**Bruno Rafael dos Santos**

# Formalização do Algoritmo Ressol em Coq

**Brasil**

**12 de setembro 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

12 de setembro de 2024

Bruno Rafael dos Santos
Formalização do Algoritmo Ressol em Coq/ Bruno Rafael dos Santos. – Brasil, 12 de setembro de 2024-
26p. : 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 , 12 de setembro 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.

**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

Brasil
12 de setembro de 2024

*"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).

# 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 $\varphi$ 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 será apresentado neste trabalho, é o algoritmo de Tonelli-Shanks, também conhecido como algoritmo **??** (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:

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

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 **??**. 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.

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 algoritmo *RESSOL* neste trabalho, se sustenta pelas seguintes justificativas:

1. Este algoritmo não está implementado e/ou formalizado na biblioteca Mathematical Components.

2. A base de teoremas e funções necessária para formalização deste algoritmo inclui diversos itens, dos quais, parte não se encontram na biblioteca Mathematical Components. A exemplo destes tem-se o conceito de *símbolo de Legendre* (algo que

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

é utilizado no algoritmo, mas que não possui implementação e por consequência nenhum teorema sobre disponível).

3. Tal base necessária abre a possibilidade para um segundo objetivo, que seria a formalização da lei da reciprocidade quadrática (que também não está disponível na biblioteca) e possui aplicações que serão apresentadas no Capitulo **??**.

Quanto ao objetivo secundário apresentado no Item 3 é interessante destacar que a prova deste teorema já foi implementada em *Lean* e *Isabelle*, estando ambas disponíveis publicamente[2,3].

[2] Implementação em *Lean*: <https://github.com/leanprover-community/mathlib4/blob/261109249151ce5651da62077c255a5c76b4941e/Mathlib/NumberTheory/LegendreSymbol/QuadraticReciprocity.lean#L121-L133>

[3] Implementação em *Isabelle*: <https://isabelle.in.tum.de/dist/library/HOL/HOL-Number_Theory/Quadratic_Reciprocity.html>

# 2 Biblioteca Mathematical Components

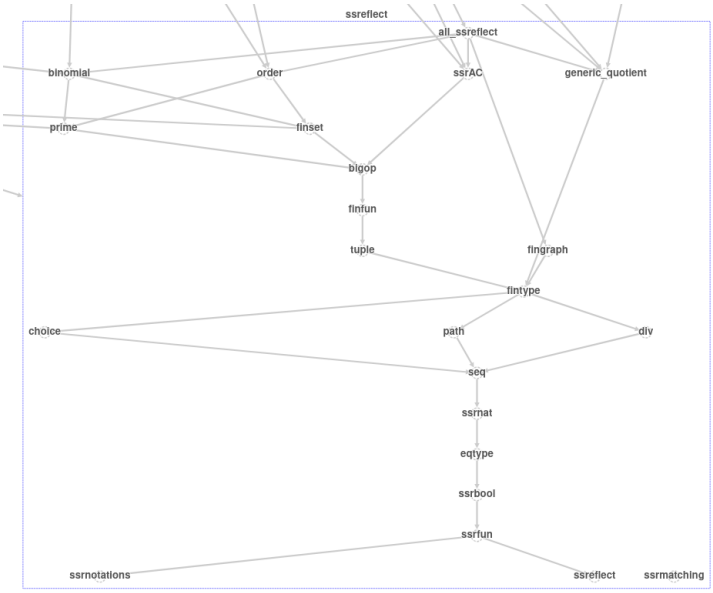
Nos capítulos seguintes será usada uma série de itens disponíveis na biblioteca Mathematical Components e outros implementados com uso da biblioteca. Todavia é necessário, por parte do leitor, um conhecimento básico sobre essa biblioteca, e para isso, neste capítulo serão explicados elementos que foram 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 a igualdade de Leibniz. 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`.

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

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

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
```

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.
```

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

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: 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 semelhante ao mencionado anteriormente (inicio da seção 2.3) que contém todos os números naturais menores que 8, porém ao invés de 8, um número $n$ qualquer. Dessa forma o primeiro tipo citado será um caso específico deste tipo (em que $n = 8$). Utilizando `Record`, temos então:

```
Record smaller (n : nat) : Type := Build
{
    x : nat;
    axiom : x < n
}.
```

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 menor que `n`. Agora, com tal declaração, visto que o objetivo da mesma é representar qualquer conjunto de números menores que um determinado natural $n$, o usuário de *Coq* provavelmente desejará que se possa escrever algo como:

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

tem-se a mensagem:

```
The command has indeed failed with message:
In environment
n : nat
a : smaller n
The term "a" has type "smaller 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 : smaller n), (x a) + 0 = (x a).
```

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

```
Definition smaller_nat (n : nat) (e : smaller 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 smaller_nat : smaller ↦ nat.
```

Agora, realizando o comando `Check` como anteriormente:

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

é gerada a mensagem:

[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, `x < n`, é equivalente a `x < n = 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 `smaller n` (para qualquer `n`) e não o valor do campo em si. Portanto a seguinte defingção poderia ser dada ser definida simplesmente como `x`.

```
forall (n : nat) (a : smaller 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 Ex9.
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 `smaller`, note que é possível provar os dois seguintes teoremas:

```
Theorem Exemplo_smaller_axiom {n} :
    ∀ (a : smaller n), a < n.

Theorem Exemplo_aplicacao_f_mod {n} :
    ∀ (f : nat → nat) (a : smaller n), (f a) %% n < n.
```

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

```
Example Exemplo_a_provar {n} :
    ∀  (a : smaller n), ((fun x ⇒ x + 8)
        (((fun x ⇒ 2 * x) a) %% n)) %% n < n = (a < n).
```

Note que, se tratando `smaller n` como um conjunto de números naturais (apesar desse não ser precisamente isso) faria sentido poder utilizar o teorema `Exemplo_smaller_axiom` para reescrever o lado esquerdo da equação como `true`, ao invés de ter que utilizar um teorema mais específico como `Exemplo_aplicacao_f_mod`. Dado que se trata de uma sequência de funções em que a última função realizada é de resto da divisão por `n`, é óbvio que o resultado da expressão:

```
((fun x ⇒ x + 8) (((fun x ⇒ 2 * x) a) %% n)) %% n
```

é menor que `n`. Entretanto, a reescrita desejada não é possível pois ocorre um problema de unificação ao tentar se usar a tática `rewrite Exemplo_aplicacao_f_mod`. 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:

```
Check (∀ n (f : nat → nat) (a : smaller n),
    let LHS := [LHS of Exemplo_aplicacao_f_mod _] in
    let RDX := (((f a) %% n) < n) in
    [unify LHS with RDX]).
```

Este comando irá falhar apresentado uma mensagem, que em parte, apresenta o seguinte conteúdo:

```
Error: In environment
n : nat
f : nat → nat
a : smaller n
LHS := [LHS of Exemplo_smaller_axiom ?a] : bool
RDX := f a %% n < 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:

```
(x ?a) < n
```

e

```
(f x a %% n) < n
```

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

```
Definition f_mod_smaller {n : nat} (f : nat → nat) (a : smaller n) : smaller n :=
    Build n ((f a) %% n) (Ex2 f a).
Canonical f_mod_smaller.
```

Retornando então a prova de motivação para introdução ao problema discutido (`Exemplo_a_provar`) e utilizando a sequência de táticas:

```
intros. rewrite Exemplo_smaller_axiom.
```

O *goal* da prova se torna:

```
(((2 * a) %% n + 8) %% n < n) = true
```

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

```
(((2 * a) %% n + 8)
```

fique na forma de uma função aplicada sobre um elemento menor que `n`. Isso é necessário para que o *Coq* possa inferir um elemento do tipo `smaller n` dado pela definição `f_mod_smaller`, permitindo assim o uso do teorema `Exemplo_smaller_axiom`. Como 8 é de tipo `nat`, o *Coq* não consegue construir um tipo `smaller 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:

$$((8 + (2 * a) \ \%\% \ n) \ \%\% \ n < n) = true$$

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

$$true = true$$

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

# 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 9.

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 2 vezes nas páginas 7 e 8.

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 9.

HUYNH, E. *Rabin's Cryptosystem*. 39 p. Monografia (Bachelor) — Linnaeus University, Department of Mathematics, Suécia, 2021. Citado na página 8.

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 8.

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 8.

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 15 e 17.

MAHBOUBI, A.; TASSI, E. *Mathematical Components*. Zenodo, 2022. Disponível em: <https://doi.org/10.5281/zenodo.7118596>. Citado 6 vezes nas páginas 9, 11, 12, 13, 14 e 21.

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 8.

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 9.

NIVEN, I.; ZUCKERMAN, H. S. *An introduction to the theory of numbers*. Estados Unidos da América: John Wiley & Sons, Inc, 1991. Citado na página 8.

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 9.

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 8.

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 8.

TEAM, T. C. D. *The Coq Reference Manual*. France, 2024. Citado na página 12.

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 8.

WOOD, A. *A Casa Sonolenta*. Original. Estados Unidos da América: Editora Ática, 1999. (Abracadabra). ISBN 9788508032761. Citado na página 5.