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

Orientador: Haniel Barbosa

Belo Horizonte, Minas Gerais
2021

Sumário

Glossário

$A \Rightarrow B$

Proposição A implica em proposição B.

$A \to B$

Tipo das funções do tipo A para o tipo B.

$[A]$

Tipo das listas de elementos do tipo A.

$[]$

Lista vazia.

$[a]_i$

i-ésimo elemento da lista a.

$a : A$

Termo a tem tipo A.

$a \not\leq b$

Negação de $a \leq b$.

$h :: t$

Lista composta pelo elemento h seguido pela lista t.

$length(l)$

Número de elementos na lista l.

$proj_{1,2}(p)$

Par contendo o primeiro e segundo elementos da tupla ordenada p.

$proj_1(p)$

Primeiro elemento da tupla ordenada p.

$proj_2(p)$

Segundo elemento da tupla ordenada p.

$proj_3(p)$

Terceiro elemento da tupla ordenada p.

HI

Hipóese Indutiva.

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:

$$M = \{A \,|\, A \in A\}$$
$$N = \{A \,|\, A \notin A\}$$
$$U = M \cup N$$

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 (1), “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 $f : A \to B$ sendo A e B tipos quaisquer. Uma regra clássica dos sistemas de tipo é algo como:

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

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* (2).

De acordo com Geuvers (3) (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 corretude 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 (4), fazendo com que fosse necessário acreditar apenas que seu kernel estaria correto. Exitem 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 (5) se identificaram como matemáticos (6) (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[1], 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[2].

Outros trabalhos que buscam desenvolver ferramentas para formalizar a complexidade temporal de algoritmos em assistentes de demonstração incluem (7) e (8), sendo o segundo trabalho especificamente com o Merge Sort. Ambos exploram a mesma técnica,

[1] <https://leanprover.zulipchat.com/#narrow/stream/113488-general/topic/BSc.20Final.20Project>
[2] <https://github.com/leanprover-community/mathlib/blob/master/src/data/list/sort.lean>

que é diferente da explorada neste trabalho, para realizar tal formalização. Ela explora fortemente os tipos dependentes para fazerem a contagem do número de operações realizadas. Tal técnica produz demonstrações mais elegantes e conceitualmente mais elaboradas, mas exige um entendimento mais aprofundado da teoria, portanto, ela será abordada em trabalhos futuros. Os resultados obtidos por ela e por aquelas usadas neste trabalho são os mesmos.

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, 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 $f : [A] \to [A]$ 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 (9). 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 (9), 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, nessa 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 $f : \mathbb{N} \to \mathbb{N}$. 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 (5) 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** (10). 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 (11).

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* (12). 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:

```
inductive Nat : Type
| zero : Nat
| succ : Nat → Nat

@[simp] def add : Nat → Nat → Nat
| Nat.zero     n := n
| (Nat.succ m) n := Nat.succ (add m n)

lemma add_zero : ∀ m : Nat , add m Nat.zero = m
| Nat.zero     := by refl
| (Nat.succ m) := by { simp, exact add_zero m }

lemma add_succ : ∀ m n : Nat , add m (Nat.succ n) = Nat.succ (add m n)
| Nat.zero     n := by refl
| (Nat.succ m) n := by { simp, exact add_succ m n, }
```

```
theorem add_comm : ∀ m n : Nat , add m n = add n m
| m (Nat.zero)   := by { simp, exact add_zero m, }
| m (Nat.succ n) := by { simp, rw ← add_comm m n, exact add_succ m n, }
```

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.

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

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

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

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

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

Usando *insert*, podemos definir o Insertion Sort. Essa função recebe uma lista xs e produz uma permutação ordenada de xs. As equações 3.5 e 3.6 representam sua versão original, enquanto 3.7 e 3.8 representam sua versão estendida.

$$insertionSort([]) = [] \tag{3.5}$$

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

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

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

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 3.9 e 3.10 representam a definição original e 3.11 e 3.12 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.

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

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

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

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

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 3.13, 3.14 e 3.15 são sua versão original, e as equações 3.16, 3.17 e 3.18 são sua versão estendida.

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

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

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

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

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

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

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 3.19, 3.20 e 3.21 representam sua definição original, enquanto as equações 3.22, 3.23 e 3.24 representam sua versão estendida.

$$mergeSort([]) = [] \tag{3.19}$$

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

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

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

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

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

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:

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

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

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:

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

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

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.

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

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

$$comparisonsMerge(x :: xs, ys) = comparisonsInsert(x, ys) + comparisonsMerge(xs, ys') \\ \textbf{onde: } ys' := removePrefix(x, ys) \tag{3.31}$$

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. Por brevidade, os teoremas sobre fatos das funções auxiliares são apresentados no apêndice A. 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

O seguinte teorema garante que, para qualquer lista, aplicar tanto o *insertionSort* quanto o *insertionSortExtended* nela produz sempre a mesma lista. Isso é importante pois, considerando a formalização em Lean, está mostrando que as novas funções definidas nesse trabalho não modificaram o comportamento daquelas que já estavam definidas na *mathlib*. Caso isso não fosse válido, não faria sentido demonstrar a sua complexidade temporal.

Teorema 4.1.1. $\forall l : [A],\ proj_1(insertionSortExtended(l)) = insertionSort(l)$

Demonstração.

Passo Base: $l = []$

$$\begin{aligned} proj_1(insertionSortExtended([]) &= proj_1([], 0) && \text{Pela equação 3.7} \\ &= [] && \text{Pela definição de } proj_1 \\ &= insertionSort([]) && \text{Pela equação 3.5} \end{aligned}$$

Passo Indutivo: $l = h :: t$
Hipótese Indutiva: $proj_1(insertionSortExtended(a, t)) = insertionSort(a, t)$.

Note: definimos $xs := proj_1(insertionSortExtended(t))$

$proj_1(insert\_sort\_extended(h :: t))$

$$\begin{aligned} &= proj_1(proj_1(insertExtended(h, xs)), \star) && \text{Pela equação 3.8} \\ &= proj_1(insert(h, xs), \star) && \text{Pelo teorema A.0.1} \\ &= insert(h, xs) && \text{Pela definição de } proj_1 \\ &= insert(h, proj_1(insertionSortExtended(t))) && \text{Pela definição de } xs \\ &= insert(h, insertionSort(t)) && \text{Por HI} \\ &= insertionSort(h :: t) && \text{Pela equação 3.6} \end{aligned}$$

□

A seguir, demonstramos um limite para o número retornado pela função *insertionSortExtended*. Como esse número representa a quantidade de comparações realizadas, esse teorema está garantindo a complexidade temporal do algoritmo, a qual está de acordo com o que é sabido sobre o Insertion Sort.

Teorema 4.1.2. $\forall l : [A],\ proj_2(insertionSortExtended(l)) \leq n^2$

Demonstração.
Passo Base: $l = []; n = 0$

$$\begin{aligned} proj_2(insertionSortExtended([])) &= proj_2(\star, 0) && \text{Pela equação 3.7} \\ &= 0 && \text{Pela definição de } proj_2 \end{aligned}$$

Passo Indutivo: $l = h :: t; n = 1 + n'$
Hipótese Indutiva: $proj_2(insertionSortExtended(t)) \leq (n')^2$
Note: definimos $m := proj_2(insertExtended(h, t))$

$proj_2(insertionSortExtended(h :: t))$

$$
\begin{aligned}
&= proj_2(\star, proj_2(insertionSortExtended(t)) + m) && \text{Pela equação 3.8}\\
&= proj_2(insertionSortExtended(t)) + m && \text{Pela definição de } proj_2\\
&\leq (n')^2 + m && \text{Por IH}\\
&\leq (n')^2 + n' && \text{Pelo teorema A.0.2}\\
&= n'(n' + 1) && \text{Por distributividade}\\
&= n'(n) && \text{Pela definição de } n'\\
&< n^2 && \text{Pois } n' < n
\end{aligned}
$$

□

4.2 Merge Sort

Os seguintes lemas serão úteis em alguns dos próximos teoremas. O lema 4.2.1 está garantindo que a função *splitExtended* nunca aumenta o tamanho da lista, e será usado no lema 4.2.2. Este afirma que a função *splitExtended*, se usada em uma lista com pelo menos dois elementos, sempre retornará duas listas de tamanho estritamente menor. Isso será útil para usar a indução transfinita nos teoremas 4.2.1 e 4.2.2.

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: $l = []$

$$
\begin{aligned}
length(proj_1(splitExtended([]))) &= length(proj_1([], \star, \star)) && \text{Pela equação 3.11}\\
&= length([]) && \text{Pela definição de } proj_1\\
&= 0 && \text{Pela definição de } length
\end{aligned}
$$

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

$length(proj_1(splitExtended(h :: t)))$

$$
\begin{aligned}
&= length(proj_1(h :: l_2, \star, \star)) && \text{Pela equação 3.12}\\
&= length(h :: l_2) && \text{Pela definição de } proj_1\\
&= 1 + length(l_2) && \text{Pela definição de } length\\
&\leq 1 + length(t) && \text{Por HI}\\
&= length(h :: t) && \text{Pela definição de } length
\end{aligned}
$$

A demonstração do segundo item é análoga. □

Lema 4.2.2. $\forall h_1, h_2 : A, t : [T]$,
$length(proj_1(splitExtended(h_1 :: h_2 :: t))) < length(h_1 :: h_2 :: t)$

Demonstração.

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

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

$$
\begin{aligned}
length(proj_1(splitExtended(h_1 :: h_2 :: t))) &= length(proj_1(h_1 :: l_1, \star, \star)) && \text{Pela equação 3.12}\\
&= length(h_1 :: l_1) && \text{Pela definição de } proj_1\\
&= 1 + length(l_1) && \text{Pela definição de } length\\
&\leq 1 + length(t) && \text{Pela lema 4.2.1}\\
&< 2 + length(t)\\
&= length(h_1 :: h_2 :: t) && \text{Pela definição de } length
\end{aligned}
$$

□

Lema 4.2.3. $\forall h_1, h_2 : A, t : [T]$,
$length(proj_2(splitExtended(h_1 :: h_2 :: t))) < length(h_1 :: h_2 :: t)$

Demonstração.

Análoga à do lema anterior.

O teorema seguinte mostra que, recebendo a mesma lista, as funções *mergeSort* e *mergeSortExtended* produzem listas idênticas. Assim como o teorema 4.1.1, ele está sendo usado para garantir que não estamos modificando o algoritmo da *mathlib*.

Teorema 4.2.1. $\forall l : [A],\ proj_1(mergeSortExtended(l)) = mergeSort(l)$

Demonstração.

Caso 1: $l = []$

$$\begin{aligned} proj_1(mergeSortExtended([])) &= proj_1([], \star) && \text{Pela equação 3.22} \\ &= [] && \text{Pela definição de } proj_1 \\ &= mergeSort([]) && \text{Pela equação 3.19} \end{aligned}$$

Caso 2: $l = [h]$

$$\begin{aligned} proj_1(mergeSortExtended([h])) &= proj_1([h], \star) && \text{Pela equação 3.23} \\ &= [h] && \text{Pela definição de } proj_1 \\ &= mergeSort([h]) && \text{Pela equação 3.20} \end{aligned}$$

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

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

Para demonstrar esse caso, será necessário usar a indução transfinita, apresentada na seção 2. Considere o predicado $P(m)$ sendo $\forall l : [A],\ length(l) = m \rightarrow proj_1(mergeSortExtended(l)) = mergeSort(l)$. O teorema 4.2.1 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 4.2.2 afirma que a função *split* diminui o tamanho de listas que tem pelo menos 2 elementos.

$proj_1(mergeSortExtended(h_1 :: h_2 :: t))$

$$\begin{aligned} &= proj_1(mergeExtended(ls_1, ls_2)) && \text{Pela equação 3.24} \\ &= merge(ls_1, ls_2) && \text{Pelo teorema A.0.4} \\ &= merge(mergeSort(l_1), mergeSort(l_2)) && \text{Por HI} \\ &= merge(mergeSort(l'_1), mergeSort(l'_2)) && \text{Pelo teorema A.0.3} \\ &= mergeSort(h_1 :: h_2 :: t) && \text{Pela equação 3.21} \end{aligned}$$

□

Para demonstrar o próximo teorema, vamos precisar do lema a seguir. Ele dita que a função *splitExtended* sempre retorna duas listas com aproximadamente a metade do tamanho da lista original. Isso indica que a função está separando a lista em duas de tamanho mais próximo o possível. Este fato é crucial para que o Merge Sort atinja sua complexidade, como será visto na demonstração do teorema 4.2.2.

Lema 4.2.4. $\forall l : [A],\ length(proj_1(splitExtended(l))) \leq \lceil \frac{n}{2} \rceil \wedge$
$length(proj_2(splitExtended(l))) \leq \lfloor \frac{n}{2} \rfloor$

Demonstração.

Passo Base: $l = []$

$length(proj_1(splitExtended([])))$

$$
\begin{aligned}
&= length(proj_1([], \star, \star)) && \text{Pela equação 3.11}\\
&= length([]) && \text{Pela definição de } proj_1\\
&= 0 && \text{Pela definição de } length
\end{aligned}
$$

A demonstração da segunda desigualdade é análoga.

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

Hipótese Indutiva: $length(proj_1(splitExtended(t))) \leq \lceil \frac{n'}{2} \rceil \wedge$
$length(proj_2(splitExtended(t))) \leq \lfloor \frac{n'}{2} \rfloor$

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

$length(proj_1(splitExtended(h :: t)))$

$$
\begin{aligned}
&= length(proj_1(h :: l_2, \star, \star)) && \text{Pela equação 3.12}\\
&= length(h :: l_2) && \text{Pela definição de } proj_1\\
&= 1 + length(l_2) && \text{Pela definição de } length\\
&\leq 1 + \left\lfloor \frac{n'}{2} \right\rfloor && \text{Por HI}\\
&= \left\lfloor \frac{n' + 2}{2} \right\rfloor\\
&= \left\lfloor \frac{n + 1}{2} \right\rfloor && \text{Pela definição de } n'\\
&= \left\lceil \frac{n}{2} \right\rceil
\end{aligned}
$$

A demonstração da segunda desigualdade é análoga.

□

O seguinte teorema estabelece um limite para o número retornado pela função *mergeSortExtended*, e, consequentemente, para a complexidade do algoritmo Merge Sort. O fator 8, que não costuma ser mencionado quando se fala da complexidade do Merge Sort, se deve principalmente às operações realizadas pela função *splitExtended*, pois, em um contexto de programação funcional é necessário ter uma função desse tipo para separar a lista em duas, enquanto que em linguagens procedurais, tal função pode ser substituída pelo uso de arrays, tornando o algoritmo mais rápido.

Teorema 4.2.2. $\forall h : A, t : [A],\ proj_2(mergeSortExtended(h :: t)) \leq 8n \log_2 n$

(nesse caso, $n = lenght(h :: t)$)

Demonstração.

Caso 1: $t = []$

$$\begin{aligned} proj_2(mergeSortExtended([h])) &= proj_2(\star, 0) && \text{Pela equação 3.23} \\ &= 0 && \text{Pela definição de } proj_2 \\ &\leq 8 * 1 * log_2(1) \end{aligned}$$

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

Usaremos o mesmo argumento do teorema 4.2.1 para usar a hipótese indutiva em $proj_1(splitExtended(h :: h' :: t'))$ e $proj_2(splitExtended(h :: h' :: t'))$.

Note: definimos

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

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

$$\begin{aligned} &= proj_2(\star, q + m_1 + m_2 + p) && \text{Pela equação 3.24} \\ &= q + e_1 + e_2 + p && \text{Pela definição de } proj_2 \\ &= q + e_1 + e_2 + n && \text{Pelo teorema A.0.5} \\ &\leq m_1 + m_2 + e_1 + e_2 + n && \text{Pelo teorema A.0.6} \\ &= n + e_1 + e_2 + n \\ &\leq 2n + 8m_1 \log_2(m_1) + 8m_2 \log_2(m_2) && \text{Por HI} \\ &\leq 2n + 8\left\lceil \frac{n}{2} \right\rceil \log_2(\left\lceil \frac{n}{2} \right\rceil) + 8\left\lfloor \frac{n}{2} \right\rfloor \log_2(\left\lfloor \frac{n}{2} \right\rfloor) && \text{Pelo lema 4.2.4} \\ &\leq 2n + 8(\frac{n+1}{2}) \log_2(\frac{n+1}{2}) + 8(\frac{n}{2}) \log_2(\frac{n}{2}) \\ &\leq 2n + 4n \log_2(n) + 4 \log_2(n) + 4n(\log_2(n) - 1) \\ &\leq 2n + 4n \log_2(n) + 2n + 4n(\log_2(n) - 1) \\ &= 4n \log_2(n) + 4n \log_2(n) - 4n + 4n \\ &= 8n \log_2(n) \end{aligned}$$

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.

□

4.3 Certificações

O próximo teorema usa o fato da função *comparisonsInsert* retornar, garantidamente, o número de comparações do *insert* em uma determinada lista para mostrar que a função *insertExtended* também retorna esse número. Desse modo, estamos garantindo que a contagem de operações está correta para o *insert*.

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

Demonstração.

Passo Base: $l = []$

$$\begin{aligned} proj_2(insertExtended(a, [])) &= proj_2(\star, 0) && \text{Pela equação 3.3} \\ &= 0 && \text{Pela definição de } proj_2 \\ &= comparisonsInsert(a, []) && \text{Pela equação 3.25} \end{aligned}$$

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: $a \leq h$

$$\begin{aligned} proj_2(insertExtended(a, h :: t)) &= proj_2(\star, 1) && \text{Pela equação 3.4} \\ &= 1 && \text{Pela definição de } proj_2 \\ &= comparisonsInsert(a, h :: t) && \text{Pela equação 3.26} \end{aligned}$$

Caso 2: $a \not\leq h$

$proj_2(insertExtended(a, h :: t))$

$$\begin{aligned} &= proj_2(\star, 1 + m) && \text{Pela equação 3.4} \\ &= 1 + m && \text{Pela definição de } proj_2 \\ &= 1 + proj_2(insertExtended(a, t)) && \text{Pela definição de } m \\ &= 1 + comparisonsInsert(a, t) && \text{Por HI} \\ &= comparisonsInsert(a, h :: t) && \text{Pela equação 3.26} \end{aligned}$$

□

O teorema seguinte tem o mesmo objetivo do anterior, verificando que a contagem está correta para a funçcão *mergeExtended*.

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: $l_1 = []$

$proj_2(mergeExtended([], l_2))$

$$\begin{aligned} &= proj_2(\star, 0) && \text{Pela equação 3.17} \\ &= 0 && \text{Pela definição de } proj_2 \\ &= comparisonsMerge([], l_2) && \text{Pela equação 3.30} \end{aligned}$$

Passo Indutivo: $l_1 = h_1 :: t_1$

Hipótese Indutiva:

$\forall l_2 : [A],\ proj_2(mergeExtended(t_1, l_2)) = comparisonsMerge(t_1, l_2)$

Para realizar o passo indutivo faremos indução em l_2.

Passo Base 2: $l_2 = []$

$proj_2(mergeExtended(h_1 :: t_1, []))$

$$\begin{aligned} &= proj_2(\star, 0) && \text{Pela equação 3.16} \\ &= 0 && \text{Pela definição de } proj_2 \\ &= comparisonsMerge(h_1 :: t_1, []) && \text{Pela equação 3.29} \end{aligned}$$

Passo Indutivo 2: $l_2 = h_2 :: t_2;\ n_2 = 1 + n_2'$

Hipótese Indutiva 2:

$\forall l_1 : [A],\ proj_2(mergeExtended(l_1, t_2)) = comparisonsMerge(l_1, t_2)$

Note: definimos

$$\begin{aligned} m_1 &:= proj_2(mergeExtended(t_1, h_2 :: t_2)) \\ m_2 &:= proj_2(mergeExtended(h_1 :: t_1, t_2)) \\ r_1 &:= removePrefix(h_1, h_2 :: t_2) \\ r_2 &:= removePrefix(h_1, t_2) \end{aligned}$$

Caso 1: $h_1 \leq h_2$

$proj_2(mergeExtended(h_1 :: t_1, h_2 :: t_2))$

$$\begin{aligned} &= proj_2(\star, 1 + m_1) && \text{Pela equação 3.18} \\ &= 1 + m_1 && \text{Pela definição de } proj_2 \\ &= 1 + comparisonsMerge(t_1, h_2 :: t_2) && \text{Por HI} \\ &= comparisonsMerge(h_1 :: t_1, h_2 :: t_2) && \text{Pela equação 3.31} \end{aligned}$$

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

Caso 2: $h_1 \not\leq h_2$

$proj_2(merge\_exteded(h_1 :: t_1, h_2 :: t_2))$

$= proj_2(\star, 1 + m_2)$ Pela equação 3.18

$= 1 + m_2$ Pela definição de $proj_2$

$= 1 + comparisonsMerge(h_1 :: t_1, t_2)$ Por HI

$= 1 + comparisonsInsert(h_1, t_2) + comparisonsMerge(t_1, r_2)$ Pela equação 3.31

$= comparisonsInsert(h_1, h_2 :: t_2) + comparisonsMerge(t_1, r_2)$ Pela equação 3.26

$= comparisonsInsert(h_1, h_2 :: t_2) + comparisonsMerge(t_1, r_1)$ Pois $h_1 \not\leq h_2$

$= comparisonsMerge(h_1 :: t_1, h_2 :: t_2)$ Pela equação 3.31

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 $comparisonsInsert(h_1, h_2 :: t_2) = 1 + comparisonsInsert(h_1, t_2)$. Pelo mesmo motivo, $removePrefix(h_1, h_2 :: t_2) = removePrefix(h_1, t_2)$.

□

5 Formalizações em Lean

Nesta 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. Isso é possível pois Lean fornece ao usuário uma série de comandos que simulam os efeitos das técnicas de demonstração que foram usadas. Tais comandos são conhecidos como táticas. Ao longo da seção apresentaremos essas táticas e como elas são usadas para simular os argumentos da seção 4.

As seguintes declarações informam ao compilador do Lean que usaremos α como um tipo genérico ao longo das definições, e que r é uma relação entre elementos de α. A segunda linha está definindo o operador $\preceq$ como um alias para a relação r. A keyword infix implica que o operador deve ser usado entre os operandos; local indica que é uma definição local, que não será importada caso outro arquivo tente importar aquele no qual essa definição foi feita e 50 é o nível de precedência do operador. A sentença [decidable_rel r] é usada para restringir r às relações que são decidíveis, isto é, para quaisquer a, b : α, 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*.

```
variables {α : Type} (r : α → α → Prop) [decidable_rel r]
local infix ` ≼ ` : 50 := r
```

5.1 Táticas

A seguir apresentamos uma breve descrição das principais táticas utilizadas. Note: um tutorial completo explicando como demonstrar teoremas em Lean pode ser encontrado em (13).

induction: Inicia uma demonstração por indução no termo fornecido. Se o objetivo é demonstrar um predicado da forma ∀ n, P(n), onde n é um natural, usar induction n irá substituir esse objetivo por dois novos, sendo o primeiro P(0) e o segundo ∀ n, P(n) → P(n + 1)

simp: Simplifica ao máximo todas as expressões no predicado que deseja-se demonstrar, usando as hipóteses no contexto. Se for possível, pode ser usada para demonstrar diretamente o predicado.

cases: Fornecendo a esta tática um par ordenado, ela produz dois novos termos, correspondentes aos elementos do par ordenado. Usualmente é usada para se ter acesso

a cada elemento do retorno de uma função que produz pares. Por exemplo, se f é uma função que recebe uma lista e retorna um par contendo um natural e uma lista, e l é uma lista, a chamada cases (f l) with n l' irá produzir os termos n (o natural retornado por f) e l' (a lista retornada)

unfold: Aplica a definição de uma função, reescrevendo seu valor. Note que, se a definição for dada por meio de casamento de padrões, é necessário que o argumento da função esteja explicitamente na forma de um dos padrões. Por exemplo, se sabemos, por hipótese, que um certo natural n não vale 0 e temos uma função f que leva, por casamento de padrões, qualquer natural que não é 0 nele mesmo, podemos usar unfold f para trocar um predicado P(f(n)) que desejamos demonstrar por P(n).

linarith: Quando o objetivo é demonstrar uma expressão aritmética que segue diretamente das regras básicas (comutatividade, distributividade e associatividade) e da aplicação direta das hipóteses no contexto, essa tática demonstra o teorema automaticamente.

have: Introduz uma nova variável no contexto. Note que essa variável pode ser tanto um objeto quanto a demonstração de um teorema.

exact: Usado para finalizar a demonstração, apresentando um termo que demonstra diretamente o teorema usando as hipóteses em contexto.

split_ifs: Quando se está demonstrando um predicado que envolve uma função que possui um if em sua definição, usa-se essa tática para separar a demonstração em duas. Na primeira, deve-se demonstrar o teorema assumindo que a condição do if é verdadeira, e na segunda, que é falsa.

rewrite: Fornecendo a esta tática uma demonstração de que um certo termo a é igual a um outro termo b, troca todas as ocorrências de a por b no predicado que se pretende demonstrar.

5.2 Funções

5.2.1 Insertion Sort

As seguintes definições correspondem à implementação das duas versões do Insert e Insertion Sort em Lean (equações 3.1 até 3.8). Na primeira linha de cada uma, temos o nome da função, seguido pelo seu tipo. Nas linhas seguintes temos sua definição, que é feita por meio do casamento de padrões com a lista de entrada.

```
def insert (a : α) : list α → list α
| []       := [a]
| (h :: t) := if a ≤ h then a :: h :: t
                       else h :: insert t

def insertion_sort : list α → list α
```

```
| []       := []
| (b :: l) := insert r b (insertion_sort l)

def insert_extended (a : α) : list α → (list α × ℕ)
| []       := ([a], 0)
| (h :: t) := if a ≺ h then (a :: h :: t, 1)
                       else let (l', n) := insert_extended t
                            in (h :: l', 1 + n)

def insertion_sort_extended : list α → (list α × ℕ)
| [] := ([], 0)
| (h :: t) := let (l', n) := (insertion_sort_extended t) ,
                  (l'', m) := insert_extended r h l'
              in (l'', n + m)
```

5.2.2 Merge Sort

As próximas definições correspondem à implementação das duas versões do Merge Sort e suas funções auxiliares em Lean (equações 3.13 até 3.24).

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 táticas 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, usamos a cláusula using_well_founded no final da definição para indicar alguma propriedade da lista que fica "menor" a cada passo da recursão. Nesse caso, estamos usando o tamanho da lista. Além disso, precisamos explicitar para o compilador demonstrações de que a lista na qual estamos fazendo a recursão tem tamanho menor do que a lista original. O teorema length_split_lt produz um par ordenado de demonstrações, afirmando que cada uma das listas produzidas pela função split é menor do que a original. Usamos a tática cases nesse teorema para separar as duas demonstrações. Ambas são necessárias pois estamos usando recursão nas duas listas produzidas pela função split.

```
def split : list α → list α × list α
| []       := ([], [])
| (a :: l) := let (l₁, l₂) := split l in (a :: l₂, l₁)

def merge : list α → list α → list α
| []       l'        := l'
| l        []        := l
```

```
| (a :: l) (b :: l') := if a ≺ b
                        then a :: merge l (b :: l')
                        else b :: merge (a :: l) l'

def merge_sort : list α → list α
| []        := []
| [a]       := [a]
| (a::b::l) := begin
  cases e : split (a::b::l) with l₁ l₂,
  cases length_split_lt e with h₁ h₂,
  exact merge r (merge_sort l₁) (merge_sort l₂)
end
using_well_founded {
  rel_tac := λ_ _, `[exact ⟨_, inv_image.wf length nat.lt_wf⟩],
  dec_tac := tactic.assumption }

def split_extended : list α → (list α × list α × ℕ)
| []        := ([], [], 0)
| (h :: t) := let (l₁, l₂, n) := split_extended t
              in (h :: l₂, l₁, n + 1)

def merge_extended : list α → list α → (list α × ℕ)
| []       l'        := (l', 0)
| l        []        := (l,  0)
| (a :: l) (b :: l') := if a ≺ b
                        then let (l'', n) := merge_extended l (b :: l')
                             in (a :: l'', n + 1)
                        else let (l'', n) := merge_extended (a :: l) l'
                             in (b :: l'', n + 1)

def merge_sort_extended : list α → (list α × ℕ)
| []        := ([],  0)
| [a]       := ([a], 0)
| (a::b::l) := begin
  cases e : split_extended (a::b::l) with l₁ l₂n,
  cases l₂n with l₂ n,
  cases length_split_lt e with h₁ h₂,
  have ms₁ := merge_sort_extended l₁,
  have ms₂ := merge_sort_extended l₂,
  have merged := merge_extended r ms₁.fst ms₂.fst,
```

```
  have split_ops := (split_extended (a::b::l)).snd.snd,
  exact ( merged.fst , split_ops + ms₁.snd + ms₂.snd + merged.snd),
end
using_well_founded {
  rel_tac := λ_ _, `[exact ⟨_, inv_image.wf list.length nat.lt_wf⟩],
  dec_tac := tactic.assumption }
```

5.2.3 Certificação

As seguintes funções são as versões em Lean correspondentes às equações usadas para certificação (equações 3.25 até 3.31).

```
def comparisons_insert (a : α) : list α → ℕ
| [] := 0
| (h :: t) := if a ≺ h
              then 1
              else 1 + comparisons_insert t

def remove_prefix (a : α) : list α → list α
| [] := []
| (h :: t) := if a ≺ h
              then h :: t
              else remove_prefix t

def comparisons_merge : list α → list α → ℕ
| []        ys        := 0
| xs        []        := 0
| (x :: xs) (y :: ys) := comparisons_insert r x (y :: ys) +
                         comparisons_merge xs (remove_prefix r x)
```

5.3 Teoremas

5.3.1 Insertion Sort

```
theorem insertion_sort_equivalence (a : α) : ∀ l : list α,
  (insertion_sort_extended r l).fst = insertion_sort r l
```

Note: .fst equivale a função $proj_1$ que estamos usando.

Equivalente ao teorema 4.1.1. Usamos a tática induction para iniciar uma demonstração por indução. O predicado do passo base pode ser demonstrado simplesmente reescrevendo a definição de insertion_sort e insertion_sort_extended, logo a tática simp é suficiente. A

demonstração do passo indutivo segue diretamente da aplicação da hipótese indutiva e da versão em Lean do teorema A.0.1. Contudo, o compilador não é capaz de reescrever a definição de insertion_sort_extended sem ajuda. Para isso, precisamos usar a tática cases para explicitar os elementos retornados pela chamada recursiva de insertion_sort_extended, possibilitando o uso da tática unfold nessa função, que irá reescrevê-la.

```
theorem insertion_sort_complexity : ∀ l : list α,
  (insertion_sort_extended r l).snd ≤ l.length * l.length :=
```

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

Equivalente ao teorema 4.1.2. Usamos a tática induction para iniciar uma demonstração por indução. No passo base, o predicado segue diretamente da simplificação das definições, logo a tática simp é suficiente. No passo indutivo, assumimos que a lista é da forma h :: t. Usamos a tática cases para explicitar ao compilador que a chamada insertion_sort_extended t irá produzir uma certa lista ts e um natural n. Isso nos permite usar a tática unfold, que irá reescrever a definição de insertion_sort_extended, usando uma chamada recursiva e a função insert_extended. Usamos a tática have para inserir no contexto uma demonstração que (insert_extended h t).snd ≤ t.length, a pártir da formalização em Lean do teorema A.0.2. Como a tática induction colocou no contexto uma demonstração de que (insertion_sort_extended).snd ≤ t.length * t.length, a tática linarith é suficiente para usar todos esses fatos e demonstrar a desigualdade desejada.

Uma diferença com a demonstração convencional foi a necessidade de demonstrar para o compilador que a função insertion_sort_extended não altera o tamanho da lista de entrada, já que a função insert_extended é usada no retorno de insertion_sort_extended t, e não diretamente em t. Na demonstração convencional simplesmente assumimos que esse fato era trivial.

5.3.2 Merge Sort

Note: em ambos os teoremas dessa função também foi necessário usar uma clausula using_well_founded, já que eles usam indução na lista produzida pelo split_extended.

```
theorem merge_sort_equivalence :
  ∀ l : list α , (merge_sort_extended r l).fst = merge_sort r l
```

Equivale ao teorema 4.2.1. Como as duas definições do Merge Sort são feitas no modo tático do Lean, foi necessário demonstrar um teorema separado, que dita essencialmente que a equação 3.24 vale na definição dessa função. Divide-se a demonstração em casos, da mesma forma que a demonstração convencional. Nos casos em que a lista é vazia ou possui um único elemento, o predicado segue diretamente das definições, bastando usar a tática unfold em cada uma delas para concluir a demonstração. No caso em que a lista

tem pelo menos dois elementos, bastou usar a tática rw no teorema citado no início desse parágrafo, seguido pela mesma tática aplicada em cada um dos teoremas que mostra a equivalência das funções auxiliares do merge_sort_extended com as funções auxiliares do merge_sort, e na hipótese indutiva.

```
theorem merge_sort_complexity : ∀ l : list α ,
  (merge_sort_extended r l).snd ≤ 8 * l.length * nat.log 2 l.length
```

Equivale ao teorema 4.2.2. A demonstração dele em Lean é consideravelmente maior que as outras. Isso se deve principalmente a uma série de pequenos detalhes aritméticos, que a tática linarith não foi capaz de demonstrar sozinha. Para demonstrá-los, foi criado um objetivo paralelo para demonstrar cada um dos detalhes usando a tática have. Uma vez que se entra no contexto desses objetivos paralelos, é possível usar o comando library_search, que busca teoremas na biblioteca do Lean que os demonstram. Todos esses detalhes puderam ser demonstrados seguindo essa estratégia.

O fato mencionado na demonstração convencional, de que a soma dos tamanhos das listas retornadas pela função split é igual ao tamanho da lista original teve que ser demonstrado explicitamente. Foram usados teoremas fornecidos pela biblioteca que ditam que a relação $\leq$ nos naturais é monotônica. Por outro lado, essa propriedade não é formalizada na biblioteca para o logaritmo, portanto tivemos que demonstrá-la. A monotonicidade dessas duas funções foi assumida na demonstração convencional.

O resto da demonstração é essencialmente igual a convencional, em que usamos teoremas que limitam o número de comparações feitas pelas funções auxiliares e pelas chamadas recursivas do merge_sort_extended.

5.3.3 Certificação

```
theorem comparisons_insert_correct (a : α) : ∀ l : list α,
  (insert_extended r a l).snd = comparisons_insert r a l
```

Equivale ao teorema 4.3.1. Usa-se a tática induction para se iniciar uma demonstração por indução. O passo base pode ser demonstrado simplesmente reescrevendo as definições das funções, logo a tática simp é suficiente. No passo indutivo, usamos a tática split_ifs para separar os casos em que o termo que está sendo inserido é menor ou não do que a cabeça da lista. Caso seja, o compilador do Lean é capaz de simplificar ambos os lados da equação para 1, sendo possível completar a demonstração com a tática simp. Caso contrário, basta usar a tática simp para simplificar ambos os lados da igualdade e reescrever a hipótese indutiva do lado direito da igualdade, usando a tática rw. Depois disso, basta usar a mesma estratégia de aplicar cases seguido por unfold em insert_extended t no lado esquerdo, analogamente ao que foi feito no teorema insertion_sort_equivalence. Isso faz com que ambos os lados fiquem iguais a 1 + (insert_extended r a t).snd.

```
theorem comparisons_merge_correct : ∀ l₁ l₂ : list α,
  (merge_extended r l₁ l₂).snd = comparisons_merge r l₁ l₂
```

Equivale ao teorema 4.3.2. Em vez de usar a tática induction, a demonstração é dividida em quatro casos, dependendo do formato de cada lista de entrada (isso não nos impede de invocar o teorema recursivamente, simulando a hipótese indutiva). Nos tres primeiros, em que pelo menos uma das listas é vazia, basta usar a tática simp seguida pela tática unfold merge, fazendo com que o compilador do Lean seja capaz de verificar que ambos os lados da igualdade são iguais a 0.

No quarto caso, em que as duas listas tem pelo menos um elemento, usa-se a tática simp seguida por unfold remove_prefix e unfold merge para substituir todas as definições e split_ifs para dividir entre os casos que a cabeça da primeira lista é ou não menor do que a cabeça da segunda lista. O restante da demonstração segue o mesmo padrão da convencional. O compilador do Lean é capaz de fazer as simplificações que justificamos ao final de cada caso dessa demonstração na seção 4 sem ajuda externa, tornando a demonstração mais simples.

6 Conclusão

Este trabalho teve como objetivo demonstrar, por meio do assistente de demonstrações Lean, limites para o número de operações dos algoritmos Insertion Sort e Merge Sort, definidos na *mathlib*. O processo completo consistiu nas seguintes etapas:

1. Reescrita dos algoritmos, para que retornassem o número de comparações.
2. Demonstração da igualdade entre as listas retornadas pelos novos algoritmos e pelos antigos.
3. Demonstração dos limites dos números retornados para cada nova função.
4. Definição de funções que garantidamente calculam o número correto de operações de cada algoritmo, por meio de observações sobre o comportamento dos mesmos.
5. Demonstração da corretude dos números retornados pelos novos algoritmos de ordenação, usando as funções definidas na etapa 4.

As etapas 2, 3 e 5 foram verificadas pelo compilador do Lean, garantindo sua corretude. Essa formalização gerou um conteúdo novo que pode ser adicionado na *mathlib*, e que, segundo seus usuários, é interessante para a completude da biblioteca.

Enquanto as formalizações dos fatos relacionados ao Insertion Sort foram relativamente simples, o Merge Sort envolveu muito mais trabalho. Foi necessário usar fortemente a ajuda dos mecanismos de demonstração automática do Lean, além de demonstrar muitos lemas relacionados as suas funções auxiliares. Um outro ponto que chamou a atenção foram os detalhes cuja formalização é exigida no assistente, mas que passam despercebidos em uma demonstração convencional. Um exemplo disso foi a necessidade da demonstração do fato do Insertion Sort não modificar o tamanho da lista, como foi mencionado na seção 5 (algo que, de fato, é necessário para se obter uma formalização completa).

Apêndices

A Teoremas Secundários da Seção 4

Neste apêndice apresentaremos demonstrações dos outros teoremas que apresentam resultados necessários para demonstrar os teoremas do capítulo 4.

Teorema A.0.1. $\forall a : A, l : [A],\ proj_1(insertExtended(a, l)) = insert(a, l)$

Demonstração.

Passo Base: $l = []$.

$$\begin{aligned} proj_1(insertExtended(a, [])) &= proj_1([a], 0) && \text{Pela equação 3.3} \\ &= [a] && \text{Pela definição de } proj_1 \\ &= insert(a, []) && \text{Pela equação 3.1} \end{aligned}$$

Passo Indutivo: $l = h :: t$.
Hipótese Indutiva: $proj_1(insertExtended(a, t)) = insert(a, t)$.
Caso 1: $a \leq h$

$$\begin{aligned} proj_1(insertExtended(a, h : t)) &= proj_1(a :: h :: t, \star) && \text{Pela equação 3.4} \\ &= a :: h :: t && \text{Pela definição de } proj_1 \\ &= insert(a, h :: t) && \text{Pela equação 3.2} \end{aligned}$$

Caso 2: $a \not\leq h$

$proj_1(insertExtended(a, h :: t))$

$$\begin{aligned} &= proj_1(h :: proj_1(insertExtended(a, t)), \star) && \text{Pela equação 3.4} \\ &= proj_1(h :: insert(a, t), \star) && \text{Por HI} \\ &= h :: insert(a, t) && \text{Pela definição de } proj_1 \\ &= insert(a, h :: t) && \text{Pela equação 3.2} \end{aligned}$$

□

Teorema A.0.2. $\forall a : A, l : [A],\ proj_2(insertExtended(a, l)) \leq n$

Demonstração.

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

$$\begin{aligned} proj_2(insertExtended(a, [])) &= proj_2(\star, 0) && \text{Pela equação 3.3} \\ &= 0 && \text{Pela definição de } proj_2 \end{aligned}$$

Passo Indutivo: $l = h :: t;\ n = 1 + n'$
Hipótese Indutiva: $proj_2(insertExtended(a, t)) \leq n'$
Caso 1: $a \leq h$

$$\begin{aligned} proj_2(insertExtended(a, h :: t)) &= proj_2(\star, 1) && \text{Pela equação 3.4} \\ &= 1 && \text{Pela definição de } proj_2 \end{aligned}$$

Como l é da forma $h :: t$, ela tem pelo menos um elemento, logo a desigualdade vale.
Caso 2: $a \not\leq h$

$proj_2(insertExtended(a, h :: t))$

$$\begin{aligned} &= proj_2(\star, 1 + proj_2(insertExtended(a, t))) && \text{Pela equação 3.4} \\ &= 1 + proj_2(insertExtended(a, t)) && \text{Pela definição de } proj_2 \\ &\leq 1 + n' && \text{Por HI} \\ &= n && \text{Pela definição de n'} \end{aligned}$$

□

Teorema A.0.3. $\forall l : [A],\ proj_{1,2}(splitExtended(l)) = split(l)$

Demonstração.
Passo Base: $l = []$

$$\begin{aligned} proj_{1,2}(splitExtended([])) &= proj_{1,2}([], [], 0) && \text{Pela equação 3.11} \\ &= ([], []) && \text{Pela definição de } proj_{1,2} \\ &= split([]) && \text{Pela equação 3.9} \end{aligned}$$

Passo Indutivo: $l = h :: t$
Hipótese Indutiva: $proj_{1,2}(splitExtended(t)) = split(t)$
Note: definimos $(l_1, l_2, \star) := splitExtended(t)$

$$\begin{aligned} proj_{1,2}(splitExtended(h : t)) &= proj_{1,2}(h :: l_2, l_1, \star) && \text{Pela equação 3.12} \\ &= (h :: l_2, l_1) && \text{Pela definição de } proj_{1,2} \\ &= split(h :: t') && \text{Pela equação 3.10} \end{aligned}$$

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

□

Teorema A.0.4. $\forall l_1, l_2 : [A],\ proj_1(mergeExtended(l_1, l_2)) = merge(l_1, l_2)$

Demonstração.

Passo Base: $l_1 = []$

$$\begin{aligned} proj_1(mergeExtended([], l_2)) &= proj_1(l_2, \star) && \text{Pela equação 3.17} \\ &= l_2 && \text{Pela definição de } proj_1 \\ &= merge([], l_2) && \text{Pela equação 3.14} \end{aligned}$$

Passo Indutivo: $l_1 = h_1 :: t_1$

Hipótese Indutiva: $\forall l_2 : [A],\ proj_1(mergeExtended(t_1, l_2)) = merge(t_1, l_2)$

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

Passo Base 2: : $l_2 = []$

$$\begin{aligned} proj_1(mergeExtended(h_1 :: t_1, [])) &= proj_1(h_1 : t_1, \star) && \text{Pela equação 3.16} \\ &= h_1 :: t_1 && \text{Pela definição de } proj_1 \\ &= merge(h_1 :: t_1, []) && \text{Pela equação 3.13} \end{aligned}$$

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: $h_1 \leq h_2$

$proj_1(mergeExtended(h_1 :: t_1, h_2 :: t_2))$

$$\begin{aligned} &= proj_1(h_1 :: l_1', \star) && \text{Pela equação 3.18} \\ &= h_1 :: l_1' && \text{Pela definição de } proj_1 \\ &= h_1 :: merge(t_1, h_2 :: t_2) && \text{Por HI} \\ &= merge(h_1 :: t_1, h_2 :: t_2) && \text{Pela equação 3.15} \end{aligned}$$

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.

□

Teorema A.0.5. $\forall l : [A],\ proj_3(splitExtended(l)) = n$

Demonstração.

Passo Base: $l = []$

$$\begin{aligned} proj_3(splitExtended([])) &= proj_3([], [], 0) && \text{Pela equação 3.11} \\ &= 0 && \text{Pela definição de } proj_3 \end{aligned}$$

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

Hipótese Indutiva: $proj_3(splitExtended(t)) = n'$

Note: definimos $m := proj_3(splitExtended(t))$

$$\begin{aligned} proj_3(splitExtended(h :: t)) &= proj_3(\star, \star, 1 + m) && \text{Pela equação 3.12} \\ &= 1 + m && \text{Pela definição de } proj_3 \\ &= 1 + n' && \text{Por HI} \\ &= n && \text{Pela definição de } n' \end{aligned}$$

□

Teorema A.0.6. $\forall l_1, l_2 : [A],\ proj_2(mergeExtended(l_1, l_2)) \leq n_1 + n_2$

Demonstração.

Passo Base: $l_1 = []$

$$\begin{aligned} proj_2(mergeExtended([], l_2)) &= proj_2(\star, 0) && \text{Pela equação 3.17} \\ &= 0 && \text{Pela definição de } proj_2 \\ &\leq 0 + n_2 \end{aligned}$$

Passo Indutivo: $l_1 = h_1 :: t_1;\ n_1 = 1 + n_1'$

Hipótese Indutiva: $\forall l_2 : [A],\ proj_2(mergeExtended(t_1, l_2)) \leq n_1' + n_2$

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

Passo Base 2: $l_2 = []$

$$\begin{aligned} proj_2(mergeExtended(h_1 :: t_1, [])) &= proj_2(\star, 0) && \text{Pela equação 3.16} \\ &= 0 && \text{Pela definição de } proj_2 \\ &\leq n_1 + 0 \end{aligned}$$

Passo Indutivo 2: $l_2 = h_2 :: t_2;\ n_2 = 1 + n_2'$

Hipótese Indutiva 2: $\forall l_1 : [A],\ proj_2(mergeExtended(l_1, t_2)) \leq n_1 + n_2'$

Note: definimos $m_1 := proj_2(mergeExtended(t_1, h_2 :: t_2))$

Caso 1: $h_1 \leq h_2$

$$\begin{aligned} proj_2(mergeExtended(h_1 :: t_1, h_2 :: t_2)) &= proj_2(\star, 1 + m_1) && \text{Pela equação 3.18} \\ &= 1 + m_1 && \text{Pela definição de } proj_2 \\ &\leq 1 + n_1' + n_2 && \text{Por HI} \\ &= n_1 + n_2 && \text{Pela definição de } n_1' \end{aligned}$$

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.

□

Referências

1 PIERCE, B. C. *Types and Programming Languages*. 1st. ed. [S.l.]: The MIT Press, 2002.

2 MILNER, R. Lcf: A way of doing proofs with a machine. In: BEČVÁŘ, J. (Ed.). *Mathematical Foundations of Computer Science 1979*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979. p. 146–159. ISBN 978-3-540-35088-0.

3 GEUVERS, H. Proof assistants: History, ideas and future. *Sadhana*, 2009.

4 XIANG, L. A formal proof of the four color theorem. *CoRR*, abs/0905.3713, 2009. Disponível em: <http://arxiv.org/abs/0905.3713>.

5 MOURA, L. de et al. The lean theorem prover (system description). *International Conference on Automated Deduction*, 2015.

6 BAANEN, A. et al. *The Hitchhiker's Guide to Logical Verification*. [S.l.: s.n.], 2020.

7 MøLLNITZ, C. B.; ELGAARD, J.; RANNES, S. Certifying time complexity of agda programs using complexity signatures. *Department of Computer Science Aalborg University*, jun. 2020.

8 LAARHOVEN, T. van. *The complete correctness of sorting*. 2013. <https://twanvl.nl/blog/agda/sorting>. Acesso em 28 de Março de 2021.

9 KNUTH, D. *The Art of Computer Programming*. 2. ed. [S.l.]: Addison-Wesley, 1998.

10 BOVE, A.; DYBJER, P. Dependent types at work. In: . [S.l.: s.n.], 2008. p. 57–99. ISBN 978-3-642-03152-6.

11 EBNER, G. et al. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, v. 1, n. ICFP, p. 34:1–34:29, 2017. Disponível em: <https://doi.org/10.1145/3110278>.

12 mathlib Community, T. The Lean mathematical library. *arXiv e-prints*, p. arXiv:1910.09336, out. 2019.

13 AVIGAD, J.; MOURA, L. de; KONG, S. *Theorem Proving in Lean*. 2021. <https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf>. Acesso em 28 de Março de 2021.