

Tomaz Gomes Mascarenhas

# **Formalização da Complexidade Temporal de Algoritmos de Ordenação em Lean**

Belo Horizonte, Minas Gerais

2021

Tomaz Gomes Mascarenhas

# **Formalização da Complexidade Temporal de Algoritmos de Ordenação em Lean**

Universidade Federal de Minas Gerais  
Instituto de Ciências Exatas  
Departamento de Ciência da Computação

Supervisor: Haniel Barbosa

Belo Horizonte, Minas Gerais  
2021

# Contents

# 1 Introdução

## 1.1 História

### Teoria dos Tipos

Em 1901, o matemático Bertrand Russell descobriu uma contradição lógica nos fundamentos da teoria dos conjuntos, a qual formava a base da matemática naquela época. Essa descoberta veio da observação de que, como não existem limitações para o conteúdo de um conjunto, nada o impede de conter a si mesmo. De fato, o conjunto de todas as ideias é uma ideia, e deve estar contido em si mesmo, por exemplo. Desse modo, é possível separar os conjuntos em duas categorias: aqueles que contém e aqueles que não contém a si próprios. Formalmente, denotando por  $U$  o conjunto de todos os conjuntos, teríamos:

$$\begin{aligned} M &= \{A \mid A \in A\} \\ N &= \{A \mid A \notin A\} \\ U &= M \cup N \end{aligned}$$

Dessa definição surge uma dúvida:  $N$  contém a si próprio? Se ele não contiver, então, sua definição implicaria que ele deveria se conter. Por outro lado, se  $N \in N$ , ele estaria contrariando sua definição, já que  $N$  só aceita conjuntos que não contém a si mesmos. Em ambos os casos, temos uma contradição. Esse é um dos exemplos mais famosos de impredicatividade, isto é, um objeto que referencia a si mesmo.

Como uma resposta para esse problema, Russell criou o campo que ficou conhecido como Teoria dos Tipos. A ideia central era dividir os conjuntos em classes, que seriam representadas por números naturais, e estabelecer regras que tornem impossível um conjunto conter a si próprio. A classe de número 0 representa todos os conjuntos que não possuem conjuntos como elementos. A classe 1 se referia a todos os conjuntos que, se possuírem conjuntos como elementos, estes deverão pertencer a classe 0. De modo geral, conjuntos da classe  $i$  só podem conter conjuntos de classes  $j$ , tal que  $j < i$ . Desse modo, não podem existir conjuntos que contém a si próprios e o paradoxo deixa de existir. Embora tenha resolvido esse problema, a teoria é menos expressiva do que sua antecessora. Um exemplo disso é a impossibilidade de representar um tipo universal, que contenha todos os outros tipos. Esse tipo de limitação fez com que a teoria de Russell não fosse amplamente adotada pelos matemáticos.

Contudo, com o surgimento da Ciência da Computação, as ideias da Teoria dos Tipos encontraram uma nova aplicação nas Linguagens de Programação. A maioria delas adota algo que é conhecido como Sistema de Tipos. Nas palavras de Pierce (??), “Um sistema de tipos é um método sintático tratável para demonstrar a isenção de certos

comportamentos em um programa por meio da classificação de sentenças de acordo com as espécies de valores que elas computam”. Por exemplo, sem um sistema desses, a expressão “word” + 2 poderia ser considerada válida em uma linguagem, embora não esteja claro o que ela quer dizer. Atribuindo ao objeto “word” o tipo String, ao objeto 2 o tipo Integer e ao operador + a propriedade de que deve ser usado com objetos de mesmo tipo, por exemplo, é possível identificar essa expressão como inválida estaticamente, isto é, sem a necessidade de tentar executar o programa e receber um erro (note a semelhança entre essas restrições relacionadas a termos e as restrições estabelecidas por Russell, relacionadas a conjuntos).

Embora esse exemplo possa parecer trivial, em softwares grandes e com relações complexas entre classes podem surgir problemas muito mais sutis e que não se manifestariam imediatamente na execução.

## Isomorfismo de Curry-Howard

Em meados do século XX, foi descoberto que os Sistemas de Tipos possuíam uma relação muito profunda com a lógica. Considere, por exemplo, uma função  $f : A \rightarrow B$  sendo  $A$  e  $B$  tipos quaisquer. Uma regra clássica dos sistemas de tipo é algo como:

$$\frac{f : A \rightarrow B \quad a : A}{f(a) : B}$$

Ou seja, se  $f$  é uma função de  $A$  para  $B$  e  $a$  é um objeto do tipo  $A$ , então aplicar  $f$  em  $a$  produz algo do tipo  $B$ . Na lógica proposicional existe uma regra de inferência semelhante a isso. Se  $P$  e  $Q$  são proposições e  $P \Rightarrow Q$  denota que a proposição  $P$  implica em  $Q$ , a regra chamada Modus Ponens estabelece a seguinte relação:

$$\frac{P \Rightarrow Q \quad P}{Q}$$

Se possuímos uma demonstração de que a proposição  $P$  é verdadeira e que  $P$  implica em  $Q$ , então podemos usar essa regra para concluir que  $Q$  é verdadeiro.

De fato, esses dois sistemas não são apenas semelhantes, mas equivalentes. Note que a única diferença entre as duas regras é a natureza dos objetos aos quais elas se referem. Se identificarmos proposições como tipos e demonstrações como termos de um determinado tipo, então não existe diferença entre elas. Podemos interpretar a proposição  $P \Rightarrow Q$  como o tipo das funções que levam demonstrações de  $P$  em demonstrações de  $Q$ .

Um isomorfismo entre duas categorias de objetos matemáticos é um mapeamento entre objetos da primeira categoria e os da segunda, preservando suas propriedades. Essa correspondência recebeu o nome Isomorfismo de Curry-Howard, em homenagem aos pesquisadores que a descobriram. Cada elemento da lógica corresponde a alguma construção nas linguagens de programação. Alguns exemplos são:

Versão em Lógica	Versão em Programação
Proposição	Tipo
Demonstração	Termo do tipo correspondente
Indução	Recursão
Conjunção	Produto de tipos
Disjunção	União de tipos
Implicação	Função
Negação	Função para um tipo vazio

## Assistentes de Demonstração

Essa equivalência contribuiu para a criação dos **Assistentes de Demonstração**. Estes são sistemas que dão suporte a certas linguagens de programação, fazendo com que elas concretizem essa correspondência e seja possível usá-las para demonstrar teoremas por meio da escrita de programas. Tais demonstrações podem ser verificadas mecanicamente pelo compilador da linguagem, fazendo com que o grau de confiabilidade que elas atinjam seja muito superior ao de demonstrações convencionais. Existem assistentes de demonstração que não usam o Isomorfismo de Curry-Howard, mas outros modelos para implementar as mesmas funcionalidades. Um desses modelos é o *Logic for Computable Functions* (??).

De acordo com Geuvers (??) (p. 4): “Uma demonstração matemática pode ser reduzida a uma série de etapas muito breves que podem ser verificadas de forma simples e irrefutável”. Tais etapas são tão pequenas que quase sempre são resumidas em demonstrações, e sua correteza fica subentendida. No entanto, é possível que esse resumo omita alguma falha lógica, invalidando toda a demonstração de uma forma difícil de ser detectada, como já aconteceu diversas vezes na história da matemática.

Ao usar um assistente, não é possível omitir tais partes da demonstração dessa maneira, tornando sua escrita mais trabalhosa. Contudo, ele verifica cada passo da dedução e o valida, tornando impossível que erros lógicos sejam aceitos. Por causa do isomorfismo de Curry-Howard, o compilador é capaz de tratar cada passo lógico como a aplicação de uma função de um certo tipo, assim, caso exista uma inconsistência no argumento usado ele será capaz de identificar uma inconsistência entre os tipos dos objetos, resultando em um erro de compilação.

## 1.2 Aplicações

Talvez o caso mais famoso de aplicação dos assistentes seja a demonstração do Teorema das Quatro Cores. Tal teorema afirma que, dado qualquer mapa plano, dividido em regiões, sempre é possível pintá-lo com no máximo quatro cores, de modo que duas regiões vizinhas nunca possuam a mesma cor. Esse fato foi conjecturado pelo matemático

Francis Guthrie em 1852 e permaneceu por mais de 100 anos sem ser demonstrado, até que em 1976, sua veracidade foi verificada, com a ajuda de um computador IBM 360. A demonstração consistia em reduzir todos os mapas possíveis em 1936 configurações, de modo que para demonstrar o teorema bastaria verificar uma por uma delas e garantir que nunca são necessárias mais do que quatro cores para pintá-las. A única possibilidade de realizar tal tarefa foi por meio da ajuda de computadores, e, mesmo os mais eficazes da época levaram mais de mil horas para isso. O uso desse tipo de técnica foi questionado por parte da comunidade matemática, principalmente pelo fato de ser necessário acreditar que os softwares utilizados para colorir e gerar os mapas estariam corretos. Anos depois, em 2005, a demonstração foi formalizada dentro de um assistente de demonstração chamado Coq (??), fazendo com que fosse necessário acreditar apenas que seu kernel estaria correto. Existem divergências quanto ao uso desse tipo de demonstração na matemática, mas a comunidade no geral tem se mostrado confiante nessa ferramenta. Por exemplo, 29 dos 62 participantes de um evento relacionado a um assistente chamado Lean (??) se identificaram como matemáticos (??) (p. 8).

Um outro campo que também tem se beneficiado desse mecanismo é a Engenharia de Software. Uma das propriedades mais desejáveis quando se desenvolve programas é a corretude, ou seja, a garantia de que, para qualquer entrada que receba, o software irá respeitar alguma especificação fornecida previamente. Usando linguagens com suporte a assistentes, é possível escrever funções, demonstrar sua corretude formalmente e exportá-la para uma outra linguagem mais usual, garantindo sua corretude. Em softwares de missão crítica, esse tipo de prática pode evitar prejuízos catastróficos.

## 1.3 Contexto

O objetivo deste trabalho é usar Lean, um dos assistentes de demonstração mais relevantes atualmente, para formalizar algum conceito que ainda não tenha sido amplamente explorado no contexto dos assistentes e que seja relevante para a comunidade. De acordo com membros ativos da comunidade que utiliza Lean<sup>1</sup>, a complexidade temporal de algoritmos é um conceito relevante que ainda não foi formalizado em nenhum módulo de sua biblioteca de formalizações. Em particular, a complexidade temporal de algoritmos básicos de ordenação, que poderia ser formalizada sem usar técnicas muito avançadas. Assim, o escopo do trabalho será a demonstração de um limite, com base no tamanho da entrada, para o número de operações feitas pelos algoritmos Insertion Sort e Merge Sort, implementados na biblioteca do Lean<sup>2</sup>.

Na seção 2, apresentamos os conceitos básicos que são abordados por este trabalho. Na seção 3, definimos matematicamente os algoritmos que serão formalizados. Em seguida,

<sup>1</sup> <<https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/BSc.20Final.20Project>>

<sup>2</sup> <<https://github.com/leanprover-community/mathlib/blob/master/src/data/list/sort.lean>>

na seção 4 apresentamos demonstrações convencionais dos principais teoremas que estão no escopo do trabalho, usando as definições da seção anterior. Finalmente, na seção 5, mostramos a implementação em Lean dos algoritmos e dos teoremas, junto com as principais ideias necessárias para escrever suas demonstrações no assistente.



## 2 Conceitos Básicos

### 2.1 Algoritmos de Ordenação

Um algoritmo de ordenação é um método que recebe como entrada uma lista de objetos comparáveis e retorna uma permutação da lista, na qual os objetos estão ordenados. Formalmente, para um método  $f : [A] \rightarrow [A]$  ser considerado um algoritmo de ordenação correto, ele deve satisfazer, para qualquer lista  $xs$ :

- $[f(xs)]_i \leq [f(xs)]_j$ , sempre que  $i \leq j$ .
- $count(x, xs) = count(x, f(xs))$ ,  $\forall x \in xs$ . Nesse caso,  $count(y, ys)$  é a função que conta o número de ocorrências do objeto  $y$  na lista  $ys$ .

Os seguintes exemplos mostram saídas corretas, considerando o problema da ordenação, para a entrada indicada:

- Entrada:  $[9, 7, 10, 2, 4]$ , Saída:  $[2, 4, 7, 9, 10]$
- Entrada:  $[1, 2, 1, 3]$ , Saída:  $[1, 1, 2, 3]$

Por outro lado, o seguinte exemplo mostra um resultado incorreto, já que temos duas ocorrências do número 2 na lista de entrada, e apenas uma na lista de saída.

- Entrada:  $[2, 2, 4, 1]$ , Saída:  $[1, 2, 4]$

Existem diversos algoritmos de ordenação conhecidos. Esse trabalho irá se restringir ao Insertion Sort e o Merge Sort (??). Essa escolha foi feita por causa de restrições de tempo para o trabalho, e pelo fato de, esses dois serem os únicos algoritmos de ordenação formalizados pela biblioteca do Lean, o que os torna os mais interessantes do ponto de vista da comunidade do assistente.

### 2.2 Complexidade Temporal

Se forem analisados como funções matemáticas, todos os algoritmos de ordenação corretos são equivalentes. Contudo, do ponto de vista computacional eles podem se diferenciar no tempo necessário para resolver o problema da ordenação.

Para que se possa avaliar formalmente a eficiência de um algoritmo, deve-se definir uma métrica que associe a execução dele em uma entrada a um número natural, tal que este esteja associado à quantidade de instruções que um computador teria que executar

para realizar aquele algoritmo naquela entrada. A métrica mais comum para algoritmos de ordenação é o número de comparações feitas entre os elementos da lista que ele está ordenando.

Uma vez que a métrica foi explicitada, deve-se encontrar alguma relação entre o seu valor para uma entrada específica e o tamanho daquela entrada. Por exemplo, no caso dos algoritmos de ordenação iremos comparar o número de comparações executadas com o número de elementos da lista de entrada. Será demonstrado que, para uma lista de tamanho  $n$ , o Insertion Sort nunca utiliza mais do que  $n^2$  comparações, enquanto o Merge Sort não passa de  $8n \log_2(n)$ . Esses dois fatos, juntos com exemplos em que os algoritmos atingem esses limites, mostram que o Merge Sort é mais eficiente do que o Insertion Sort.

Na literatura, existem demonstrações de que o Merge Sort utiliza menos do que  $n \log_2(n)$  comparações no pior caso (??), o que é um limite mais forte do que o apresentado neste trabalho. Contudo, na literatura é considerada uma implementação que usa um vetor em vez de uma lista, fazendo com que não seja necessário usar operações para dividir a lista em duas. No caso do Insertion Sort, sabe-se que ele nunca faz mais do que  $n(n-1)/2$  comparações, o que é um limite um pouco mais forte do que o deste trabalho.

## 2.3 Polimorfismo Paramétrico

Existem diversas linguagens de programação que suportam polimorfismo sobre seus tipos. Isso significa que é possível definir funções sem fixar concretamente alguns dos tipos sobre a qual elas agem, fazendo com que se possa usá-las em objetos de qualquer tipo. Além disso, algumas linguagens aceitam que, nessas funções, se restrinja os tipos aceitos àqueles que possuam alguma propriedade. Esta é uma variação do polimorfismo, conhecido como polimorfismo paramétrico.

Como foi mencionado no início dessa seção, os algoritmos de ordenação que estamos considerando agem sobre listas de objetos de qualquer tipo, desde que seja possível comparar elementos desse tipo. Para expressar essa restrição nas linguagens de programação, usa-se o polimorfismo paramétrico, restringindo os tipos aceitos àqueles que implementam alguma função que compare elementos daquele tipo. Para que o algoritmo funcione corretamente, essa função deve induzir uma ordem total sobre o tipo. Isso significa que, se a função de comparação sobre um tipo  $T$  é definida pelo operador binário “ $\leq$ ”, as seguintes propriedades devem valer:

- $\forall t : T, \quad t \leq t$
- $\forall t_1, t_2 : T, \quad t_1 \leq t_2 \wedge t_2 \leq t_1 \leftrightarrow t_1 = t_2$
- $\forall t_1, t_2, t_3 : T, \quad t_1 \leq t_2 \wedge t_2 \leq t_3 \rightarrow t_1 \leq t_3$

Além disso, a função deve ser total, isto é, deve estar definida para todos os pares de elementos daquele tipo. Sem essa restrição, podem existir listas para as quais não faz sentido definir se estão ordenadas ou não, o que torna impossível o desenvolvimento de algoritmos para resolver esse problema.

## 2.4 Indução Transfinita

Uma tática muito comum para demonstrar teoremas é a indução. Dado um predicado  $P(n)$  (que depende do número natural  $n$ ), ao se demonstrar a veracidade de  $P(0)$ , e de  $P(k) \Rightarrow P(k+1)$ , sendo  $k$  um número natural qualquer, o princípio da indução matemática afirma que  $P(n)$  é verdadeiro para todo  $n$ . Neste trabalho, será usado amplamente uma versão generalizada desse princípio, conhecida como Indução Transfinita. Nesse caso, deve-se supor que  $P(k)$  é verdadeiro para todo  $k < n$ . Se essa hipótese implicar na veracidade de  $P(n)$ , então, pelo princípio de Indução Transfinita, obtêm-se a veracidade de  $P(m)$ , para todo  $m$  natural.

Eventualmente será necessário usar a indução com uma hipótese da forma  $P(f(n))$ , para algum predicado  $P$  e função  $f : \mathbb{N} \rightarrow \mathbb{N}$ . Isso é válido, desde que se demonstre que  $f$  é estritamente decrescente. Com essa propriedade, temos a garantia que  $f(n) < n$ , logo a hipótese de indução se aplica a  $f(n)$ . Caso exista algum  $n$  tal que  $f(n) \geq n$ , estaríamos usando uma hipótese que não temos para demonstrar o teorema, o que é inválido.

## 2.5 Lean

Lean (??) é um assistente de demonstrações. Estes são linguagens de programação que possibilitam que seu usuário enuncie e demonstre teoremas, os quais são verificados pelo compilador do assistente, que é capaz de determinar se a demonstração é válida ou não.

Assim como muitos outros assistentes, Lean é baseado em **tipos dependentes** (??). Ou seja, cada proposição que se enuncia dentro da linguagem é representada internamente por um tipo, o qual depende de valores de outros tipos. Um exemplo para essa idéia é a igualdade. Ao enunciar um teorema como  $a = b$ , sendo  $a, b : T$ , o símbolo “=” age como um construtor de tipos, que toma dois valores do tipo  $T$  como parâmetro e retorna o tipo relativo a igualdade entre os dois valores. Nesse contexto, construir um termo do tipo “ $a = b$ ” equivale a demonstrar a igualdade entre esses dois valores.

O processo de demonstrar teoremas usando apenas tipos dependentes se resume, na maior parte das vezes, a compor funções e analisar casos para conseguir um termo de um determinado tipo, algo que é muito difícil de ser praticado em teoremas que envolvem uma longa argumentação, além de produzir demonstrações pouco legíveis. O modo convencional de realizar essa tarefa é muito diferente. Nele, o matemático pode

estruturar a demonstração da forma que preferir, sendo possível organizar ideias complexas de forma menos confusa, o que facilita o trabalho de quem está escrevendo e de quem irá ler o argumento depois. Lean fornece abstrações ao usuário para que o processo de demonstrar teoremas se torne semelhante ao convencional. Tais abstrações são conhecidas como táticas. Além de prover as táticas correspondentes as técnicas de demonstração mais comuns (como indução, demonstração por contradição, etc.), o assistente também permite que o usuário desenvolva suas próprias táticas, por meio de metaprogramação (??).

Lean também apresenta um certo nível de automatização de demonstrações. Dado um certo fato que deseja-se demonstrar, é possível requisitar ao assistente que se pesquise na biblioteca de teoremas já demonstrados alguma combinação que, ou sirva como demonstração direta do que se pretende demonstrar, ou funcione como uma etapa intermediária, simplificando o fato a ser demonstrado. Tal funcionalidade é especialmente útil se usada em conjunto com a biblioteca mantida pela comunidade, conhecida como *mathlib* (??). Nela são mantidas formalizações extensas de diversas áreas da matemática e da computação, as quais podem ser usadas livremente para ajudar nas demonstrações de teoremas que o usuário desejar. Em particular, nessa biblioteca existe a formalização dos algoritmos de ordenação, as quais serão usadas ao longo deste trabalho. Tais formalizações não incluem propriedades relacionadas a complexidade temporal desses algoritmos. Isso não acontece apenas nesses dois algoritmos, esse tipo de propriedade ainda não foi formalizada para nenhum algoritmo dessa biblioteca.

O seguinte código apresenta a definição dos números naturais, da adição sobre eles e a demonstração da comutatividade dessa operação em Lean, junto com dois lemas necessários para essa demonstração:

## 3 Definições dos Algoritmos

Como foi mencionado, nesse trabalho será abordada a complexidade temporal de dois algoritmos: Insertion Sort e Merge Sort. Nessa seção eles serão definidos em linguagem matemática.

A definição original desses algoritmos (que é usada pela *mathlib*) não registra a quantidade de comparações que são feitas, o que torna impossível que se demonstre teoremas sobre esses números em Lean. Por causa disso, definimos novas funções que produzem, além de listas, números naturais que determinam o número de operações executadas.

Este trabalho irá apresentar demonstrações que as listas produzidas por essas novas funções são idênticas às produzidas pelas funções definidas originalmente, dado que elas receberam a mesma entrada. Além disso, serão definidas funções que usam a estrutura da lista para determinar exatamente quantas comparações são efetivamente realizadas pelos algoritmos. Com isso, demonstraremos que o número natural retornado pelo algoritmo modificado está contando essas comparações corretamente. Finalmente, demonstraremos um limite para esse número, em função do tamanho da lista de entrada. Considerando uma lista de tamanho  $n$ , mostraremos que o Insertion Sort se limita a  $n^2$  comparações, enquanto o Merge Sort nunca passa de  $8n \log_2(n)$ .

Para definir os algoritmos, é necessário definir formalmente como se constroem as listas que serão usadas. Usaremos o símbolo  $[]$  para indicar uma lista sem nenhum elemento e o símbolo  $::$  para indicar a construção de uma lista, que terá como primeiro elemento a entidade a esquerda do símbolo, seguido pela lista de elementos a direita dele. Por exemplo, a construção  $3 :: 2 :: 6 :: []$  representa a lista que contém o número 3, seguido por 2, seguido por 6.

### 3.1 Insertion Sort

A seguir apresentamos a definição original da função auxiliar do Insertion Sort, que chamamos de *insert*. Dado um elemento  $x$  e uma lista ordenada  $xs$ ,  $insert(x, xs)$  produz uma nova lista, inserindo  $x$  em  $xs$  de modo que o resultado permaneça ordenado. Assim como todas as outras funções que serão definidas, sua definição é dada por meio de casamento de padrões, dependendo da forma da lista de entrada. No caso da lista ter mais de um elemento, usamos a função recursivamente.

$$insert(x, []) = [x] \tag{3.1}$$

$$\text{insert}(x, y : ys) = \begin{cases} x :: y :: ys, & \text{se } x \leq y \\ y :: \text{insert}(x, ys), & \text{caso contrário} \end{cases} \quad (3.2)$$

A versão estendida funciona exatamente da mesma forma, mas produz também um número natural, que conta o número de operações realizadas.

$$\text{insertExtended}(x, []) = ([x], 0) \quad (3.3)$$

$$\text{insertExtended}(x, y :: ys) = \begin{cases} (x :: y :: ys, 1), & \text{se } x \leq y \\ (y :: l, 1 + n), & \text{caso contrário,} \\ \quad \textbf{onde: } (l, n) := \text{insertExtended}(x, ys) \end{cases} \quad (3.4)$$

Usando *insert*, podemos definir o Insertion Sort. Essa função recebe uma lista *xs* e produz uma permutação ordenada de *xs*. As equações ?? e ?? representam sua versão original, enquanto ?? e ?? representam sua versão estendida.

$$\text{insertionSort}([]) = [] \quad (3.5)$$

$$\text{insertionSort}(x :: xs) = \text{insert}(x, \text{insertionSort}(xs)) \quad (3.6)$$

$$\text{insertionSortExtended}([]) = ([], 0) \quad (3.7)$$

$$\begin{aligned} \text{insertionSortExtended}(x :: xs) &= (l', n + m), \\ \textbf{onde: } (l, m) &:= \text{insertionSortExtended}(xs) \\ (l', n) &:= \text{insertExtended}(x, l) \end{aligned} \quad (3.8)$$

## 3.2 Merge Sort

Agora vamos definir o Merge Sort. Para isso, precisaremos de duas funções auxiliares. A primeira, chamada *split*, recebe uma lista e produz um par de listas, na qual a primeira contém todos os elementos em posições pares da lista de entrada e a segunda os elementos de posição ímpar. As equações ?? e ?? representam a definição original e ?? e ?? a versão estendida. Note que, apesar do *split* não executar nenhuma comparação, é necessário contar quantas vezes ele é usado, pois este número tem a mesma ordem de grandeza do tamanho da lista, logo essa função pode interferir na performance do Merge Sort.

$$\text{split}([]) = ([], []) \quad (3.9)$$

$$\begin{aligned} \text{split}(x :: xs) &= (x :: l_2, l_1), \\ \text{onde: } (l_1, l_2) &:= \text{split}(xs) \end{aligned} \quad (3.10)$$

$$\text{splitExtended}([]) = ([], [], 0) \quad (3.11)$$

$$\begin{aligned} \text{splitExtended}(x :: xs) &= (x :: l_2, l_1, 1 + n) \\ \text{onde: } (l_1, l_2, n) &:= \text{splitExtended}(xs) \end{aligned} \quad (3.12)$$

A segunda função auxiliar, que chamamos de *merge*, recebe duas listas ordenadas e produz uma nova lista, intercalando todos os elementos das listas de entrada, de modo que a lista final esteja ordenada. As equações ??, ?? e ?? são sua versão original, e as equações ??, ?? e ?? são sua versão estendida.

$$\text{merge}(xs, []) = xs \quad (3.13)$$

$$\text{merge}([], ys) = ys \quad (3.14)$$

$$\text{merge}(x :: xs, y :: ys) = \begin{cases} x :: \text{merge}(xs, y :: ys), & \text{se } x \leq y \\ y :: \text{merge}(x :: xs, ys), & \text{caso contrário} \end{cases} \quad (3.15)$$

$$\text{mergeExtended}(xs, []) = (xs, 0) \quad (3.16)$$

$$\text{mergeExtended}([], ys) = (ys, 0) \quad (3.17)$$

$$\begin{aligned} \text{mergeExtended}(x :: xs, y :: ys) &= \begin{cases} (x :: l_1, 1 + n), & \text{se } x \leq y \\ (y :: l_2, 1 + m), & \text{caso contrário} \\ \text{onde: } (l_1, n) := \text{mergeExtended}(xs, y :: ys) \\ \quad (l_2, m) := \text{mergeExtended}(x :: xs, ys) \end{cases} \end{aligned} \quad (3.18)$$

Com essas duas funções, podemos finalmente definir o Merge Sort. Assim como o Insertion Sort, o Merge Sort recebe uma lista *xs* e produz uma permutação ordenada de *xs*. As equações ??, ?? e ?? representam sua definição original, enquanto as equações ??, ?? e ?? representam sua versão estendida.

$$\text{mergeSort}([]) = [] \quad (3.19)$$

$$\text{mergeSort}([x]) = [x] \quad (3.20)$$

$$\begin{aligned} \text{mergeSort}(x_1 :: x_2 :: xs) &= \text{merge}(l'_1, l'_2), \\ \text{onde: } (l_1, l_2) &:= \text{split}(x_1 :: x_2 :: xs), \\ l'_1 &:= \text{mergeSort}(l_1), \\ l'_2 &:= \text{mergeSort}(l_2) \end{aligned} \quad (3.21)$$

$$\text{mergeSortExtended}([]) = ([], 0) \quad (3.22)$$

$$\text{mergeSortExtended}([x]) = ([x], 0) \quad (3.23)$$

$$\begin{aligned} \text{mergeSortExtended}(x_1 :: x_2 :: xs) &= (l, n + m_1 + m_2 + p), \\ \text{onde: } (l_1, l_2, n) &:= \text{splitExtended}(x_1 :: x_2 :: xs), \\ (l'_1, m_1) &:= \text{mergeSortExtended}(l_1), \\ (l'_2, m_2) &:= \text{mergeSortExtended}(l_2), \\ (l, p) &:= \text{mergeExtended}(l'_1, l'_2) \end{aligned} \quad (3.24)$$

### 3.3 Funções para Certificação

Também é necessário garantir que o número que estamos calculando está capturando corretamente nossa intenção, que é contar o número de operações realizadas por cada algoritmo. Para isso, também precisamos definir funções que calculam, para uma determinada lista, quantas operações os algoritmos fazem.

Sabemos que *insert* realiza uma comparação para cada elemento do prefixo da lista de entrada que é menor do que o elemento que estamos inserindo, e mais uma para o primeiro que não é menor (caso ele exista). A seguinte função usa essa informação para produzir o número exato de comparações:

$$\text{comparisonsInsert}(x, []) = 0 \quad (3.25)$$

$$\text{comparisonsInsert}(x, y :: ys) = \begin{cases} 1, & \text{se } x \leq y \\ 1 + \text{comparisonsInsert}(x, ys), & \text{caso contrário} \end{cases} \quad (3.26)$$

No caso do *merge*, para cada elemento da primeira lista, devemos encontrar o primeiro elemento da segunda que não é menor do que ele. Ao encontrarmos, repetimos



o processo com o próximo elemento da primeira lista e o sufixo que sobrou da segunda lista. A seguir, definimos uma função auxiliar para contar as operações do *merge*. Ela recebe um elemento e uma lista ordenada, e remove o prefixo da lista que é menor do que o elemento recebido:

$$\text{removePrefix}(x, []) = [] \quad (3.27)$$

$$\text{removePrefix}(x, y :: ys) = \begin{cases} y :: ys, & \text{se } x \leq y \\ \text{removePrefix}(x, ys), & \text{caso contrário} \end{cases} \quad (3.28)$$

Agora vamos definir a função que calcula o número de comparações em uma execução do *merge*. Note que, pela similiaridade entre o *insert* e as operações que o *merge* realiza em cada um dos elementos, podemos usar a função *comparisonsInsert* para ajudar a contar esse número.

$$\text{comparisonsMerge}(xs, []) = 0 \quad (3.29)$$

$$\text{comparisonsMerge}([], ys) = 0 \quad (3.30)$$

$$\begin{aligned} \text{comparisonsMerge}(x :: xs, ys) &= \text{comparisonsInsert}(x, ys) + \text{comparisonsMerge}(xs, ys') \\ \text{onde: } ys' &:= \text{removePrefix}(x, ys) \end{aligned} \quad (3.31)$$

## 4 Demonstrações Convencionais

Nesta seção iremos apresentar e demonstrar, de forma convencional, os principais teoremas deste trabalho. Isso é importante pois a argumentação utilizada nessas demonstrações é essencialmente a mesma que a do código que apresentaremos na seção 5, assim, poderemos focar apenas no código em si na próxima seção. Além disso, essa apresentação irá tornar mais clara as semelhanças entre as demonstrações convencionais e o processo de demonstrar teoremas em assistentes como o Lean. Os fatos que estamos interessados em formalizar são os seguintes:

- O primeiro elemento do par retornado nas definições estendidas é igual ao retorno das definições originais correspondentes.
- O segundo elemento do par retornado nas definições estendidas é limitado por uma função (a ser especificada) do tamanho da lista de entrada.
- O segundo elemento do par retornado nas definições estendidas é igual ao retorno das funções de certificação correspondentes.

Ao longo dos teoremas, usaremos  $A$  para indicar um tipo genérico que possui uma ordem total induzida pelo operador “ $\leq$ ”. Quase todas as demonstrações são feitas por meio de indução na lista de entrada, portanto, omitiremos essa informação nelas. Sempre usaremos  $n$  para indicar  $length(l)$ ,  $n_1$  para indicar  $length(l_1)$  e  $n_2$  para indicar  $length(l_2)$ .

### 4.1 Insertion Sort

**Teorema 4.1.1.**  $\forall a : A, l : [A], \text{proj}_1(\text{insertExtended}(a, l)) = \text{insert}(a, l)$

*Demonstração.*

**Passo Base:**  $l = []$ .

$$\begin{aligned}
 \text{proj}_1(\text{insertExtended}(a, [])) &= \text{proj}_1([a], 0) && \text{Pela equação ??} \\
 &= [a] && \text{Pela definição de } \text{proj}_1 \\
 &= \text{insert}(a, []) && \text{Pela equação ??}
 \end{aligned}$$

**Passo Indutivo:**  $l = h :: t$ .

**Hipótese Indutiva:**  $\text{proj}_1(\text{insertExtended}(a, t)) = \text{insert}(a, t)$ .

Caso 1:  $a \leq h$

$$\begin{aligned}
 \text{proj}_1(\text{insertExtended}(a, h :: t)) &= \text{proj}_1(a :: h :: t, \star) && \text{Pela equação ??} \\
 &= a :: h :: t && \text{Pela definição de } \text{proj}_1 \\
 &= \text{insert}(a, h :: t) && \text{Pela equação ??}
 \end{aligned}$$

Caso 2:  $a \not\leq h$

$$\begin{aligned}
& \text{proj}_1(\text{insertExtended}(a, h :: t)) \\
&= \text{proj}_1(h :: \text{proj}_1(\text{insertExtended}(a, t)), \star) && \text{Pela equação ??} \\
&= \text{proj}_1(h :: \text{insert}(a, t), \star) && \text{Por HI} \\
&= h :: \text{insert}(a, t) && \text{Pela definição de } \text{proj}_1 \\
&= \text{insert}(a, h :: t) && \text{Pela equação ??}
\end{aligned}$$

□

**Teorema 4.1.2.**  $\forall l : [A], \text{proj}_1(\text{insertionSortExtended}(l)) = \text{insertionSort}(l)$

*Demonstração.*

**Passo Base:**  $l = []$

$$\begin{aligned}
\text{proj}_1(\text{insertionSortExtended}([])) &= \text{proj}_1([], 0) && \text{Pela equação ??} \\
&= [] && \text{Pela definição de } \text{proj}_1 \\
&= \text{insertionSort}([]) && \text{Pela equação ??}
\end{aligned}$$

**Passo Indutivo:**  $l = h :: t$

**Hipótese Indutiva:**  $\text{proj}_1(\text{insertionSortExtended}(a, t)) = \text{insertionSort}(a, t)$ .

Note: definimos  $xs := \text{proj}_1(\text{insertionSortExtended}(t))$

$$\begin{aligned}
& \text{proj}_1(\text{insert\_sort\_extended}(h :: t)) \\
&= \text{proj}_1(\text{proj}_1(\text{insertExtended}(h, xs)), \star) && \text{Pela equação ??} \\
&= \text{proj}_1(\text{insert}(h, xs), \star) && \text{Pelo teorema ??} \\
&= \text{insert}(h, xs) && \text{Pela definição de } \text{proj}_1 \\
&= \text{insert}(h, \text{proj}_1(\text{insertionSortExtended}(t))) && \text{Pela definição de } xs \\
&= \text{insert}(h, \text{insertionSort}(t)) && \text{Por HI} \\
&= \text{insertionSort}(h :: t) && \text{Pela equação ??}
\end{aligned}$$

□

**Teorema 4.1.3.**  $\forall a : A, l : [A], \text{proj}_2(\text{insertExtended}(a, l)) \leq n$

*Demonstração.*

**Passo Base:**  $l = []; n = 0$

$$\begin{aligned}
\text{proj}_2(\text{insertExtended}(a, [])) &= \text{proj}_2(\star, 0) && \text{Pela equação ??} \\
&= 0 && \text{Pela definição de } \text{proj}_2
\end{aligned}$$

**Passo Indutivo:**  $l = h :: t; n = 1 + n'$

**Hipótese Indutiva:**  $\text{proj}_2(\text{insertExtended}(a, t)) \leq n'$

Caso 1:  $a \leq h$

$$\begin{aligned} \text{proj}_2(\text{insertExtended}(a, h :: t)) &= \text{proj}_2(\star, 1) && \text{Pela equação ??} \\ &= 1 && \text{Pela definição de } \text{proj}_2 \end{aligned}$$

Como  $l$  é da forma  $h :: t$ , ela tem pelo menos um elemento, logo a desigualdade vale.

Caso 2:  $a \not\leq h$

$$\begin{aligned} &\text{proj}_2(\text{insertExtended}(a, h :: t)) \\ &= \text{proj}_2(\star, 1 + \text{proj}_2(\text{insertExtended}(a, t))) && \text{Pela equação ??} \\ &= 1 + \text{proj}_2(\text{insertExtended}(a, t)) && \text{Pela definição de } \text{proj}_2 \\ &\leq 1 + n' && \text{Por HI} \\ &= n && \text{Pela definição de } n' \end{aligned}$$

□

**Teorema 4.1.4.**  $\forall l : [A], \text{proj}_2(\text{insertionSortExtended}(l)) \leq n^2$

*Demonstração.*

**Passo Base:**  $l = []; n = 0$

$$\begin{aligned} \text{proj}_2(\text{insertionSortExtended}([])) &= \text{proj}_2(\star, 0) && \text{Pela equação ??} \\ &= 0 && \text{Pela definição de } \text{proj}_2 \end{aligned}$$

**Passo Indutivo:**  $l = h :: t; n = 1 + n'$

**Hipótese Indutiva:**  $\text{proj}_2(\text{insertionSortExtended}(t)) \leq (n')^2$

Note: definimos  $m := \text{proj}_2(\text{insertExtended}(h, t))$

$$\begin{aligned} &\text{proj}_2(\text{insertionSortExtended}(h :: t)) \\ &= \text{proj}_2(\star, \text{proj}_2(\text{insertionSortExtended}(t)) + m) && \text{Pela equação ??} \\ &= \text{proj}_2(\text{insertionSortExtended}(t)) + m && \text{Pela definição de } \text{proj}_2 \\ &\leq (n')^2 + m && \text{Por IH} \\ &\leq (n')^2 + n' && \text{Pelo teorema ??} \\ &= n'(n' + 1) && \text{Por distributividade} \\ &= n'(n) && \text{Pela definição de } n' \\ &< n^2 && \text{Pois } n' < n \end{aligned}$$

□

## 4.2 Merge Sort

**Teorema 4.2.1.**  $\forall l : [A], \text{proj}_{1,2}(\text{splitExtended}(l)) = \text{split}(l)$

*Demonstração.*

**Passo Base:**  $l = []$

$$\begin{aligned} \text{proj}_{1,2}(\text{splitExtended}([])) &= \text{proj}_{1,2}([], [], 0) && \text{Pela equação ??} \\ &= ([], []) && \text{Pela definição de } \text{proj}_{1,2} \\ &= \text{split}([]) && \text{Pela equação ??} \end{aligned}$$

**Passo Indutivo:**  $l = h :: t$

**Hipótese Indutiva:**  $\text{proj}_{1,2}(\text{splitExtended}(t)) = \text{split}(t)$

Note: definimos  $(l_1, l_2, \star) := \text{splitExtended}(t)$

$$\begin{aligned} \text{proj}_{1,2}(\text{splitExtended}(h :: t)) &= \text{proj}_{1,2}(h :: l_2, l_1, \star) && \text{Pela equação ??} \\ &= (h :: l_2, l_1) && \text{Pela definição de } \text{proj}_{1,2} \\ &= \text{split}(h :: t') && \text{Pela equação ??} \end{aligned}$$

No último termo,  $t'$  indica alguma lista tal que  $\text{split}(t') = (l_1, l_2)$ . Por hipótese indutiva,  $t$  satisfaz esse requerimento. Além disso,  $\text{split}$  é claramente uma função injetiva. Logo,  $t$  é a única lista que satisfaz essa igualdade, e temos  $\text{split}(h :: t') = \text{split}(h :: t)$ .

□

**Teorema 4.2.2.**  $\forall l_1, l_2 : [A], \text{proj}_1(\text{mergeExtended}(l_1, l_2)) = \text{merge}(l_1, l_2)$

*Demonstração.*

**Passo Base:**  $l_1 = []$

$$\begin{aligned} \text{proj}_1(\text{mergeExtended}([], l_2)) &= \text{proj}_1(l_2, \star) && \text{Pela equação ??} \\ &= l_2 && \text{Pela definição de } \text{proj}_1 \\ &= \text{merge}([], l_2) && \text{Pela equação ??} \end{aligned}$$

**Passo Indutivo:**  $l_1 = h_1 :: t_1$

**Hipótese Indutiva:**  $\forall l_2 : [A], \text{proj}_1(\text{mergeExtended}(t_1, l_2)) = \text{merge}(t_1, l_2)$

Para demonstrar o passo indutivo, faremos indução na segunda lista:

**Passo Base 2:**  $l_2 = []$

$$\begin{aligned}
proj_1(mergeExtended(h_1 :: t_1, [])) &= proj_1(h_1 : t_1, \star) && \text{Pela equação ??} \\
&= h_1 :: t_1 && \text{Pela definição de } proj_1 \\
&= merge(h_1 :: t_1, []) && \text{Pela equação ??}
\end{aligned}$$

**Passo Indutivo 2:**  $l_2 = h_2 :: t_2$

**Hipótese Indutiva 2:**  $\forall l_1 : [A], proj_1(mergeExtended(l_1, t_1)) = merge(l_1, t_1)$

Note: definimos  $l'_1 := proj_1(mergeExtended(t_1, h_2 :: t_2))$

Caso 1:  $h_1 \leq h_2$

$$\begin{aligned}
proj_1(mergeExtended(h_1 :: t_1, h_2 :: t_2)) & \\
&= proj_1(h_1 :: l'_1, \star) && \text{Pela equação ??} \\
&= h_1 :: l'_1 && \text{Pela definição de } proj_1 \\
&= h_1 :: merge(t_1, h_2 :: t_2) && \text{Por HI} \\
&= merge(h_1 :: t_1, h_2 :: t_2) && \text{Pela equação ??}
\end{aligned}$$

Observe que, na terceira igualdade estamos usando a primeira hipótese de indução, tomando  $l_2 = h_2 :: t_2$ . A demonstração do segundo caso, em que  $h_1 \not\leq h_2$ , é totalmente análoga, usando a segunda hipótese de indução.

□

Os seguintes lemas serão úteis em alguns dos próximos teoremas:

**Lema 4.2.1.**  $\forall l : [A], length(proj_1(splitExtended(l))) \leq length(l) \wedge$   
 $length(proj_2(splitExtended(l))) \leq length(l)$

*Demonstração.*

**Passo Base:**  $l = []$

$$\begin{aligned}
length(proj_1(splitExtended([]))) &= length(proj_1([], \star, \star)) && \text{Pela equação ??} \\
&= length([]) && \text{Pela definição de } proj_1 \\
&= 0 && \text{Pela definição de } length
\end{aligned}$$

A demonstração do segundo item é análoga.

**Passo Indutivo:**  $l = h :: t$

**Hipótese Indutiva:**  $length(proj_1(splitExtended(t))) \leq length(t) \wedge$   
 $length(proj_2(splitExtended(t))) \leq length(t)$

Note: definimos  $(l_1, l_2, \star) := splitExtended(t)$

$$\begin{aligned}
& \text{length}(\text{proj}_1(\text{splitExtended}(h :: t))) \\
&= \text{length}(\text{proj}_1(h :: l_2, \star, \star)) && \text{Pela equação ??} \\
&= \text{length}(h :: l_2) && \text{Pela definição de } \text{proj}_1 \\
&= 1 + \text{length}(l_2) && \text{Pela definição de } \text{length} \\
&\leq 1 + \text{length}(t) && \text{Por HI} \\
&= \text{length}(h :: t) && \text{Pela definição de } \text{length}
\end{aligned}$$

A demonstração do segundo item é análoga.  $\square$

**Lema 4.2.2.**  $\forall h_1, h_2 : A, t : [T],$

$$\text{length}(\text{proj}_1(\text{splitExtended}(h_1 :: h_2 :: t))) < \text{length}(h_1 :: h_2 :: t)$$

*Demonstração.*

Note: esta é a única demonstração que não usa indução.

Definimos  $(l_1, l_2, \star) := \text{splitExtended}(t)$

$$\begin{aligned}
\text{length}(\text{proj}_1(\text{splitExtended}(h_1 :: h_2 :: t))) &= \text{length}(\text{proj}_1(h_1 :: l_1, \star, \star)) && \text{Pela equação ??} \\
&= \text{length}(h_1 :: l_1) && \text{Pela definição de } \text{proj}_1 \\
&= 1 + \text{length}(l_1) && \text{Pela definição de } \text{length} \\
&\leq 1 + \text{length}(t) && \text{Pela lema ??} \\
&< 2 + \text{length}(t) \\
&= \text{length}(h_1 :: h_2 :: t) && \text{Pela definição de } \text{length}
\end{aligned}$$

$\square$

**Lema 4.2.3.**  $\forall h_1, h_2 : A, t : [T],$

$$\text{length}(\text{proj}_2(\text{splitExtended}(h_1 :: h_2 :: t))) < \text{length}(h_1 :: h_2 :: t)$$

*Demonstração.*

Análoga à do lema anterior.

**Teorema 4.2.3.**  $\forall l : [A], \text{proj}_1(\text{mergeSortExtended}(l)) = \text{mergeSort}(l)$

*Demonstração.*

Caso 1:  $l = []$

$$\begin{aligned}
\text{proj}_1(\text{mergeSortExtended}([])) &= \text{proj}_1([], \star) && \text{Pela equação ??} \\
&= [] && \text{Pela definição de } \text{proj}_1 \\
&= \text{mergeSort}([]) && \text{Pela equação ??}
\end{aligned}$$

Caso 2:  $l = [h]$

$$\begin{aligned}
 \text{proj}_1(\text{mergeSortExtended}([h])) &= \text{proj}_1([h], \star) && \text{Pela equação ??} \\
 &= [h] && \text{Pela definição de } \text{proj}_1 \\
 &= \text{mergeSort}([h]) && \text{Pela equação ??}
 \end{aligned}$$

Caso 3:  $l = h_1 :: h_2 :: t$

Note: definimos

$$\begin{aligned}
 (l'_1, l'_2) &:= \text{split}(l) \\
 (l_1, l_2, \star) &:= \text{splitExtended}(l) \\
 (ls_1, \star) &:= \text{mergeSortExtended}(l_1) \\
 (ls_2, \star) &:= \text{mergeSortExtended}(l_2)
 \end{aligned}$$

Para demonstrar esse caso, será necessário usar a indução transfinita, apresentada na seção 2. Considere o predicado  $P(m)$  sendo  $\forall l : [A], \text{length}(l) = m \rightarrow \text{proj}_1(\text{mergeSortExtended}(l)) = \text{mergeSort}(l)$ . O teorema ?? equivale a  $\forall m, P(m)$ . Vamos assumir que o teorema vale para toda lista que tem tamanho menor que  $l$ . Isso implica que ele vale para  $l_1$  e  $l_2$ , pois o lema ?? afirma que a função *split* diminui o tamanho de listas que tem pelo menos 2 elementos.

$$\begin{aligned}
 &\text{proj}_1(\text{mergeSortExtended}(h_1 :: h_2 :: t)) \\
 &= \text{proj}_1(\text{mergeExtended}(ls_1, ls_2)) && \text{Pela equação ??} \\
 &= \text{merge}(ls_1, ls_2) && \text{Pelo teorema ??} \\
 &= \text{merge}(\text{mergeSort}(l_1), \text{mergeSort}(l_2)) && \text{Por HI} \\
 &= \text{merge}(\text{mergeSort}(l'_1), \text{mergeSort}(l'_2)) && \text{Pelo teorema ??} \\
 &= \text{mergeSort}(h_1 :: h_2 :: t) && \text{Pela equação ??}
 \end{aligned}$$

□

**Teorema 4.2.4.**  $\forall l : [A], \text{proj}_3(\text{splitExtended}(l)) = n$

*Demonstração.*

**Passo Base:**  $l = []$

$$\begin{aligned}
 \text{proj}_3(\text{splitExtended}([])) &= \text{proj}_3([], [], 0) && \text{Pela equação ??} \\
 &= 0 && \text{Pela definição de } \text{proj}_3
 \end{aligned}$$

**Passo Indutivo:**  $l = h :: t; n = 1 + n'$

**Hipótese Indutiva:**  $\text{proj}_3(\text{splitExtended}(t)) = n'$

Note: definimos  $m := \text{proj}_3(\text{splitExtended}(t))$



$$\begin{aligned}
proj_3(splitExtended(h :: t)) &= proj_3(\star, \star, 1 + m) && \text{Pela equação ??} \\
&= 1 + m && \text{Pela definição de } proj_3 \\
&= 1 + n' && \text{Por HI} \\
&= n && \text{Pela definição de } n'
\end{aligned}$$

□

**Teorema 4.2.5.**  $\forall l_1, l_2 : [A], \text{proj}_2(\text{mergeExtended}(l_1, l_2)) \leq n_1 + n_2$

*Demonstração.*

**Passo Base:**  $l_1 = []$

$$\begin{aligned}
proj_2(\text{mergeExtended}([], l_2)) &= proj_2(\star, 0) && \text{Pela equação ??} \\
&= 0 && \text{Pela definição de } proj_2 \\
&\leq 0 + n_2
\end{aligned}$$

**Passo Indutivo:**  $l_1 = h_1 :: t_1; n_1 = 1 + n'_1$

**Hipótese Indutiva:**  $\forall l_2 : [A], \text{proj}_2(\text{mergeExtended}(t_1, l_2)) \leq n'_1 + n_2$

Mais uma vez, para demonstrar o passo indutivo, usaremos indução na segunda lista.

**Passo Base 2:**  $l_2 = []$

$$\begin{aligned}
proj_2(\text{mergeExtended}(h_1 :: t_1, [])) &= proj_2(\star, 0) && \text{Pela equação ??} \\
&= 0 && \text{Pela definição de } proj_2 \\
&\leq n_1 + 0
\end{aligned}$$

**Passo Indutivo 2:**  $l_2 = h_2 :: t_2; n_2 = 1 + n'_2$

**Hipótese Indutiva 2:**  $\forall l_1 : [A], \text{proj}_2(\text{mergeExtended}(l_1, t_2)) \leq n_1 + n'_2$

Note: definimos  $m_1 := \text{proj}_2(\text{mergeExtended}(t_1, h_2 :: t_2))$

Caso 1:  $h_1 \leq h_2$

$$\begin{aligned}
proj_2(\text{mergeExtended}(h_1 :: t_1, h_2 :: t_2)) &= proj_2(\star, 1 + m_1) && \text{Pela equação ??} \\
&= 1 + m_1 && \text{Pela definição de } proj_2 \\
&\leq 1 + n'_1 + n_2 && \text{Por HI} \\
&= n_1 + n_2 && \text{Pela definição de } n'_1
\end{aligned}$$

Observe que, na terceira igualdade estamos usando a primeira hipótese de indução, tomando  $l_2 = h_2 :: t_2$ . A prova do segundo caso, em que  $h_1 \not\leq h_2$  é totalmente análoga, usando a segunda hipótese de indução.

□

Para demonstrar o próximo teorema, vamos precisar do seguinte lema:

**Lema 4.2.4.**  $\forall l : [A], \text{length}(\text{proj}_1(\text{splitExtended}(l))) \leq \lceil \frac{n}{2} \rceil \wedge$   
 $\text{length}(\text{proj}_2(\text{splitExtended}(l))) \leq \lfloor \frac{n}{2} \rfloor$

*Demonstração.*

**Passo Base:**  $l = []$

$$\text{length}(\text{proj}_1(\text{splitExtended}([])))$$

$$= \text{length}(\text{proj}_1([], \star, \star))$$

Pela equação ??

$$= \text{length}([])$$

Pela definição de  $\text{proj}_1$

$$= 0$$

Pela definição de  $\text{length}$

A demonstração da segunda desigualdade é análoga.

**Passo Indutivo:**  $l = h :: t; n = 1 + n'$

**Hipótese Indutiva:**  $\text{length}(\text{proj}_1(\text{splitExtended}(t))) \leq \lceil \frac{n'}{2} \rceil \wedge$   
 $\text{length}(\text{proj}_2(\text{splitExtended}(t))) \leq \lfloor \frac{n'}{2} \rfloor$

Note: definimos  $(l_1, l_2, \star) := \text{splitExtended}(t)$

$$\text{length}(\text{proj}_1(\text{splitExtended}(h :: t)))$$

$$= \text{length}(\text{proj}_1(h :: l_2, \star, \star))$$

Pela equação ??

$$= \text{length}(h :: l_2)$$

Pela definição de  $\text{proj}_1$

$$= 1 + \text{length}(l_2)$$

Pela definição de  $\text{length}$

$$\leq 1 + \left\lfloor \frac{n'}{2} \right\rfloor$$

Por HI

$$= \left\lfloor \frac{n' + 2}{2} \right\rfloor$$

$$= \left\lfloor \frac{n + 1}{2} \right\rfloor$$

Pela definição de  $n'$

$$= \left\lceil \frac{n}{2} \right\rceil$$

A demonstração da segunda desigualdade é análoga.

□

**Teorema 4.2.6.**  $\forall h : A, t : [A], \text{proj}_2(\text{mergeSortExtended}(h :: t)) \leq 8n \log_2 n$

(nesse caso,  $n = \text{length}(h :: t)$ )

*Demonstração.*

Caso 1:  $t = []$

$$\text{proj}_2(\text{mergeSortExtended}([h])) = \text{proj}_2(\star, 0)$$

Pela equação ??

$$= 0$$

Pela definição de  $\text{proj}_2$

$$\leq 8 * 1 * \log_2(1)$$

Caso 2:  $t = h' :: t'$

Usaremos o mesmo argumento do teorema ?? para usar a hipótese indutiva em  $proj_1(splitExtended(h :: h' :: t'))$  e  $proj_2(splitExtended(h :: h' :: t'))$ .

Note: definimos

$$\begin{aligned} (l_1, l_2, p) &:= splitExtended(h :: h' :: t') \\ m_1 &:= length(l_1) \\ m_2 &:= length(l_2) \\ e_1 &:= proj_2(mergeSortExtended(l_1)) \\ e_2 &:= proj_2(mergeSortExtended(l_2)) \\ q &:= proj_2(mergeExtended(l_1, l_2)) \end{aligned}$$

$$proj_2(mergeSortExtended(h :: h' :: t'))$$

$$\begin{aligned} &= proj_2(\star, q + m_1 + m_2 + p) && \text{Pela equação ??} \\ &= q + e_1 + e_2 + p && \text{Pela definição de } proj_2 \\ &= q + e_1 + e_2 + n && \text{Pelo teorema ??} \\ &\leq m_1 + m_2 + e_1 + e_2 + n && \text{Pelo teorema ??} \\ &= n + e_1 + e_2 + n \\ &\leq 2n + 8m_1 \log_2(m_1) + 8m_2 \log_2(m_2) && \text{Por HI} \\ &\leq 2n + 8 \left\lceil \frac{n}{2} \right\rceil \log_2 \left( \left\lceil \frac{n}{2} \right\rceil \right) + 8 \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left( \left\lfloor \frac{n}{2} \right\rfloor \right) && \text{Pelo lema ??} \\ &\leq 2n + 8 \left( \frac{n+1}{2} \right) \log_2 \left( \frac{n+1}{2} \right) + 8 \left( \frac{n}{2} \right) \log_2 \left( \frac{n}{2} \right) \\ &\leq 2n + 4n \log_2(n) + 4 \log_2(n) + 4n(\log_2(n) - 1) \\ &\leq 2n + 4n \log_2(n) + 2n + 4n(\log_2(n) - 1) \\ &= 4n \log_2(n) + 4n \log_2(n) - 4n + 4n \\ &= 8n \log_2(n) \end{aligned}$$

Observe que na décima linha estamos usando  $4 \log_2(n) \leq 2n$ . Além disso, na quarta linha estamos usando o fato da soma dos tamanhos das duas listas geradas por *split* ser igual ao tamanho da lista original, o que é simples de ser verificado usando indução.  $\square$

### 4.3 Certificações

**Teorema 4.3.1.**  $\forall a : A, l : [A], proj_2(insertExtended(l)) = comparisonsInsert(l)$

*Demonstração.*

**Passo Base:**  $l = []$

$$\begin{aligned}
proj_2(insertExtended(a, [])) &= proj_2(\star, 0) && \text{Pela equação ??} \\
&= 0 && \text{Pela definição de } proj_2 \\
&= comparisonsInsert(a, []) && \text{Pela equação ??}
\end{aligned}$$

**Passo Indutivo:**  $l = h : t$

**Hipótese Indutiva:**  $proj_2(insertExtended(a, t)) = comparisonsInsert(a, t)$

Note: definimos  $m := proj_2(insertExtended(a, t))$ .

Caso 1:  $a \leq h$

$$\begin{aligned}
proj_2(insertExtended(a, h :: t)) &= proj_2(\star, 1) && \text{Pela equação ??} \\
&= 1 && \text{Pela definição de } proj_2 \\
&= comparisonsInsert(a, h :: t) && \text{Pela equação ??}
\end{aligned}$$

Caso 2:  $a \not\leq h$

$$\begin{aligned}
proj_2(insertExtended(a, h :: t)) &= proj_2(\star, 1 + m) && \text{Pela equação ??} \\
&= 1 + m && \text{Pela definição de } proj_2 \\
&= 1 + proj_2(insertExtended(a, t)) && \text{Pela definição de } m \\
&= 1 + comparisonsInsert(a, t) && \text{Por HI} \\
&= comparisonsInsert(a, h :: t) && \text{Pela equação ??}
\end{aligned}$$

□

**Teorema 4.3.2.**  $\forall l_1, l_2 : [A], proj_2(mergeExtended(l_1, l_2)) = comparisonsMerge(l_1, l_2)$

*Demonstração.*

**Passo Base:**  $l_1 = []$

$$\begin{aligned}
proj_2(mergeExtended([], l_2)) &= proj_2(\star, 0) && \text{Pela equação ??} \\
&= 0 && \text{Pela definição de } proj_2 \\
&= comparisonsMerge([], l_2) && \text{Pela equação ??}
\end{aligned}$$

**Passo Indutivo:**  $l_1 = h_1 :: t_1$

**Hipótese Indutiva:**

$$\forall l_2 : [A], \text{proj}_2(\text{mergeExtended}(t_1, l_2)) = \text{comparisonsMerge}(t_1, l_2)$$

Para realizar o passo indutivo faremos indução em  $l_2$ .

**Passo Base 2:**  $l_2 = []$

$$\text{proj}_2(\text{mergeExtended}(h_1 :: t_1, []))$$

$$= \text{proj}_2(\star, 0) \quad \text{Pela equação ??}$$

$$= 0 \quad \text{Pela definição de } \text{proj}_2$$

$$= \text{comparisonsMerge}(h_1 :: t_1, []) \quad \text{Pela equação ??}$$

**Passo Indutivo 2:**  $l_2 = h_2 :: t_2; n_2 = 1 + n'_2$

**Hipótese Indutiva 2:**

$$\forall l_1 : [A], \text{proj}_2(\text{mergeExtended}(l_1, t_2)) = \text{comparisonsMerge}(l_1, t_2)$$

Note: definimos

$$m_1 := \text{proj}_2(\text{mergeExtended}(t_1, h_2 :: t_2))$$

$$m_2 := \text{proj}_2(\text{mergeExtended}(h_1 :: t_1, t_2))$$

$$r_1 := \text{removePrefix}(h_1, h_2 :: t_2)$$

$$r_2 := \text{removePrefix}(h_1, t_2)$$

Caso 1:  $h_1 \leq h_2$

$$\text{proj}_2(\text{mergeExtended}(h_1 :: t_1, h_2 :: t_2))$$

$$= \text{proj}_2(\star, 1 + m_1) \quad \text{Pela equação ??}$$

$$= 1 + m_1 \quad \text{Pela definição de } \text{proj}_2$$

$$= 1 + \text{comparisonsMerge}(t_1, h_2 :: t_2) \quad \text{Por HI}$$

$$= \text{comparisonsMerge}(h_1 :: t_1, h_2 :: t_2) \quad \text{Pela equação ??}$$

Na terceira igualdade estamos usando a primeira hipótese indutiva com  $l_2 = h_2 :: t_2$ . Como  $h_1 \leq h_2$ , temos  $\text{comparisonsInsert}(h_1, h_2 :: t_2) = 1$  e  $\text{removePrefix}(h_1, h_2 :: t_2) = h_2 :: t_2$ , logo a última igualdade vale.

Caso 2:  $h_1 \not\leq h_2$

$$\begin{aligned}
& \text{proj}_2(\text{merge\_exteded}(h_1 :: t_1, h_2 :: t_2)) \\
&= \text{proj}_2(\star, 1 + m_2) && \text{Pela equação ??} \\
&= 1 + m_2 && \text{Pela definição de } \text{proj}_2 \\
&= 1 + \text{comparisonsMerge}(h_1 :: t_1, t_2) && \text{Por HI} \\
&= 1 + \text{comparisonsInsert}(h_1, t_2) + \text{comparisonsMerge}(t_1, r_2) && \text{Pela equação ??} \\
&= \text{comparisonsInsert}(h_1, h_2 :: t_2) + \text{comparisonsMerge}(t_1, r_2) && \text{Pela equação ??} \\
&= \text{comparisonsInsert}(h_1, h_2 :: t_2) + \text{comparisonsMerge}(t_1, r_1) && \text{Pois } h_1 \not\leq h_2 \\
&= \text{comparisonsMerge}(h_1 :: t_1, h_2 :: t_2) && \text{Pela equação ??}
\end{aligned}$$

Na terceira igualdade estamos usando a segunda hipótese indutiva com  $l_1 = h_1 :: t_1$ . Note que, como  $h_1 \not\leq h_2$ , vale que  $\text{comparisonsInsert}(h_1, h_2 :: t_2) = 1 + \text{comparisonsInsert}(h_1, t_2)$ . Pelo mesmo motivo,  $\text{removePrefix}(h_1, h_2 :: t_2) = \text{removePrefix}(h_1, t_2)$ .

□

## 5 Formalizações em Lean

Nessa seção iremos definir as funções da seção 3 usando Lean. Além disso, apresentaremos o código do enunciado dos principais teoremas da seção 4 em Lean, assim como as principais táticas usadas para demonstrá-los. O argumento de todas as demonstrações em Lean é essencialmente o mesmo da seção 4.

As seguintes declarações informam ao compilador do Lean que usaremos  $\alpha$  como um tipo genérico ao longo das definições, e que  $r$  é uma relação entre elementos de  $\alpha$ . A segunda linha está definindo o operador  $\preceq$  como um alias para a relação  $r$ . A sentença `[decidable_rel r]` é usada para restringir  $r$  às relações que são decidíveis, isto é, para quaisquer  $a, b : \alpha$ ,  $a \preceq b$  nunca é indefinido, sendo sempre verdadeiro ou falso. Isto é necessário pois, como foi argumentado,  $r$  deve sempre ser uma relação de ordem total. Uma possível melhoria seria implementar as outras restrições que limitariam  $r$  para apenas relações de ordem total. Tais restrições não são implementadas na *mathlib*.

### 5.1 Funções

#### 5.1.1 Insertion Sort

As seguintes definições correspondem a implementação das duas versões do Insert e Insertion Sort em Lean (equações ?? até ??). Na primeira linha de cada uma, temos o nome da função, seguido pelo seu tipo. Nas linhas seguintes temos sua definição definida por meio do casamento de padrões com a lista de entrada.

#### 5.1.2 Merge Sort

As próximas definições correspondem a implementação das duas versões do Merge Sort e suas funções auxiliares em Lean (equações ?? até ??).

Nas duas versões do Merge Sort foi necessário usar o modo de táticas do Lean. Esse modo, usualmente usado para demonstrar teoremas, permite que o usuário use algumas expressões especiais para ajudar na definição que está sendo feita. No caso do Merge Sort, ele foi usado pois fazemos recursão na lista gerada pela função `split`. Nesse caso, é necessário fornecer ao compilador uma demonstração que essa recursão irá, eventualmente, terminar. Para isso, antes de definir o que será retornado, usamos o lema `length_split_lt` (que afirma que o tamanho da lista produzida por `split` é menor do que o tamanho da lista original), juntamente com o comando `using_well_founded`, para indicar ao compilador que a métrica usada para determinar o fim da recursão é o tamanho da lista.

### 5.1.3 Certificação

As seguintes funções são as versões em Lean correspondentes às equações usadas para certificação (equações ?? até ??)

## 5.2 Teoremas

### 5.2.1 Insertion Sort

Note: `.snd` equivale a função  $proj_2$  que estamos usando.

Equivalente ao teorema ??. A tática `induction` é usada para iniciar uma demonstração por indução. No passo indutivo, usa-se a tática `split_ifs` para dividir entre os casos  $a \leq h$  e  $a \not\leq h$ . Por causa do `let` usado na definição do `insertExtended`, é necessário usar a tática `cases`, que troca a chamada recursiva `insertExtended t` pelo par  $(l', n)$ , seguido pela tática `unfold`, que reescreve a definição de `insertExtended`. A tática `linarith` é usada quando é necessário apenas realizar cálculos para completar a demonstração.

### 5.2.2 Merge Sort

### 5.2.3 Certificação



## 6 Conclusão

## Apêndices

## A Teoremas Secundários da Seção 4