p-1}{2}$), portanto:

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p-1}{4}} = (-1)^{2 \cdot j} = 1$$

Como $\frac{p-1}{4} = k = 2 \cdot j$ é par, então, $\frac{p-1}{4} \cdot \frac{p+1}{2} = \frac{p^2-1}{8}$ também é par, logo

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p-1}{4}} = (-1)^{2 \cdot j} = 1 = (-1)^{\frac{p^2-1}{8}}$$

Agora, caso $p \equiv 5 \pmod 8$ (que é a outra única possibilidade no caso de $p \equiv 1 \pmod 4$) tem-se que $p \equiv -3 \pmod 8$, portanto existe $j \in \mathbb{Z}$ tal que:

$$\begin{aligned} p \equiv -3 \pmod 8 &\iff p = 8 \cdot j - 3 \\ &\iff 4 \cdot k + 1 = 8 \cdot j - 3 \\ &\iff 4 \cdot k = 8 \cdot j - 4 \\ &\iff k = 2 \cdot j - 1 \end{aligned}$$

Novamente, aplicando o Lema 12 para $a = 2$ tem-se $\left(\frac{2}{p}\right) = (-1)^s$, onde $s = \frac{p-1}{4}$, e como $\frac{p-1}{4}$ é ímpar (pois $\frac{p-1}{4} = k = 2 \cdot j - 1$):

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p-1}{4}} = -1$$

E como $2 \cdot k + 1 = \frac{p-1}{2} + 1 = \frac{p+1}{2}$ é ímpar, então $\frac{p-1}{4} \cdot \frac{p+1}{2} = \frac{p^2-1}{8}$ também é ímpar (pois resulta da multiplicação de números ímpares), ou seja,

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p-1}{4}} = -1 = (-1)^{\frac{p^2-1}{8}}$$

Logo, se concluí que:

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 \text{ se } p \equiv 1 \pmod 8 \\ -1 \text{ se } p \equiv -3 \pmod 8 \end{cases} \tag{B.6}$$

Por fim, para o caso de $p \equiv 3 \pmod 4$, existe $k \in \mathbb{Z}$ tal que $p = 4 \cdot k + 3$ e por consequência, $\frac{p-1}{2} = 2 \cdot k + 1$. Assim, pelo Lema 12, note antes que $\frac{p-1}{4}$ não é inteiro e o inteiro mais próximo (e maior) deste valor é $\frac{1}{2} \cdot (\frac{p-1}{2} + 1) = \frac{p+1}{4} = k+1$, portanto para $k+1 = \frac{p+1}{4} \leq i \leq \frac{p-1}{2} = 2 \cdot k + 1$ tem-se $\frac{p-1}{2} < 2 \cdot i \leq p - 1$, logo há $2 \cdot k + 1 - (k+1) + 1 = k+1 = \frac{p-1}{2} - \frac{p+1}{4} + 1 = \frac{p+1}{4}$ restos maiores que $\frac{p-1}{2}$, então $s = \frac{p+1}{4}$.

Tratando então da possibilidade de que $p \equiv 3 \pmod 8$, em que então existe $j \in \mathbb{Z}$ tal que:

$$\begin{aligned} p \equiv 3 \pmod 8 &\iff p = 8 \cdot j + 3 \\ &\iff 4 \cdot k + 3 = 8 \cdot j + 3 \\ &\iff 4 \cdot k = 8 \cdot j \\ &\iff k = 2 \cdot j \end{aligned}$$

Assim, $s = \frac{p+1}{4} = k + 1 = 2 \cdot j + 1$, portanto $\frac{p+1}{4}$ é ímpar, e então:

$$\left(\frac{2}{p}\right) = (-1)^s = (-1)^{\frac{p+1}{4}} = -1$$

Como visto anteriormente $\frac{p-1}{2} = 2 \cdot k + 1$, logo $\frac{p-1}{2}$ é ímpar e então $\frac{p+1}{4} \cdot \frac{p-1}{2} = \frac{p^2-1}{8}$ é ímpar também (pois resulta do produto entre números ímpares), assim:

$$\left(\frac{2}{p}\right) = (-1)^s = (-1)^{\frac{p+1}{4}} = -1 = (-1)^{\frac{p^2-1}{8}}$$

Agora, se for o caso em que $p \equiv 7 \pmod 8$ (pois $p \equiv 3 \pmod 4$), então $p \equiv -1 \pmod 8$, logo existe $j \in \mathbb{Z}$ tal que:

$$\begin{aligned} p \equiv -1 \pmod 8 &\iff p = 8 \cdot j + 1 \\ &\iff 4 \cdot k + 3 = 8 \cdot j - 1 \\ &\iff 4 \cdot k = 8 \cdot j - 4 \\ &\iff k = 2 \cdot j - 1 \end{aligned}$$

Então, $s = \frac{p+1}{4} = k + 1 = 2 \cdot j$, portanto $\frac{p+1}{4}$ é par, logo:

$$\left(\frac{2}{p}\right) = (-1)^s = (-1)^{\frac{p+1}{4}} = 1$$

Finalmente, $\frac{p+1}{4} \cdot \frac{p-1}{2} = \frac{p^2-1}{8}$ é também par (pois resulta do produto entre um número par e um número qualquer), assim:

$$\left(\frac{2}{p}\right) = (-1)^s = (-1)^{\frac{p+1}{4}} = 1 = (-1)^{\frac{p^2-1}{8}}$$

Em que então se conclui que:

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 \text{ se } p \equiv -1 \pmod 8 \\ -1 \text{ se } p \equiv 3 \pmod 8 \end{cases} \tag{B.7}$$

Juntando B.6 e B.7, tem-se:

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 \text{ se } p \equiv \pm 1 \pmod 8 \\ -1 \text{ se } p \equiv \pm 3 \pmod 8 \end{cases}$$

Portanto está provado o Item 2. ■