Uma introdução à lógica assistida por computador

Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília[1]

13 de setembro de 2023

---

[1] flaviomoura@unb.br

Capítulo 1

Introdução

Este material foi desenvolvido para dar suporte às disciplinas que tenham lógica (proposicional e/ou de primeira ordem) e uma introdução ao estudo de algoritmos como parte do seu programa. Neste contexto, podemos citar diversas disciplinas de graduação da Universidade de Brasília que podem utilizar este material integralmente ou em parte como, por exemplo, Fundamentos Teóricos da Computação, Lógica Computacional e Projeto e Análise de Algoritmos do Departamento de Ciência da Computação, Lógica Matemática do Departamento de Matemática e Lógica do Departamento de Filosofia. Normalmente, o público que cursa estas disciplinas pertence aos cursos de Computação, Matemática e Engenharias em geral, mas acreditamos que este material seja útil para todos que tenham interesse nos temas de lógica e algoritmos.

A primeira parte deste material apresenta os conceitos básicos de lógica partindo da lógica proposicional e chegando na lógica de primeira ordem. A lógica de primeira ordem pode ser vista como a "lógica padrão"utilizada, ainda que informalmente, em Matemática e Computação. Este estudo inicial de lógica será útil para a segunda parte que utiliza os conceitos estudados na resolução de problemas computacionais. Tanto o estudo de lógica quanto o de algoritmos dá especial atenção à construção de provas (matemáticas). Neste contexto, as provas são inicialmente feitas em papel e lápis (provas informais), e posteriormente em computador (provas formais). A construção de provas é um tema que costuma ser espinhoso, de forma que aqui tentaremos facilitar o processo de familiarização com este tema partindo de provas simples, para em seguida explorarmos situações mais complexas.

Linguagens naturais, como o Português por exemplo, são ambíguas por natureza, e para evitarmos possíveis dúvidas na leitura de fórmulas ou propriedades utilizaremos linguagens mais restritas. Iniciaremos com a linguagem da lógica proposicional (LP), que nos permitirá resolver diversos problemas interessantes. Estes problemas serão estudados também no contexto computacional. Com isto, queremos dizer que resolveremos problemas manualmente, isto é, em papel e lápis, e também no computador.

Apesar da LP possuir limitações de expressividade, ela será útil para que possamos entender a dinâmica da construção de provas, mas a lógica efetivamente usada no dia a dia do matemático ou do cientista da computação é a Lógica de Primeira Ordem (LPO), que nos permitirá expressar propriedades de algoritmos de forma mais natural. Durante esta caminhada, estudaremos um assunto fundamental que está presente em diversos contextos: *indução*. Intuitivamente, o conceito de indução é bastante simples, mas a sua aplicação em situações específicas costuma gerar muita dúvida.

A construção de provas mecânicas, ou seja, provas feitas em computador, é uma atividade que tem despertado interesse crescente nas últimas décadas em função da forma como a computação tem se infiltrado no nosso dia a dia. Mas aqui precisamos de uma pequena pausa para explicarmos o que queremos dizer com provas feitas por computador. Esta explicação se faz necessária porque existem pelo menos duas abordagens distintas no que se refere a este assunto: os provadores automáticos de teoremas por um lado, e os assistentes de prova por outro.

Um provador automático de teoremas é um programa munido de uma heurística que recebe um teorema como argumento e tenta, de forma automática, encontrar uma prova para o teorema dado [29, 18, 21]. Um assistente de provas por outro lado, consiste em um programa que requer a orientação/interação do usuário para poder construir uma prova. Ou seja, o usuário vai guiando o sistema na construção de prova, enquanto o sistema verifica se cada passo dado/sugerido pelo usuário está correto. São exemplos de assistentes de prova o PVS[25], o Isabelle/HOL[23], o Lean[9] e o Coq[31]. Neste material trabalharemos com o assistente de provas Coq, que é um sistema de código aberto e que pode ser instalado em sistemas Linux, MacOs e Windows, ou até mesmo ser executado via browser[1].

Existem materiais muito interessantes que servem como tutoriais do Coq, como por exemplo, [28], ou [5]. Este material não é um tutorial do Coq, mas a sua utilização enriquecerá bastante nosso estudo. Nosso foco é o estudo da lógica proposicional e de primeira ordem, assim como a sua utilização/aplicação no estudo de algoritmos. Para isto utilizaremos o Coq como ferramenta de apoio mostrando como um assistente de provas pode ser útil nesta caminhada. Aqui é importante observar também que um assistente de provas é basicamente uma linguagem de programação juntamente com uma linguagem de especificação, ou seja, além da linguagem de programação existem uma camada lógica adicional, chamada de linguagem de especificação, que nos permite expressar os lemas e teoremas, por exemplo. A camada lógica do Coq é baseada em um formalismo conhecido como *cálculo de construções indutivas* [26] que é muito mais expressivo do que a lógica de primeira ordem que estudaremos aqui. Neste sentido, utilizaremos apenas uma pequena parte do poder de computacional do Coq.

Não assumimos nenhum conhecimento prévio de Coq, e sua utilização é opcional. Ou seja, é perfeitamente possível utilizar apenas a parte teórica deste material. A ideia aqui é que você possa reproduzir os temas abordados em Coq a partir do zero: simplesmente abra o Coq com a interface de sua preferência, e siga as orientações das atividades propostas. Nem tudo que será abordado aqui terá uma versão correspondente em Coq, já que alguns temas serão puramente teóricos e foram pensados para serem feitos apenas em papel e lápis. E mesmo a etapa de construção de provas em um assistente de provas deve ser precedida de um esboço em papel e lápis. As atividades a serem realizadas no assistente de provas têm um arquivo correspondente de apoio cujo *link* é fornecido junto com o texto da atividade.

No contexto de algoritmos e desenvolvimento de *software* é comum a utilização de testes como método de validação. Ou seja, o programa (ou *software*) é executado com diversas entradas distintas, e se nenhum problema é encontrado, o programa é considerado bom o suficiente para ser utilizado. De fato, a primeira coisa que fazemos após implementar um algoritmo é testá-lo para diversas entradas, e caso alguma resposta seja incorreta, uma revisão da implementação é feita para corrigir o erro, e então novos testes são realizados. Este processo é repetido até que o programador sinta confiança na implementação, mas depois de todos estes testes é possível dizer que o programa é correto? Certamente não! Pensando no caso particular da implementação de um algoritmo de ordenação listas de naturais ou inteiros (ou qualquer estrutura munida de uma ordem total), sabemos que existe uma infinidade de listas de inteiros que podem ser utilizadas nos testes, e portanto não é possível testar todas elas. Em se tratando de programas utilizados em sistemas críticos (aviação, medicina, sistemas bancários, etc), por menores que sejam as chances de erros, falhas não são toleradas. O que fazer então para garantir a correção de um programa? Uma abordagem possível consiste em utilizar a lógica para **provar** a correção do programa! Uma prova de uma propriedade de um programa fornece a garantia de que o programa satisfaz a propriedade provada **sempre**! Esta é a abordagem que utilizaremos aqui, e que tem se mostrado cada vez mais importante para o desenvolvimento da Matemática[13, 11, 2, 3] e Computação[19, 27, 24]. Para concluir esta seção e começarmos a colocar a mão na massa, listamos três exemplos famosos de erros em sistemas computacionais:

1. **Therac-25**: Uma máquina de radioterapia controlada por computador causou a morte de pelo menos 6 pacientes entre 1985 e 1987 por overdose de radiação.

2. **Pentium FDIV**: Um erro na construção da unidade de ponto flutuante do processador Pentium da Intel causou um prejuízo de aproximadamente 500 milhões de dólares para a empresa que se viu forçada a substituir os processadores que já estavam no mercado em 1994.

3. **Ariane 5**: Um foguete que custou aproximadamente 7 bilhões de dólares para ser construído explodiu no seu primeiro voo em 1996 devido ao reuso sem verificação apropriada de partes do código do seu predecessor.

Já deixamos claro que vamos **provar** muita coisa aqui. Mas o que é uma prova? Uma resposta possível "é um argumento feito para convencer alguém"[30]. O problema deste argumento é que pessoas diferentes podem ter compreensões distintas sobre o argumento, de forma que o argumento pode ser uma prova para uma pessoa, mas não para a outra... estranho, não? Uma definição geral e abstrata para a noção de prova não é uma tarefa fácil, mas forneceremos uma definição precisa em um contexto mais restrito, a saber, o da lógica simbólica[16, 32].

Capítulo 2

Lógica

O que é lógica? Esta é uma pergunta difícil de responder de forma direta e precisa, mas em um contexto bem amplo, podemos dizer que é um ramo da filosofia que estuda os princípios e métodos de inferência e demonstração, que nos permitem determinar as condições em que determinados fatos podem, ou não, serem concluídos a partir de outros [22]. Ao longo deste material trabalharemos para deixar esta ideia mais clara! Além disso, a lógica tem aplicações em muitas disciplinas além da filosofia, como matemática, ciência da computação, linguística e até mesmo em algumas ciências sociais, onde ela é usada para ajudar a estruturar e analisar argumentos e inferências. Na próxima seção, iniciaremos nossos estudo com a lógica proposicional, que nos fornecerá um ambiente relativamente simples, mas rigoroso, para as construções de nossas primeiras inferências.

2.1 A Lógica Proposicional

A lógica proposicional (LP) é baseada na noção **proposição**, que é uma sentença que pode ser qualificada como verdadeira ou falsa, mas nunca ambos. Por exemplo, são proposições:

- $2+2 = 4$.
- $1+3 < 0$.
- 2 é um número primo.
- João tem 20 anos e Maria tem 22 anos.

Mas nem toda sentença é uma proposição. De fato, a sentença "Feche a porta!", ou ainda a pergunta "Qual é o seu nome?"não podem ser qualificadas como verdadeira ou falsa, e portanto não são proposições. Estudaremos diversas lógicas distintas, e todas elas utilizam *fórmulas* como elementos fundamentais. As fórmulas são expressões ou *strings* (sequência de símbolos) utilizados para representar uma proposição. Por exemplo, utilizaremos letras latinas minúsculas para representar proposições que não podem ser subdivididas em proposições menores como "2+2=4", mas não "João tem 20 anos e Maria tem 22 anos". Chamamos estas letras latinas minúsculas de *variáveis proposicionais*. Assim, a variável proposicional é o elemento básico utilizado na construção das fórmulas.

Faremos um estudo incremental partindo de lógicas mais simples e pouco expressivas, e aos poucos vamos incrementando o poder de expressividade destas lógicas tendo o contexto computacional como referência. Trabalharemos com a lógica simbólica de forma que o significado de uma variável proposicional (ou a informação que ela codifica) será irrelevante daqui para frente. Outro aspecto importante é que o foco do nosso estudo é a construção de provas. Para isto utilizaremos um sistema dedutivo conhecido como *dedução natural*. Este sistema foi criado pelo lógico alemão Gerhard Gentzen (1909-1945), e consiste em um sistema lógico composto por um conjunto de regras de inferência que tenta capturar o raciocínio matemático da forma mais *natural* possível. Iniciaremos a construção de provas em uma lógica denominada o *fragmento implicacional da lógica proposicional*.

2.1.1 O fragmento implicacional da lógica proposicional

As fórmulas do fragmento implicacional da lógica proposicional são formadas pelas variáveis proposicionais apresentadas anteriormente e pelo conectivo binário chamado *implicação*. Podemos representar as fórmulas deste fragmento pela seguinte gramática:

$$\varphi ::= p \mid (\varphi \to \varphi) \tag{2.1}$$

onde p denota um elemento de um conjunto enumerável de variáveis proposicionais, *i.e.* podemos utilizar qualquer número finito de variáveis proposicionais para construir uma fórmula. O construtor $\varphi \to \varphi$ diz que uma fórmula implicacional é construída a partir de duas fórmulas já construídas anteriormente. Por exemplo, se p e q denotam variáveis proposicionais então podemos concluir que $p \to q$ é uma fórmula, e neste caso, chamamos p de *antecedente*, e q de *sucedente* da implicação. Utilizando esta nova fórmula, podemos construir uma nova implicação a partir dela, e por exemplo, p, obtendo $(p \to q) \to p$ ou $p \to (p \to q)$, e assim por diante. Como indicado pela gramática (2.1), utilizaremos letras gregas minúsculas para representar fórmulas do fragmento implicacional da lógica proposicional.

Nosso objetivo agora é raciocinar sobre as estas fórmulas. Mais especificamente, queremos obter (ou derivar) novas informações a partir de informações conhecidas. Tudo isto em um contexto abstrato onde os símbolos proposicionais utilizados podem representar qualquer informação que corresponda a uma proposição. Utilizaremos a notação de *sequentes* para separar as informações (fórmulas) dadas da nova informação (fórmula) que queremos concluir. Chamaremos as fórmulas dadas de *premissas*, e a fórmula a ser derivada de *conclusão*, assim um sequente é formado por duas partes: um conjunto finito de fórmulas (que são as premissas), digamos Γ, e uma fórmula que é a conclusão, digamos φ, que escrevemos como $\Gamma \vdash \varphi$. Assim, se $\varphi_1, \varphi_2, \ldots, \varphi_n$ são as premissas de um sequente, e se ψ é a sua conclusão, então escrevemos $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ para representar o sequente que tem ψ como conclusão, e o conjunto $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ de premissas. O conjunto $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$, isto é, a primeira componente do sequente $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ também pode ser chamado de *contexto* ao longo do texto, e normalmente será escrito sem as chaves que usualmente são usadas para representar conjuntos. Este é um abuso de linguagem usado para deixar a notação mais leve. Assim, se Γ denota um conjunto finito de fórmulas, ao invés de $\Gamma \cup \{\varphi\} \vdash \psi$, escreveremos simplesmente $\Gamma, \varphi \vdash \psi$, onde Γ, φ deve então ser lido como a união Γ com o conjunto unitário $\{\varphi\}$.

O conceito de prova agora será definido de forma mais precisa. Concretamente, uma prova (ou uma derivação) de um sequente da forma $\Gamma \vdash \psi$ é uma sequência de passos dedutivos, e um passo dedutivo consiste na aplicação de uma *regra de inferência* que possui a seguinte forma:

$$\frac{\Gamma_1 \vdash \gamma_1 \;\; \Gamma_2 \vdash \gamma_2 \ldots \Gamma_k \vdash \gamma_k}{\Gamma \vdash \psi}$$

onde $k \geq 0$. Quando $k = 0$ e $\psi \in \Gamma$ a regra corresponde a um *axioma*:

$$\frac{}{\Gamma \vdash \psi}\ (\text{Ax}),\ \text{se } \psi \in \Gamma$$

Uma prova (*i.e.* uma sequência de passos dedutivos) pode ser representada por meio de uma estrutura de árvore, onde os nós são anotados com sequentes. A raiz da árvore é anotada com o sequente que queremos provar, digamos, $\Gamma \vdash \psi$, e as são axiomas. Quais são as regras de inferência que podem ser utilizadas no fragmento implicacional da lógica proposicional? Além do axioma apresentado acima, temos duas regras para a implicação. Antes de apresentá-las devemos lembrar que o sistema dedutivo que utilizaremos se chama *dedução natural*, e as regras deste sistema são divididas em dois tipos: *introdução* e *eliminação*. A regra de eliminação da implicação é conhecida pelo nome *modus ponens* e tem a seguinte estrutura:

$$\frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$$

ou seja, para construirmos uma prova de um sequente com a forma $\Gamma \vdash \psi$ utilizando esta regra, precisamos construir duas outras provas: uma do sequente $\Gamma \vdash \varphi \to \psi$, e outra do sequente $\Gamma \vdash \psi$. Ou seja, na leitura desta regra de baixo para cima (*i.e.* da conclusão para as premissas) reduzimos o problema de provar $\Gamma \vdash \psi$ a dois outros problemas (potencialmente) mais simples. Esta regra também pode ser lida de cima para baixo (*i.e.* das premissas para a conclusão), e neste caso precisamos de uma prova de uma implicação, a saber $\Gamma \vdash \varphi \to \psi$, e de uma prova do antecedente desta implicação, a saber $\Gamma \vdash \psi$ para construirmos uma prova da conclusão da implicação, ou seja, uma prova de $\Gamma \vdash \psi$.

A regra de introdução é bastante intuitiva e, em certo sentido, nos fornece uma definição da implicação:

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_{\text{i}})$$

ou seja, na leitura de baixo para cima, para construirmos a prova de uma implicação precisamos construir uma prova do sucedente assumindo que temos uma prova do antecedente. Na leitura de cima para baixo, precisamos transformar uma prova do antecedente em uma prova do sucedente.

O interesse computacional do fragmento implicacional está diretamente relacionado ao algoritmo de inferência de tipos em linguagens funcionais[15]. O fundamento teórico destas linguagens é o cálculo λ[**?**] desenvolvido por Alonzo Church em 1936 [6, 7]. Para mais detalhes veja o Capítulo 1 de [4]. Vejamos um exemplo.

Exemplo 1. *Considere o sequente* $\vdash (p \to q) \to (q \to r) \to p \to r$. *A primeira observação a ser feita aqui é que a implicação é associativa à direita, ou seja,* $\varphi \to \psi \to \gamma$ *deve ser lido como* $\varphi \to (\psi \to \gamma)$, *e não como* $(\varphi \to \psi) \to \gamma$. *Portanto, o sequente que queremos provar deve ser lido como* $\vdash (p \to q) \to ((q \to r) \to (p \to r))$. *Utilizando inicialmente a regra* $(\to_{\mathrm{i}})$, *temos a seguinte situação:*

$$\dfrac{p \to q \vdash (q \to r) \to (p \to r)}{\vdash (p \to q) \to ((q \to r) \to (p \to r))}\,(\to_{\mathrm{i}})$$

Agora podemos aplicar novamente a regra $(\to_{\mathrm{i}})$*:*

$$\dfrac{\dfrac{p \to q, q \to r \vdash p \to r}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

E mais uma vez, já que a conclusão do sequente é ainda uma implicação:

$$\dfrac{\dfrac{\dfrac{p \to q, q \to r, p \vdash r}{p \to q, q \to r \vdash p \to r}\,(\to_{\mathrm{i}})}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

Agora não é mais possível utilizar a regra $(\to_{\mathrm{i}})$ *porque a conclusão* r *não é uma implicação, mas podemos utilizar a hipótese* $q \to r$ *para obter* r, *desde que tenhamos uma prova de* q *para utilizarmos* $(\to_e)$. *Neste ponto, a árvore é bifurcada em dois ramos e precisamos dividir o contexto de forma adequada em cada um dos ramos.*

$$\dfrac{\dfrac{\dfrac{\dfrac{p \to q, q \to r, p \vdash q \qquad \dfrac{}{p \to q, q \to r, p \vdash q \to r}\,(\mathrm{Ax})}{p \to q, q \to r, p \vdash r}\,(\to_e)}{p \to q, q \to r \vdash p \to r}\,(\to_{\mathrm{i}})}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

Observe que o ramo da direita consiste em um axioma já que a fórmula $q \to r$ *pertence ao conjunto de hipóteses. No ramo da esquerda podemos obter* q *por meio da regra* $(\to_e)$ *com as hipóteses* $p \to q$ *e* p. *A prova completa é dada a seguir:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \to q, q \to r, p \vdash p}\,(\mathrm{Ax}) \quad \dfrac{}{p \to q, q \to r, p \vdash p \to q}\,(\mathrm{Ax})}{p \to q, q \to r, p \vdash q}\,(\to_e) \quad \dfrac{}{p \to q, q \to r, p \vdash q \to r}\,(\mathrm{Ax})}{p \to q, q \to r, p \vdash r}\,(\to_e)}{p \to q, q \to r \vdash p \to r}\,(\to_{\mathrm{i}})}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

Este primeiro exemplo possui uma característica importante: a escolha da regra a ser aplicada em cada passo é única, e portanto não temos outra opção para a construção sesta prova! Se todo sequente tivesse esta característica, poderíamos construir um programa de computador que fizesse este trabalho para nós. Mesmo programas que não foram desenvolvidos especificamente para a construção destas provas podem resolver facilmente estas situações particulares. Por exemplo, solicitei ao ChatGPT[1] a construção de uma prova para o sequente acima e obtive a seguinte resposta:

ChatGPT (modelo gpt-4)

1. $p \to q$ (suposição)
2. $q \to r$ (suposição)
3. p (suposição)
4. q $(\to E, 1, 3)$
5. r $(\to E, 2, 4)$
6. $p \to r$ $(\to I, 3-5)$
7. $(q \to r) \to (p \to r)$ $(\to I, 2-6)$
8. $(p \to q) \to (q \to r) \to (p \to r)$ $(\to I, 1-7)$

Observe que, como nossa prova, a resposta do ChatGPT possui 3 aplicações da regra $(\to_{\mathrm{i}})$ e 2 de $(\to_e)$, o que já dá uma boa indicação de que as duas provas são iguais. De fato, as linhas 1, 2 e 3 são suposições, isto é, fórmulas que são assumidas, e considerando as folhas da nossa árvore, estas 3 fórmulas fazem parte do contexto. A linha 4 constrói uma prova da fórmula q a partir de uma aplicação da regra $(\to_e)$ utilizando as linhas 1 e 3, ou seja, utilizando as fórmulas p e $(p \to q)$. Isto corresponde à aplicação de $(\to_e)$ mais à esquerda na nossa árvore. Na linha 5, uma prova de r é feita utilizando $(\to_e)$ a partir das linhas 2 e 4, ou seja, utilizando as fórmulas q e $q \to r$. Isto corresponde à aplicação de $(\to_e)$ mais à direita na nossa árvore. A linha 6 constrói uma prova da fórmula $p \to r$ com uma aplicação da regra $(\to_{\mathrm{i}})$ a partir da subprova que vai da linha 3 até a linha 5. Esta subprova assume p (linha 3), e conclui r (linha 5), e isto corresponde à primeira aplicação de $(\to_{\mathrm{i}})$ (de cima para baixo) na nossa árvore. Na linha 7, uma prova de $((q \to r) \to (p \to r))$ é feita com uma aplicação da regra $(\to_{\mathrm{i}})$ a partir da subprova que vai da linha 2 até a linha 6. Esta subprova assume $(q \to r)$ (linha 2), e conclui $(p \to r)$ (linha 6), e isto corresponde à segunda aplicação de $(\to_{\mathrm{i}})$ na nossa árvore. Por fim, a linha 8 corresponde à terceira aplicação de $(\to_{\mathrm{i}})$ na nossa árvore, e portanto a prova dada pelo ChatGPT é a mesma prova apresentada no exemplo escrita de outra forma. Será que o ChatGPT ou uma inteligência artificial análoga é capaz de, neste momento, resolver todas as questões que serão propostas neste curso? Responderemos esta questão ao longo deste material, mas note que uma resposta afirmativa pode ser vista como uma possibilidade de substituição (parcial ou total) por máquinas, ainda que não de forma imediata, dos profissionais responsáveis por estas tarefas. Vejamos um outro exemplo.

---

[1] Foi utilizado o modelo GPT-4 em 04/06/2023.

Exemplo 2. *Considere o sequente* $\vdash (p \to q) \to (p \to p \to q)$. *Como no exemplo anterior, iniciaremos com uma aplicação da regra* $(\to_{\mathrm{i}})$*:*

$$\dfrac{p \to q \vdash p \to p \to q}{\vdash (p \to q) \to (p \to p \to q)}\;(\to_{\mathrm{i}})$$

Podemos aplicar mais uma vez a regra $(\to_{\mathrm{i}})$*:*

$$\dfrac{\dfrac{\dfrac{}{p \to q, p \vdash p \to q}\;(\mathrm{Ax})}{p \to q \vdash p \to p \to q}\;(\to_{\mathrm{i}})}{\vdash (p \to q) \to (p \to p \to q)}\;(\to_{\mathrm{i}})$$

E a prova está completa porque $(p \to q)$ *é uma das fórmulas do contexto. No entanto, poderíamos ter continuado com outra aplicação de* $(\to_{\mathrm{i}})$*:*

$$\dfrac{\dfrac{\dfrac{p \to q, p \vdash q}{p \to q, p \vdash p \to q}\;(\to_{\mathrm{i}})}{p \to q \vdash p \to p \to q}\;(\to_{\mathrm{i}})}{\vdash (p \to q) \to (p \to p \to q)}\;(\to_{\mathrm{i}})$$

Como o contexto é um conjunto de fórmulas, a nova introdução de p *não adiciona nenhuma fórmula no contexto porque os conjuntos* $\{p \to q, p\}$ *e* $\{p \to q, p, p\}$ *são idênticos. E agora, podemos concluir esta prova com uma aplicação de* $(\to_e)$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \to q, p \vdash p}\;(\mathrm{Ax}) \qquad \dfrac{}{p \to q, p \vdash p \to q}\;(\mathrm{Ax})}{p \to q, p \vdash q}\;(\to_e)}{p \to q, p \vdash p \to q}\;(\to_{\mathrm{i}})}{p \to q \vdash p \to p \to q}\;(\to_{\mathrm{i}})}{\vdash (p \to q) \to (p \to p \to q)}\;(\to_{\mathrm{i}})$$

No exemplo anterior, construímos duas provas para o mesmo sequente, ainda que bastante similares. Será que o ChatGPT consegue construir alguma destas provas? Vejamos:

ChatGPT (modelo gpt-4)

1. $p \to q$ (suposição)
2. p (suposição)
3. q $(\to E, 1, 2)$
4. $p \to q$ $(\to I, 2-3)$
5. $p \to p \to q$ $(\to I, 2-4)$
6. $(p \to q) \to (p \to p \to q)$ $(\to I, 1-5)$

Sim! Ele construiu uma derivação correspondente à nossa segunda prova. Vejamos mais um exemplo antes de finalizarmos esta seção.

Exemplo 3. *Sejam φ e ψ fórmulas quaisquer da LP. Considere o sequente $\psi \vdash \varphi \to \psi$. Queremos provar que a partir de uma prova de ψ podemos provar qualquer implicação que tenha ψ como conclusão:*

$$\dfrac{\dfrac{}{\psi, \varphi \vdash \psi}\,(\mathrm{Ax})}{\psi \vdash \varphi \to \psi}\,(\to_{\mathrm{i}})$$

O sequente do exemplo anterior nos diz que podemos construir a prova de uma implicação a partir de uma prova do consequente desta implicação. Esta prova é utilizada com muita frequência em outras provas, e por esta razão promovemos este sequente ao *status* de regra derivada:

$$\dfrac{\psi}{\varphi \to \psi}\,(\to_i)\;\emptyset$$

Podemos agora refazer a prova do exemplo anterior utilizando esta regra derivada:

$$\dfrac{\dfrac{\dfrac{}{p \to q \vdash p \to q}\,(\mathrm{Ax})}{p \to q \vdash p \to p \to q}\,(\to_i)\;\emptyset}{\vdash (p \to q) \to (p \to p \to q)}\,(\to_{\mathrm{i}})$$

Observe que esta prova corresponde à primeira prova do exemplo anterior: de fato, basta substituir a aplicação da regra derivada $(\to_i)\;\emptyset$ pela prova correspondente. Agora vejamos se o ChatGPT consegue fazer o mesmo:

ChatGPT (modelo gpt-4)

Pergunta: Prove o sequente $\psi \vdash \varphi \to \psi$.

Resposta: Para provar o sequente $\psi \vdash \varphi \to \psi$, podemos usar as regras de inferência do fragmento implicacional da lógica proposicional. Aqui está a prova, em um formato de derivação:

$$\begin{array}{lll}
1. & \psi & (\text{suposição}) \\
2. & \varphi & (\text{suposição}) \\
3. & \psi & (\text{repetição}, 1) \\
4. & \varphi \to \psi & (\to I, 2-3)
\end{array}$$

Nesta prova, utilizamos a regra de introdução da implicação (→I) para fazer a suposição φ (linha 2), e repetimos a suposição original ψ (linha 3). Finalmente, usamos a regra de introdução da implicação (→I) para inferir a proposição $\varphi \to \psi$ (linha 4).

Ok, a prova construída é análoga à que fizemos anteriormente, exceto pela linha 3 que repete a fórmula assumida na linha 1. Esta diferença ocorre porque o ChatGPT não está escrevendo as provas com a mesma notação de sequentes, mas a relação entre elas é mais ou menos direta. Retomando a conversa com o ChatGPT:

ChatGPT (modelo gpt-4)

Pergunta: A prova anterior nos diz que podemos provar uma implicação qualquer desde que tenhamos uma prova do seu consequente, correto?

Resposta: Sim, a prova anterior ilustra que, dado um consequente ψ, podemos sempre provar a implicação $\varphi \to \psi$ para qualquer antecedente φ, no fragmento implicacional da lógica proposicional. Isso ocorre porque a regra de introdução da implicação (→I) nos permite assumir qualquer proposição como uma suposição temporária. Portanto, mesmo que não tenhamos nenhuma informação sobre φ (o antecedente), ainda podemos provar a implicação $\varphi \to \psi$ se já temos uma prova de ψ.

Pergunta: Ótimo, então você pode construir uma nova prova do sequente $\vdash (p \to q) \to (p \to p \to q)$ utilizando a ideia dada na resposta anterior?

Resposta: Sim, podemos construir uma prova do sequente $\vdash (p \to q) \to (p \to p \to q)$ usando a ideia que discutimos anteriormente. Essencialmente, podemos provar uma implicação de qualquer antecedente, desde que tenhamos uma prova de seu consequente. Aqui está a prova, em um formato de derivação:

1. $p \to q \quad (\text{suposição})$
2. $p \quad (\text{suposição})$
3. $p \to q \quad (\text{repetição}, 1)$
4. $p \to p \to q \quad (\to I, 2-3)$
5. $(p \to q) \to (p \to p \to q) \quad (\to I, 1-4)$

Nesta prova, usamos a regra de introdução da implicação (→I) para fazer as suposições $p \to q$ (linha 1) e p (linha 2), e repetimos a suposição original $p \to q$ (linha 3). Depois, usamos a regra de introdução da implicação (→I) para inferir a proposição $p \to p \to q$ (linha 4). Finalmente, usamos novamente a regra de introdução da implicação (→I) para inferir a proposição $(p \to q) \to (p \to p \to q)$ (linha 5).

Opa! Aqui o ChatGPT se confundiu e não utilizou o raciocínio anterior de forma direta, ainda que a prova apresentada esteja correta. Parece que ele não consegue abstrair a ideia de que uma implicação pode ser provada diretamente a partir do seu consequente. Ou seja, ele não conseguiu reproduzir a prova que fizemos acima utilizando a regra $(\to_i)$. Vamos dar mais uma ajuda?

ChatGPT (modelo gpt-4)

Pergunta: Que tal chamar de ($\to i2$) a nova regra que nos permite provar uma implicação a partir de uma prova do seu consequente? Com esta informação, pode tentar novamente construir a prova utilizando a nova regra ($\to i2$)?

Resposta: Entendi sua sugestão. A nova regra ($\to i2$) permite a introdução de uma implicação a partir de uma prova do consequente. Dessa forma, podemos construir uma prova mais direta do sequente $\vdash (p \to q) \to (p \to p \to q)$. Aqui está a prova, em um formato de derivação:

1. $p \to q$ (suposição)
2. $p \to p \to q$ $(\to i2, 1)$
3. $(p \to q) \to (p \to p \to q)$ $(\to I, 1-2)$

Nesta prova, usamos a regra de introdução da implicação (→I) para fazer a suposição $p \to q$ (linha 1). Depois, usamos a nova regra (→i2) para inferir a proposição $p \to p \to q$ (linha 2). Finalmente, usamos novamente a regra de introdução da implicação (→I) para inferir a proposição $(p \to q) \to (p \to p \to q)$ (linha 3).

Excelente! Com este empurrãozinho o ChatGPT conseguiu utilizar a regra derivada. Os testes acima nos mostram que, apesar de ser uma ferramenta com potencial para o ensino, ainda deve ser utilizada com muita cautela. Devemos ter em mente que esta ferramenta responde a qualquer pergunta, e portanto algumas (ou muitas) delas podem conter erros. Durante este curso, utilizaremos o ChatGPT para enfatizar a importância da criatividade humana no processo de construção de uma prova, especialmente de uma prova não trivial. Mesmo em exemplos simples como os anteriores, o ChatGPT não conseguiu abstrair a ideia da nova regra e utilizá-la diretamente em uma prova sem ajuda, mas inicialmente ele disse que tinha utilizado a nova ideia. Agora é a sua vez! Deixando o ChatGPT de lado, e utilizando as regras (Ax), ($\to_\mathrm{i}$) e ($\to_e$), resolva os exercícios a seguir:

Exercício 1. *Prove o sequente* $\vdash (p \to p \to q) \to p \to q$.

Exercício 2. *Prove o sequente* $\vdash (q \to r \to t) \to (p \to q) \to p \to r \to t$.

Exercício 3. *Prove o sequente* $\vdash (p \to q \to r) \to (p \to q) \to p \to r$.

Exercício 4. *Prove o sequente* $\vdash (p \to q \to r) \to (q \to p \to r)$.

Exercício 5. *Prove o sequente* $\vdash (p \to r) \to p \to q \to r$.

Exercício 6. *Prove o sequente* $\vdash (p \to q) \to (p \to r) \to (q \to r \to t) \to p \to t$.

Na próxima seção veremos como resolver os sequentes do fragmento implicacional da lógica proposicional utilizando um programa de computador feito especificamente para a construção de provas! Utilizaremos o assistente de provas Coq (`https://coq.inria.fr/`) que é uma ferramenta de código aberto e que pode ser facilmente instalada em qualquer sistema operacional. A próxima seção é opcional, mas pode ajudar bastante no estudo da lógica.

2.1.2 O fragmento implicacional no Coq

O assistente de provas Coq (`https://coq.inria.fr`) é um sistema que implementa, entre outras coisas, as lógicas que estudaremos por aqui. Um assistente de provas, como o nome sugere, é uma ferramenta que, sob nossa orientação, nos permitirá mecanizar uma prova. Ou seja, nos permitirá passar uma prova manuscrita para o computador. Mas qual é a vantagem de refazer uma prova no computador? A grande vantagem é ter a garantia de que a prova está efetivamente correta. Provas manuscritas são mais susceptíveis a erros. No entanto, como comentado na Introdução, um assistente de provas não é adequado para encontrar uma nova prova. Ou seja, não deve ser utilizado de forma não planejada para eventualmente conseguir completar uma prova. A utilização eficiente consiste em primeiro ter uma prova manuscrita (ou pelo menos na cabeça) para, em seguida, reconstruí-la no assistente de prova. Provas feitas em assistentes de prova são bastante confiáveis, e amplamente aceitas na comunidade científica. Um caso interessante que levou o matemático americano Thomas Hales ao mundo dos assistentes de prova. Em 1998, Hales submeteu um artigo para publicação com aproximadamente 250 páginas contendo uma prova da Conjectura de Kepler. Esta conjectura foi formulada em 1611 pelo matemático Johannes Kepler e permaneceu sem solução até então, ou seja, por quase 400 anos! Uma comissão de especialistas foi montada para analisar o artigo e concluiu, após quase 2 anos depois, que tinha 99% de certeza de que a prova apresentada estava correta. Conta-se, que o então diretor do periódico onde se pleiteava a publicação, disse a Hales que se ele fornecesse algum ponto adicional de que a prova estava correta, o artigo seria publicado. Mas o que pode fornecer uma evidência maior da prova de um teorema do que o próprio artigo? Sim, uma prova formal! Em 2003, Halles começou a trabalhar no projeto Flyspeck que tinha como objetivo formalizar sua prova utilizando os assistentes de prova Isabelle[23] e HOL/Light[2], mas o artigo acabou sendo publicado em 2005[14]. O projeto tinha uma previsão inicial de durar 20 anos, mas em 2014, isto é 11 anos depois, o grupo do projeto Flyspeck anunciou que tinha completado a formalização da prova da Conjectura de Kepler[13].

Voltando ao Coq, é importante observarmos que ele é uma ferramenta de pesquisa, e não foi desenvolvida exclusivamente para o ensino. Grandes projetos foram e são desenvolvidos nele como a formalização do Teorema das 4 cores[11], do Teorema de Feit-Thompson (??) e a verificação de um compilador C[19]. Ainda, assim, veremos que ela nos será bastante útil. O Coq pode ser utilizado via uma *interface web* sem a necessidade de instalação[3], no entanto a instalação local costuma ser bastante simples (independente do sistema operacional). Assumiremos que você tem o Coq[4] instalado em sua máquina ou que tem acesso à internet para utilizar a *interface web*.

Utilizaremos algumas notações para facilitar a identificação de diferentes contextos, principalmente no que se refere ao assistente de provas Coq[31]. Os códigos do Coq são escritos em `verbatim`, e uma sessão típica do Coq possui três janelas:

---

[2] `https://www.cl.cam.ac.uk/~jrh13/hol-light/`

[3] Veja `https://coq.vercel.app/`

[4] Utilizamos a versão 8.16 durante a elaboração deste material.

```
Require Import Arith.
```

```
U:---  *goals*        All L1     (Coq Goals Wrap)
[Loading ML file ring_plugin.cmxs ...
</infomsg>
```

```
U:**-  teste.v        All L1     (Coq Script(0-) Holes Wrap.)
U:%%-  *response*     All L1     (Coq Response Wrap)
menu-bar Coq Electric Terminator
```

A janela da esquerda é a janela de especificação, ou seja, onde escrevemos as definições, os lemas e as provas. As especificações serão escritas em uma caixa azul:

janela de especificação

A janela do canto superior direito nos mostra o *status* atual da prova; e a do canto inferior direito nos mostra mensagens do sistema. O texto correspondente a estas janelas aparecerá em uma caixa verde:

janela de prova ou mensagens do sistema

Existem diversas razões para a utilização do Coq, mas aqui apresentaremos duas: a primeira é que o Coq implementa o sistema de dedução natural e portanto, é o candidato natural para explorar computacionalmente este sistema dedutivo; segundo, que o Coq possui uma comunidade muito grande e sempre disponível para ajudar com as dúvidas[5]. Apresentaremos as regras do fragmento implicacional da LP e faremos uma analogia entre o sistema de dedução natural que apresentamos aqui e o Coq sempre que possível, mas como veremos, esta analogia não é feita via uma correspondência direta entre as regras em dedução natural e as *regras* do Coq que são chamadas de *táticas*. De fato, as táticas são desenvolvidas para realizarem vários passos de prova, incluindo simplificações, de só uma vez porque isto facilita o processo de construção de provas em sistemas mais complexos.

A regra de introdução da implicação ($\to_i$) é simulada por meio da tática `intro`. Por exemplo, para provarmos uma implicação, digamos:

```
============================
 phi -> psi
```

---

[5] Existem diversos canais disponíveis para dúvidas sobre o Coq como, por exemplo, `https://coq.discourse.group/` e `https://proofassistants.stackexchange.com/`.

podemos utilizar a tática `intro` que vai mover o antecedente `phi` da implicação para as hipóteses, reduzindo assim, o problema de provar `phi -> psi` ao problema de provar `psi` assumindo `phi`:

```
H : phi
  ============================
  psi
```

A regra de *eliminação da implicação* $(\to_e)$, isto é, a regra *modus ponens* corresponde à tática `apply`. Considere uma aplicação da regra $(\to_e)$ como a seguir:

$$\dfrac{\dfrac{}{\varphi \to \psi, \varphi \vdash \varphi \to \psi}\,(\mathrm{Ax}) \qquad \dfrac{}{\varphi \to \psi, \varphi \vdash \varphi}\,(\mathrm{Ax})}{\varphi \to \psi, \varphi \vdash \psi}\,(\to_e)$$

A situação correspondente em Coq é dada a seguir:

```
H1 : phi -> psi
H2 : phi
============================
psi
```

Podemos aplicar a tática `apply H1` que vai reduzir a prova de `psi` a uma prova de `phi` já que o consequente da implicação `phi -> psi` coincide com a fórmula que queremos provar. Em seguida, concluímos a prova com a utilização da tática `assumption` porque `phi` é a hipótese `H2`.

Agora vamos refazer aqui os exemplos da seção anterior. Iniciaremos com o $\vdash (p \to q) \to ((q \to r) \to (p \to r))$. Precisamos declarar as variáveis `p`, `q` e `r` e, em seguida enunciarmos o lema a ser provado:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
```

A janela de prova correspondente é:

```
1 goal (ID 1)

============================
(p -> q) -> (q -> r) -> p -> r
```

Repetindo os passos da prova construída na seção anterior, devemos aplicar 3 vezes a regra $(\to_i)$. Na primeira aplicação, a fórmula `p -> q` é colocada no contexto. Em outras palavras, introduzimos a fórmula `p -> q`:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
  intro H1.
```

```
  1 goal (ID 2)

  H1 : p -> q
  ============================
  (q -> r) -> p -> r
```

Em seguida, introduzimos a fórmula `q -> r`:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
  intro H1. intro H2.
```

```
 1 goal (ID 3)

  H1 : p -> q
  H2 : q -> r
  ============================
  p -> r
```

E então, introduzimos a fórmula `p`:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
  intro H1. intro H2. intro H3.
```

```
1 goal (ID 4)

  H1 : p -> q
  H2 : q -> r
  H3 : p
  ============================
  r
```

Alternativamente, podemos compor os 3 passos de introdução de uma única vez usando a variante `intros`:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
  intros H1 H2 H3.
```

```
1 goal (ID 4)

  H1 : p -> q
  H2 : q -> r
  H3 : p
  ============================
  r
```

Agora podemos provar `r` via a regra ($\rightarrow_e$) utilizando a hipótese `H2` que tem `r` como consequente. Isto corresponde a utilizar a tática `apply H2`, que resulta na obrigação de provar `q`:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
  intros H1 H2 H3. apply H2.
```

```
1 goal (ID 5)

  H1 : p -> q
  H2 : q -> r
  H3 : p
  ============================
  q
```

Agora utilizamos novamente a regra ($\rightarrow_e$) com a hipótese `H1`, ou seja, aplicamos a tática `apply H1`, e concluímos com `assumption` já que `p` corresponde à hipótese `H3`:

```
Parameter p q r: Prop.

Lemma ex1: (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
  intros H1 H2 H3. apply H2. apply H1.
  assumption.
Qed.
```

Será que o ChaGPT também consegue fazer provas em Coq? Vejamos:

ChatGPT (modelo gpt-4)

Pergunta: Construa uma prova para o sequente $\vdash (p \to q) \to ((q \to r) \to (p \to r))$ no assistente de provas Coq.

Resposta: Claro, eu posso ajudar com isso. Coq é uma linguagem de programação funcional que também serve como assistente de provas. Aqui está como você provaria o sequente em Coq:

```
Theorem example : forall (p q r : Prop), (p -> q) -> ((q -> r) -> (p -> r)).
Proof.
 intros p q r.
 intros Hpq Hqr Hp.
 apply Hqr.
 apply Hpq.
 apply Hp.
Qed.
```

Uau! A prova está correta! A diferença para o que fizemos se restringe à forma como as variáveis p, q e r foram declaradas: nos declaramos como parâmetro, e ele declarou utilizando quantificação universal. A tática `apply Hp` corresponde à nossa utilização de `assumption`. A utilização de ferramentas como o ChatGPT como um assistente para os estudos pode ser interessante, se utilizadas com senso crítico porque da mesma forma que recebemos respostas corretas também recebemos respostas que não fazem o menor sentido, mas como a ferramenta responde com uma linguagem persuasiva, o seu uso acrítico pode ser bastante prejudicial.

Os exercícios propostos na seção anterior também podem ser feitos em Coq. O arquivo com os enunciados encontra-se disponível em **removido**[6].

2.1.3 A Lógica Proposicional Minimal

Agora vamos ampliar o poder de expressividade do fragmento implicacional da LP adicionando novos construtores gramaticais e novas regras de inferência. Assim, a nova gramática terá, além da implicação, a constante ($\bot$), a negação ($\neg$), a conjunção ($\wedge$) e a disjunção ($\vee$):

$$\varphi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \tag{2.2}$$

A constante $\bot$ é utilizada para representar a negação: $\neg\varphi$ é o mesmo que $\varphi \to \bot$. Ou seja, temos duas maneiras distintas de escrever a negação, e portanto a gramática acima possui redundâncias. De fato, veremos que existem outras redundâncias na gramática (2.2), mas elas são úteis porque simplificam a escrita das fórmulas.

A gramática (2.2) define as fórmulas da LP, e a partir dela consideraremos 3 sublógicas da LP: a minimal, a intuicionista e a clássica. Nesta seção estudaremos a lógica proposicional minimal (LPM), que assim como no fragmento implicacional visto anteriormente, possui uma regra de introdução e uma regra de eliminação para cada um dos conectivos lógicos. Ou seja, uma regra de introdução e uma de eliminação para cada um dos construtores recursivos da gramática (2.2).

Apesar da gramática apresentada acima não incluir a bi-implicação, este é um conectivo bastante utilizado, e pode ser escrito em função dos outros conectivos: $\varphi \leftrightarrow \psi$ é o mesmo que $(\varphi \to \psi) \wedge (\psi \to \varphi)$.

---

[6] Removido para não identificar a autoria, conforme o item 7.2(c) do edital conjunto DEG/EDU 04/2023.

As regras da negação são análogas às regras da implicação, uma vez que uma negação, digamos $(\neg\varphi)$ é definida como $(\varphi \rightarrow \bot)$.

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\;(\neg_i) \qquad\qquad \frac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\;(\neg_e)$$

Veremos posteriormente que apenas com a negação e implicação podemos expressar todos os outros conectivos apresentados na gramática (2.2), que portanto é uma gramática redundante. No entanto, esta redundância é interessante porque nos permite expressar fórmulas complexas de forma compacta.

Exercício 7. *Seja* φ *uma fórmula da LP. Prove o sequente* $\varphi \vdash \neg\neg\varphi$ *na LPM.*

A prova do exercício anterior ocorre com alta frequência como parte de outras provas, e por isto ela será promovida ao *status* de *regra derivada*, ou seja, regras que podem ser provadas a partir das regras básicas:

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \neg\neg\varphi}\;(\neg\neg_i)$$

Talvez você esteja esperando agora a derivação da eliminação da dupla negação para se juntar a regra anterior, mas infelizmente isto não é possível neste momento porque o poder de expressividade que temos até agora com as regras $(\rightarrow_e)$, $(\rightarrow_\mathrm{i})$, $(\neg_\mathrm{i})$ e $(\neg_e)$ não é suficiente para provarmos a eliminação da dupla negação.

Exemplo 4. *Ainda não temos expressividade suficiente para provar uma regra geral de eliminação da dupla negação, mas podemos provar a dupla eliminação de fórmulas negadas. Ou seja, provaremos o seguinte sequente:* $\neg\neg\neg\varphi \vdash \neg\varphi$*.*

$$\dfrac{\dfrac{(\neg\neg_i)\;\dfrac{\dfrac{}{\neg\neg\neg\varphi, \varphi \vdash \varphi}\;(\text{Ax})}{\neg\neg\neg\varphi, \varphi \vdash \neg\neg\varphi} \qquad \dfrac{}{\neg\neg\neg\varphi, \varphi \vdash \neg\neg\neg\varphi}\;(\text{Ax})}{\neg\neg\neg\varphi, \varphi \vdash \bot}\;(\neg_e)}{\neg\neg\neg\varphi \vdash \neg\varphi}\;(\neg_i)$$

Voltaremos a falar da eliminação da dupla negação na seção sobre lógica proposicional clássica.

Exercício 8. *Sejam* φ *e* ψ *fórmulas da LP. Prove o sequente* $\varphi \rightarrow \psi \vdash (\neg\neg\varphi) \rightarrow (\neg\neg\psi)$ *na LPM.*

Exercício 9. *Sejam* φ *e* ψ *fórmulas da LP. Prove o sequente* $\neg\neg(\varphi \rightarrow \psi) \vdash (\neg\neg\varphi) \rightarrow (\neg\neg\psi)$ *na LPM.*

A *regra de introdução da conjunção*, denotada por $(\wedge_i)$, nos diz o que precisamos fazer para construir uma prova de um sequente que possui uma conjunção na conclusão, isto é, um sequente da forma $\Gamma \vdash \varphi_1 \wedge \varphi_2$, onde Γ é um conjunto finito de fórmulas da LP, e φ_1 e φ_2 são fórmulas da LP. A regra $(\wedge_i)$ é dada pela seguinte regra de inferência:

$$\frac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\,(\wedge_i)$$

ou seja, uma prova de $\Gamma \vdash \varphi_1 \wedge \varphi_2$ é construída a partir de uma prova de $\Gamma \vdash \varphi_1$ e de uma prova de $\Gamma \vdash \varphi_2$.

Existem duas regras de eliminação para a conjunção já que podemos extrair qualquer uma das componentes de uma conjunção:

$$\frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_1}\,(\wedge_{e_1}) \qquad\qquad \frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_2}\,(\wedge_{e_2})$$

Estas duas regras podem ser representadas de forma mais concisa da seguinte forma:

$$\frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i \in \{1,2\}}}\,(\wedge_e)$$

Usaremos o nome $(\wedge_e)$ para designar a utilização da regra de eliminação da conjunção quando não quisermos especificar qual das regras $(\wedge_{e_1})$ ou $(\wedge_{e_2})$ foi utilizada.

Exemplo 5. *Com as regras da conjunção já podemos fazer um exercício interessante: provar a comutatividade da conjunção, isto é, queremos construir uma prova para o sequente* $\varphi \wedge \psi \vdash \psi \wedge \varphi$*, onde* φ *e* ψ *são fórmulas quaisquer da LP. A construção da prova é feita inicialmente de baixo para cima com a aplicação da regra* $(\wedge_i)$*:*

$$\frac{\dfrac{?}{\varphi \wedge \psi \vdash \psi} \qquad \dfrac{?}{\varphi \wedge \psi \vdash \varphi}}{\varphi \wedge \psi \vdash \psi \wedge \varphi}\,(\wedge_i)$$

Concluímos com a regra de eliminação da conjunção e o axioma:

$$\frac{\dfrac{\dfrac{}{\varphi \wedge \psi \vdash \varphi \wedge \psi}\,(\mathrm{Ax})}{\varphi \wedge \psi \vdash \psi}\,(\wedge_e) \qquad \dfrac{\dfrac{}{\varphi \wedge \psi \vdash \varphi \wedge \psi}\,(\mathrm{Ax})}{\varphi \wedge \psi \vdash \varphi}\,(\wedge_e)}{\varphi \wedge \psi \vdash \psi \wedge \varphi}\,(\wedge_i)$$

Exercício 10. *Prove que a conjunção é associativa, isto é, prove o sequente* $(\varphi \wedge \psi) \wedge \rho \vdash \varphi \wedge (\psi \wedge \rho)$ *na LPM, onde* φ, ψ *e* ρ *são fórmulas quaisquer da LP.*

Exercício 11. *Sejam* φ *e* ψ *fórmulas quaisquer da LP. Prove os sequentes* $\neg\neg(\varphi \wedge \psi) \dashv\vdash (\neg\neg\varphi) \wedge (\neg\neg\psi)$ *na LPM.*

Uma vez que uma bi-implicação corresponde a uma conjunção de duas implicações, ela pode ser decomposta com a regra de eliminação da conjunção:

$$\frac{\Gamma \vdash \varphi_1 \leftrightarrow \varphi_2}{\Gamma \vdash \varphi_1 \rightarrow \varphi_2}\ (\wedge_{e_1}) \qquad \frac{\Gamma \vdash \varphi_1 \leftrightarrow \varphi_2}{\Gamma \vdash \varphi_2 \rightarrow \varphi_1}\ (\wedge_{e_2})$$

Vejamos agora as regras para a disjunção. A *regra de introdução da disjunção* nos permite construir a prova de uma disjunção a partir da prova de qualquer uma das suas componentes:

$$\frac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_{i_1}) \qquad \frac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_{i_2})$$

Como no caso da regra de eliminação da conjunção podemos representar estas duas regras de forma mais compacta:

$$\frac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$$

A *regra de eliminação da disjunção* nos permite construir a prova de uma fórmula, digamos γ, a partir de uma disjunção. Para isto, precisamos de duas provas distintas de γ, cada uma assumindo uma das componentes da disjunção separadamente:

$$\frac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\ (\vee_e)$$

Assim, para que tenhamos uma prova de γ a partir de Γ precisamos de uma prova de γ a partir de φ e das fórmulas em Γ e de outra prova de γ a partir de φ_2 e das fórmulas em Γ. Observe como os contextos mudam em cada um dos sequentes que compõem esta regra.

A Tabela 2.1 resume todas as regras da LPM, isto é, as regras de introdução e eliminação dos conectivos lógicos apresentados na gramática 2.2.

	Regras de introdução	Regras de eliminação
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \land \varphi_2}\,(\land_i)$	$\dfrac{\Gamma \vdash \varphi_1 \land \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\,(\land_e)$
2	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \lor \varphi_2}\,(\lor_i)$	$\dfrac{\Gamma \vdash \varphi_1 \lor \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\,(\lor_\mathrm{e})$
3	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\,(\to_i)$	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\,(\to_e)$
4	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\,(\neg_i)$	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\,(\neg_e)$

Tabela 2.1: Regras da Lógica Minimal

Exemplo 6. *Vamos mostrar que a disjunção é comutativa, ou seja, queremos construir uma prova para o sequente* $\varphi \lor \psi \vdash \psi \lor \varphi$*. A ideia aqui é utilizarmos a regra* $(\lor_\mathrm{e})$*. Para isto podemos instanciar* Γ *com o conjunto unitário contendo a fórmula* $\varphi \lor \psi$*. Em função da estrutura da regra* $(\lor_\mathrm{e})$*, precisamos construir duas provas distintas de* $\psi \lor \varphi$*: uma a partir de* φ*, e outra a partir de* ψ*. Podemos fazer isto com a ajuda da regra* $(\lor_i)$*:*

$$\dfrac{(\mathrm{Ax})\,\dfrac{}{\varphi \lor \psi \vdash \varphi \lor \psi} \qquad \dfrac{\dfrac{}{\varphi \vdash \varphi}\,(\mathrm{Ax})}{\varphi \vdash \psi \lor \varphi}\,(\lor_i) \quad \dfrac{\dfrac{}{\psi \vdash \psi}\,(\mathrm{Ax})}{\psi \vdash \psi \lor \varphi}\,(\lor_i)}{\varphi \lor \psi \vdash \psi \lor \varphi}\,(\lor_\mathrm{e})$$

Exercício 12. *Sejam* φ*,* ψ *e* ρ *fórmulas quaisquer da LP. Prove que a disjunção é associativa, isto é, prove o sequente* $(\varphi \lor \psi) \lor \rho \vdash \varphi \lor (\psi \lor \rho)$ *na LPM.*

Exemplo 7. *Considere o sequente $\varphi \to \psi, \neg\psi \vdash \neg\varphi$. Como a fórmula do consequente é uma negação, vamos aplicar a regra de introdução da negação na construção de uma prova de baixo para cima, isto é, da raiz para as folhas da árvore:*

$$\cfrac{\cfrac{?}{\varphi \to \psi, \neg\psi, \varphi \vdash \bot}}{\varphi \to \psi, \neg\psi \vdash \neg\varphi}\,(\neg_{\mathrm{i}})$$

Agora, precisamos construir uma prova do absurdo, e portanto podemos tentar utilizar a regra $(\neg_e)$. Para isto precisamos escolher uma fórmula do contexto para fazer o papel de φ da regra 8 da Tabela 2.1. A princípio temos três opções: $\varphi \to \psi$, $\neg\psi$ e φ. A boa escolha neste caso é $\neg\psi$ porque podemos facilmente provar ψ a partir deste contexto:

$$\cfrac{\cfrac{(\to_e)\,\cfrac{\cfrac{}{\varphi \to \psi \vdash \varphi \to \psi}\,(\text{Ax}) \quad \cfrac{}{\varphi \vdash \varphi}\,(\text{Ax})}{\varphi \to \psi, \varphi \vdash \psi} \qquad \cfrac{}{\neg\psi \vdash \neg\psi}\,(\text{Ax})}{\varphi \to \psi, \neg\psi, \varphi \vdash \bot}\,(\neg_e)}{\varphi \to \psi, \neg\psi \vdash \neg\varphi}\,(\neg_{\mathrm{i}})$$

Depois de concluída a prova é fácil entender o que queríamos dizer com boa escolha *acima: Uma boa escolha é um caminho que vai nos permitir concluir uma prova. Mas como fazer uma boa escolha? Isto depende do problema a ser resolvido. Em alguns casos pode ser simples, mas em outros, bastante complicado. O ponto importante a compreender é que existem caminhos possíveis distintos na construção de provas da lógica proposicional, e muito deste processo depende da nossa criatividade.*

O sequente que acabamos de provar ocorre com certa frequência em outras provas, assim como a regra derivada $(\neg\neg_i)$. As regras que são obtidas a partir das regras da Tabela 2.1 são chamadas de *regras derivadas*. Este é o caso da regra conhecida como *modus tollens* (MT) obtida a partir do sequente do exemplo anterior, onde cada antecedente é generalizado como uma premissa da regra:

$$\cfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \neg\psi}{\Gamma \vdash \neg\varphi}\,(\text{MT})$$

Exemplo 8. *Considere o sequente $\varphi \to \psi \vdash \neg\psi \to \neg\varphi$. Inicialmente, devemos observar que a fórmula que queremos provar é uma implicação, e portanto, o mais natural é tentar aplicar a regra* $(\to_{\mathrm{i}})$*, e em seguida aplicar* (MT) *(na construção de baixo para cima) para poder completar a prova:*

$$\cfrac{\cfrac{(\text{Ax})\,\cfrac{}{\varphi \to \psi \vdash \varphi \to \psi} \qquad \cfrac{}{\neg\psi \vdash \neg\psi}\,(\text{Ax})}{\varphi \to \psi, \neg\psi \vdash \neg\varphi}\,(\text{MT})}{\varphi \to \psi \vdash \neg\psi \to \neg\varphi}\,(\to_{\mathrm{i}})$$

A prova que acabamos de fazer é outro caso que aparece com frequência, e corresponde a uma regra conhecida como *contrapositiva*:

$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \neg\neg\varphi}\ (\neg\neg_i)$	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \neg\psi}{\Gamma \vdash \neg\varphi}\ (\mathrm{MT})$	$\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)\ \emptyset$	$\dfrac{\Gamma \vdash \varphi \to \psi}{\Gamma \vdash \neg\psi \to \neg\varphi}\ (\mathrm{CP})$

Tabela 2.2: Regras derivadas da Lógica Proposicional Minimal

$$\dfrac{\Gamma \vdash \varphi \to \psi}{\Gamma \vdash \neg\psi \to \neg\varphi}\ (\mathrm{CP})$$

Exercício 13. *Sejam* φ *e* ψ *fórmulas da LP. Prove o sequente* $\varphi \to \neg\psi \vdash \psi \to \neg\varphi$ *na LPM.*

Exercício 14. *Sejam* φ *e* ψ *fórmulas da LP. Prove o sequente* $\vdash ((((\varphi \to \psi) \to \varphi) \to \varphi) \to \psi) \to \psi$ *na LPM.*

Exercício 15. *Sejam* φ *e* ψ *fórmulas da LP. Prove o sequente* $\varphi, \neg\varphi \vdash \neg\psi$ *na LPM.*

Exemplo 9. *Neste exemplo, veremos que é possível fazer a introdução de uma implicação sem precisar descartar uma hipótese, se tivermos uma prova do consequente da implicação que queremos construir. Ou seja, se temos uma prova de* ψ *então podemos construir uma prova de* $\varphi \to \psi$*, qualquer que seja a fórmula* φ*. Em outras palavras, queremos construir uma prova para o sequente* $\psi \vdash \varphi \to \psi$*:*

$$\dfrac{\dfrac{}{\psi, \varphi \vdash \psi}\ (\mathrm{Ax})}{\psi \vdash \varphi \to \psi}\ (\to_\mathrm{i})$$

Como este raciocínio aparece com frequência nas provas, vamos colocá-lo como uma regra derivada:

$$\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)\ \emptyset$$

A Tabela 2.2 apresenta as regras derivadas já provadas.

Exercício 16. *Sejam* φ *e* γ *fórmulas da LP. Construa uma prova para o sequente* $\varphi, \neg\varphi \vdash \neg\gamma$ *na LPM.*

Exercício 17. *Seja* φ *uma fórmula da LP. Construa uma prova para o sequente* $\neg\neg\neg\varphi \vdash \neg\varphi$ *na LPM.*

Exercício 18. *Sejam* φ *e* ψ *fórmulas da LP. Construa uma prova para o sequente* $\neg(\varphi \vee \psi) \vdash (\neg\varphi) \wedge (\neg\psi)$ *na LPM.*

Exercício 19. *Sejam* φ *e* ψ *fórmulas da LP. Construa uma prova para o sequente* $(\neg\varphi) \wedge (\neg\psi) \vdash \neg(\varphi \vee \psi)$ *na LPM.*

Exercício 20. *Sejam* φ, ψ *e* δ *fórmulas da LP. Construa uma prova para o sequente* $\varphi \rightarrow \psi \vdash (\delta \vee \varphi) \rightarrow (\delta \vee \psi)$ *na LPM.*

Exercício 21. *Sejam* φ *e* ψ *fórmulas da LP. Construa uma prova para o sequente* $\varphi \rightarrow \psi \vdash \neg(\varphi \wedge \neg\psi)$ *na LPM.*

Exercício 22. *Sejam* φ *e* ψ *fórmulas da LP. Construa uma prova para o sequente* $\varphi \wedge \psi \vdash \neg(\neg\varphi \vee \neg\psi)$ *na LPM.*

Exercício 23. *Sejam* φ *e* γ *fórmulas da LP. Construa uma prova para os sequentes* $\neg(\varphi \vee \gamma) \vdash (\neg\varphi) \wedge (\neg\gamma)$ *e* $(\neg\varphi) \wedge (\neg\gamma) \vdash \neg(\varphi \vee \gamma)$ *na LPM.*

Exercício 24. *Sejam* φ *e* γ *fórmulas da LP. Construa uma prova para o sequente* $(\neg\varphi) \vee (\neg\gamma) \vdash \neg(\varphi \wedge \gamma)$ *na LPM.*

Exercício 25. *Sejam* φ *e* γ *fórmulas da LP. Construa uma prova para o sequente* $\neg\neg(\varphi \wedge \gamma) \vdash (\neg\neg\varphi) \wedge (\neg\neg\gamma)$ *na LPM.*

Exercício 26. *Sejam* φ *e* γ *fórmulas da LP. Construa uma prova para o sequente* $(\neg\neg\varphi) \wedge (\neg\neg\gamma) \vdash \neg\neg(\varphi \wedge \gamma)$ *na LPM.*

Exercício 27. *Sejam* φ, ψ *e* γ *fórmulas da LP. Prove o sequente* $\varphi \vee (\psi \wedge \gamma) \vdash (\varphi \vee \psi) \wedge (\varphi \vee \gamma)$ *na LPM.*

Exercício 28. *Sejam* φ, ψ *e* γ *fórmulas da LP. Prove o sequente* $(\varphi \vee \psi) \wedge (\varphi \vee \gamma) \vdash \varphi \vee (\psi \wedge \gamma)$ *na LPM.*

Exercício 29. *Sejam* φ, ψ *e* γ *fórmulas da LP. Prove o sequente* $\varphi \wedge (\psi \vee \gamma) \vdash (\varphi \wedge \psi) \vee (\varphi \wedge \gamma)$ *na LPM.*

Exercício 30. *Sejam* φ, ψ *e* γ *fórmulas da LP. Prove o sequente* $(\varphi \wedge \psi) \vee (\varphi \wedge \gamma) \vdash \varphi \wedge (\psi \vee \gamma)$ *na LPM.*

Exercício 31. *Seja* φ *uma fórmula da LP. Prove o sequente* $\vdash \neg\neg(\varphi \vee \neg\varphi)$ *na LPM.*

Exercício 32. *Seja* φ *uma fórmula da LP. Prove o sequente* $\vdash \neg(\varphi \wedge \neg\varphi)$ *na LPM.*

Exercício 33. *Sejam* φ *e* γ *fórmulas da LP. Construa uma prova para o sequente* $(\varphi \to \gamma) \wedge \neg(\varphi \wedge \gamma) \vdash \neg\varphi$ *na LPM.*

Exercício 34. *Sejam* φ, ψ *e* γ *fórmulas da LP. Prove o sequente* $\varphi \leftrightarrow \psi, \psi \to \gamma, \neg\gamma \vdash (\neg\varphi) \wedge (\neg\psi)$ *na LPM.*

2.1.4 A Lógica Proposicional Minimal no Coq

Iniciaremos com alguns sequentes simples que nos mostram como utilizar as regras dadas na Tabela 2.1 no Coq. Aqui é importante lembrar que o Coq possui uma série de comandos de prova (táticas) que normalmente fazem várias coisas ao mesmo tempo como, por exemplo, simplificação e combinação de aplicação de regras. Assim, ainda que o sistema dedutivo do Coq seja o sistema de dedução natural, não temos necessariamente uma correspondência direta entre as táticas e as regras da Tabela 2.1. Iniciaremos com a regra de introdução da conjunção que pode ser simulada da seguinte forma: vamos declarar duas variáveis `phi` e `psi`, e em seguida, criaremos uma seção que vai delimitar o escopo da prova. Chamaremos esta seção de `landi`, e então declaramos as hipóteses e o lema propriamente dito dentro da seção:

```
Variables phi psi: Prop.

Section landi.
 Hypothesis H1: phi.
 Hypothesis H2: psi.
 Lemma landi: phi /\ psi.

End landi.
```

Esta é uma forma de declarar o sequente `phi, psi` $\vdash$ `phi` $\wedge$ `psi` no Coq. De fato, na janela de prova temos o sequente como esperado:

```
H1 : phi
H2 : psi
============================
phi /\ psi
```

Portanto uma prova deste sequente é o que vai corresponder a uma aplicação da regra ($\wedge_i$). A tática `split` é utilizada para dividir a prova da conjunção nas subprovas das suas componentes (já que a construção em Coq é sempre feita de baixo para cima, isto é da raiz para as folhas da árvore) que por sua vez são hipóteses, e a prova de algo que já faz parte do conjunto de hipóteses pode ser concluída com a tática `assumption`:

```
Variables phi psi: Prop.

Section landi.
 Hypothesis H1: phi.
 Hypothesis H2: psi.
 Lemma landi: phi /\ psi.
 Proof.
   split.
   - assumption.
   - assumption.
 Qed.
End landi.
```

Logo, a regra ($\wedge_i$) está relacionada com a tática `split` e o axioma com a tática `assumption`. Uma maneira de ver isto de forma mais explícita consiste em comparar a árvore de prova do sequente `phi, psi` $\vdash$ `phi` $\wedge$ `psi`:

$$\dfrac{(\mathrm{Ax})\ \dfrac{}{\texttt{phi},\texttt{psi} \vdash \texttt{phi}} \qquad \dfrac{}{\texttt{phi},\texttt{psi} \vdash \texttt{psi}}\ (\mathrm{Ax})}{\texttt{phi},\texttt{psi} \vdash \texttt{phi} \wedge \texttt{psi}}\ (\wedge_i) \qquad \texttt{assumption}\ \dfrac{\dfrac{}{\texttt{phi},\texttt{psi} \vdash \texttt{phi}} \qquad \dfrac{}{\texttt{phi},\texttt{psi} \vdash \texttt{psi}}\ \texttt{assumption}}{\texttt{phi},\texttt{psi} \vdash \texttt{phi} \wedge \texttt{psi}}\ \texttt{split}$$

Neste caso, temos uma correspondência direta entre uma regra do sistema de dedução natural e o Coq, mas veremos que isto nem sempre é possível.

O caminho natural na construção das provas em Coq consiste em partir da raiz da árvore de dedução (que é o sequente que queremos provar), e subir até as folhas (que são os axiomas). Esta construção de baixo para cima se torna mais difícil na medida em que um ajuste nas hipóteses se faz necessário. Assim como em papel e lápis, as provas em Coq podem ser construídas tanto de baixo para cima (da raiz para as folhas) quanto de cima para baixo: a cada instante da construção de uma prova podemos tanto dar um passo de baixo para cima aplicando uma tática que altera o objetivo (raiz da árvore que estamos construindo) quanto uma tática que altera uma hipótese que corresponde a um passo de cima para baixo na construção da prova. Vejamos um exemplo de tática do Coq que altera uma hipótese. Para isto, considere o sequente que tem uma conjunção como hipótese, e uma das componentes desta conjunção como conclusão: `phi` $\wedge$ `psi` $\vdash$ `phi`:

```
Variables phi psi: Prop.

Section landel.
 Hypothesis H: phi /\ psi.
 Lemma landel: phi.
 Proof.
```

Neste momento a janela de prova tem a seguinte forma:

```
1 subgoal (ID 1)

 H : phi /\ psi
 ============================
 phi
```

Podemos usar tática `inversion H` para decompor a hipótese `H`, e obtemos:

```
1 subgoal (ID 1)

 H : phi /\ psi
 H0 : phi
 H1 : psi
 ============================
 phi
```

E a prova pode ser concluída com a tática `assumption`. Não entraremos neste momento nos detalhes técnicos da tática `inversion`, mas a grosso modo, ela gera as condições necessárias para a construção da hipótese onde ela está sendo aplicada. Para mais detalhes recomendamos que o leitor consulte o manual do usuário do Coq[7]. Existem outras táticas que podemos usar no lugar de `inversion` no exemplo anterior, como `destruct` e `elim`, mas elas não serão abordadas agora. Assim, táticas diferentes podem ser usadas para construir provas diferentes de um mesmo sequente, assim como ocorre em papel e lápis.

É importante observar que o Coq, ou assistentes de prova em geral, não são ferramentas para nos ajudar a fazer uma prova, mas sim para verificar provas. A ideia é utilizar os assistentes de prova para mecanizarmos uma prova que já tenha sido feito em papel e lápis, ou uma prova que temos na cabeça (mesmo que apenas um esboço). Iniciar uma prova em um assistente de provas sem saber inicialmente que caminho seguir, tentando a sorte, em geral não é uma boa ideia.

---

[7] `https://coq.inria.fr/distrib/current/refman/`

Exemplo 10. *Provaremos que a disjunção é comutativa. Para isto, precisamos declarar duas variáveis, digamos* `phi` *e* `psi`*, e a hipótese* `H: phi \/ psi`*:*

```
Variables phi psi: Prop.

Section or_comm.
Hypothesis H: phi \/ psi.
Lemma or_comm: psi \/ phi.
Proof.

End or_comm.
```

Temos então o seguinte sequente para ser provado:

```
1 subgoal (ID 1)

 H : phi \/ psi
 ============================
 psi \/ phi
```

A tática `destruct H` *vai dividir a prova em duas subprovas. Na primeira subprova precisamos provar* `psi \/ phi` *a partir de* `phi`*:*

```
 H : phi \/ psi
 H0 : phi
 ============================
 psi \/ phi
```

Aqui podemos utilizar a tática `right`*, que seleciona a componente da direita de uma disjunção, e em seguida, concluímos a prova com* `assumption`*. Na segunda subprova precisamos provar* `psi \/ phi` *a partir de* `psi`*:*

```
 H : phi \/ psi
 H0 : psi
 ============================
 psi \/ phi
```

A seleção da componente da esquerda de uma disjunção consiste em uma aplicação da tática `left`*, e em seguida a prova pode ser concluída com uma aplicação da tática* `assumption`*.*

Exercício 35. *Utilize os seus conhecimentos de Coq para provar que a conjunção é comutativa e associativa.*

O arquivo Coq com o enunciado dos exercícios desta seção está disponível em **removido**[8].

---

[8] Removido para não identificar a autoria, conforme o item 7.2(c) do edital conjunto DEG/EDU 04/2023.

	Contexto explícito	Contexto implícito
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \land \varphi_2}\ (\land_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \land \varphi_2}\ (\land_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \land \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\land_e)$	$\dfrac{\varphi_1 \land \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\land_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \lor \varphi_2}\ (\lor_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \lor \varphi_2}\ (\lor_i)$
4	$\dfrac{\Gamma \vdash \varphi_1 \lor \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\ (\lor_e)$	$\dfrac{\varphi_1 \lor \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\lor_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \bot\end{array}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$
9	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot_e)$	$\dfrac{\bot}{\varphi}\ (\bot_e)$

Tabela 2.3: Regras da Lógica Intuicionista

2.1.5 A Lógica Proposicional Intuicionista

Agora vamos estender a lógica proposicional minimal com uma nova regra chamada de *regra da explosão* ou *regra da eliminação do absurdo intuicionista.* Esta regra nos permite concluir qualquer fórmula a partir do absurdo:

$$\frac{\bot}{\varphi}\ (\bot_e)$$

ou seja, agora a constante $\bot$ tem uma semântica diferente porque agora a partir dela podemos concluir qualquer fórmula. A lógica obtida adicionando-se a regra da explosão à lógica proposicional minimal é denominada *lógica proposicional intuicionista.* Observe que a lógica proposicional minimal possui uma versão mais fraca de regra de explosão. De fato, podemos na lógica proposicional minimal concluir qualquer fórmula negada a partir do absurdo (???). A lógica proposicional intuicionista é conhecida por corresponder à noção de lógica construtiva que é particularmente interessante para a Computação. De forma simplificada, a lógica proposicional intuicionista pode ser vista como a lógica que rejeita a lei do terceiro excluído, ou seja, nesta lógica o sequente $\vdash \varphi \lor \neg\varphi$ não tem prova, quando φ é uma fórmula arbitrária.

Vejamos um exemplo de prova na lógica proposicional intuicionista:

Exemplo 11. *Considere o seguinte sequente $\neg\varphi \lor \gamma \vdash \varphi \to \gamma$. Iniciando esta prova de baixo para cima, isto é, partindo do consequente, podemos aplicar a regra de introdução da implicação:*

$$\dfrac{\begin{array}{c}\neg\varphi \lor \gamma \qquad [\varphi]^u \\ \\ ? \\ \\ \gamma\end{array}}{\varphi \to \gamma}\,(\to_i)\ u$$

Agora precisamos construir uma prova de γ tendo as fórmulas $\neg\varphi \lor \gamma$ e $[\varphi]^u$ como contexto. Uma ideia possível é usar a regra de eliminação da disjunção porque com o lado esquerdo, isto é, com $\neg\varphi$ e com $[\varphi]^u$ temos o absurdo, e com a regra da explosão podemos concluir γ como queríamos. O lado direito da disjunção já é igual a γ, e assim concluímos a prova:

$$\dfrac{\dfrac{\neg\varphi \lor \gamma \qquad \dfrac{\dfrac{[\neg\varphi]^v \qquad [\varphi]^u}{\bot}\,(\neg_e)}{\gamma}\,(\bot_e) \qquad [\gamma]^w}{\gamma}\,(\lor_e)\ v, w}{\varphi \to \gamma}\,(\to_i)\ u$$

Agora vamos refazer esta prova no Coq.

Exemplo 12. *Precisamos declarar duas variáveis, digamos* `phi` *e* `psi`*, e a hipótese* `(~phi)\/psi`*:*

```
Variables phi psi: Prop.

Section or_to_imp.
 Hypothesis H: (~phi) \/ psi.
 Lemma or_to_imp: phi -> psi.
 Proof.
```

Neste momento estamos com a seguinte janela de prova:

```
H : ~ phi \/ psi
============================
phi -> psi
```

Reproduzindo a prova anterior (de baixo para cima), devemos iniciar com a tática `intro` que corresponde à regra ($\rightarrow_i$) , para em seguida dividirmos a prova em função da disjunção na hipótese `H` com a tática `destruct H`. O primeiro subcaso consiste em construir uma prova de `psi` tendo `phi` e `~phi` no contexto. Neste momento podemos utilizar a regra da explosão por meio da tática `contradiction`. O outro ramo é trivial:

```
Variables phi psi: Prop.

Section or_to_imp.
 Hypothesis H: (~phi) \/ psi.
 Lemma or_to_imp: phi -> psi.
 Proof.
   intro H'.
   destruct H.
   - contradiction.
   - assumption.
 Qed.
End or_to_imp.
```

Exercício 36. *Sejam* φ *e* ψ *fórmulas da LP. Construa uma prova para o sequente* $(\neg\neg\varphi) \rightarrow (\neg\neg\psi) \vdash \neg\neg(\varphi \rightarrow \psi)$ *na LPI.*

Exercício 37. *Seja* φ *uma fórmula da LP. Construa uma prova para o sequente* $\vdash \neg\neg(\neg\neg\varphi \rightarrow \varphi)$ *na LPI.*

Exercício 38. *Sejam* φ *e* ψ *fórmulas da LP. Construa uma prova para o sequente* $\vdash \neg\neg(((\varphi \rightarrow \psi) \rightarrow \varphi) \rightarrow \varphi)$ *na LPI.*

2.1.6 A Lógica Proposicional Clássica

Vamos caminhar na direção de mais uma extensão, agora da lógica intuicionista para a lógica clássica. Iniciamos com a lógica proposicional minimal, depois a estendemos para a lógica proposicional intuicionista, e agora vamos estendê-la com a lei do terceiro excluído, obtendo assim a lógica proposicional clássica. Na Tabela 2.4 apresentamos também as regras com contexto explícito para que tenhamos sempre em mente como os contextos mudam de acordo com a aplicação das regras.

	Notação com sequentes	Notação padrão
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\ (\wedge_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\wedge_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \vee \varphi_2}\ (\vee_i)$
4	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\ (\vee_\mathrm{e})$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_\mathrm{i})$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_\mathrm{i})$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \bot\end{array}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$
9	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot_e)$	$\dfrac{\bot}{\varphi}\ (\bot_e)$
10	$\dfrac{}{\vdash \varphi \vee \neg\varphi}\ (\mathrm{LEM})$	$\dfrac{}{\varphi \vee \neg\varphi}\ (\mathrm{LEM})$

Tabela 2.4: Regras da Lógica Clássica

Exemplo 13. *Neste exemplo, vamos construir uma prova de uma regra conhecida como* prova por contradição *(PBC). A ideia desta regra é negar o que se quer provar, e então gerar uma contradição. O sequente a ser provado é o seguinte* $(\neg\varphi) \to \bot \vdash \varphi$*. Veja que queremos provar* φ*, e para isto estamos assumindo que a negação de* φ *nos leva a uma contradição. Vamos então tomar uma instância da* (LEM)*, e provar* φ *via a eliminação da disjunção:*

$$\dfrac{(\text{LEM})\ \dfrac{}{\varphi \vee \neg\varphi} \qquad [\varphi]^u \qquad \dfrac{\dfrac{(\neg\varphi) \to \bot \qquad [\neg\varphi]^v}{\bot}\ (\to_e)}{\varphi}\ (\bot_e)}{\varphi}\ (\vee_e)\ u, v$$

A regra de prova por contradição é dada a seguir. Observe como o contexto muda por conta do descarte de hipóteses:

Notação com sequentes	*Notação padrão*
$\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}$ (PBC)	$\begin{array}{c}[\neg\varphi]^u \\ \vdots \\ \dfrac{\bot}{\varphi}\end{array}$ $(PBC)\ u$

Agora vamos construir esta prova em Coq, mas precisamos de alguns cuidados porque a lógica implementada no Coq é construtiva, e portanto não temos táticas que correspondam a uma aplicação da lei do terceiro excluído. Neste caso, vamos adicionar a lei do terceiro excluído como um axioma:

```
Section pbc.
 Variable phi: Prop.

 Axiom lem: phi \/ ~phi.

 Hypothesis H: ~phi -> False.
 Lemma pbc: phi.
 Proof.
```

e o contexto de prova correspondente é como a seguir:

```
phi : Prop
H : ~ phi -> False
============================
phi
```

Podemos adicionar um axioma ou lema no contexto via a tática `pose proof`. Neste caso, usamos `pose proof lem` para obtermos o seguinte contexto:

```
phi : Prop
H : ~ phi -> False
H0 : phi \/ ~ phi
============================
phi
```

Agora podemos dividir a prova em duas subprovas com a tática `destruct H0`. A primeira subprova é trivial porque o que queremos provar está nas hipóteses. Na segunda subprova, temos pelo menos dois caminhos possíveis para seguir. O primeiro consiste em manipular as hipóteses (raciocínio de cima para baixo) para gerar o absurdo nas hipóteses por meio da tática `apply` no seguinte contexto:

```
phi : Prop
 H : ~ phi -> False
 H0 : ~ phi
 ============================
 phi
```

Como resultado, temos o absurdo como hipótese e podemos provar qualquer coisa via a regra da explosão (tática `contradiction`):

```
 phi : Prop
 H : ~ phi -> False
 H0 : False
 ============================
 phi
```

A prova completa é dada a seguir:

```
Variable phi: Prop.

Axiom lem: phi \/ ~phi.

Hypothesis H: ~phi -> False.
Lemma pbc: phi.
Proof.
 pose proof lem.
 destruct H0.
 - assumption.
 - apply H in H0.
   contradiction.
Qed.
```

O segundo caminho consiste em gerar o absurdo como objetivo a ser provado, ou seja, aplicamos a regra da explosão de baixo para cima na prova. Isto pode ser feito com a tática `apply False_ind` que simplesmente troca o objetivo atual (qualquer que seja ele) pelo absurdo. Neste caso, podemos aplicar a hipótese H (com a tática `apply H`) e concluir a prova com `assumption`.

```
Variable phi: Prop.

Axiom lem: phi \/ ~phi.

Hypothesis H: ~phi -> False.
Lemma pbc: phi.
Proof.
 pose proof lem.
 destruct H0.
 - assumption.
 - apply False_ind.
   apply H.
   assumption.
Qed.
```

Exercício 39. *Acabamos de caracterizar a lógica proposicional clássica como sendo a lógica proposicional intuicionista juntamente com a lei do terceiro excluído (ver Tabela 2.4), mas outras caracterizações são possíveis. Por exemplo, a lógica minimal juntamente com a regra de prova por contradição (PBC) também nos dá a lógica proposicional clássica. Ou seja, a Tabela 2.5 nos dá outra caracterização da lógica proposicional clássica. Para mostrarmos que esta é, de fato, uma caracterização da lógica proposicional clássica precisamos provar tanto a regra da explosão quanto a lei do terceiro excluído a partir das regras da Tabela 2.5. Sendo assim, prove os sequentes a seguir utilizando as regras da Tabela 2.5:*

1. $\bot \vdash \varphi$ *(regra da explosão)*
2. $\vdash \varphi \lor \neg\varphi$ *(lei do terceiro excluído)*
3. *Refaça estas duas provas no Coq.*

Uma terceira caracterização possível para a lógica proposicional clássica é com a regra de eliminação da dupla negação:

Notação com sequentes	Notação padrão
$\dfrac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}\ (\neg\neg_e)$	$\dfrac{\neg\neg\varphi}{\varphi}\ (\neg\neg_e)$

Exercício 40. *Substitua a regra 9 (PBC) na Tabela 2.5 pela regra* $(\neg\neg_e)$*, e prove os seguintes sequentes:*

1. $\bot \vdash \varphi$ *(regra da explosão)*
2. $\vdash \varphi \lor \neg\varphi$ *(lei do terceiro excluído)*
3. $\neg\varphi \to \bot \vdash \varphi$ *(prova por contradição)*
4. *Refaça estas três provas no Coq.*

Considerando que a negação, digamos $\neg\varphi$, é o mesmo que $\varphi \to \bot$, é fácil ver que as regras de eliminação da dupla negação e prova por contradição são maneiras diferentes de escrever a mesma coisa (por que?). Uma outra caracterização possível da lógica proposicional clássica envolve a chamada *lei de Peirce* (LP), como detalhado no exemplo a seguir:

	Contexto explícito	Contexto implícito
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\ (\wedge_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\wedge_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \vee \varphi_2}\ (\vee_i)$
4	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\ (\vee_\mathrm{e})$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_\mathrm{i})$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_\mathrm{i})$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \bot\end{array}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$
9	$\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}\ (\mathrm{PBC})$	$\dfrac{\begin{array}{c}[\neg\varphi]^u \\ \vdots \\ \bot\end{array}}{\varphi}\ (PBC)\ u$

Tabela 2.5: Regras da Lógica Clássica (versão 2)

Notação com sequentes	Notação padrão
$\cfrac{}{\vdash ((\varphi \to \psi) \to \varphi) \to \varphi}$ (LP)	$\cfrac{}{((\varphi \to \psi) \to \varphi) \to \varphi}$ (LP)

Exercício 41. *Assuma a regra (LP) acima, e prove o sequente* $\vdash \varphi \vee \neg\varphi$ *utilizando as regras da Tabela 2.3*

Exercício 42. *Sejam* φ *e* ψ *fórmulas da LP. Prove* $\varphi \wedge \psi \dashv\vdash \neg(\neg\varphi \vee \neg\psi)$.

Exercício 43. *Sejam* φ *e* ψ *fórmulas da LP. Prove* $\varphi \to \psi \dashv\vdash (\neg\varphi) \vee \psi$.

Exercício 44. *Sejam* φ *e* ψ *fórmulas da LP. Prove o sequente* $(\neg\varphi) \to \psi \vdash (\neg\psi) \to \varphi$.

Exercício 45. *Sejam* φ *e* ψ *fórmulas quaisquer da LP. Prove os sequentes* $\neg\neg(\varphi \vee \psi) \dashv\vdash (\neg\neg\varphi) \vee (\neg\neg\psi)$.

Exercício 46. $\varphi \leftrightarrow \neg\varphi \vdash \bot$[a]

---

[a] Note que a bi-implicação $\psi \leftrightarrow \gamma$ é apenas uma notação compacta para $(\psi \to \gamma) \wedge (\gamma \to \psi)$.

Exercício 47. *Sejam* φ *e* ψ *fórmulas quaisquer da LP. Mostre que o sequente* $\varphi \to \psi \vdash (\neg\varphi) \vee \psi$ *não possui uma derivação intuicionista.*

Exemplo 14. *Considere o seguinte problema: Em uma ilha moram apenas dois tipos de pessoas: as honestas, que sempre falam a verdade; e as desonestas, que sempre mentem. Um viajante, ao passar por esta ilha encontra três moradores chamados A, B e C. O viajante pergunta para o morador A:*
"Você é honesto ou desonesto?" A responde algo incompreensível, e o viajante pergunta para B: "O que ele disse?" B então responde "Ele disse que é desonesto". Neste momento C se manifesta: "Não acredito nisto! Isto é uma mentira!". Questão: C é honesto ou desonesto?

Para resolver este problema pense no que ocorre se um morador desta ilha, digamos X, disser "Eu sou desonesto"? Isto nos levaria a uma contradição! De fato, se X for honesto então ele disse a verdade, e portanto é desonesto. Por outro lado, se X é desonesto então ele mentiu, e portanto é honesto. Assim, como A não poderia ter dito que é desonesto, podemos concluir que B é desonesto! E portanto, C é honesto! Vamos construir uma prova de que este raciocínio está em correto usando a teoria que estudamos? O ponto de partida é construir um sequente que corresponda ao enunciado deste problema. Que variáveis proposicionais vamos precisar? Certamente precisamos de variáveis que nos permitam caracterizar quando um morador é ou não honesto. Assim, utilizaremos três variáveis proposicionais com a seguinte semântica:

- *a: o morador A é honesto;*
- *b: o morador B é honesto;*
- *c: o morador C é honesto.*

Desta forma, a negação de qualquer destas variáveis significa que o morador correspondente é desonesto. Agora precisamos representar o que foi dito por cada um dos moradores por meio de uma fórmula da lógica proposicional. Considere o que disse o morador B: "Ele disse que é desonesto", quer dizer, o morador B disse que o morador A disse que era desonesto. Como codificar este fato por meio de uma fórmula da LP? Vamos iniciar considerando uma situação geral e mais simples. Digamos que um morador X tenha dito Y, isto é, "X disse Y". Que fórmula da LP corresponde a este fato? Suponha que a variável x codifica a proposição "X é honesto". Então observe que, se X for honesto então o que ele disse é verdade, ou seja, tanto x quanto Y são verdade. Por outro lado, se X for desonesto então Y é falso, e tanto x quanto Y são falsos. Assim, podemos concluir que as variáveis x e Y são equivalentes, no sentido que ou ambas são verdadeiras, ou ambas são falsas. Assim, podemos representar a afirmação "X disse Y"pela fórmula $x \leftrightarrow Y$. Voltando então ao nosso problema original, podemos agora representar o fato de que o morador B disse que o morador A disse que era desonesto pela fórmula $b \leftrightarrow (a \leftrightarrow (\neg a))$. O morador C por sua vez, disse que B mentiu, o que corresponde a fórmula $c \leftrightarrow (\neg b)$. Com isto podemos montar o sequente a ser provado: $b \leftrightarrow (a \leftrightarrow (\neg a)), c \leftrightarrow (\neg b) \vdash c$.

Exercício 48. *Prove o sequente $b \leftrightarrow (a \leftrightarrow (\neg a)), c \leftrightarrow (\neg b) \vdash c$ construído no exemplo anterior.*

Exercício 49. *Considere uma ilha onde moram apenas dois tipos de pessoas: as honestas, e que portanto sempre falam a verdade; e as desonestas, que sempre mentem. Um viajante, ao passar por esta ilha encontra três moradores chamados A, B e C. O viajante pergunta para o morador A: "Quantos, dentre vocês três, são desonestos?" A responde algo incompreensível, e o viajante pergunta para B: "O que ele disse?" B então responde "Ele disse que exatamente dois de nós somos desonestos". Neste momento C se manifesta: "Não acredito nisto! Isto é uma mentira!". Questão: C é honesto ou desonesto?*

Exercício 50. *Em uma ilha moram apenas dois tipos de habitantes: os honestos, que sempre falam a verdade; e os desonestos, que sempre mentem. Você encontra dois habitantes desta ilha, digamos João e José. João diz que José é desonesto. José diz "Nem João nem eu somos desonestos". Você consegue determinar qual dos dois é honesto e qual é desonesto?*

No exemplo anterior, utilizamos a associação do valor de verdade (verdadeiro ou falso) de uma variável proposicional para resolver um problema. Esta abordagem está relacionado com a semântica da lógica proposicional clássica que nos fornece os meios para concluir quando uma fórmula é verdadeira ou falsa. A gramática (2.2) define como são as fórmulas da LP, a partir de seis construtores:

1. O primeiro denota uma variável proposicional, e caracteriza uma fórmula atômica, i.e. uma fórmula que não pode ser subdividida em uma fórmula menor.

2. O segundo construtor é uma constante que denota o absurdo ($\perp$), que também é uma fórmula atômica. O absurdo é utilizado para representar uma fórmula que tem valor de verdade "falso (F)". É importante observar que podemos associar a qualquer fórmula da LP apenas dois valores de verdade, a saber: verdadeiro (T) ou falso (F).

3. O terceiro construtor denota a negação e nos permite construir uma nova fórmula a partir de uma fórmula dada. Assim, dada uma fórmula φ, podemos construir a sua negação $(\neg\varphi)$. A semântica da negação é a que conhecemos intuitivamente: se uma fórmula φ é verdadeira (T) então sua negação é falsa (F), e vice-versa. Normalmente, representamos este fato via a seguinte tabela:

φ	$(\neg\varphi)$
T	F
F	T

4. O quarto construtor denota a conjunção e nos permite construir uma nova fórmula a partir de duas fórmulas dadas. Assim, dadas duas fórmulas φ_1 e φ_2, podemos construir a sua conjunção $(\varphi_1 \wedge \varphi_2)$. A semântica da conjunção também é a usual, isto é, a conjunção $(\varphi_1 \wedge \varphi_2)$ é verdadeira somente quando φ_1 e φ_2 são simultaneamente verdadeiras:

φ_1	φ_2	$(\varphi_1 \wedge \varphi_2)$
T	T	T
T	F	F
F	T	F
F	F	F

Aqui é importante observar que a leitura da construção da conjunção na gramática 2.2 não diz que suas componentes são iguais (apesar da utilização do mesmo símbolo φ nas duas componenetes). Lembre-se que a leitura desta construção em 2.2 é: dadas duas fórmulas (não necessariamente iguais!), podemos construir a sua conjunção. Alternativamente, poderíamos ter escrito a gramática 2.2 da seguinte forma equivalente:

$$\varphi, \psi ::= p \mid \perp \mid (\neg\varphi) \mid (\varphi \wedge \psi) \mid (\varphi \vee \psi) \mid (\varphi \rightarrow \psi) \tag{2.3}$$

5. O quinto construtor denota a disjunção e, como no caso anterior, nos permite construir uma nova fórmula a partir de duas fórmulas dadas. Assim, dadas duas fórmulas φ_1 e φ_2, podemos construir a sua disjunção $(\varphi_1 \vee \varphi_2)$, cuja semântica é dual à semântica da conjunção: a disjunção $(\varphi_1 \vee \varphi_2)$

é falsa somente quando φ_1 e φ_2 são simultaneamente falsas.

φ_1	φ_2	$(\varphi_1 \vee \varphi_2)$
T	T	T
T	F	T
F	T	T
F	F	F

6. O sexto construtor é a implicação. Assim, dadas duas fórmulas φ_1 e φ_2, podemos construir a sua implicação $(\varphi_1 \rightarrow \varphi_2)$ com a semântica dada na tabela abaixo.

φ_1	φ_2	$(\varphi_1 \rightarrow \varphi_2)$
T	T	T
T	F	F
F	T	T
F	F	T

O sentido usual da implicação assume implicitamente uma relação de causa e efeito, ou causa e consequência no sentido de que o antecedente φ_1 é o que gera o consequente φ_2 como em "Se eu não beber água então ficarei desidratado". No entanto, o sentido da implicação na lógica é um pouco diferente pois tem como fundamento a *preservação da verdade*, que não necessariamente possui uma relação de causa e efeito. Por exemplo, a proposição "Se 2+2=4 então o dia tem 24 horas"é verdadeira, mas não existe relação causal entre a igualdade 2+2=4 e o fato de o dia ter 24 horas de duração.

Uma gramática como 2.2 (ou 2.3) nos fornece as regras sintáticas para a construção das fórmulas da LP. São quatro construtores recursivos (negação, conjunção, disjunção e implicação) também chamados de conectivos lógicos, e dois não recursivos.

Apesar da gramática apresentada acima não incluir a bi-implicação, este é um conectivo bastante utilizado. De fato, a bi-implicação, já utilizada em exemplos anteriores, pode ser reescrita em usando a implicação e conjunção. Como exercício construa a tabela verdade da bi-implicação e observe que $\varphi \leftrightarrow \psi$ é verdadeira somente quando φ e ψ possuem o mesmo valor de verdade. Adicionalmente, dizemos que duas fórmulas φ e ψ são **equivalentes** quando a fórmula $\varphi \leftrightarrow \psi$ é uma tautologia:

Tautologia	Uma fórmula que é sempre verdadeira, independentemente dos valores de verdade associados às suas variáveis.
Contradição	Uma fórmula que é sempre falsa, independentemente dos valores de verdade associados às suas variáveis.
Contingência	Uma fórmula que pode ser tanto verdadeira quanto falsa dependendo dos valores de verdade associados às suas variáveis.

As tautologias e as contradições são particularmente importantes, e possuem símbolos especiais para representá-las. Nas gramáticas 2.2 e 2.3 já vimos que a constante $\perp$ é o representante das contradições. As tautologias, por sua vez, podem ser representadas pelo símbolo $\top$.

Agora chegamos em um momento chave do nosso estudo. Considere um sequente arbitrário, digamos $\Gamma \vdash \varphi$, onde Γ é um conjunto finito de fórmulas da LP. Podemos então perguntar: é possível provar este sequente? Ou em outras palavras, qualquer sequente possui uma prova? A resposta certamente é não. Se tudo pudesse ser provado então não teríamos razão para estudar a lógica proposicional. Como então é possível separar os sequentes que têm prova dos que não podem ser provados? Para responder esta pergunta precisamos inicialmente compreender a noção de **consequência lógica**. Dizemos que uma fórmula φ é consequência lógica da fórmula ψ, notação $\psi \models \varphi$, se φ for verdadeira sempre que ψ for verdadeira. Este conceito pode ser facilmente estendido para um conjunto Γ de fórmulas, de forma que $\Gamma \models \varphi$, isto é, φ é consequência lógica do conjunto Γ se φ for verdadeira sempre que as fórmulas em Γ forem verdadeiras. Agora podemos enunciar dois teoremas importantes que nos permitirão responder à questão anterior:

Teorema 15 (Correção da LP)**.** *Sejam Γ um conjunto, e φ uma fórmula da lógica proposicional. Se $\Gamma \vdash \varphi$ então $\Gamma \models \varphi$.*

A prova deste teorema é por indução em $\Gamma \vdash \varphi$. Não detalharemos aqui esta prova, que pode ser encontrada por exemplo em [4].

Teorema 16 (Completude da LP)**.** *Sejam Γ um conjunto, e φ uma fórmula da lógica proposicional. Se $\Gamma \models \varphi$ então $\Gamma \vdash \varphi$.*

A prova do teorema de completude da LP é um pouco mais complexa do que a prova de correção, e também pode ser encontrada em [4]. Note que este lema responde a nossa pergunta anterior: um sequente tem prova exatamente quando seu consequente for consequência lógica do seu antecedente.

A lógica proposicional nos permite resolver diversos problemas, e constitui a base de tudo o que faremos nos próximos capítulos. Apesar de muito importante como ponto de partida no estudo que estamos fazendo, a lógica proposicional possui limitações importantes, como a impossibilidade de quantificar de forma explícita sobre elementos de um conjunto. Por exemplo, podemos representar a sentença "Todo mundo gosta de Matemática"na LP via uma variável proposicional, mas esta representação não expressa a quantificação universal "Todo mundo"de forma explícita. O mesmo vale para uma sentença da forma "Existe um número natural que não é primo". O próprio princípio da indução, tão importante em Matemática e Computação, precisa de uma linguagem mais expressiva do que a proposicional. A lógica que nos permitirá expressar este tipo de quantificação (tanto existencial quanto universal) é conhecida como *Lógica de Primeira Ordem* (LPO), ou *Lógica de Predicados* que estudaremos no próximo capítulo.

2.2 A Lógica de Primeira Ordem

Nesta seção vamos em um certo sentido estender a Lógica Proposicional para ganhar em poder de expressividade. Como é a gramática da Lógica de Primeira Ordem (LPO)? Isto é, qual a linguagem que precisamos para conseguir expressar quantificação universal e existencial? Inicialmente, precisamos representar os elementos que podem ser quantificados. Assim, diferentemente do caso proposicional, temos duas classes de objetos na LPO: *termos* e *fórmulas.* Os termos são representados pela seguinte gramática:

$$t ::= x \mid f(t, \ldots, t) \tag{2.4}$$

ou seja, os termos são construídos a partir de variáveis (no sentido usual da palavra em Matemática) e, funções com uma certa aridade (i.e número de argumentos). Observe que os termos vão representar os elementos do conjunto sobre o qual podemos quantificar e caracterizar por meio de propriedades. Por exemplo, considere o conjunto dos números naturais $\mathbb{N}$. Neste caso, as variáveis representam números naturais, e exemplos de funções são: sucessor (aridade 1), soma (aridade 2), etc. O conjunto das variáveis de um termo t, notação (t), consiste no conjunto das variáveis que ocorrem em t, e pode ser definido indutivamente por:

Definição 17. *O conjunto $var(t)$ das variáveis que ocorrem no termo t é definido indutivamente como a seguir:*

1. $var(x) = \{x\}$;
2. $var(f(t_1, t_2, \ldots, t_n)) = var(t_1) \cup var(t_2) \cup \ldots \cup var(t_n)$.

Denotaremos por $t[[x/u]]$ o termo obtido ao se substituir todas as ocorrências da variável x pelo termo u no termo t.

As fórmulas da LPO utilizam os mesmos conectivos da LP e são definidas pela seguinte gramática:

$$\varphi ::= p(t, \ldots, t) \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \mid \exists_x\varphi \mid \forall_x\varphi \tag{2.5}$$

onde o primeiro construtor representa uma fórmula atômica, e os dois últimos representam, respectivamente, a quantificação existencial e universal. Note que as fórmulas atômicas representam fórmulas que não podem ser decompostas, e que têm termos como argumentos. Em uma fórmula atômica da forma $p(t_1, \ldots, t_n)$, p é um *predicado* de aridade n, e t_1, ..., t_n são termos. A LPO é a lógica utilizada no dia a dia dos matemáticos, ainda que de maneira informal. Com os predicados podemos expressar propriedades dos termos. Por exemplo, ainda no conjunto dos números naturais, podemos expressar a propriedade de um número natural ser primo por meio de um predicado unário, digamos p. Desta forma, a fórmula $p(x)$ pode expressar o fato de x ser primo. Outros exemplos de fórmulas atômicas incluem os predicados $\leq$, $\geq$, $<$ e $>$ que normalmente usamos em notação infixa como em $2 \leq 5$, por exemplo.

Observe que agora existem dois tipos de variáveis na linguagem da Lógica de Primeira Ordem. Por exemplo, considere as fórmulas $q(x)$ e $\forall_x p(x)$. Em $\forall_x p(x)$ ocorrência da variável x em $p(x)$ está **ligada** ao quantificador universal, enquanto que na fórmula $q(x)$, a variável x está **livre**. De uma forma geral, dizemos que uma ocorrência de uma variável é ligada, se ela estiver no escopo de um quantificador (universal ou existencial), e livre, se a ocorrência não estiver no escopo de nenhum quantificador. Observe que uma variável pode ocorrer livre e ligada em uma mesma fórmula: $q(x) \vee \forall_x p(x)$. O conjunto das variáveis livres de uma fórmula φ, notação $FV(\varphi)$, é definido indutivamente como segue:

Definição 18. *Seja φ uma fórmula da LPO. O conjunto $FV(\varphi)$ das variáveis livres da fórmula φ é definido indutivamente na estrutura de φ por:*

1. $FV(p(t_1, t_2, \ldots, t_n)) = var(t_1) \cup var(t_2) \cup \ldots \cup var(t_n)$;
2. $FV(\bot) = \{\}$;
3. $FV(\neg\psi) = FV(\psi)$;
4. $FV(\psi \star \gamma) = FV(\psi) \cup FV(\gamma)$, *onde* $\star \in \{\wedge, \vee, \to\}$;
5. $FV(Q_x\psi) = FV(\psi) \backslash \{x\}$, *onde* $Q \in \{\forall, \exists\}$.

De maneira análoga podemos definir o conjunto das variáveis ligadas de uma fórmula:

Definição 19. *Seja φ uma fórmula da LPO. O conjunto $BV(\varphi)$ das variáveis ligadas da fórmula φ é definido indutivamente na estrutura de φ por:*

1. $BV(p(t_1, t_2, \ldots, t_n)) = \{\}$;
2. $BV(\bot) = \{\}$;
3. $BV(\neg\psi) = BV(\psi)$;
4. $BV(\psi \star \gamma) = BV(\psi) \cup BV(\gamma)$, *onde* $\star \in \{\wedge, \vee, \rightarrow\}$;
5. $BV(Q_x\psi) = BV(\psi) \cup \{x\}$, *onde* $Q \in \{\forall, \exists\}$.

Estas noções são importantes porque a operação de substituição na Lógica de Primeira Ordem é definida de tal forma a evitar captura de variáveis, diferentemente da substituição feita em termos vista anteriormente. Isto significa que, por exemplo, se quisermos substituir a ocorrência de y em $\forall_x p(x, y)$ por x, o resultado não pode ser $\forall_x p(x, x)$ já que neste caso a segunda ocorrência de x que era livre, passou a ser ligada depois da substituição, ou seja, a segunda ocorrência de x foi capturada. Para evitar este problema, podemos renomear as variáveis ligadas de uma fórmula sempre que necessário. De fato, observe que as fórmulas $\forall_x q(x)$, $\forall_y q(y)$ e $\forall_z q(z)$ têm todas a mesma semântica. Isto significa que o renomeamento de variáveis ligadas não muda o sentido, ou significado, de uma fórmula. Para enfatizarmos a operação de substituição que definiremos a seguir, denotaremos por $\varphi[x/t]$ o resultado de substituir todas as ocorrências livres de x na fórmula φ pelo termo t. Quando a variável a ser substituída não precisar ser enfatizada (por exemplo, por poder ser facilmente obtida do contexto), escreveremos simplesmente $\varphi(t)$ ao invés de $\varphi[x/t]$.

Definição 20. *Seja φ uma fórmula da LPO. A operação de substituir todas as ocorrências livres da variável x pelo termo t em φ, notação $\varphi[x/t]$ é definida indutivamente na estrutura da fórmula φ da seguinte forma:*

1. $p(t_1, t_2, \ldots, t_n)[x/t] = p(t_1[[x/t]], t_2[[x/t]], \ldots, t_n[[x/t]])$;
2. $\bot[x/t] = \bot$;
3. $(\neg\psi)[x/t] = \neg(\psi[x/t])$;
4. $(\psi \star \gamma)[x/t] = (\psi[x/t]) \star (\gamma[x/t])$, *onde* $\star \in \{\vee, \wedge, \rightarrow\}$;
5. $(Q_y\psi)[x/t] = \begin{cases} Q_y\psi, & \text{se } x = y; \\ Q_y(\psi[x/t]), & \text{se } y \notin var(t); \\ Q_z(\psi[y/z][x/t]), & \text{c.c.} \end{cases}$
   onde z é uma variável nova, e $Q \in \{\forall, \exists\}$.

Observe que o primeiro caso do item 5 da definição anterior, a substituição não tem nenhum efeito sobre a fórmula quando a variável da substituição coincide com a variável do quantificador ($x = y$), e portanto variáveis ligadas não são substituídas. O caso em que $y \notin var(t)$ faz a propagação da substituição para dentro do corpo do quantificador já que não há possibilidade de captura de variável. Por fim, quando $x \neq y$ e $y \in var(t)$ a variável do quantificador é renomeada para um nome novo, no caso z, as ocorrências de y em ψ são renomeadas para z e então a substituição é propagada para dentro do corpo do quantificador.

O sistema de dedução natural na LPO possui as mesmas regras utilizadas no caso proposicional, mas agora aplicadas a fórmulas da LPO, e adicionalmente temos as regras de introdução e eliminação para os quantificadores que apresentamos a seguir.

A regra de introdução do quantificador universal permite a construção de uma prova de uma fórmula da forma $\forall_x\varphi(x)$, ou seja, queremos concluir que a propriedade φ é satisfeita por qualquer elemento x do domínio. Mas o que precisamos para garantir que todo elemento x do domínio tenha a propriedade φ? Uma maneira seria tentar a construção individual de cada uma destas provas, ou seja, suponha que o domínio seja o conjunto $\{x_0, x_1, x_2 \ldots\}$ que pode ser finito ou infinito, e considere uma prova de $\varphi(x_0)$, isto é, uma prova de que x_0 satisfaz a propriedade φ. Seria possível repetir esta prova para x_1, x_2, e assim sucessivamente? Se pudermos repetir a mesma prova para todos os elementos do domínio então certamente podemos concluir $\forall_x\varphi(x)$. Para que uma generalização desta forma seja possível precisamos que a prova de $\varphi(x_0)$ não dependa de hipótese que assuma alguma informação sobre x_0.

$$\frac{\varphi(x_0)}{\forall_x\varphi(x)}\ (\forall_i) \qquad \text{se a prova de } \varphi(x_0) \text{ não depende de hipótese não-descartada que contenha } x_0.$$

A regra de eliminação do quantificador universal nos permite instanciar a variável quantificada universalmente x com qualquer elemento t do domínio.

$$\frac{\forall_x \varphi(x)}{\varphi(t)}\ (\forall_e)$$

A analogamente, a regra de introdução do quantificador existencial nos permite concluir que existe um elemento que satisfaz a propriedade φ a partir da prova de que algum elemento do domínio, digamos t, satisfaça a propriedade φ.

$$\frac{\varphi(t)}{\exists_x \varphi(x)}\ (\exists_i)$$

Por fim, a regra de eliminação do quantificador existencial é dada como a seguir:

$$\frac{\exists_x \varphi(x) \qquad \begin{matrix} [\varphi(x_0)]^u \\ \vdots \\ \gamma \end{matrix}}{\gamma}\ (\exists_e)\ u \qquad \text{onde } x_0 \text{ é uma variável nova que não ocorre em } \gamma.$$

Nesta regra provamos γ a partir de uma prova de $\exists_x \varphi(x)$, e de uma prova de γ a partir da suposição $\varphi(x_0)$. Ou seja, como temos uma prova de $\exists_x \varphi(x)$, então temporariamente assumimos que x_0 (um novo elemento que, portanto, não pode ter sido utilizado antes) satisfaz a propriedade φ. Se a partir desta suposição pudermos provar uma fórmula, digamos γ, que não dependa de x_0 então podemos concluir γ após descartar a suposição $\varphi(x_0)$.

Exercício 51. *Apresente derivações em Dedução Natural para os sequentes* $\forall_x \neg\varphi \dashv\vdash \neg\exists_x \varphi$ *na LPO minimal.*

Exercício 52. *Apresente derivações em Dedução Natural para os sequentes* $\neg\forall x\, \phi \dashv\vdash \exists x\, \neg\phi$*, e em seguida classifique cada prova como minimal, intuicionista ou clássica.*

Exercício 53. *Apresente derivações em Dedução Natural para os sequentes* $\forall x\, \phi \dashv\vdash \neg\exists x\, \neg\phi$*, e em seguida classifique cada prova como minimal, intuicionista ou clássica.*

Exercício 54. *Apresente derivações em Dedução Natural para os sequentes* $\exists x\, \phi \dashv\vdash \neg\forall x\, \neg\phi$*, e em seguida classifique cada prova como minimal, intuicionista ou clássica.*

Exercício 55. *Apresente derivações em Dedução Natural para os sequentes a seguir assumindo que x não ocorre livre em ψ, e em seguida classifique cada prova como minimal, intuicionista ou clássica.*

1. $(\forall x\, \phi) \wedge \psi \vdash \forall x\, (\phi \wedge \psi)$
2. $(\exists x\, \phi) \wedge \psi \vdash \exists x\, (\phi \wedge \psi)$
3. $\forall x\, (\psi \to \phi) \vdash \psi \to \forall x\, \phi$
4. $\forall x\, (\phi \to \psi) \vdash (\exists x\, \phi) \to \psi$

Exercício 56. *Prove que não existe uma derivação intuicionista para os sequentes a seguir:*

1. $\neg\exists_x \neg\varphi \vdash \forall_x \varphi$
2. $\neg\forall_x \neg\varphi \vdash \exists_x \varphi$
3. $\forall_x \neg\neg\varphi \vdash \neg\neg\forall_x \varphi$

Assim como a LP, a LPO é correta e completa, mas estes resultados não serão provados aqui (Veja, por exemplo, [4]).

2.2.1 Indução

Indução é uma técnica de prova muito poderosa que desempenha um papel fundamental tanto em Matemática quanto em Computação. Estudaremos esta técnica partindo dos conjuntos definidos indutivamente, mas o leitor interessado em se aprofundar no tema pode consultar, por exemplo, os livros [10, 17]. Um conjunto, digamos, A, é *definido indutivamente* se seus elementos podem ser construídos a partir de um conjunto finito de regras de inferência da forma:

$$\frac{}{a \in A} \qquad\qquad \frac{a_1 \in A \qquad a_2 \in A \ldots\ a_n \in A}{a \in A}$$

A regra da esquerda é um axioma, e diz que a é um elemento do conjunto A, enquanto que na regra da direita temos que se $a_1, a_2, \ldots, a_n$ são elementos de A então a também é um elemento de A. Por exemplo, qualquer conjunto finito pode ser definido indutivamente com um axioma para cada elemento. De fato, o conjunto *Sem* dos dias da semana pode ser definido indutivamente via 7 axiomas:

$$\frac{}{\text{domingo} \in \mathit{Sem}}\ (\textsc{dom}) \qquad \frac{}{\text{segunda-feira} \in \mathit{Sem}}\ (\textsc{seg}) \qquad \frac{}{\text{terça-feira} \in \mathit{Sem}}\ (\textsc{ter})$$

$$\frac{}{\text{quarta-feira} \in \mathit{Sem}}\ (\textsc{qua}) \qquad \frac{}{\text{quinta-feira} \in \mathit{Sem}}\ (\textsc{qui}) \qquad \frac{}{\text{sexta-feira} \in \mathit{Sem}}\ (\textsc{sex})$$

$$\frac{}{\text{sábado} \in \mathit{Sem}}\ (\textsc{sab})$$

Alternativamente, podemos utilizar uma notação mais compacta definir o conjunto *Sem*: se d é uma variável que representa um elemento qualquer de *Sem* então a gramática a seguir é equivalente à definição via as 7 regras de inferência apresentadas acima:

$$d ::= \text{domingo} \mid \text{segunda-feira} \mid \text{terça-feira} \mid \text{quarta-feira} \mid \text{quinta-feira} \mid \text{sexta-feira} \mid \text{sábado}$$

O conjunto *bool*, muito popular em Ciência da Computação, possui apenas dois elementos:

$$\frac{}{\text{true} \in \mathit{bool}}\ (\textsc{true}) \qquad \frac{}{\text{false} \in \mathit{bool}}\ (\textsc{false})$$

Alternativamente, se b denota um valor booleano, podemos descrever o conjunto *bool* pela gramática:

$$b ::= \text{true} \mid \text{false}$$

Mas conjuntos definidos indutivamente também podem ser infinitos. O exemplo mais conhecido provavelmente é o conjunto dos números naturais $\mathbb{N}$, que pode ser definido pelas regras de inferência a seguir:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}$$

Neste caso, a regra da esquerda é um axioma que diz que o 0 é um número natural, enquanto que a regra da direita diz que se n é um número natural então $S\ n$ (o sucessor de n) também é um número natural. A gramática equivalente a estas duas regras é dada por:

$$n ::= 0 \mid S\ n \tag{2.6}$$

Agora que sabemos o que são conjuntos definidos indutivamente podemos voltar ao tema da indução, que é uma técnica de prova que nos permite provar propriedades de conjuntos definidos indutivamente. Como provar que os elementos de um conjunto definido indutivamente, digamos A, satisfazem uma dada propriedade P? Se o conjunto A for finito então podemos testar individualmente se cada elemento satisfaz a propriedade P. Mesmo que A seja um conjunto grande, depois de uma quantidade finita de tempo teremos uma prova de que os elementos de A satisfazem a propriedade P. E se o conjunto A for infinito? A ideia é bastante intuitiva: suponha que os elementos deste conjunto possam ser colocados um após o outro como peças de um dominó, de tal forma que, se uma peça qualquer for derrubada então a peça que está logo em seguida também é derrubada. Então podemos concluir, que se a primeira peça for derrubada então **todas** as outras serão derrubadas. Ou seja, voltando ao contexto de propriedades de elementos de um conjunto, a ideia é provar que se um elemento arbitrário do conjunto satisfaz a propriedade então o próximo elemento também satisfaz a propriedade. Se esta prova puder ser feita juntamente com a prova de que o primeiro elemento do conjunto também satisfaz a propriedade então podemos concluir que todos os elementos do conjunto satisfazem a propriedade.

Vamos iniciar este estudo sobre indução no contexto dos números naturais, onde esta noção de ordem é bem clara: o primeiro elemento é o 0, em seguida vem o 1, depois o 2, etc. De uma forma geral, depois de um número natural k vem o natural $S\ k$, o sucessor de k que também escrevemos como $k + 1$. A indução no contexto dos números naturais é conhecida como *indução matemática*, e será explorada na próxima seção.

A gramática (2.6) possui dois construtores: 0 e S. O primeiro diz que 0 é um número natural, e o segundo diz que a partir de um natural já construído, digamos n, podemos construir um outro natural, a saber, $S\ n$, ou seja, o sucessor de n. Muito bem, agora considere uma propriedade qualquer dos números naturais. Por exemplo, a que diz que a soma dos n primeiros números ímpares é igual a n^2. Como podemos provar esta propriedade? Isto mesmo, por indução! O que diz mesmo o princípio de indução para os números naturais? Diz que se uma propriedade P vale para 0 (base da indução), e se, supondo que P vale para um natural arbitrário k (hipótese de indução), podemos provar que ela vale também para $S\ k$ (o sucessor de k)[9] (passo indutivo) então podemos concluir que P vale para todos os números naturais. Esquematicamente, podemos apresentar este princípio, denominado *Princípio da Indução Matemática (PIM)*, como a seguir:

$$\frac{P\ 0 \qquad\qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, P\ n}\ (PIM)$$

---

[9] Note que o sucessor de k pode ser escrito como $S\ k$ ou $k + 1$.

Exemplo 21. *Queremos provar que a soma dos n primeiros números ímpares é igual a n^2. Esta propriedade vale trivialmente para o 0 (a soma dos 0 primeiros números ímpares é igual a 0^2). Agora suponha que a soma dos k primeiros números ímpares seja igual a k^2 (hipótese de indução). O $(k+1)$-ésimo número ímpar é igual a $2.k+1$ (por que?), e portanto a soma dos $k+1$ primeiros números ímpares é $k^2+2.k+1=(k+1)^2$, como queríamos provar.*

Uma outra forma de resolver este problema em um contexto mais formal pode ser feita a partir de uma definição formal da soma dos n primeiros números ímpares por meio do somatório $\sum_{i=1}^{n}(2.i-1)$, que por definição é igual a 0, se $n=0$. Queremos provar que $\sum_{i=1}^{n}(2.i-1)=n^2$, para todo número natural n. Aplicando o princípio da indução, teremos 2 casos para analisar:

- *(**Base da indução**): A base da indução se dá quando $n=0$, e é trivial porque o lado esquerdo da igualdade é igual a 0 por definição.*
- *(**Passo indutivo**): O passo indutivo é a parte interessante de qualquer prova por indução. Neste caso específico, vamos assumir que a propriedade que queremos provar vale para um número natural arbitrário, digamos k, e provaremos que esta propriedade continua valendo para o natural $k+1$. Ou seja, assumimos que $\sum_{i=1}^{k}(2.i-1)=k^2$, e vamos provar que $\sum_{i=1}^{k+1}(2.i-1)=(k+1)^2$. Partindo do lado esquerdo desta igualdade, podemos decompor o somatório da seguinte forma $\sum_{i=1}^{k+1}(2.i-1)=\sum_{i=1}^{k}(2.i-1)+(2.k+1)$, e agora podemos utilizar a hipótese de indução (h.i.) para assim chegarmos ao lado direito da igualdade: $\sum_{i=1}^{k+1}(2.i-1)=\sum_{i=1}^{k}(2.i-1)+(2.k+1)\overset{h.i.}{=}k^2+(2.k+1)=(k+1)^2$.*

Por fim, apresentamos esta prova na forma de árvore:

$$\dfrac{\dfrac{0=0}{\sum\limits_{i=1}^{0}(2.i-1)=0^2} \qquad \dfrac{\dfrac{\dfrac{[\sum\limits_{i=1}^{k}(2.i-1)=k^2]^u}{\sum\limits_{i=1}^{k}(2.i-1)+(2.k+1)=(k+1)^2}}{\sum\limits_{i=1}^{k+1}(2.i-1)=(k+1)^2}}{\sum\limits_{i=1}^{k}(2.i-1)=k^2\to\sum\limits_{i=1}^{k+1}(2.i-1)=(k+1)^2}\,(\to_i)\ u}{\sum\limits_{i=1}^{n}(2.i-1)=n^2}\,(\text{Ind. em } n)$$

Existem propriedades que valem apenas para um subconjunto próprio dos números naturais:

Por exemplo, $2^n<n!$ só vale para $n\geq 4$. Para este tipo de problema utilizamos uma generalização do PIM onde a base de indução não precisa ser o 0. Chamaremos esta variação de *Princípio da Indução Generalizado (PIG)*:

$$\frac{P\ m \qquad\qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, n \geq m \Longrightarrow P\ n}\ (PIG)$$

Exemplo 22. *Prove que* $2^n < n!, \forall n \geq 4$.

1. *(Base de indução) A propriedade vale para* $n = 4$, *o que é trivial, e;*
2. *(Passo indutivo) Mostraremos que* $2^{(S\ k)} < (S\ k)!$ *assumindo que* $2^k < k!, \forall k \geq 4$. *De fato, temos que* $2^{(S\ k)} = 2.2^k \overset{(h.i)}{<} 2.k! \overset{(*)}{<} (S\ k).k! = (S\ k)!$, *onde a desigualdade (*) se justifica pelo fato de* k *ser maior ou igual a 4.*

Uma variação do PIM bastante útil é conhecida como *Princípio da Indução Forte (PIF)*:

$$\frac{\forall k, (\forall m, m < k \Longrightarrow P\ m) \Longrightarrow P\ k}{\forall n, P\ n}\ (PIF)$$

Exercício 57. *Prove que qualquer inteiro* $n \geq 2$ *é um número primo ou pode ser escrito como um produto de primos (não necessariamente distintos), i.e. na forma* $n = p_1.p_2.\cdots.p_r$, *onde os fatores* p_i $(1 \leq i \leq r)$ *são primos.*

Exercício 58. *Mostre que PIM e PIF são princípios equivalentes.*

Exercício 59. *Prove que a soma dos* n *primeiros números naturais é igual a* $\frac{n(n+1)}{2}$. *Ou seja, mostre que* $\displaystyle\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Exercício 60. *Prove que a soma dos* n *primeiros quadrados é igual a* $\frac{n(n+1)(2n+1)}{6}$. *Ou seja, mostre que* $\displaystyle\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

Exercício 61. *Prove que* $\displaystyle\sum_{i=0}^{n} i(i+1) = \frac{n.(n+1).(n+2)}{3}$.

Exercício 62. *Prove que* $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

Exercício 63. *Prove que a soma dos n primeiros cubos é igual ao quadrado da soma de 1 até n, ou seja, que* $1^3 + 2^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2$.

Exercício 64. *Prove que* $2^n - 1$ *é múltiplo de 3, para todo número natural* n *par.*

Exercício 65. *Prove que* $3 \mid (2^{2n} - 1)$ *para todo* $n \geq 0$.

Exercício 66. *Prove que* $3^n \geq n^2 + 3$ *para todo* $n \geq 2$.

Exercício 67. *Prove que* $n^2 < 4^{n-1}$ *para todo* $n \geq 3$.

Exercício 68. *Prove que* $n! > 3^n$ *para todo* $n \geq 7$.

Exercício 69. *Prove que* $n! \leq n^n$ *para todo* $n \geq 1$.

Indução no assistente de provas Coq

Nesta seção, veremos como construir provas utilizando a indução matemática no assistente de provas Coq (`https://coq.inria.fr`). Um conjunto indutivamente definido pode ser construído com a palavra reservada `Inductive`. Por exemplo, o conjunto finito *Sem* dos dias da semana apresentado na seção anterior pode ser definido como a seguir:

```
Inductive Sem :=
| domingo: Sem
| segunda_feira: Sem
| terca_feira: Sem
| quarta_feira: Sem
| quinta_feira: Sem
| sexta_feira: Sem
| sabado: Sem.
```

O conjunto dos números naturais é uma construção nativa do Coq, *i.e.* ela está disponível uma vez que o sistema é iniciado, e pode ser vista a partir do comando `Print nat.`

```
Inductive nat : Set :=  O : nat | S : nat -> nat.
```

Ou seja, o conjunto `nat` dos números naturais é definido indutivamente e possui 2 construtores: o `O` (zero), e `S` (sucessor). Claramente, esta definição corresponde à gramática (2.6). Toda definição indutiva possui um princípio indutivo associado, e que é automaticamente gerado pelo Coq. Por padrão, o nome do princípio indutivo associado a uma definição indutiva, digamos `mdef`, é `mdef_ind`. No caso de `nat` podemos acessar este princípio pelo comando `Print nat_ind`:

```
nat_ind =
fun (P : nat -> Prop) (f : P 0) (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
                         | 0 => f
                         | S n0 => f0 n0 (F n0)
                         end
:forall P: nat -> Prop, P 0 ->
                        (forall n: nat, P n -> P (S n)) ->
                        forall n: nat, P n
```

A parte que nos interessa desta saída está em azul. Como em *(PIM)*, a base de indução diz que `P 0`, e o passo indutivo corresponde ao trecho `(forall n : nat, P n -> P (S n))`. A conclusão como esperado, diz que `forall n : nat, P n`.

A seguir faremos a prova de que a soma dos n primeiros números ímpares é igual a n^2. Inicialmente precisamos expressar a "soma dos n primeiros números ímpares"em Coq. Para isto, definiremos um somatório. Antes disto carregamos a biblioteca `Lia`, que vai nos ajudar com a simplificação de expressões aritméticas nos inteiros.

```
Require Import Lia.

Fixpoint msum (n:nat) :=
 match n with
 | 0 => 0
 |S k => (msum k) + (2*k+1)
 end.
```

A palavra reservada `Fixpoint` é utilizada para definir funções recursivas. Note que $\sum_{i=1}^{n}(2.i - 1)$ corresponde a `msum n`. Podemos fazer alguns testes com esta definição:

```
Eval compute in (msum 1).
```

```
     = 1
   : nat
```

O valor retornado é 1 porque que é igual ao primeiro número ímpar.

```
Eval compute in (msum 2).
```

```
     = 4
   : nat
```

Aqui a resposta é igual a 4 porque corresponde à soma dos dois primeiros números ímpares, ou seja, 1+3.

```
Eval compute in (msum 3).
```

```
     = 9
   : nat
```

O valor retornado corresponde à soma dos 3 primeiros números ímpares: 1+3+5=9.

```
Eval compute in (msum 4).
```

```
     = 16
   : nat
```

Por fim, temos que a soma dos 4 primeiros números ímpares é 1+3+5+7=16.

De acordo com estes testes, nossa definição de somatório está funcionando corretamente, e portanto podemos escrever o lema que queremos provar, a saber, que a soma dos `n` primeiros números naturais é igual a `n*n`:

```
Lemma msum_square: forall n, msum n = n*n.
```

Faremos a mesma prova apresentada na árvore construída na seção anterior. Iniciamos a prova por indução em `n` com a tática (ou comando) `induction n`.

```
Lemma msum_square: forall n, msum n = n*n.
Proof.
 induction n.
```

```
2 goals (ID 11)

 ============================
 msum 0 = 0 * 0

goal 2 (ID 14) is:
msum (S n) = S n * S n
```

Temos 2 casos para analisar (`2 goals`): o primeiro corresponde a base da indução, e o segundo é o passo indutivo.

O primeiro caso é trivial, e a tática `reflexivity` é capaz de concluir que os lados esquerdo e direito da igualdade são iguais a 0, uma vez que `msum 0` é igual a 0.

No segundo caso, temos como hipótese de indução que a soma dos `k` primeiros números ímpares é igual a `k*k`, e precisamos provar que a soma dos `(S k)` primeiros números ímpares é igual a `(S k)*(S K)`:

```
1 goal (ID 14)

k : nat
IHk : msum k = k * k
============================
msum (S k) = S k * S k
```

Podemos, por exemplo, aplicar a tática `simpl` para simplificar a expressão `msum (S k)`, ou seja, para aplicarmos a definição de `msum`. Agora podemos substituir o lado esquerdo da hipótese de indução pelo lado direito via o comando `rewrite IHn`. A expressão resultante é uma igualdade envolvendo somas e multiplicações de números naturais:

```
1 goal (ID 24)

k : nat
IHk : msum k = k * k
============================
k * k + (k + (k + 0) + 1) = S (k + k * S k)
```

As simplificações algébricas necessárias para que possamos concluir que os lados esquerdo e direito da igualdade coincidem são feitas pela tática `lia`, e a prova completa, disponível no arquivo `pim.v`[10] , tem a seguinte forma:

---

[10] `http://flaviomoura.info/files/lca/pim.v`. Os exercícios propostos na seção anterior também estão disponíveis neste arquivo.

```
Lemma msum_square: forall n, msum n = n*n.
Proof.
 induction n.
 - reflexivity.
 - simpl. rewrite IHn. lia.
Qed.
```

Agora vamos estabelecer a equivalência entre PIM e o PIG:

Exercício 70. *Complete a prova a seguir:*

```
Lemma PIG: forall (P : nat -> Prop) (k : nat), P k ->
              (forall n, n >= k -> P n -> P (S n)) ->
                       forall n : nat, n >= k -> P n.
Proof.
 intros P k H1 IH n H2.
 assert (H := nat_ind (fun n => n >= k -> P n)).
 Admitted.
```

Observe que a prova do exercício anterior utiliza o PIM via o comando `nat_ind`, e portanto temos uma prova de PIG via PIM. No outro sentido, vamos enunciar PIM como um lema:

Exercício 71. *Complete a prova a seguir:*

```
Lemma PIM : forall P: nat -> Prop,
   (P 0) ->
   (forall n, P n -> P (S n)) ->
   forall n, P n.
Proof.
 intros P H IH n.
 apply PIG with 0.
 Admitted.
```

Indução Estrutural

Nesta seção veremos que o princípio de indução matemática (PIM) visto anteriormente é um caso particular de um princípio geral que está associado a qualquer conjunto definido indutivamente. Vimos dois tipos de regras utilizadas na construção de um conjunto definido indutivamente:

1. As regras não recursivas, ou seja, aquelas que definem diretamente um elemento do conjunto definido indutivamente;

2. As regras recursivas, ou seja, aquelas que constroem novos elementos a partir de elementos já construídos.

Como veremos no próximo exemplo, estas regras podem fazer uso de elementos de outros conjuntos previamente definidos. Formalmente, se $A_1, A_2, \ldots$ são conjuntos então a estrutura geral das regras de um conjunto definido indutivamente B é como a seguir:

1. Inicialmente temos as regras não recursivas que definem diretamente os elementos $b_1, \ldots, b_m$ de B:

$$\frac{a_1 \in A_1 \qquad a_2 \in A_2 \ldots\ a_{j_1} \in A_{j_1}}{b_1[a_1, \ldots, a_{j_1}] \in B} \qquad \ldots \qquad \frac{a_1 \in A_1 \qquad a_2 \in A_2 \ldots\ a_{j_m} \in A_{j_m}}{b_m[x_1, \ldots, x_{j_m}] \in B}$$

2. Em seguida, temos as regras recursivas que constroem novos elementos a partir de elementos já construídos:

$$\frac{a_1 \in A_1 \ldots\ a_{j'_1} \in A_{j'_1}\ d_1, \ldots, d_{k_1} \in B}{c_1[x_1, \ldots, x_{j'_1}, d_1, \ldots, d_{k_1}] \in B} \qquad \ldots \qquad \frac{a_1 \in A_1 \ldots\ a_{j_n} \in A_{j_n}\ d_1, \ldots, d_{k_n} \in B}{c_n[a_1, \ldots, a_{j_n}, d_1, \ldots, d_{k_n}] \in B}$$

Qualquer elemento de um conjunto definido indutivamente pode ser construído após um número finito de aplicações das regras que o definem (e somente com estas regras). Os elementos $d_1, d_2, \ldots, d_{k_i}$ são ditos *estruturalmente menores* do que o elemento $c_i[a_1, \ldots, a_{j_i}, d_1, \ldots, d_{k_i}]$. Isto significa que os elementos $d_1, d_2, \ldots, d_{k_i}$ são subtermos próprios de $c_i[a_1, \ldots, a_{j_i}, d_1, \ldots, d_{k_i}]$.

Podemos associar um princípio de indução a qualquer conjunto definido indutivamente. No contexto genérico acima, teremos um caso base (base da indução) para cada regra não recursiva, e um passo indutivo para cada regra recursiva. O esquema simplificado (omitindo os parâmetros por falta de espaço) tem a seguinte forma:

$$\frac{\overbrace{P(b_1) \ldots P(b_m)}^{\text{casos base}}\ \overbrace{(\forall d_1 \ldots d_{k_1}, P(d_1), \ldots, P(d_{k_1}) \Rightarrow P(c_1)) \ldots (\forall d_1 \ldots d_{k_1}, P(d_1), \ldots, P(d_{k_n}) \Rightarrow P(c_n))}^{\text{casos indutivos}}}{\forall x \in B, P\ x}$$

Retornando ao caso do conjunto dos números naturais, temos um princípio indutivo com apenas um caso base e um caso indutivo:

$$\frac{P\ 0 \qquad\qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, P\ n}$$

O conjunto dos booleanos possui um princípio indutivo com dois casos base, e nenhum caso indutivo:

$$\frac{P\ \text{true} \qquad\qquad P\ \text{false}}{\forall b, P\ b}$$

A gramática 2.2 nos diz como as fórmulas da LP podem ser construídas. Observe, em particular, seus os construtores recursivos: por exemplo, a negação de uma fórmula é construída a partir de outra fórmula já construída; a conjunção, a disjunção e a implicação são construídas a partir de duas fórmulas previamente construídas. Podemos derivar o princípio de indução para o conjunto das fórmulas da LP de maneira análoga aos casos anteriores, ou seja, como um caso particular da generalização apresentada acima. Como temos dois construtores não recursivos (variáveis proposicionais e a constante $\bot$) e quatro construtores recursivos (negação, conjunção, disjunção e implicação), o princípio indutivo correspondente terá a seguinte forma, considerando uma propriedade Q qualquer das fórmulas da LP:

$$\frac{(Q\ p) \qquad (Q\ \bot) \qquad (\forall \varphi, Q\ \varphi \Longrightarrow Q\ (\neg\varphi)) \qquad (\forall \varphi_1, Q\ \varphi_1 \wedge \forall \varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))}{\forall \varphi, Q\ \varphi}$$

onde $\star \in \{\wedge, \vee, \rightarrow\}$. Chamamos o princípio de indução construído a partir de uma gramática recursiva de *indução estrutural*. Note que, para cada um dos conectivos binários (conjunção, disjunção e implicação) temos duas hipóteses de indução.

No exemplo a seguir, vamos mostrar que a gramática acima possui redundâncias, isto é, que existem conectivos que podem ser escritos a partir de outros:

Exemplo 23. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\vee$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Dizemos que duas fórmulas φ e ψ da LP são equivalentes se $\varphi \leftrightarrow \psi$ é uma tautologia. Provaremos este exercício por indução estrutural, isto é, indução na estrutura de φ:

- *Se φ é uma variável proposicional ou a constante $\bot$ então tome $\varphi' = \varphi$.*

- *Se $\varphi = \neg\psi$ então, por hipótese de indução, existe uma fórmula ψ' equivalente a ψ construída apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em ψ. Neste caso, basta tomar $\varphi' = \neg\psi'$, e estamos prontos.*

- *Se $\varphi = \psi_1 \vee \psi_2$ então, por hipótese de indução, existem fórmulas $\psi'_i (i = 1, 2)$, equivalentes respectivamente a $\psi_i (i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i (i = 1, 2)$. Neste caso, basta tomar $\varphi' = \psi'_1 \vee \psi'_2$ e estamos prontos.*

- *Se $\varphi = \psi_1 \wedge \psi_2$ então, por hipótese de indução, existem fórmulas $\psi'_i (i = 1, 2)$, equivalentes respectivamente a $\psi_i (i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i (i = 1, 2)$. Pelo exercício 2.1.6 sabemos que $\psi_1 \wedge \psi_2 \dashv\vdash \neg(\neg\psi_1 \vee \neg\psi_2)$. Então basta tomar $\varphi' = \neg(\neg\psi'_1 \vee \neg\psi'_2)$, e estamos prontos.*

- *Por fim, se $\varphi = \psi_1 \rightarrow \psi_2$ então, por hipótese de indução, existem fórmulas $\psi'_i (i = 1, 2)$, equivalentes respectivamente a $\psi_i (i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i (i = 1, 2)$. Pelo exercício 2.1.6 da lista sabemos que $\psi_1 \rightarrow \psi_2 \dashv\vdash (\neg\psi_1) \vee \psi_2$. Então basta tomar $\varphi' = (\neg\psi'_1) \vee \psi'_2$ e estamos prontos.*

Agora é a sua vez! Resolva o exercícios a seguir:

Exercício 72. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\rightarrow$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Exercício 73. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\land$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Considere a gramática da LPO:

$$\varphi ::= p(t,\ldots,t) \mid \bot \mid (\neg\varphi) \mid (\varphi \land \varphi) \mid (\varphi \lor \varphi) \mid (\varphi \rightarrow \varphi) \mid \exists_x\varphi \mid \forall_x\varphi$$

o princípio de indução correspondente é dado como a seguir:

$$\frac{(\forall t_1,\ldots,t_n, Q\ p(t_1,\ldots,t_n)) \qquad (Q\ \bot) \qquad (\forall\varphi, Q\ \varphi \Longrightarrow Q\ (\neg\varphi)) \qquad (*) \qquad (**)}{\forall\varphi, Q\ \varphi}$$

onde

$(*)$ é igual a $(\forall\varphi_1, Q\ \varphi_1 \land \forall\varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))$, $\star \in \{\land, \lor, \rightarrow\}$;

$(**)$ é igual a $(\forall x, \varphi, Q\ \varphi(x) \Longrightarrow Q\ (R_x\varphi(x)))$, $R \in \{\exists, \forall\}$.

Exercício 74. *Seja φ uma fórmula da lógica de predicados. Definimos a tradução negativa de Gödel-Gentzen de φ, denotada por φ^N, indutivamente por:*

$$\varphi^N = \begin{cases} \neg\neg\varphi & \textit{se } \varphi \textit{ é uma fórmula atômica, ou a constante } \bot \\ \neg(\psi^N) & \textit{se } \varphi = \neg\psi \\ \varphi_1^N \land \varphi_2^N & \textit{se } \varphi = \varphi_1 \land \varphi_2 \\ \neg(\neg(\varphi_1^N) \land \neg(\varphi_2^N)) & \textit{se } \varphi = \varphi_1 \lor \varphi_2 \\ \varphi_1^N \rightarrow \varphi_2^N & \textit{se } \varphi = \varphi_1 \rightarrow \varphi_2 \\ \forall_x(\psi^N) & \textit{se } \varphi = \forall_x\psi \\ \neg(\forall_x\neg(\psi^N)) & \textit{se } \varphi = \exists_x\psi \end{cases}$$

Construa uma prova intuicionista para o sequente a seguir: $\neg\neg(\varphi^N) \vdash_i \varphi^N$

Exercício 75. *Uma fórmula da lógica de predicados* ϕ *pertence ao* fragmento negativo *se* ϕ *pode ser construída a partir da seguinte gramática, onde* $t_1, t_2, \ldots, t_n$ $(n > 0)$ *são termos:*

$$\phi ::= \neg p(t_1, t_2, \ldots, t_n) \mid \bot \mid (\neg \phi) \mid (\phi \wedge \phi) \mid (\phi \to \phi) \mid (\forall_x \phi)$$

Prove na lógica minimal que $\neg\neg\theta \vdash_m \theta$ *para qualquer fórmula* θ *pertencente ao fragmento negativo. Use indução na estrutura de* θ*.*

Nos próximos capítulos estudaremos diversos algoritmos que utilizam a estrutura de lista encadeada, definida pela seguinte gramática $l ::= nil \mid a :: l$, onde nil representa a lista vazia, e $a :: l$ representa a lista com primeiro elemento a e cauda l. Como esta gramática possui um construtor não recursivo, e um construtor recursivo, teremos um princípio de indução com um caso base, e um passo indutivo:

$$\frac{P\ nil \qquad \forall l\ h, P\ l \Longrightarrow P\ (h :: l)}{\forall l, P\ l}$$

O comprimento de uma lista, isto é, o número de elementos que a lista possui, é definido recursivamente por:

$$|l| = \begin{cases} 0, & \text{se } l = nil \\ 1 + |l'|, & \text{se } l = a :: l' \end{cases}$$

Uma operação importante que nos permite construir uma nova lista a partir de duas listas já construídas é a concatenação. Podemos definir a concatenação de duas listas por meio da seguinte função recursiva:

$$l_1 \circ l_2 = \begin{cases} l_2, & \text{se } l_1 = nil \\ a :: (l' \circ l_2), & \text{se } l_1 = a :: l' \end{cases}$$

Por fim, o reverso de uma lista é definido recursivamente por:

$$rev(l) = \begin{cases} l, & \text{se } l = nil \\ (rev(l')) \circ (a :: nil), & \text{se } l = a :: l' \end{cases}$$

Os exercícios a seguir expressam diversas propriedades envolvendo estas operações. Resolva cada um deles utilizando indução.

Exercício 76. *Prove que* $|l_1 \circ l_2| = |l_1| + |l_2|$*, quaisquer que sejam as listas* l_1, l_2*.*

Exercício 77. *Prove que* $l \circ nil = l$*, qualquer que seja a lista* l*.*

Exercício 78. *Prove que a concatenação de listas é associativa, isto é,* $(l_1 \circ l_2) \circ l_3) = l_1 \circ (l_2 \circ l_3)$ *quaisquer que sejam as listas* l_1, l_2 *e* l_3*.*

Exercício 79. *Prove que* $|rev(l)| = |l|$*, qualquer que seja a lista* l*.*

Exercício 80. *Prove que* $rev(l_1 \circ l_2) = (rev(l_2)) \circ (rev(l_1))$*, quaisquer que sejam as listas* l_1, l_2*.*

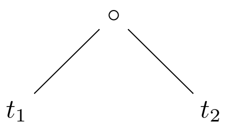
Exercício 81. *Prove que* $rev(rev(l)) = l$*, qualquer que seja a lista* l*.*

Outra estrutura de dados importante em Computação é a estrutura de árvores. O caso particular do conjunto *btree* das árvores binárias, isto é, as árvores cujos nós têm dois filhos, ou são folhas (não têm filhos) pode ser definido indutivamente pelas seguintes regras:

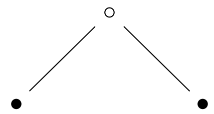
$$\frac{}{\bullet \in btree} \qquad\qquad \frac{t_1 \in btree \qquad t_2 \in btree}{\circ(t_1, t_2)}$$

A gramática correspondente é dada por $t ::= \bullet \mid \circ(t, t)$.

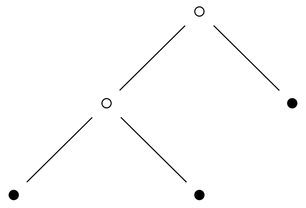
Assim, um nó sem filhos representa uma árvore (caso não recursivo), e t_1 e t_2 são duas árvores binárias então podemos construir uma nova árvore como na figura abaixo:



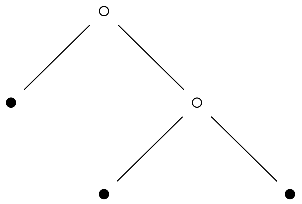
Observe que a árvore



é escrita como $\circ(\bullet, \bullet)$ na sintaxe da regra recursiva acima. Enquanto que



corresponde a $\circ(\circ(\bullet, \bullet), \bullet)$. Ou ainda,



corresponde a $\circ(\bullet, \circ(\bullet, \bullet))$.

O princípio de indução sobre *btree* terá um caso base, e um caso indutivo com duas hipóteses de indução. Nesta representação a raiz da árvore está no topo, e as folhas ficam para baixo (como ser a árvore estivesse de cabeça para baixo), mas veremos outras situações em que a raiz fica na parte inferior, e as folhas ficam no topo da árvore (como ocorre na natureza). As duas representações são utililizadas em Computação, como veremos.

Podemos definir a altura de uma árvore binária da seguinte forma:

$$h(t) = \begin{cases} 0, & \text{se } t = \bullet \\ 1 + \max(h(t_1), h(t_2)), & \text{se } t = \circ(t_1, t_2) \end{cases}$$

O número de nós de uma árvore binária é dado por:

$$n(t) = \begin{cases} 1, & \text{se } t = \bullet \\ 1 + n(t_1) + n(t_2)), & \text{se } t = \circ(t_1, t_2) \end{cases}$$

Exercício 82. *Mostre que* $n(t) \leq 2^{h(t)+1} - 1$*, para qualquer árvore binária* t*.*

Podemos flexibilizar um pouco a definição de árvore binária e permitir que um nó tenha, no máximo, dois filhos. Neste caso, acrescentaremos mais uma regra à nossa definição:

$$\frac{}{\bullet \in btree} \qquad\qquad \frac{t \in btree}{\circ(t)} \qquad\qquad \frac{t_1 \in btree \qquad t_2 \in btree}{\circ(t_1, t_2)}$$

A gramática correspondente é dada por $t ::= \bullet \mid \circ(t) \mid \circ(t, t)$.

Podemos estender esta definição para árvores cujos nóes possuem até $k \geq 0$ filhos, mas na prática ficaremos restritos a $k = 3$ por conta da estrutura das regras do sistema de dedução natural tanto na lógica proposicional quanto na lógica de predicados. Como exemplo, provaremos o Teorema de Glivenko:

Exemplo 24. *O teorema de Glivenko diz que se* $\Gamma \vdash_c \varphi$ *então* $\Gamma \vdash_i \neg\neg\varphi$ *na lógica proposicional, ou seja, se* φ *tem uma prova clássica a partir de* Γ*, então* $\neg\neg\varphi$ *tem uma prova intuicionista a partir de* Γ *na lógica proposicional. Faremos a prova deste teorema por indução na derivação* $\Gamma \vdash_c \varphi$*, isto é, indução na estrutura da árvore correspondente à derivação clássica de* φ *a partir de* Γ*. Queremos provar* $\Gamma \vdash_i \neg\neg\varphi$*, quais quer que sejam* Γ *e* φ*.*

Capítulo 3

Algoritmos

3.1 Introdução

Nesta seção vamos utilizar a LPO no estudo de algoritmos. Mais especificamente, utilizaremos a LPO para expressarmos propriedades dos algoritmos. Os computadores e seus algoritmos estão presentes na maioria das atividades quotidianas, utilizamos computadores para fazer compras, transações bancárias, etc. Os carros, aviões e equipamentos hospitalares modernos também possuem diversos sistemas embarcados. À medida que estes equipamentos se tornam mais comuns em nosso dia a dia, aumenta também o seu poder de processamento e de armazenamento. Diante desta realidade é natural perguntar: como a lógica (em geral), e a LPO em particular, pode ser útil no estudo dos algoritmos? Porque precisamos garantir que os algoritmos são corretos. Mas o que significa dizer que um algoritmo é correto? Intuitivamente, significa que o algoritmo sempre fornece respostas corretas para qualquer entrada possível. Por exemplo, se AlgOrd é um algoritmo de ordenação de inteiros, então espera-se que para qualquer vetor de inteiros A dado, AlgOrd(A) retorne uma permutação de A que esteja ordenada. Isto significa que as respostas dadas pelo algoritmo são corretas para todas as entradas possíveis, e que estas respostas são geradas em tempo finito. Como veremos, em alguns casos a prova de correção é simples, mas em geral, tanto definir a noção de correção quanto provar a propriedade são tarefas complexas. Uma ferramenta que será utilizada frequentemente nas provas de correção de algoritmos é a indução matemática estudada no capítulo anterior.

3.1.1 Busca sequencial

Como primeiro exemplo, considere o problema de buscar um elemento em um vetor de números naturais. O pseudocódigo a seguir recebe como argumentos o vetor $A[0..n-1]$ contendo n números naturais e o natural x procurado, e faz a busca sequencial de x em A: se A possui uma ocorrência de x então o algoritmo retorna a posição $0 \leq i \leq n-1$ da primeira ocorrência encontrada. Caso contrário, o algoritmo retorna o valor -1.

```
$i \leftarrow 0$;
while $i < n$ and $A[i] \neq x$ do
    $i \leftarrow i + 1$;
end
if $i < n$ then
    return $i$;
else
    return -1
end
```

Algoritmo 1: SequentialSearch($A[0..n-1], x$)

Parece bastante intuitivo dizer que este algoritmo é correto, mas como **provar** isto? Inicialmente temos que expressar a noção de correção como uma propriedade, e em seguida, precisamos provar esta propriedade. Em programas contendo laços utilizamos as *invariantes de laço*, que são propriedades satisfeitas durante toda a execução de um laço. A prova de uma invariante de laço consiste em três etapas, e é análoga a uma prova indutiva:

1. **Inicialização**: Nesta etapa mostramos que a propriedade é satisfeita antes da primeira execução do laço. Esta etapa é equivalente à base da indução em uma prova indutiva;

2. **Manutenção**: Esta é a etapa mais delicada da prova porque corresponde ao passo indutivo. Aqui assumimos por hipótese (de indução) que a invariante vale antes de uma iteração arbitrária do laço (depois da primeira) e mostramos que a invariante continua válida antes da próxima iteração. Ou seja, assumimos que a invariante vale antes da k-ésima iteração ($k > 0$) e mostramos que ao final desta iteração, isto é, antes da $(k+1)$-ésima iteração, a invariante continua satisfeita;

3. **Finalização**: Nesta etapa concluímos que a invariante é satisfeita durante toda a execução do laço, e esta informação é utilizada para estabelecer a correção do algoritmo.

Com estas informações, construiremos uma invariante para o laço **while** (linhas 2-4) que nos permita concluir a correção ao final da execução do algoritmo. A construção da invariante costuma ser a etapa mais difícil porque precisamos pensar em uma propriedade que seja verdadeira antes da execução do laço (inicialização), permaneça verdadeira durante toda a execução do laço (manutenção) e tal que sua validade ao final da execução do laço nos permita concluir sua correção (finalização).

No caso da busca sequencial queremos mostrar que ao final da execução, o algoritmo SequentialSearch retorna -1 se x não ocorre em A, e retorna $0 \leq i \leq n-1$ se a primeira ocorrência de x é na posição i do vetor A. Precisamos adaptar esta propriedade de forma que possa ser utilizada ao longo da execução do algoritmo, e não apenas ao final de sua execução. Sabemos que uma nova iteração do laço **while** só ocorrerá se o elemento x não foi ainda encontrado, pois caso contrário o algoritmo para. A partir destas observações, considere a seguinte invariante para o laço **while** do algoritmo SequentialSearch:

Antes da i-ésima iteração do laço **while**, o subvetor $A[0..i-1]$ não possui ocorrências de x.

A prova de uma invariante é construída pelos 3 passos citados acima:

Demonstração. A prova é dividida em 3 passos:

- **Inicialização**: Precisamos mostrar que antes da primeira iteração do laço **while**, o subvetor $A[0..i-1]$ não possui ocorrências de x. Este passo é trivial porque antes da primeira iteração, temos que $i = 0$, e portanto o subvetor $A[0..i-1]$ é vazio.

- **Manutenção**: Este é o passo que exige mais cuidado na prova. Observe que antes da primeira iteração o valor de i é 0. Considerando que as condições do laço sejam satisfeitas, i será incrementado, e portanto antes da segunda iteração o valor de i é 1, e assim sucessivamente. Logo, antes da k-ésima iteração $k > 1$, o valor de i é $k-1$ e podemos assumir por hipótese que o subvetor $A[0..k-2]$ não possui ocorrências de x. Para que a próxima iteração ocorra, precisamos que $k < n$ e $A[k-1] \neq x$. Nestas condições, temos que o subvetor $A[0..k-1]$ não possui ocorrências de x preservando assim, a invariante.

- **Finalização**: Ao final da execução do laço, a condição "$i < n$ and $A[i] = x$ $(0 \leq i < n)$"não é mais satisfeita, e portanto temos que $i \geq n$ ou $A[i] = x$ $(0 \leq i < n)$. Se $i \geq n$ então o vetor A não possui ocorrências de x e o algoritmo retorna -1 de acordo com a linha 5. Se $A[i] = x$ $(0 \leq i < n)$ então a posição i é retornada, uma vez que o elemento procurado está na posição i do vetor A. Isto finaliza a prova de correção do algoritmo SequentialSearch.

□

Agora vamos analisar a complexidade em tempo e espaço deste algoritmo. Observe que a execução do algoritmo não demanda espaço adicional, ou seja, o espaço utilizado para a sua execução é o espaço alocado para armazenar o vetor A e nada mais. Neste caso, dizemos que a complexidade em espaço do algoritmo SequentialSearch é constante. Uma complexidade, seja em tempo ou espaço, constante é a melhor situação que podemos ter, ou seja, a mais eficiente possível. As classes básicas de eficiência que utilizaremos para analisar algoritmos são listadas a seguir em ordem crescente de complexidade em função do tamanho n da entrada:

Classe	**Nome**
1	constante
$\log n$	logarítmica
n	linear
$n.\log n$	linearítmica
n^2	quadrática
n^3	cúbica
n^k	polinomial ($k \geq 1$ e finito)
a^n	exponencial ($a \geq 2$)
$n!$	fatorial

A análise da complexidade de tempo do algoritmo SequentialSearch não é tão imediata quanto a análise feita para a complexidade de espaço, ainda que seja simples. Podemos começar com a seguinte pergunta: qual o custo de execução de cada linha do algoritmo SequentialSort? A linha 1 faz uma atribuição, cujo custo não depende do tamanho n do vetor A, e portanto é razoável dizer que este custo é constante, digamos c_1, uma constante positiva. Observe que esta constante não depende do parâmetro n, mas do computador e da linguagem de programação. As linhas 2-4 constituem um laço cujo corpo contém apenas uma atribuição. Ainda que o custo da linha 3 possa ser o mesmo da linha 1, vamos denotá-lo pela constante positiva c_3. Quantas vezes a linha 3 é executada? Isto depende tanto do vetor A quanto da chave x. De fato, se x ocorre na primeira posição de A, isto é, se $A[0]$ é igual a x então a condição do laço é executada uma única vez, mas a linha 3 não é executada nenhuma vez independente de existirem outras ocorrências de x em A. Esta é a situação constitui o melhor caso possível, e por isso é chamada de *análise do melhor caso*. Se $A[0] \neq x$ e x ocorre na segunda posição de A então a linha 3 é executada uma única vez, enquanto que a linha 2 é executada duas vezes. Em geral, observe que a linha que define um laço é sempre executada uma vez a mais do que as linhas que compõem o seu corpo. Por fim, se x não ocorre no vetor A então a linha 2 será executada $n + 1$ vezes enquanto que a linha 3 será executada n vezes. Esta situação vai configurar a *análise do pior caso*. Por fim, o condicional da linha 5 será executado uma única vez a um custo constante, digamos c_5, e apenas uma das linhas 6 ou 8 será executada uma única vez. Juntando todas estas informações podemos então dividir a análise em 2 casos:

Análise do melhor caso na busca sequencial

Como vimos anteriormente, o melhor caso ocorre quando o elemento procurado ocorre na primeira posição do vetor A. Os custos associados por linha nesta situação são apresentados na seguinte tabela:

Linha	Custo	Observação
1	c_1	
2	c_2	
3	0	não é executada
5	c_5	
6	c_6	
8	0	não é executada
Total	$c_1 + c_2 + c_5 + c_6$	

Denotando por $T_b(n)$ o custo no melhor caso (*best case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_b(n) = c_1 + c_2 + c_5 + c_6$. Neste caso dizemos que o custo da busca sequencial é constante em função do tamanho n da entrada.

Análise do pior caso na busca sequencial

Agora vamos compilar as informações discutidas anteriormente considerando que o laço da linha 2 é executado o maior número de vezes possível, o que acontece quando o elemento procurado não ocorre no vetor:

Linha	Custo
1	c_1
2	$c_2.(n+1)$
3	$c_3.n$
5	c_5
6	0
8	c_8
Total	$c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$

Denotando por $T_w(n)$ o custo no pior caso (*worst case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_w(n) = c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$. Neste caso dizemos que o custo da busca sequencial é linear em função do tamanho n da entrada. Antes de refinarmos a análise e apresentarmos as definições precisas das análises de melhor e pior caso, vejamos um outro exemplo considerando agora o problema da ordenação de um vetor.

3.1.2 O algoritmo de ordenação por inserção (*insertion sort*)

Nesta seção estamos interessados em ordenar $n > 0$ números naturais em ordem crescente. Suponha que estes números estão armazenados no vetor $A[0..n-1]$. Então, ao final do processo queremos obter uma permutação de $A[0..n-1]$, digamos $A'[0..n-1]$ tal que $A'[i-1] \leq A'[i]$, para todo $1 \leq i < n$. Estudaremos diversas formas distintas de abordar este problema nas próximas seções, mas aqui vamos iniciar com o algoritmo de ordenação por inserção (*insertion sort*), cujo pseudocódigo é dado a seguir:

```
for $j = 1$ to $n - 1$ do
    $key \leftarrow A[j]$;
    $i \leftarrow j - 1$;
    while $i \geq 0$ and $A[i] > key$ do
        $A[i+1] \leftarrow A[i]$;
        $i \leftarrow i - 1$;
    end
    $A[i+1] \leftarrow key$;
end
```

Algoritmo 2: InsertionSort($A[0..n-1]$)

A primeira pergunta que precisamos responder é: este algoritmo é correto? Isto é, ele satisfaz as especificações do problema que propõe resolver?

A correção do algoritmo de ordenação por inserção

Queremos ordenar os elementos de um vetor, assim esperamos que os elementos do vetor gerado após a execução do algoritmo coincidam com os elementos do vetor original, e que o vetor resultante esteja ordenado. Como vimos no exemplo anterior, em algoritmos iterativos utilizamos as invariantes de laço para estabelecer a correção do algoritmo. Dada a dinâmica do algoritmo InsertionSort, considere a seguinte invariante de laço:

> Antes da j-ésima iteração do laço **for** (linhas 1-9), o subvetor $A[0..j-1]$ está ordenado e contém os mesmos elementos do vetor original $A[0..j-1]$.

Assim, se esta propriedade for válida ao final da execução do laço **for**, *i.e.* antes da $n+1$-ésima iteração, teremos que o vetor gerado consiste dos elementos do vetor original $A[0..n-1]$ ordenado. Isto corresponde a dizer que InsertionSort é correto.

Como então provar esta invariante para InsertionSort? A prova é por indução no número de iterações do laço **for**:

- **Inicialização** (Base da indução):
  Antes da primeira iteração do laço **for**, temos que $j = 1$ (condição necessária para iniciar o laço), e portanto a invariante é trivial porque o subvetor unitário $A[0]$ está ordenado por definição.

- **Manutenção** (Passo indutivo):
  Considere a k-ésima iteração, isto é, $j = k$ $(1 < k < n)$. Temos como hipótese que "Antes da k-ésima iteração do laço **for** o subvetor $A[0..k-1]$ é uma permutação que está ordenada do subvetor original $A[0..k-1]$."Assim, durante a k-ésima iteração, o laço **while** vai deslocar cada elemento maior do que $A[k]$, i.e. key, uma posição para a direita até encontrar a posição correta onde o elemento $A[k]$ deve ser inserido, de forma que neste momento o subvetor $A[0..k]$ está ordenado e possui os mesmos elementos do subvetor $A[0..k]$ original. A incrementarmos o valor de k para a próxima iteração, a invariante é reestabelecida. Informalmente estamos dizendo que o laço **while** encontra a posição correta para inserir $A[j]$ (que está armazenado na variável key). Provaremos este fato com a ajuda de uma invariante para o laço **while**:

  > Antes de cada iteração do laço **while**, o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a key.

  A prova é também por indução no número de iterações do laço **while**:

  1. **Inicialização**: Antes da primeira iteração do **while** temos que $i + 1 = j = k$, e como $key = A[j]$ a invariante está satisfeita.

  2. **Manutenção**: Por hipótese de indução temos que o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a *key*. Durante uma iteração do laço, o elemento $A[i]$ é copiado na posição $i+1$ do vetor A, e portanto a invariante continua valendo.

  3. **Finalização**: Ao final da execução do laço, temos que i é, de fato, a posição correta para inserir o elemento $A[k]$ já que todos os elementos do subvetor $A[i+1..j]$ são maiores ou iguais a key. É importante observar que a inserção do elemento $A[k]$ na posição i não elimina nenhum elemento do vetor original porque o elemento que está na posição i foi copiado para a posição $i+1$, se o laço **while** foi executado pelo menos uma vez, ou ele é o próprio elemento armazenado em *key*, quando o laço não é executado.

- **Finalização**: Ao final da execução do laço **for**, temos $j = n$, e portanto a invariante corresponde a dizer que o vetor $A[0..n-1]$ obtido ao final da execução do algoritmo está ordenado, e é uma permutação do vetor original $A[0..n-1]$. Assim, concluímos a prova da correção do algoritmo InsertionSort.

Exercício 83. *Prove que o algoritmo BubbleSort a seguir é correto.*

```
for $i = 0$ to $n - 2$ do
    for $j = 0$ to $n - 2 - i$ do
        if $A[j + 1] < A[j]$ then
            swap $A[j]$ and $A[j + 1]$;
        end
    end
end
```

Algoritmo 3: BubbleSort($A[0..n-1]$)

Exercício 84. *Prove que o algoritmo BubbleSort2 [8] a seguir é correto.*

```
for $i = 0$ to $n - 2$ do
    for $j = n - 1$ downto $i + 1$ do
        if $A[j] < A[j - 1]$ then
            swap $A[j]$ and $A[j - 1]$;
        end
    end
end
```

Algoritmo 4: BubbleSort2($A[0..n-1]$)

Exercício 85. *Prove que o algoritmo SelectionSort a seguir é correto.*

```
for $i = 0$ to $n - 2$ do
    $min \leftarrow i$;
    for $j = i + 1$ to $n - 1$ do
        if $A[j] < A[min]$ then
            $min \leftarrow j$;
        end
    end
    swap $A[i]$ and $A[min]$;
end
```

Algoritmo 5: SelectionSort($A[0..n-1]$)

A complexidade do algoritmo de ordenação por inserção

Agora faremos uma análise da complexidade do algoritmo de ordenação por inserção semelhante análoga à feita para a busca sequencial. Certamente, ordenar um vetor com 1000 demanda mais tempo do que ordenar apenas 3 elementos, assim é usual descrever o tempo de execução de um algoritmo em função do tamanho da entrada que neste caso é o número n de elementos a serem ordenados. Novamente assumiremos que cada linha do pseudocódigo é executada em tempo constante, mas este tempo pode diferir de uma linha para outra. Assim, denotaremos por c_i a constante que corresponde ao tempo de execução da i-ésima linha do pseudocódigo. Vejamos, então, o custo de execução do algoritmo InsertionSort. O laço **for** da linha 1 é executado n vezes, enquanto que o corpo do laço é executado $n - 1$ vezes, uma vez para cada $j = 1, \ldots, n - 1$. Denotaremos por t_j o número de vezes que o teste do laço **while** da linha 4 é executado, de forma que temos o seguinte custo por linha:

Portanto, o custo total, que denotaremos por $T(n)$ é dado por:

Linha	Custo	Número de execuções	Custo total
1	c_1	n	$c_1.n$
2	c_2	$n-1$	$c_2.(n-1)$
3	c_3	$n-1$	$c_3.(n-1)$
4	c_4	$\sum_{j=1}^{n-1} t_j$	$c_4. \sum_{j=1}^{n-1} t_j$
5	c_5	$\sum_{j=1}^{n-1}(t_j-1)$	$c_5. \sum_{j=1}^{n-1}(t_j-1)$
6	c_6	$\sum_{j=1}^{n-1}(t_j-1)$	$c_6. \sum_{j=1}^{n-1}(t_j-1)$
8	c_8	$n-1$	$c_8.(n-1)$

$$T(n) = c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.\sum_{j=2}^{n} t_j + c_5.\sum_{j=2}^{n}(t_j-1) + c_6.\sum_{j=2}^{n}(t_j-1) + c_8.(n-1)$$

Agora note que, mesmo para entradas de mesmo tamanho, o tempo de execução pode mudar. De fato, um vetor que tenha mais elementos a serem reposicionados terá um custo maior para ser ordenado. Portanto, a análise do melhor caso se dá quando o vetor já estiver ordenado pois $t_j = 1$, para todo $2 \leq j \leq n$:

$$\begin{aligned} T_b(n) &= c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.(n-1) + c_8.(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8).n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

ou seja, uma função linear de n. Por outro lado, a análise do pior caso se dá quando o vetor estiver ordenado decrescentemente pois $t_j = j$ (por que?), e portanto

$$T_w(n) = c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\tfrac{(n-1).n}{2}) + (c_5 + c_6).(\tfrac{(n-2).(n-1)}{2})$$

ou seja, uma função quadrática de n.

A forma de análise feita para InsertionSort acima, assim como para SequentialSearch na seção anterior, apresenta alguns problemas porque as constantes utilizadas podem mudar dependendo do computador, da linguagem de programação ou mesmo do estilo de programação utilizados. Uma maneira de ignorar estas especificidades, e fazer uma análise que seja independente destes aspectos, consiste na utilização de uma notação adequada, a *notação assintótica*, que considera o comportamento de funções no limite, isto é, para valores suficientemente grandes do parâmetro n. A ideia é que possamos pegar uma função como $T_w(n) = c_1 + c_2.n + c_3.(n-1) + c_6 + c_8$ que expressa o custo no pior caso do algoritmo de busca sequencial, e dizer que ela cresce como n, sem a necessidade de considerar as constantes. Faremos isto considerando o conjunto das funções que são limitadas superiormente por um múltiplo constante de n. Observe que podemos facilmente construir uma cota superior para a função $T_w(n)$ da seguinte forma $T_w(n) = c_1+c_2.n+c_3.(n-1)+c_6+c_8 \leq c_1.n+c_2.n+c_3.n-c_3+c_6.n+c_8.n \leq (c_1+c_2+c_3+c_6+c_8).n \leq c.n$ para qualquer constante $c \geq c_1 + c_2 + c_3 + c_6 + c_8$ e $n \geq 1$. Neste caso, dizemos que a função $T_w(n)$ é $O(n)$, ou seja, que $T_w(n)$ é de ordem n. Formalmente, temos a seguinte definição para o conjunto $O(g(n))$ que contém todas as funções que são da ordem de $g(n)$:

Definição 25. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $O(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $f(n) \leq c \cdot g(n), \forall n \geq n_0$. Alternativamente, $O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq c{\cdot}g(n), \forall n \geq n_0.\}$*

Observe que uma função $f(n)$ pode estar em $O(g(n))$ mesmo que $f(n) > g(n), \forall n$. O ponto importante é que $f(n)$ tem que ser limitada por um múltiplo constante de $g(n)$. A relação entre $f(n)$ e $g(n)$ para valores pequenos de n também é desconsiderada. Intuitivamente, os termos de menor ordem de uma função assintoticamente positiva podem ser ignorados na determinação da cota superior porque são insignificantes para valores grandes do parâmetro n. Assim, quando n é grande qualquer porção ou fração do termo de maior ordem é suficiente para dominar os termos de menor ordem.

Normalmente, escrevemos $T(n) \in O(n^2)$ para dizer que $T(n)$ é $O(n^2)$ já que $O(n^2)$ é um conjunto. No entanto, é comum encontrarmos o uso da igualdade $T(n) = O(n^2)$ ao invés de $T(n) \in O(n^2)$. A conveniência do uso da igualdade será vista posteriormente, mas o importante aqui é entender que esta igualdade é unidirecional, e portanto não pode ser confundida com a igualdade tradicional. Por exemplo, escrevemos $T(n) = O(n^2)$, mas $O(n^2) = T(n)$ não é correto. O número de funções anônimas em uma expressão é igual ao número de vezes que a notação assintótica aparece: por exemplo, na expressão $\sum_{i=1}^{n} O(i)$ contém apenas uma função anônima (a função que tem parâmetro i), e portanto esta expressão não é o mesmo que $O(1) + O(2) + \ldots + O(n)$ (que não possui uma interpretação clara). A notação assintótica também pode aparecer do lado esquerdo de uma equação: $2n^2 + O(n) = O(n^2)$. Neste caso, independentemente da forma como as funções anônimas são escolhidas do lado esquerdo da equação, existe uma forma de escolher funções anônimas do lado direito da equação de forma que a equação se verifique. No caso do exemplo acima, temos que para qualquer $f(n) = O(n)$, existe uma função $g(n) = O(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$.

Equações também podem ser encadeadas como em $2n^2 + 3n + 1 = 2n^2 + O(n) = O(n^2)$, e podem ser interpretadas separadamente de acordo com as regras anteriores. Assim, a primeira equação nos diz que existe alguma função $f(n) = O(n)$ para a qual a equação se verifica para todo n. A segunda equação nos diz que para toda função $g(n) = O(n)$, existe uma função $h(n) = O(n^2)$ tal que a equação se verifica para todo n. Este encadeamento é transitivo, ou seja, podemos concluir que $2n^2+3n+1 = O(n^2)$.

O lema a seguir nos permite utilizar limites para utilizar a notação assintótica:

Lema 26. *Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$*, incluindo o caso em que* $c = 0$*, então* $f(n) = O(g(n))$.

Demonstração. Exercício.

□

Observe que a outra direção do lema anterior não vale: de fato, considere $f(n) = n$ e $g(n) = 2^{\lfloor \lg n \rfloor}$, onde $\lg n$ é o logaritmo de n na base 2. Temos que $f(n) = O(g(n))$ porque $f(n) = n = 2^{\lg n} \leq 2^{\lfloor \lg n \rfloor + 1} = 2.2^{\lfloor \lg n \rfloor} = 2.g(n), \forall n$. No entanto, o limite $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ não existe, já que o quociente $\frac{f(n)}{g(n)}$ oscila.

Teorema 27.
1. *Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$*, onde* c *é uma constante real positiva, então* $f(n) = O(g(n))$ *e* $g(n) = O(f(n))$*;*
2. *Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *então* $f(n) = O(g(n))$*, mas* $g(n) \neq O(f(n))$*;*
3. *Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ *então* $f(n) \neq O(g(n))$*, mas* $g(n) = O(f(n))$.

Demonstração. Exercício.

□

No caso de InsertionSort, a análise do pior caso nos dá a função

$T_w(n) \quad = \quad c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\frac{(n-1).n}{2}) + (c_5 + c_6).(\frac{(n-2).(n-1)}{2})$ que é $O(n^2)$.

Como exercício, mostre em detalhes que a complexidade do pior caso de InsertionSort é $O(n^2)$.

Assim, considerando as expressões (ou polinômios) construídas(os) até agora, observamos que a classe de complexidade é obtida considerando-se o monômio de maior grau sem levar em conta o coeficiente. Portanto, a construção do polinômio a partir do custo de cada linha do algoritmo não é uma estratégia eficiente porque no final consideraremos apenas a parcela mais significativa, ou seja, o monômio de maior grau. Vamos então buscar diretamente a parte do algoritmo que nos dá este monômio de maior grau. Observando a Tabela 9 concluímos que o termo quadrático vem da linha 4, mais precisamente da comparação $A[i] > key$ que é executada em cada iteração do laço **for**. Então podemos fazer uma análise bem mais direta do que a feita anteriormente para chegarmos à mesma conclusão. Como durante a i-ésima iteração do laço **for**, a linha 4 é executada i vezes, temos:

$$T_w(n) = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} = O(n^2).$$

A análise do melhor caso também pode ser feita da mesma forma considerando que a cada iteração do laço **for**, a linha 4 é executada uma única vez:

$$T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n).$$

Da mesma forma, na busca sequencial o custo linear do pior caso pode ser obtido calculando diretamente o número de comparações feitas na linha 2. A notação O nos dá uma cota superior para o custo de execução de algoritmos, mas ela também pode ser utilizada para estabelecer uma cota para a complexidade de espaço utilizado durante a execução de um algoritmo. Tanto a busca sequencial quanto o algoritmo de ordenação por inserção não necessitam de espaço adicional de armazenamento, e portanto, em ambos os casos a complexidade é constante, ou seja, é igual a $O(1)$. Dizemos que algoritmos de ordenação que não demandam espaço adicional fazem a ordenação *in place*. Posteriormente estudaremos algoritmos que necessitam de espaço adicional. Na tabela abaixo, resumimos as análises feitas até agora:

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$

Uma ferramenta bastante útil na análise assintótica é conhecida como *regra do máximo*:

$$O(f(n) + g(n)) = O(max(f(n), g(n))) \tag{3.1}$$

Depois de alguns exercícios, e de apresentarmos mais alguns detalhes sobre a notação assintótica, estudaremos um pouco da chamada análise do caso médio. A análise do melhor caso nos dá uma ideia de situações específicas em que o algoritmo tem a melhor performance possível, mas a análise do melhor caso não costuma ser muito informativa e normalmente não é relevante. A análise do pior caso, por outro lado, tem bastante relevância e será explorada exaustivamente nas próximas seções. Ela é importante porque nos fornece o pior cenário possível para o algoritmo. Com isto sabemos que o algoritmo não pode ter um comportamento menos eficiente do que o apresentado pela análise do pior caso. No entanto, esta análise pode ser excessivamente pessimista considerando uma situação mais realista. Por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A análise do caso médio pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis, o que não corresponde à média entre as análises do melhor e pior casos.

Por fim, é importante ter em mente que a notação assintótica nos permite analisar a taxa de crescimento, ou ordem de crescimento do tempo de execução de um algoritmo, e portanto as simplificações feitas na obtenção da cota superior não devem ser esquecidas em situações práticas. Por exemplo, considere dois algoritmos A e B com complexidades, respectivamente, iguais a $O(n^2)$ e $O(n^3)$. Qual dos dois algoritmos é mais eficiente? Para valores grandes de n certamente o algoritmo A é mais eficiente, mas devemos levar em consideração que no cálculo destas classes de complexidade diversas constantes foram ignoradas. Se soubéssemos, por exemplo, que o algoritmo A realiza $100.n^2$ operações, enquanto que o algoritmo B realiza $5.n^3$ operações para resolver o mesmo problema, então agora sabemos que para $n < 20$ o algoritmo B é mais eficiente.

Exercício 86. *Complete a tabela abaixo considerando os pseudocódigos apresentados nos exercícios 7 e 9.*

Algoritmo	*tempo (melhor caso)*	*tempo (pior caso)*	*espaço*
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$
Bubble sort			
Selection sort			

Exercício 87. *Mostre que* $n = O(n^2)$.

Exercício 88. *Mostre que* $100n + 5 = O(n^2)$.

Exercício 89. *Mostre que* $\frac{n(n-1)}{2} = O(n^2)$.

Exercício 90. *Mostre que* $n^3 \neq O(n^2)$.

Exercício 91. *Sejam* $f(n), g(n)$ *e* $h(n)$ *funções dos inteiros não-negativos nos reais positivos. Mostre que se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$.

Assim, como $O(g(n))$ estabelece uma cota superior para funções, o conjunto $\Omega(g(n))$ estabelece uma cota inferior para as funções:

Definição 28. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Então* $\Omega(g(n))$ *é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real* $c > 0$ *e uma constante inteira* $n_0 > 0$ *satisfazendo a desigualdade* $c \cdot g(n) \leq f(n), \forall n \geq n_0$. *Alternativamente,* $\Omega(g(n)) = \{f(n) :$ *existem constantes positivas* c *e* n_0 *tais que* $0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.\}$

Quando dizemos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que independentemente da entrada de tamanho n, o tempo de execução desta entrada é pelo menos uma constante multiplicada por $g(n)$ para n suficientemente grande. Ou seja, estamos fornecendo uma cota inferior no melhor caso. Por exemplo, no melhor caso, o algoritmo InsertionSort é $\Omega(n)$, e portanto, o tempo de execução do algoritmo InsertionSort está entre $\Omega(n)$ e $O(n^2)$. A definição alternativa para o conjunto $\Omega(g(n))$ em termos de limites é dada pelo lema a seguir:

Lema 29. *Uma função* $f(n) = \Omega(g)$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, *incluindo o caso em que o limite é igual a* ∞.

Demonstração. Exercício.

□

A forma mais precisa de expressar o comportamento assintótico de um algoritmo é fornecendo cotas superiores e inferiores ao mesmo tempo. No parágrafo anterior, apresentamos uma cota superior e uma cota inferior para o algoritmo InsertionSort. No entanto, estas cotas são de classes diferentes, o conjunto $\Theta(g(n))$, definido a seguir, é utilizado quando ambas as cotas são da mesma classe.

Definição 30. *Seja g uma função dos inteiros não-negativos nos reais positivos. Então $\Theta(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem constantes reais positivas c_1 e c_2, e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$. Alternativamente, $\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.\}$*

Como qualquer constante pode ser vista como um polinômio de grau 0, podemos representar funções constantes como $\Theta(n^0)$, ou simplesmente, $\Theta(1)$. O lema a seguir apresenta uma caracterização do conjunto $\Theta(g(n))$ em termos de limite:

Lema 31. *Uma função $f(n) = \Theta(g(n))$ se $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c$, para alguma constante $0 < c < \infty$.*

Demonstração. Exercício.

□

Teorema 32.

1. *Se $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = \Theta(g(n))$;*

2. *Se $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $f(n) \neq \Theta(g(n))$;*

3. *Se $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = +\infty$ então $f(n) = \Omega(g(n))$, mas $f(n) \neq O(g(n))$.*

Demonstração. Exercício.

□

Exercício 92. *Prove que $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$ para qualquer inteiro $k \geq 0$ fixado.*

Teorema 33. *Dadas funções $f(n)$ e $g(n)$, temos que $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.*

Demonstração. Exercício.

□

Lema 34.

1. *$f(n) = O(g(n))$ se, e somente se $g(n) = \Omega(f(n))$;*

2. *Se $f(n) = \Theta(g(n))$ então $g(n) = \Theta(f(n))$;*

3. *Θ define uma relação de equivalência sobre as funções. Cada conjunto $\Theta(f(n))$ é uma classe de equivalência que chamamos de classe de complexidade;*

4. *$\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$;*

5. *$\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$;*

Definição 35. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Definimos, $o(g(n)) = \{f(n)$: para qualquer constante positiva c, existe uma constante positiva n_0 tal que $0 \leq f(n) < c \cdot g(n), \forall n \geq n_0.\}$*

Lema 36. *Uma função* $f(n) = o(g)$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Definição 37. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Definimos,* $\omega(g(n)) = \{f(n) :$ *para qualquer constante positiva* c, *existe uma constante positiva* n_0 *tal que* $0 \leq c{\cdot}g(n) < f(n), \forall n \geq n_0.\}$

Lema 38. *Uma função* $f(n) = \omega(g(n))$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *se este limite existir.*

Lema 39. *Se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$, *ou seja, a notação* O *é transitiva. Também são transitivos* Ω, Θ, o *e* ω.

Teorema 40. $\lg n = o(n^\alpha), \forall \alpha > 0$. *Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de* n *(incluindo potências fracionárias)*

Teorema 41. $n^k = o(2^n), \forall k > 0$. *Ou seja, potências de* n *crescem mais lentamente que a função exponencial* 2^n. *Mais ainda, potências de* n *crescem mais lentamente do que qualquer função exponencial* $c^n, c > 1$.

Exercício 93. *Mostre que* $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 94. *Mostre que* $6n^3 \neq \Theta(n^2)$.

Exercício 95. *Sejam* $f(n)$, $g(n)$ *e* $h(n)$ *funções não-negativas tais que* $f(n) = O(h(n))$ *e* $g(n) = O(h(n))$. *Prove que* $f(n) + g(n) = O(h(n))$.

3.1.3 Análise do caso médio

Nas sessões anteriores fizemos a análise do melhor caso e do pior caso para a busca sequencial, e para o algoritmo de ordenação por inserção. Nesta seção definiremos cada uma destas noções, assim como a análise do caso médio. A complexidade do pior caso consiste em considerar, dentre todas as entradas possíveis de um dado tamanho, aquela ou aquelas entradas para as quais o algoritmo em consideração executa o maior número de passos possível, ou seja, o maior número de operações básicas:

Definição 42. *Sejam* D_n *o conjunto das entradas de tamanho* n *para o algoritmo em questão, e* $I \in D_n$. *Seja* $t(I)$ *o número de operações básicas executadas pelo algoritmo na entrada* I. *Definimos a função* W, *que denota a complexidade do pior caso, por:*

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

Analogamente, a complexidade do melhor caso é dada por:

Definição 43. *Sejam* D_n *o conjunto das entradas de tamanho* n *para o algoritmo em questão, e* $I \in D_n$. *Seja* $t(I)$ *o número de operações básicas executadas pelo algoritmo na entrada* I. *Definimos a função* B, *que denota a complexidade do melhor caso, por:*

$$B(n) = \min\{t(I) \mid I \in D_n\}$$

Como comentamos anteriormente, a análise do melhor caso não costuma ser muito informativa porque pode ocorrer de forma esporádica, e normalmente não é relevante. Em outras palavras, a análise do melhor caso faz pouco sentido porque sempre é possível construir um algoritmo que é rápido para uma entrada particular. Por outro lado, a análise do pior caso nos fornece uma cota superior para o tempo de execução do algoritmo que, por si só, já é bastante útil. No entanto, esta análise pode ser excessivamente pessimista: por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A tentativa de fugir destes extremos nos leva à análise do caso médio, que pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis. É importante observar que a análise do caso médio não corresponde à média entre as análises do melhor e pior casos. Formalmente, temos a seguinte definição:

Definição 44. *Sejam D_n o conjunto das entradas de tamanho n para o algoritmo em questão, e $I \in D_n$. Seja $t(I)$ o número de operações básicas executadas pelo algoritmo na entrada I, e $p(I)$ a probabilidade de ocorrência da entrada I. Definimos a complexidade do caso médio por:*

$$A(n) = \sum_{I \in D_n} p(I).t(I)$$

Portanto, a complexidade do caso médio consiste em uma espécie de média ponderada onde os pesos são probabilidades. Antes de fazermos um exemplo, vamos fazer algumas considerações sobre a complexidade do caso médio. Ainda que seja importante pelas razões apontadas no parágrafo anterior, a complexidade do caso médio não é tão comum quanto a complexidade do pior caso. A principal razão é que ela é em geral difícil de ser determinada. Sempre que possível incluiremos a complexidade do caso médio em nossas análises, mas na maioria dos casos ficaremos restritos à análise do pior caso. Como primeiro exemplo de análise do caso médio, vamos fazer esta análise para o algoritmo de busca sequencial. Para simplificar este primeiro exemplo, suponha que a chave procurada ocorra no vetor, e também assumiremos que a chave pode ocorrer com mesma probabilidade em qualquer uma das n posições do vetor A. Antes de prosseguirmos com este exemplo, vamos revisar algumas noções importantes de probabilidade.

Definição 45. *Um experimento é* determinístico *quando repetido em condições semelhantes conduz a resultados essencialmente idênticos. Já os experimentos que quando repetidos sob as mesmas condições produzem resultados geralmente diferentes são ditos* aleatórios. *Um* espaço amostral *é um conjunto que contém os resultados possíveis de um* experimento aleatório. *Os elementos de um espaço amostral são chamados de* eventos elementares*, enquanto que subconjuntos do espaço amostral são chamados de* eventos.

Por exemplo, considere o seguinte experimento aleatório: jogar um dado e observar o número mostrado na face de cima. O espaço amostral deste experimento é o conjunto $\Omega = \{1, 2, 3, 4, 5, 6\}$. Qualquer um dos elementos deste conjunto é um evento elementar, enquanto que o subconjunto $A = \{2, 4, 6\}$ corresponde ao evento que acontece se o número mostrado na face de cima é par. Os experimentos aleatórios que consideraremos aqui possuem as seguintes características:

1. Há um número finito de eventos elementares, e a união de todos os eventos elementares é o espaço amostral;

2. Os eventos elementares são igualmente prováveis;

3. Todo evento é a união de eventos elementares.

Definição 46. *Definimos a* probabilidade *de um evento A, notação $p(A)$, como sendo o quociente*

$$p(A) = \frac{\#(A)}{\#(\Omega)}$$

Como consequência desta definição temos as seguintes propriedades:

1. Para todo evento A, $0 \leq p(A) \leq 1$;

2. $p(\Omega) = 1$;

3. $p(\emptyset) = 0$;

4. Se $A \cap B = \emptyset$ então $p(A \cup B) = p(A) + p(B)$.

Voltando para o exemplo da busca sequencial, observe que ao assumirmos que a probabilidade da chave ocorrer em qualquer das n posições do vetor A, estamos dizendo que esta probabilidade é igual a $\frac{1}{n}$. O espaço amostral que estamos considerando é o conjunto D_n das entradas de tamanho n. Denotaremos por I_j o evento correspondente ao caso em que a chave ocorre na $j+1$-ésima $(0 \leq j \leq n-1)$ posição do vetor A, e portanto a união destes eventos é igual ao espaço amostral:

$$\bigcup_{j=0}^{n-1} I_j = D_n$$

O número de comparações realizadas para I_j é $t(I_j) = j + 1$, e portanto aplicando a definição da complexidade do caso médio, onde $p(I \mid succ)$ denota a probabilidade da entrada I para o caso de sucesso, temos:

$$A_s(n) = \sum_{I \in D_n} p(I \mid succ).t(I) = \sum_{j=0}^{n-1} p(I_j \mid succ).t(I_j) = \sum_{j=0}^{n-1} \frac{1}{n}.(j+1) = \sum_{j=1}^{n} \frac{1}{n}.j = \frac{1}{n}\frac{n.(n+1)}{2} = \frac{n+1}{2}.$$

Agora vejamos como a teoria da probabilidade pode nos ajudar a completar a análise do caso médio. A definição a seguir apresenta as noções de esperança e esperança condicional a partir de variáveis aleatórias que são variáveis reais que dependem do evento elementar que ocorreu. Em outras palavras, é uma função definida para eventos elementares, isto é, $X : \Omega \to \mathbb{R}$. No caso da busca sequencial, a função t que recebe um evento elementar de $I \in D_n$ e retorna o número de comparações feitas é uma variável aleatória. Ou seja, o nosso espaço amostral D_n contém as entradas de tamanho n, e cada possível entrada é um evento elementar.

Definição 47. *Seja $f(e)$ uma variável aleatória definida sobre um conjunto de eventos elementares $e \in U$. A* esperança *de f, denotada por $E(f)$, é definida por*

$$E(f) = \sum_{e \in U} f(e).p(e)$$

Normalmente, a esperança de f é chamada de valor médio de f. A esperança condicional *de f dado um evento S, denotada por $E(f \mid S)$, é definida por*

$$E(f \mid S) = \sum_{e \in U} f(e).p(e \mid S) = \sum_{e \in S} f(e).p(e \mid S)$$

onde a última igualdade se justifica pelo fato de que a probabilidade condicional de qualquer evento que não esteja em S é igual a 0.

Portanto a esperança de t (número de comparações) é igual a $A_s(n)$ calculado acima. Quando a chave não ocorre no vetor, n comparações são feitas, e denotaremos esta informação por $A_f(n) = n$. A análise do caso médio da busca sequencial é feita juntando estas informações de acordo com a lei da esperança condicional enunciada a seguir:

Lema 48. *Sejam $f(e)$ e $g(e)$ variáveis aleatórias definidas sobre um conjunto de eventos elementares $e \in U$, e S um evento qualquer. Então:*

1. $E(f + g) = E(f) + E(g)$

2. $E(f) = p(S).E(f \mid S) + p(\neg S).E(f \mid \neg S)$

Voltando ao nosso exemplo, consideraremos dois eventos: *succ* que consiste das instâncias de D_n (eventos elementares) que possuem a chave procurada, e $\neg \succ$ que contém as instâncias de D_n que não possuem a chave procurada. Assumindo que $p(succ) = q$, temos:

$$A(n) = p(succ).A_s(n) + p(\neg succ).A_f(n) = q.(\frac{n+1}{2}) + (1-q).n = n.(1 - \frac{q}{2}) + \frac{q}{2}.$$

Assim, se a probabilidade da chave ocorrer no vetor for a mesma de não ocorrer, i.e. se $q = \frac{1}{2}$ então $A(n) = \frac{3n+1}{4} = O(n)$, ou seja, a busca sequencial tem complexidade linear no caso médio.

O exemplo acima nos mostra como o conjunto D_n, das entradas de tamanho n, deve ser interpretado. Ao invés de considerar todos os possíveis vetores de tamanho n, identificamos as propriedades das entradas que afetam o comportamento do algoritmo. No exemplo, isto corresponde a considerar quando a chave é um elemento do vetor, e em que posições ocorre. Um elemento $I \in D_n$ pode ser visto como um conjunto (ou uma classe de equivalência) de todos os vetores e chaves que ocorrem em uma posição específica do vetor, ou que não ocorre no vetor. Então $t(I)$ é o número de operações realizadas para qualquer das entradas em I.

Exercício 96. *Considere o problema de busca em um vetor ordenado. Como o algoritmo de busca sequencial poderia ser melhorado para realizar a busca neste caso? Faça a análise do pior caso e caso médio para o seu algoritmo modificado.*

3.1.4 Casamento de padrões

Dados um texto t (sequência de n caracteres) e uma palavra p (*string*) (sequência de $m \leq n$ caracteres) que chamaremos de padrão, queremos encontrar a posição i da primeira ocorrência de p em t (*string matching*), ou retornar -1 caso não existam ocorrências de p em t. Mais precisamente, i corresponde à posição do caractere mais à esquerda da primeira ocorrência de p em t tal que $t_i = p_0$, ..., $t_{i+j} = p_j$, ..., $t_{i+m-1} = p_{m-1}$.

Caso outras possíveis ocorrências de p precisem ser encontradas, basta que o algoritmo continue até que todo o texto seja percorrido. Um algoritmo força-bruta para resolver este problema funciona da seguinte forma: alinhe o primeiro caractere da palavra p com o primeiro caractere do texto t e inicie a comparação entre os caracteres correspondentes. Quando os caracteres coincidem, dizemos que houve um casamento (*matching*), e prosseguimos para o caractere seguinte até, eventualmente parar depois do casamento do último caractere da palavra p. Quando durante este processo, um caractere do texto diverge do caractere correspondente da palavra, dizemos que ocorreu um *mismatching*, e neste caso, precisamos deslocar a palavra uma posição à direita da posição atual, para então reiniciarmos o processo. Observe que a última posição do texto t a partir da qual ainda pode o casamento com a palavra p é $n - m$ já que depois desta posição não temos mais caracteres suficientes para casar com a palavra p. A seguir, apresentamos o pseudocódigo que implementa esta ideia:

```
for $i = 0$ to $n - m$ do
    $j \leftarrow 0$;
    while $j < m$ and $P[j] = T[i + j]$ do
        $j \leftarrow j + 1$;
    end
    if $j = m$ then
        return $i$;
    end
end
return $-1$;
```

Algoritmo 6: BFStringMatch($T[0..n-1], P[0..m-1]$)

Exercício 97. *Qual é a operação básica realizada por este algoritmo? Lembre-se que a operação básica é aquela que é mais relevante para a análise do custo do algoritmo.*

Exercício 98. *Qual é a complexidade deste algoritmo no melhor caso? Observe que o número de operações básicas a serem realizadas depende tanto do tamanho n do texto quanto do tamanho m da palavra, e portanto esta análise deve levar em conta os dois parâmetros.*

Exercício 99. *Qual é a complexidade deste algoritmo no pior caso? Observe que o número de operações básicas a serem realizadas depende tanto do tamanho n do texto quanto do tamanho m da palavra, e portanto esta análise deve levar em conta os dois parâmetros.*

3.2 Recursão

Recursão é um método para resolver problemas computacionais que depende das soluções de instâncias menores do mesmo problema[12] Nesta seção estudaremos alguns algoritmos recursivos, e iniciaremos pelas versões recursivas da busca sequencial e da ordenação por inserção. Como a utilização da recursão é muito natural no contexto funcional, isto é, da programação funcional, apresentaremos os algoritmos recursivos diretamente no assistente de provas Coq.

3.2.1 Busca sequencial recursiva

Considere a seguinte função recursiva:

$$seq\_search\ x\ l := \begin{cases} FALSE, & \text{se } l = nil; \\ TRUE, & \text{se } l = h :: l' \text{ e } h = x; \\ seq\_search\ x\ l', & \text{se } l = h :: l' \text{ e } h \neq x; \end{cases}$$

> Prove que se x for um elemento de l então *seq_search* x l retorna TRUE, e retorna FALSE, quando x não ocorre em l.

A busca sequencial pode ser definida recursivamente como $seq\_search\ l\ x := seq\_search\_aux\ l\ x\ 0$, onde:

Em Coq, temos:

```
Definition seq_search (l: list nat) (x: nat) := seq_search_aux l x 1.

Fixpoint seq_search_aux (l: list nat) (x pos: nat)  :=
 match l with
 | nil => 0
 | h::tl => if h =? x then pos else seq_search_aux tl x (S pos)
 end.
```

A função recursiva `seq_search` é definida em termos da função auxiliar `seq_search_aux` que recebe uma lista `l` de naturais, e os naturais `x` e `pos` como argumentos, e retorna 0 se `l` for a lista vazia. Ou seja, estamos assumindo que a primeira posição válida de uma lista é 1. Caso contrário, isto é, se `l` tem a forma `h::tl` então comparamos o primeiro elemento da lista `h` com o elemento procurado `x`, e se eles são iguais retornamos a posição atual `pos`. Caso contrário, continuamos recursivamente a busca de `x` na cauda `tl` da lista original depois de atualizar a variável `pos` em uma unidade.

TODO A correção do algoritmo de busca sequencial

TODO A complexidade do algoritmo de busca sequencial

3.2.2 TODO Busca binária

A busca binária é mais eficiente que a busca sequencial, mas este algoritmo assume que o vetor, onde a busca será realizada a busca, está ordenado. O pseudocódigo da busca binária em um vetor ordenado de inteiros com n elementos é dado a seguir:

```
if $high < low$ then
    return -1;
end
$mid = \lfloor (high + low)/2 \rfloor$;
if $key > A[mid]$ then
    return BinarySearch($A, mid + 1, high, key$);
end
else
    if $key < A[mid]$ then
        return BinarySearch($A, low, mid - 1, key$);
    end
    else
        return $mid$;
    end
end
```

Algoritmo 7: BinarySearch($A[1..n], low, high, key$)

Exercício 100. *Faça a análise da complexidade do melhor caso para este algoritmo.*

Exercício 101. *Faça a análise da complexidade do pior caso para este algoritmo.*

A correção deste algoritmo pode ser estabelecida em duas etapas. A primeira dela consiste em provar que se a chave *key* não ocorre no vetor $A[1..n]$, então BinarySearch($A[1..n], 1, n, key$) retorna o valor -1, e a segunda consiste em provar que o algoritmo retorna a posição correta do elemento procurado.

Exercício 102. *Prove o lema a seguir:*

Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key não ocorre em $A[1..n]$, então BinarySearch($A[1..n], 1, n, key$) retorna o valor -1.

***Dica**: Indução (forte) sobre o tamanho n do vetor.*

Exercício 103. *Prove o lema a seguir:*

Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key ocorre em $A[1..n]$, então BinarySearch($A[1..n], 1, n, key$) retorna o valor mid tal que $A[mid] = key$.

3.2.3 Ordenação por inserção recursivo

Nesta seção estudaremos o algoritmo de ordenação por inserção recursivo. A estrutura de dados utilizada é a de listas, e para simplificar trabalharemos com números naturais, mas as ideias são as mesmas para ordenarmos qualquer estrutura que possua uma ordem total. Vimos no capítulo anterior que as listas de naturais possuem dois construtores: *nil* para representar a lista vazia, e o operador :: que nos permite construir uma nova lista a partir de um número natural e de uma lista. Assim, a lista unitária contendo apenas o natural 5 é representada por $5 :: nil$, enquanto que a lista $1 :: (5 :: nil)$, ou simplesmente $1 :: 5 :: nil$, representa a lista que tem 1 como primeiro elemento, e a lista $5 :: nil$ como cauda.

A operação que dá nome ao algoritmo é a operação de inserção porque a cada passo queremos inserir um novo elemento em uma lista já ordenada. Suponha, por exemplo, que queiramos inserir o número 3 na lista $1 :: 5 :: nil$, isto é, o nosso objetivo final é obter a lista ordenada $1 :: 3 :: 5 :: nil$. Para isto, precisamos inicialmente comparar o 3 com o primeiro elemento da lista, e o resultado desta comparação nos diz que o 3 deve ser inserido depois do 1, ou seja, em algum lugar da cauda da lista. Em seguida, comparamos o 3 com o primeiro elemento da cauda, ou seja, com 5, e como $3 < 5$, sabemos que ele deve ser inserido antes do 5. Esta ideia está implementada na função *insere* definida a seguir:

Definição 49. *Sejam x um número natural, e l uma lista de números naturais. A função $(insere\ x\ l)$ que insere o natural x na lista l é definida recursivamente como a seguir:*

$$insere\ x\ l = \begin{cases} x :: nil, & se\ l = nil \\ x :: l, & se\ l = h :: tl\ \ e\ x \leq h \\ h :: (insere\ x\ tl), & se\ l = h :: tl\ \ e\ x > h \end{cases}$$

O algoritmo de ordenação por inserção então consiste em recursivamente, dada uma lista não vazia $h :: tl$, inserir o primeiro elemento h na versão ordenada da cauda tl. Ou seja, o algoritmo de ordenação por inserção que será implementado pela função de *ord_insercao* vai receber como argumento uma lista l para ordenar. Se l for a lista vazia não há nada a fazer, e caso contrário, recursivamente ordenamos a cauda da lista para então inserir o novo elemento:

Definição 50. *Seja l uma lista de números naturais. A função ord_insercao é definida recursivamente como a seguir:*

$$ord\_insercao\ l = \begin{cases} nil, & se\ l = nil \\ insere\ h\ (ord\_insercao\ tl), & se\ l = h :: tl \end{cases}$$

Vejamos como este algoritmo funciona na prática. Suponha que queiramos ordenar a lista $3 :: 2 :: 1 :: nil$. Ao fornecermos esta lista como argumento para a função *ord_insercao*, temos:

$$\begin{array}{ll} ord\_insercao\ (3 :: 2 :: 1 :: nil) = & (\text{def. } ord\_insercao) \\ insere\ 3\ (ord\_insercao\ (2 :: 1 :: nil)) = & (\text{def. } ord\_insercao) \\ insere\ 3\ (insere\ 2\ (ord\_insercao\ (1 :: nil))) = & (\text{def. } ord\_insercao) \\ insere\ 3\ (insere\ 2\ (insere\ 1\ (ord\_insercao\ nil))) = & (\text{def. } ord\_insercao) \\ insere\ 3\ (insere\ 2\ (insere\ 1\ nil)) = & (\text{def. } insere) \\ insere\ 3\ (insere\ 2\ (1 :: nil)) = & (\text{def. } insere) \\ insere\ 3\ (1 :: (insere\ 2\ nil)) = & (\text{def. } insere) \\ insere\ 3\ (1 :: 2 :: nil) = & (\text{def. } insere) \\ 1 :: (insere\ 3\ (2 :: nil)) = & (\text{def. } insere) \\ 1 :: 2 :: (insere\ 3\ nil) = & (\text{def. } insere) \\ 1 :: 2 :: 3 :: nil & \end{array}$$

Veja que o algoritmo ordenou corretamente a lista $3 :: 2 :: 1 :: nil$, mas será que ele ordena corretamente qualquer lista de números naturais? Para responder esta pergunta, vamos analisar se o algoritmo é correto ou não.

A correção do algoritmo de ordenação por inserção

Nesta seção vamos provar que o algoritmo de ordenação por inserção apresentado na seção anterior é correto. Para isto precisaremos definir algumas noções que serão utilizadas também em outros algoritmos de ordenação. A primeira noção que precisamos definir formalmente é a de ordenação. Ou seja, o que significa dizer que uma lista está ordenada? A definição a seguir apresenta o predicado *sorted* que caracteriza a noção de lista ordenada:

Definição 51. *Sejam x e y números naturais, e l uma lista de números naturais. O predicado sorted, que caracteriza o fato de uma lista estar ordenada, é definido por meio das seguintes regras de inferência:*

$$\frac{}{\mathit{sorted}\ \mathit{nil}}\ (\mathit{sorted\_nil}) \qquad\qquad \frac{}{\mathit{sorted}\ x :: \mathit{nil}}\ (\mathit{sorted\_one})$$

$$\frac{x \leq y \qquad \mathit{sorted}\ y :: l}{\mathit{sorted}\ x :: y :: l}\ (\mathit{sorted\_all})$$

A regra (*sorted_nil*) é um axioma que estabelece que a lista vazia está ordenada. A regra (*sorted_one*) também é um axioma que estabelece que listas unitárias estão ordenadas. A regra (*sorted_all*) possui duas condições para que uma lista da forma $x :: y :: l$ esteja ordenada: $x \leq y$ e a lista $y :: l$ tem que estar ordenada. Em outras palavras, a regra (*sorted_all*) diz que uma lista com pelo menos dois elementos está ordenada, se o primeiro elemento é menor ou igual ao segundo elemento, e a cauda da lista (ou seja, a lista do segundo elemento em diante) está ordenada. Note que as variáveis x, y e a lista l estão implicitamente quantificadas universalmente na Definição 51. Segundo esta definição, a lista $(1 :: 2 :: 3 :: \mathit{nil})$ está ordenada. De fato, a prova deste fato é dada pela seguinte árvore de derivação:

$$\dfrac{1 \leq 2 \qquad \dfrac{2 \leq 3 \qquad \dfrac{}{\mathit{sorted}\ (3 :: \mathit{nil})}\ (\mathit{sorted\_one})}{\mathit{sorted}\ (2 :: 3 :: \mathit{nil})}\ (\mathit{sorted\_all})}{\mathit{sorted}\ (1 :: 2 :: 3 :: \mathit{nil})}\ (\mathit{sorted\_all})$$

Com esta definição em mãos, podemos provar uma propriedade da função *insere* que ficou implícita:

Lema 52. *Sejam x um número natural, e l uma lista de números naturais. Se l está ordenada então $(\mathit{insere}\ x\ l)$ também está ordenada.*

Demonstração. A prova é por indução na estrutura da lista l. Se l for a lista vazia então $(\mathit{insere}\ x\ l)$ é a lista unitária $x :: \mathit{nil}$ que está ordenada por definição (regra *sorted_one*). Se l é da forma $h :: tl$ então temos dois casos a considerar:

- $x \leq h$: Neste caso, $\mathit{insere}\ x\ (h :: tl)$ retorna a lista $x :: h :: tl$ que está ordenada já que $x \leq h$ e, por hipótese, a lista $h :: tl$ está ordenada:

$$\frac{x \leq h \qquad \dfrac{}{\mathit{sorted}\ h :: tl}\ (\mathit{hip.})}{\mathit{sorted}\ x :: h :: tl}\ (\mathit{sorted\_all})$$

- $x > h$: Neste caso, x será inserido na cauda tl, e por hipótese de indução temos que a lista $(\mathit{insere}\ x\ tl)$ está ordenada. Como a lista $h :: tl$ está ordenada, então h é menor ou igual a todo elemento de tl. Logo h é menor ou igual que todo elemento da lista $(\mathit{insere}\ x\ tl)$, e portanto a lista $h :: (\mathit{insere}\ x\ tl)$ está ordenada.

□

Agora vamos refazer esta prova no Coq. Note que uma das grandes vantagens da utilização de um assistente de provas é justamente a possibilidade de verificação de que uma prova feita manualmente está, de fato, correta. Isto é muito importante em provas mais complexas onde erros podem passar despercebidos. Iniciaremos definindo a função *insere* em Coq. Funções recursivas são definidas usando a palavra reservada `Fixpoint`:

```
Require Import List Arith.

Fixpoint insere x l :=
   match l with
    | nil => x::nil
    | h::tl => if x <=? h then x::l
                else h :: (insere x tl)
   end.
```

Na primeira linha importamos duas bibliotecas, a primeira chamada `List` nos permite usar a notação `x::l` para representar uma lista com cabeça `x` e cauda `l`. A segunda biblioteca, chamada `Arith`, nos permite usar a comparação booleana `<=?`. Observe que a definição acima é a mesma função da Definição 49: ambas retornam a lista unitária `x::nil` quando `l` é a lista vazia, e quando `l` tem a forma `h::tl`, a função retorna `x::l` quando `x` $\leq$ `h`, e `h::(insere x tl)`, caso contrário. O predicado *sorted* é definido em Coq utilizando a palavra reservada `Inductive`. Neste caso, cada regra da Definição 51 aparece como sendo um construtor da definição:

```
Inductive sorted: list nat -> Prop :=
| sorted_nil: sorted nil
| sorted_one: forall x, sorted (x::nil)
| sorted_all: forall x y l, x <= y -> sorted (y::l) -> sorted (x::y::l).
```

Agora estamos prontos para enunciar o Lema 52 em Coq, e iniciar a prova fazendo indução na estrutura da lista `l`. Observe que utilizamos o comando `induction l as [|h tl]` para utilizarmos os mesmos nomes que aparecem na prova do Lema 52, mas este detalhe não muda nada na estrutura da prova. Poderíamos ter utilizado apenas o comando `induction l`, e a única diferença é que o Coq utilizaria nomes de variáveis diferentes para se referir à cabeça e cauda da lista `l`.

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
```

Temos então dois casos a considerar:

```
2 goals (ID 37)

 ============================
 forall x : nat, sorted nil -> sorted (insere x nil)

goal 2 (ID 41) is:
forall x : nat, sorted (h :: tl) -> sorted (insere x (h :: tl))
```

O primeiro caso se dá quando a lista `l` é a lista vazia. Então basta introduzirmos a variável `x` e a hipótese `sorted nil` no contexto, e aplicarmos a definição de `insere` via a tática `simpl` para concluirmos com a aplicação da regra `sorted_one`:

```
 Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
  induction l as [|h tl].
  - intros x H.
    simpl.
    apply sorted_one.
```

O segundo caso se dá quando a lista `l` tem a forma `(h::tl)`. Após introduzirmos a variável `x` e a hipótese `sorted (h::tl)`, precisamos provar que `sorted (insere x (h::tl))`:

```
  h : nat
  tl : list nat
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  ============================
  sorted (insere x (h :: tl))
```

Observe a hipótese de indução `IHtl` que foi gerada pelo princípio de indução aplicado à estrutura da lista `l`: ela tem exatamente a mesma forma do lema (ou seja, expressa a mesma propriedade) aplicada à cauda `tl` da lista `l`. De acordo com a prova que fizemos para o Lema 52, neste ponto precisamos comparar `x` e `h` para decidir o que fazer de acordo com a definição da função `insere`. Podemos então aplicar a tática `simpl` (de simplificação) para que a definição de `insere` seja aplicada no objetivo atual:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
```

A janela de prova correspondente é:

```
  h : nat
  tl : list nat
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  ============================
  sorted (if x <=? h then x :: h :: tl else h :: insere x tl)
```

Agora precisamos lidar com o condicional `if` para dividirmos a prova nos dois subcasos esperados. Para isto, utilizamos a tática `destruct` com `(x <=? h)` como argumento:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
```

Observe que a tática `destruct` foi utilizada com dois argumentos: `(x <=? h)` e `eqn:Hle`. O primeiro argumento divide a prova nos dois subcasos desejados, ou seja, `x` $\leq$ `h` e `x` $>$ `h`. Já o argumento `eqn:Hle` mantém a informação dos casos que estão sendo analisados no contexto, isto é, na janela de prova:

```
 2 goals (ID 61)

  h : nat
  tl : list nat
```

```
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  Hle : (x <=? h) = true
  ============================
  sorted (x :: h :: tl)

goal 2 (ID 62) is:
sorted (h :: insere x tl)
```

Se a informação da hipótese `Hle` não fosse relevante para a prova, poderíamos ter utilizado a tática `destruct` apenas com o argumento `(x <=? h)`. Mas observe que para provarmos `sorted (x::h::tl)` precisamos, de acordo com a regra `sorted_all`, mostrar que $\texttt{x} \leq \texttt{h}$ e que `sorted (h::tl)`, e para isto precisaremos das hipóteses `Hle` e `H`, respectivamente.

No primeiro subcaso precisamos provar `sorted (x::h::tl)`, ou seja, que uma lista com pelo menos dois elementos está ordenada. Então, aplicamos a regra `sorted_all`. Isto vai dividir a prova em dois novos subcasos: no primeiro precisamos provar que `x <= h`, e no segundo, `sorted (h::tl)`. O segundo caso é imediato da hipótese `H`, então vamos focar no primeiro caso. Observe que `x <= h` é essencialmente o que diz a hipótese `Hle: (x <=? h) = true`, mas escrito de outra forma. Estas diferentes formas de escrever a mesma informação estão relacionadas com a teoria por trás do Coq e fogem do escopo deste livro. Então o que precisamos fazer é descobrir como passar de uma notação para outra. O Coq tem diversos comandos de busca, dentre os quais está o comando `Search`. Para resolver o nosso problema precisamos encontrar lemas que envolvam a relação `<=`. Podemos fazer esta busca com o comando `Search le` ou `Search "_ <= _"`. Em ambos os casos, o Coq vai exibir uma janela com o resultado da busca. A seguir aparecem três dos resultado listados que estão relacionados com os operadores `<=` e `<=?`:

```
leb_complete: forall m n : nat, (m <=? n) = true -> m <= n
leb_correct: forall m n : nat, m <= n -> (m <=? n) = true
Nat.leb_le: forall n m : nat, (n <=? m) = true <-> n <= m
```

Note que o lema `leb_correct` não serve para nossos propósitos porque não temos o operador `<=` nas hipóteses, e sim na conclusão. Mas tanto `leb_complete` quanto `Nat.leb_le` resolvem este caso, e ambos podem ser aplicados tanto na conclusão quanto na hipótese `Hle`. Por exemplo, para aplicar a tática `leb_complete` na conclusão utilizamos o comando `apply leb_complete`. Abaixo temos a prova com a aplicação do lema `leb_complete` na hipótese `Hle`:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
   + apply sorted_all.
     * apply leb_complete in Hle.
       assumption.
     * assumption.
```

Retornando para a prova do lema `insere_sorted`, precisamos analisar o caso em que $\texttt{x} > \texttt{h}$. Veja como esta condição aparece na hipótese `Hle` neste momento da prova:

```
h : nat
tl : list nat
IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
x : nat
H : sorted (h :: tl)
Hle : (x <=? h) = false
============================
sorted (h :: insere x tl)
```

Para utilizarmos aqui o mesmo argumento da prova do Lema 52, precisaremos encontrar uma forma de representar que um elemento é menor ou igual a todo elemento de uma lista, e relacionar este fato com o predicado `sorted`. Para isto, definimos o predicado `le_all`, de forma que `le_all x l` representa o fato de que `x` é menor ou igual a todo elemento de `l`, da seguinte forma:

```
Definition le_all x l := forall y, In y l -> x <= y.
```

Ou seja, `x` é menor ou igual a todo elemento de `l` se `x` $\leq$ `y`, para todo `y` que seja elemento de `l`. O predicado `In` está definido em Coq, de forma que `In y l` significa que `y` é um elemento de `l`. A definição do predicado `In` pode ser vista com o comando `Print In`:

```
In =
fun A : Type =>
fix In (a : A) (l : list A) {struct l} : Prop :=
 match l with
 | nil => False
 | b :: m => b = a \/ In a m
 end
    : forall A : Type, A -> list A -> Prop
```

A palavra reservada `fix` denota que `In` é uma função recursiva (exatamente como fazemos com `Fixpoint`). Assim, se a lista dada como segundo argumento em `In a l` for a lista vazia a função retorna `False`, ou seja, `a` não é um elemento de `l`. Caso contrário, suponha que `l` tem a forma `b::m`, e neste caso verificamos se `b=a` ou então recursivamente continuamos a busca pelo elemento `a` na cauda `m`.

Agora podemos enunciar o argumento que precisamos para completar a prova do lema `insere_sorted`, ou seja, precisamos provar que se `l` é uma lista ordenada, e `a` é um natural menor ou igual a todo elemento de `l` então a lista `(a::l)` está ordenada:

```
Lemma le_all_sorted: forall l a, le_all a l -> sorted l -> sorted (a::l).
```

Este lema pode ser provado por análise de casos sobre a estrutura da lista `l`, isto é, basta inspecionarmos o que ocorre quando `l` é a lista vazia e quando é uma lista não vazia. Você pode estar se perguntando: mas não é isto que fazemos em uma prova por indução? Sim, a diferença é que na análise de casos não precisamos da hipótese de indução. Enquanto uma prova por indução é feita com a tática `induction`, a análise de casos é feita com a tática `case`:

```
Lemma le_all_sorted: forall l a, le_all a l -> sorted l -> sorted (a::l).
Proof.
 intro l.
 case l.
```

Neste ponto a prova é divida em dois subcasos, um quando `l` é a lista vazia, e outro quando `l` é uma lista não vazia:

```
2 goals (ID 42)

 l : list nat
 ============================
 forall a : nat, le_all a nil -> sorted nil -> sorted (a :: nil)

goal 2 (ID 43) is:
 forall (n : nat) (l0 : list nat) (a : nat),
 le_all a (n :: l0) -> sorted (n :: l0) -> sorted (a :: n :: l0)
```

O restante desta prova será deixado como exercício, e como dica fica a sugestão de dar uma olhada no lema `in_eq` que pode ser útil para completar a prova.

Exercício 104. *Complete a prova do lema `le_all_sorted`.*

Agora podemos retomar a prova do lema `insere_sorted` e aplicar o lema `le_all_sorted` que acabamos de provar. Isto vai dividir a prova em dois subcasos:

```
2 goals (ID 114)

 h : nat
 tl : list nat
 IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
 x : nat
 H : sorted (h :: tl)
 Hle : (x <=? h) = false
 ============================
 le_all h (insere x tl)

goal 2 (ID 115) is:
 sorted (insere x tl)
```

No primeiro subcaso precisamos provar que `h` é menor ou igual a todo elemento da lista `(insere x tl)`, e no segundo, que a lista `(insere x tl)` está ordenada. Como provar `le_all h (insere x tl)`? Isto é, como provar que `h` é menor ou igual a todo elemento da lista `(insere x tl)`? A hipótese `Hle` diz, usando a comparação booleana, que `h` $<$ `x`. Adicionalmente, a hipótese `H` diz que a lista `(h::tl)` está ordenada, e portanto `h` tem que ser menor do que todo elemento em `tl`. Estes dois fatos nos permitem concluir informalmente o que queremos, mas como fazer isto em Coq? A ideia é novamente enunciar um lema auxiliar que será provado separadamente:

```
Lemma le_all_insere: forall l x y, y <= x -> le_all y l -> le_all y (insere x l).
Proof.
Admitted.
```

O lema `le_all_insere` expressa a propriedade que precisamos para continuar a prova do lema `insere_sorted`. Ao invés de provarmos este lema agora, vamos usá-lo para ver se realmente conseguimos avançar na prova de `insere_sorted`. Sua prova só será feita depois de verificarmos que o ele é realmente útil. Esta é uma estratégia importante no desenvolvimento de provas formais porque evita gastarmos energia na prova de um lema que eventualmente precise ser modificado, ajustado ou mesmo eliminado em um momento posterior. O comando `Admitted` é utilizado nesta situação: permite que a utilização do lema ainda que ele não esteja provado. Ao aplicarmos o lema `le_all_insere` ao contexto atual, isto é, ao primeiro dos objetivos gerados na aplicação do lema `le_all_sorted`, obtemos uma nova bifurcação da prova:

```
    h : nat
    tl : list nat
    IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
    x : nat
    H : sorted (h :: tl)
    Hle : (x <=? h) = false
    ============================
    h <= x

   goal 2 (ID 117) is:
   le_all h tl
```

O primeiro subobjetivo, a saber `h <= x` pode ser provado a partir da hipótese `Hle`:

```
 Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
  induction l as [|h tl].
  - intros x H.
    simpl.
    apply sorted_one.
  - intros x H.
    simpl.
    destruct (x <=? h) eqn:Hle.
    + apply sorted_all.
      * apply leb_complete in Hle.
        assumption.
      * assumption.
    + apply le_all_sorted.
      * apply le_all_insere.
        ** apply leb_complete_conv in Hle.
           apply Nat.lt_le_incl in Hle.
           assumption.
```

No segundo ramo da prova precisamos provar `le_all h tl`, ou seja, que `h` é menor ou igual a todo elemento da lista `tl`. Precisamos então de uma propriedade semelhante ao lema `le_all_sorted`, mas na outra direção:

```
 Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
 Proof.
 Admitted.
```

Também deixaremos a prova deste lema para um momento posterior, mas é importante estar seguro de que todos os lemas deixados em aberto expressam propriedades corretas. Este é o caso do lema `sorted_le_all` porque se a lista `(a::l)` está ordenada então o primeiro elemento tem que ser menor ou igual a todos os elementos da cauda. Este segundo ramo é concluído de forma imediata com a ajuda deste lema:

```
 Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
 Proof.
  induction l as [|h tl].
  - intros x H.
    simpl.
    apply sorted_one.
  - intros x H.
    simpl.
```

```
    destruct (x <=? h) eqn:Hle.
    + apply sorted_all.
      * apply leb_complete in Hle.
        assumption.
      * assumption.
    + apply le_all_sorted.
      * apply le_all_insere.
        ** apply leb_complete_conv in Hle.
           apply Nat.lt_le_incl in Hle.
           assumption.
        ** apply sorted_le_all.
           assumption.
```

O segundo caso gerado na aplicação do lema `le_all_sorted` consiste na prova de que a lista `(insere x tl)` está ordenada:

```
 1 goal (ID 115)

  h : nat
  tl : list nat
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  Hle : (x <=? h) = false
  ============================
  sorted (insere x tl)
```

Note que podemos obter `sorted (insere x tl)` da hipótese de indução `IHtl` desde que a lista `tl` esteja ordenada, isto é, desde que tenhamos uma prova de `sorted tl`. Esta prova pode ser obtida da hipótese `H`, pois se a lista `(h::tl)` está ordenada então sua cauda `tl` também está ordenada. Apesar deste fato ser óbvio, precisamos provar mais este resultado auxiliar no Coq:

```
Lemma sorted_sublist: forall l a, sorted (a::l) -> sorted l.
```

A prova deste lema pode ser feita via análise de casos na estrutura da lista `l` e é deixada como exercício:

Exercício 105. *Prove o lema `sorted_sublist`.*

Agora podemos concluir a prova do lema `insere_sorted`:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
   + apply sorted_all.
     * apply leb_complete in Hle.
       assumption.
     * assumption.
   + apply le_all_sorted.
```

```
      * apply le_all_insere.
        ** apply leb_complete_conv in Hle.
           apply Nat.lt_le_incl in Hle.
           assumption.
        ** apply sorted_le_all.
           assumption.
      * apply IHtl.
        apply sorted_sublist in H.
        assumption.
Qed.
```

Agora que sabemos que os lemas `le_all_insere` e `sorted_le_all` são efetivamente úteis em nossa formalização, podemos prová-los.
Vamos iniciar com a prova do lema `sorted_le_all`, que é feita por indução na estrutura da lista `l`:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l.
```

Quando a lista `l` é a lista vazia (base da indução), precisamos provar que o natural `a` é menor ou igual a todo elemento da lista vazia. Como a lista vazia não possui nenhum elemento, dizemos que este fato é verdadeiro por vacuidade, isto é, porque não existe nenhum elemento que o contradiz. Para ver como este tipo de situação ocorre em Coq, vamos abrir a definição de `le_all` com o comando `unfold le_all`:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l as [|h tl].
 - intros a H.
   unfold le_all.
```

O comando `unfold le_all` simplesmente substitui a expressão `le_all a l` pela expressão correspondente à definição de `le_all`:

```
1 goal (ID 79)

 a : nat
 H : sorted (a :: nil)
 ============================
 forall y : nat, In y nil -> a <= y
```

Depois de fazermos as introduções possíveis, temos a hipótese `In y nil`. Como a lista vazia não possui elementos, a tática `inversion` nos permite concluir este ramo da prova.

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l as [|h tl].
 - intros a H.
   unfold le_all.
   intros y Hnil.
   inversion Hnil.
```

No passo indutivo, a lista `l` tem a forma `h::tl`, e a janela de prova após fazermos as introduções possíveis é a seguinte:

```
 1 goal (ID 86)

  h : nat
  tl : list nat
  IHtl : forall a : nat, sorted (a :: tl) -> le_all a tl
  a’ : nat
  H : sorted (a’ :: h :: tl)
  ============================
  le_all a’ (h :: tl)
```

Então precisamos provar que `a’` é menor ou igual a todo elemento da lista `(h::tl)`. Dividiremos esta tarefas em dois passos: primeiro mostraremos que `a’` é menor ou igual a `h`, e depois que `a’` é menor ou igual a todo elemento da lista `tl`. Para isto vamos enunciar mais um lema auxiliar cuja prova será deixada como exercício:

Exercício 106. *Prove o lema a seguir utilizando análise de casos na estrutura da lista `l`:*

```
Lemma le_le_all: forall l x y, y <= x -> le_all y l -> le_all y (x::l).
```

A aplicação do lema `le_le_all` divide a prova nos dois subcasos descritos acima. A prova de que `a’` $\leq$ `h` pode ser obtida da hipótese `H` porque a regra `sorted_all` diz que para uma lista com dois ou mais elementos estar ordenada, o primeiro elemento precisa ser menor ou igual ao segundo. Então utilizamos a tática `inversion` para que esta condição seja gerada a partir da hipótese `H`:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l as [|h tl].
 - intros a H.
   unfold le_all.
   intros y Hnil.
   inversion Hnil.
 - intros a’ H.
   apply le_le_all.
   + inversion H; subst.
     assumption.
```

O segundo subcaso gerado consiste em provar `le_all a’ tl`. Para isto podemos utilizar a hipótese de indução. Veja a janela de prova atual:

```
  h : nat
  tl : list nat
  IHtl : forall a : nat, sorted (a :: tl) -> le_all a tl
  a’ : nat
  H : sorted (a’ :: h :: tl)
  ============================
  le_all a’ tl
```

Com o comando `apply IHtl` aplicamos a hipótese de indução ao objetivo atual, e o novo objetivo a ser provado passa a ser o antecedente da implicação que compõe a hipótese de indução considerando que a variável universal `a` de `IHtl` foi instanciada com `a’`:

```
  h : nat
  tl : list nat
  IHtl : forall a : nat, sorted (a :: tl) -> le_all a tl
  a’ : nat
```

```
H : sorted (a' :: h :: tl)
============================
sorted (a' :: tl)
```

A prova de `sorted (a'::tl)` pode ser obtida a partir da hipótese `H`, se pudéssemos remover o segundo elemento da lista `(a'::h::tl)`, mas como fazer isto? Exatamente, através de um lema auxiliar já que a extração do segundo elemento de uma lista ordenada não decorre diretamente das definições que temos. Esta tarefa fica como exercício e pode ser feita por análise de casos:

Exercício 107. *Complete a prova do lema* `sublist_sorted`*:*

```
Lemma sublist_sorted: forall l a1 a2, sorted (a1 :: a2 :: l) -> sorted (a1 :: l).
Proof.
intro l; case l.
```

A última pendência em relação à prova do lema `insere_sorted` é o lema auxiliar `le_all_insere`. Esta prova será deixada como exercício já que sua prova pode ser feita com a ajuda dos lemas auxiliares já apresentados, ou seja, nenhum lema auxiliar adicional é necessário.

Exercício 108. *Prove o lema a seguir utilizando indução na estrutura da lista* `l`*:*

```
Lemma le_all_insere: forall l x y, y <= x -> le_all y l -> le_all y (insere x l).
```

Seguimos um longo caminho até completarmos uma versão formal (ou mecânica) da prova do Lema 52. Uma pergunta natural é: existe um caminho mais curto, ou em outras palavras, existe uma outra prova possível para este lema? A resposta é sim! A seguir apresentamos uma prova alternativa que não requer lemas auxiliares:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
   + apply sorted_all.
     * apply leb_complete in Hle.
       assumption.
     * assumption.
   + generalize dependent tl.
     intro tl; case tl.
     * intros IH H.
       simpl.
       apply sorted_all.
       ** apply leb_complete_conv in Hle.
          apply Nat.lt_le_incl in Hle.
          assumption.
       ** apply sorted_one.
     * intros n l IH H.
       simpl in *.
       destruct (x <=? n) eqn:H'.
       ** apply sorted_all.
```

```
            *** apply leb_complete_conv in Hle.
                apply Nat.lt_le_incl in Hle.
                assumption.
            *** apply sorted_all.
                **** apply leb_complete.
                     assumption.
                **** inversion H; subst.
                     assumption.
         ** inversion H; subst.
            apply sorted_all.
            *** assumption.
            *** specialize (IH x).
                apply IH in H4.
                rewrite H' in H4.
                assumption.
Qed.
```

Você compreendeu o que esta prova faz de diferente? Como exercício vamos fazer o inverso do que foi feito com o Lema 52.

Exercício 109. *Construa uma prova em linguagem natural que corresponda a estratégia utilizada na prova em Coq acima.*

Nosso próximo passo é provar que o algoritmo de ordenação por inserção efetivamente ordena qualquer lista de naturais dada como entrada:

Lema 53. *O algoritmo de ordenação por inserção da Definição 50 ao receber uma lista l de números naturais como argumento retorna uma lista ordenada. Em outras palavras, a lista $(ord\_insercao\ l)$ está ordenada, para qualquer lista l.*

Demonstração. A prova é por indução na estrutura da lista l. Se l é a lista vazia (base de indução) então, por definição temos que $ord\_insercao\ nil = nil$, e não há o que fazer porque a lista vazia está ordenada. No passo indutivo suponha que l tem a forma $h :: tl$. Temos $ord\_insercao\ (h :: tl) = insereh(ord\_insercao\ tl)$, e por hipótese de indução temos que a lista $(ord\_insercao\ tl)$. Então, pelo Lema 52 concluímos que a lista $insereh(ord\_insercao\ tl)$ está ordenada, e portanto $ord\_insercao\ (h :: tl)$ está ordenada.

□

Exercício 110. *Refaça a prova acima utilizando a estrutura de árvore. Em outras palavras, prove o sequente $\vdash sorted(ord\_insercao\ l)$.*

Agora prove o Lemma 53 em Coq:

Exercício 111. `Lemma ord_insercao_ordena: forall l, sorted (ord_insercao l).`

A segunda parte da prova da correção de um algoritmo de ordenação consiste em mostrar que o algoritmo retorna uma lista que é uma permutação da lista de entrada. Assim, um algoritmo de ordenação será correto se, para qualquer lista l dada como entrada, a saída for uma permutação de l que esteja ordenada. Ou seja, a resposta do algoritmo tem que ser uma lista que contém exatamente os mesmos elementos da lista de entrada e que adicionalmente esteja ordenada.

Como então definir a noção de permutação? Temos pelo menos duas opções. A primeira é simplesmente contar o número de ocorrências de cada elemento e ver que qualquer elemento tem que ocorrer o mesmo número de vezes nas duas listas para que uma seja uma permutação da outra. De maneira mais precisa, podemos definir o número de ocorrências de x na lista l, notação $num\_oc\ x\ l$ da seguinte forma:

Definição 54. *Seja x um número natural, e l uma lista de números naturais. Definimos recursivamente o número de ocorrências de x em l por:*

$$num\_oc\ x\ l = \begin{cases} 0, & se\ l = nil \\ 1 + num\_oc\ x\ tl, & se\ l = x :: tl \\ num\_oc\ x\ tl, & caso\ contr\acute{a}rio. \end{cases}$$

O predicado *perm*, que define quando duas lista, digamos l e l' são permutações uma da outra.

Definição 55. *Sejam l e l' listas de números naturais. Definimos o predicado perm em função de num_oc por $perm\ l\ l' := \forall x, num\_oc\ x\ l = num\_oc\ x\ l'$.*

De acordo com esta definição, a lista l' é uma permutação da lista l se o número de ocorrências de x em l é igual ao número de ocorrências de x em l'. Nosso objetivo agora é mostrar que o algoritmo de ordenação por inserção gera uma lista que é uma permutação da lista de entrada, ou seja, queremos provar o seguinte teorema:

Teorema 56. *Seja l uma lista de números naturais. O algoritmo de ordenação por inserção gera como saída uma lista que é permutação da lista de entrada, ou seja, o sequente $\vdash perm\ l\ (ord\_insercao\ l)$ é válido.*

Demonstração. A prova é por indução na estrutura da lista l. Quando l é a lista vazia (base da indução), temos que $num\_oc\ x\ (ord\_insercao\ nil) = num\_oc\ x\ nil$ para todo x, ou seja, nil é uma permutação de $(ord\_insercao\ nil)$. Suponha que l tenha a forma $h :: tl$ (passo indutivo). Precisamos provar que $(h :: tl)$ é uma permutação da lista $(ord\_insercao\ (h :: tl))$, que pela definição de $ord\_insercao$ é igual a $(insere\ h\ (ord\_insercao\ tl))$. Por hipótese de indução temos que tl é uma permutação da lista $(ord\_insercao\ tl)$. Considerando que a função $(insere\ h\ (ord\_insercao\ tl))$ apenas adiciona o elemento h à lista $(ord\_insercao\ tl)$, concluímos que a lista $(h :: tl)$ é uma permutação da lista $insere\ h\ (ord\_insercao\ tl)$, que por sua vez é igual a $(ord\_insercao\ (h :: tl))$, como queríamos demonstrar.

□

Como seria a representação desta prova em forma de árvore? O primeiro passo é aplicar o princípio de indução para listas:

$$\dfrac{(def.)\ \dfrac{(\text{Refl})\ \dfrac{}{perm\ nil\ nil}}{perm\ nil\ (ord\_insercao\ nil)} \qquad \dfrac{\dfrac{(*)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}}{perm\ (h :: tl)\ (ord\_insercao\ (h :: tl))}\ (def.)}{perm\ l\ (ord\_insercao\ l)}\ (ind.\ em\ l)$$

A folha do ramo esquerdo, não é nada mais do que a igualdade $\forall x, num\_oc\ x\ nil = \_oc\ x\ nil$, e portanto este ramo da prova está completo pelo axioma da reflexividade da igualdade apresentado abaixo, onde t é um termo qualquer:

$$\dfrac{}{t = t}\ (\text{Refl})$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{perm\ tl\ (ord\_insercao\ tl)}\ (h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\ (def.)}{num\_oc\ h\ tl = num\_oc\ h\ (ord\_insercao\ tl)}\ (\forall_e)}{S(num\_oc\ h\ tl) = S(num\_oc\ h\ (ord\_insercao\ tl))}}{num\_oc\ h\ (h :: tl) = num\_oc\ h\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{(*)}$$

A continuação da prova do ramo direito segue com a aplicação da definição de *perm* na leitura de baixo para cima. Em seguida aplicamos a definição de *num_oc*, e adicionamos a função sucessor em ambos os lados da igualdade já que h ocorre tanto do lado esquerdo quanto do lado direito. A regra utilizada acima sem nome (quarta linha de baixo para cima) corresponde à uma propriedade algébrica que não precisa ser detalhada aqui (injetividade da igualdade), mas é importante notar que a passagem da prova em linguagem natural (que normalmente chamamos de prova informal) para a prova em forma de árvore já exige uma disciplina maior porque cada regra aplicada na árvore exige uma justificativa mais detalhada. Por fim, aplicamos mais uma vez a definição de `perm` para chegarmos na hipótese de indução.

Agora vamos formalizar esta prova em Coq. Como sabemos, a disciplina exigida é ainda maior do que a que foi necessária para a construção da árvore de derivação. Iniciaremos com a definição da função *num_oc* em Coq:

```
Fixpoint num_oc n l  :=
 match l with
   | nil => 0
   | h :: tl =>
     if n =? h then S(num_oc n tl) else  num_oc n tl
 end.
```

Após se certificar que esta definição faz exatamente o mesmo que a Definição 54, podemos construir a versão em Coq da definição 55:

```
Definition perm l l' := forall n:nat, num_oc n l = num_oc n l'.
```

Agora vamos refazer a prova do Teorema 56 em Coq. A prova é por indução na estrutura da lista `l`, e a base de indução, como na prova informal (e na árvore de derivação), é imediata. Basta abrirmos a definição com a tática `unfold` e aplicarmos o axioma da reflexividade da igualdade (tática `reflexivity`).

```
 Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
Proof.
  induction l as [|h tl].
  - simpl.
    unfold perm.
    reflexivity.
```

No passo indutivo, precisamos aplicar a definição de *perm* para que tenhamos o objetivo em função de *num_oc*, ou seja, precisamos aplicar a tática `unfold`. A janela de prova correspondente é mostrada a seguir:

```
h : nat
tl : list nat
IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
============================
forall n : nat,
```

```
   num_oc n (h :: tl) = num_oc n (insere h (ord_insercao tl))
```

Aqui é possível ver um problema na árvore de dedução acima. A aplicação da definição de *perm* (na primeira linha de baixo para cima) está **errada**! De fato, a definição de *perm* é feita sobre uma variável quantificada universalmente, enquanto que na árvore acima esta variável está instanciada como h, ou seja, é um caso particular da definição e portanto não serve como prova. Esta situação simples, mas serve para mostrar como uma formalização pode ajudar a corrigir erros de uma prova informal. A nossa estratégia será completar primeiro a prova em Coq, e a partir daí refazer a árvore de dedução. Após introduzirmos a variável universal `n`, precisamos comparar `n` com `h` para saber se o contador precisa ou não ser incrementado. Podemos, depois de `intro n`, usar a tática `simpl` para aplicar a definição de `num_oc` e gerar condicional que vai nos permitir dividir a prova em dois casos:

```
   h : nat
   tl : list nat
   IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
   n : nat
   ============================
   (if n =? h then S (num_oc n tl) else num_oc n tl) =
   num_oc n (insere h (ord_insercao tl))
```

Com o comando `destruct (n =?  h) eqn:H.` dividimos a prova em dois casos e guardamos a informação do caso em andamento na hipótese `H`. O primeiro caso é quando `n` é igual a `h`, e corresponde ao caso analisado em nossa árvore de dedução. No entanto, a árvore não analisou o caso em que `n` e `h` são distintos. Utilizaremos o lema `beq_nat_true` para transformar a comparação booleana em igualdade sintática, e assim substituir (tática `subst`) todas as ocorrências de `n` por `h`:

```
Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
Proof.
 induction l as [|h tl].
 - simpl.
   unfold perm.
   reflexivity.
 - simpl.
   unfold perm in *.
   intro n.
   simpl.
   destruct (n =? h) eqn:H.
   + apply beq_nat_true in H.
     subst.
```

E a janela de prova correspondente é a seguinte:

```
   h : nat
   tl : list nat
   IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
   ============================
   S (num_oc h tl) = num_oc h (insere h (ord_insercao tl))
```

Agora precisamos que o Coq transformar `num_oc h (insere h (ord_insercao tl))` em `S (num_oc h (ord_insercao tl))`. No entanto, esta transformação não é trivial do ponto de vista formal porque não sabemos de antemão a posição da lista `(ord_insercao tl)` em que `h` será inserido. Ou seja, esta transformação não pode ser obtida de forma imediata das definições de `num_oc` e `insere`. Portanto, precisamos de um lema auxiliar que faça isto:

```
Lemma num_oc_insere: forall l x, num_oc x (insere x l) = S (num_oc x l).
```

A prova deste lema será deixada como exercício, e pode ser feita por indução na estrutura da lista `l`.

Exercício 112. *Prove o lema* `num_oc_insere`*.*

Como o lema `num_oc_insere` é uma igualdade então utilizamos a tática `rewrite`, e depois disto fechamos este ramo da prova com a hipótese de indução:

```
Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
Proof.
 induction l as [|h tl].
 - simpl.
   unfold perm.
   reflexivity.
 - simpl.
   unfold perm in *.
   intro n.
   simpl.
   destruct (n =? h) eqn:H.
   + apply beq_nat_true in H.
     subst.
     rewrite num_oc_insere.
     rewrite IHtl.
     reflexivity.
```

Por fim, podemos analisar o caso que faltou na nossa árvore de derivação, a saber, o caso em que `n` é diferente de `h`:

```
h : nat
tl : list nat
IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
n : nat
H : (n =? h) = false
============================
num_oc n tl = num_oc n (insere h (ord_insercao tl))
```

Este ramo pode ser provado facilmente, desde que consigamos transformar `num_oc n (insere h (ord_insercao tl))` em `num_oc n (ord_insercao tl)`, o que é verdade já que `n` é diferente de `h`. Novamente precisaremos de um resultado auxiliar cuja prova será deixada como exercício:

```
Lemma num_oc_insere_diff: forall l x y, (x =? y) = false -> num_oc x (insere y l) = num_oc x l.
```

Exercício 113. *Prove o lema* `num_oc_insere_diff`*.*

Com este lema e a hipótese de indução conseguimos completar a prova:

```
Theorem perm_ord_insercao: forall l, perm l (ord_insercao l).
Proof.
 induction l as [|h tl].
 - simpl.
   unfold perm.
   reflexivity.
 - simpl.
```

```
    unfold perm in *.
    intro n.
    simpl.
    destruct (n =? h) eqn:H.
    + apply beq_nat_true in H.
      subst.
      rewrite num_oc_insere.
      rewrite IHtl.
      reflexivity.
    + rewrite num_oc_insere_diff.
      * apply IHtl.
      * assumption.
Qed.
```

Agora podemos corrigir o ramo da árvore de dedução que corresponde ao passo indutivo. Note que a aplicação da definição de *perm* (primeira regra de baixo para cima) gera uma fórmula quantificada universalmente.

$$
\dfrac{
  \dfrac{
    \dfrac{
      (h = x)\ \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{
              \dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\ (h.i.)
            }{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\ (def.)
          }{num\_oc\ h\ tl = num\_oc\ h\ (ord\_insercao\ tl)}\ (\forall_e)
        }{S(num\_oc\ h\ tl) = S(num\_oc\ h\ (ord\_insercao\ tl))}
      }{num\_oc\ h\ (h :: tl) = num\_oc\ h\ (insere\ h\ (ord\_insercao\ tl))}
      \qquad (**)
    }{num\_oc\ x\ (h :: tl) = num\_oc\ x\ (insere\ h\ (ord\_insercao\ tl))}\ (h = x \vee h \neq x)
  }{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)
}{(*)}
$$

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\ (h.i.)
      }{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\ (def.)
    }{num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\ (\forall_e)
  }{num\_oc\ x\ (h :: tl) = num\_oc\ x\ (insere\ h\ (ord\_insercao\ tl))}\ (h \neq x)
}{(**)}
$$

Os lemas *ord_insercao_ordena* e $ord\_insercao_perm$ juntos caracterizam a correção do algoritmo de ordenação por inserção. Em Coq, temos:

```
Theorem ord_insercao_correcao: forall l, sorted (ord_insercao l) /\ perm l (ord_insercao l).
Proof.
 intro l. split.
 - apply ord_insercao_ordena.
 - apply ord_insercao_perm.
Qed.
```

Para finalizar esta seção, mostre que *perm* é uma relação de equivalência sobre a estrutura de listas, isto é, mostre que *perm* é reflexiva, simétrica e transitiva.

Exercício 114. *Mostre que o predicado perm da Definição 55 é uma relação de equivalência sobre a estrutura de listas, isto é, mostre:*

1. *perm l l, para qualquer lista l (reflexividade)*

2. *Se perm l l' então perm l' l, quaisquer que sejam as listas l e l' (simetria)*

3. *Se perm l l' e perm l' l'' então perm l l'', quaisquer que sejam as listas l, l' e l'' (transitividade)*

4. *Refaça suas provas no Coq:*

```
Lemma perm_refl: forall l, perm l l.

Lemma perm_sym: forall l l’, perm l l’ -> perm l’ l.

Lemma perm_trans: forall l l’ l’’, perm l l’ -> perm l’ l’’ -> perm l l’’.
```

Existem formas distintas de definirmos o mesmo conceito, ou seja, existem formas distintas de escrever a mesma coisa. A consequência é um conjunto de provas diferentes que podem ser mais simples ou mais complexas. No contexto de provas informais, a mudança de uma definição pode não ter muito impacto, mas o contexto formal é muito mais sensível a este tipo de mudança. Além disto, a mudança ou mesmo um ajuste em uma definição ou teorema durante uma formalização normalmente implica em ter que refazer todas as provas que dependem daquela mudança. Por isto, um bom planejamento é fundamental antes de iniciar uma formalização. Para exemplificar como definições distintas podem impactar em uma formalização, apresentaremos uma definição indutiva da noção de permutação de listas.

Definição 57. *Sejam x e y números naturais, e l, l' e l'' listas de números naturais. O predicado binário* `permutation` *é definido pelas regras de inferência seguintes:*

$$\frac{}{permutation\ nil\ nil}\ (permutation\_nil)$$

$$\frac{permutation\ l\ l'}{permutation\ (x :: l)\ (x :: l')}\ (permutation\_skip)$$

$$\frac{}{permutation\ (y :: x :: l)\ (x :: y :: l)}\ (permutation\_swap)$$

$$\frac{permutation\ l\ l' \qquad permutation\ l'\ l''}{permutation\ l\ l''}\ (permutation\_trans)$$

Você pode estar se perguntando se as definições *perm* e *permutation* são equivalentes. A resposta é sim, e a conclusão desta seção será justamente a prova desta equivalência. Isto significa que a utilização de uma ou outra não fará diferença do ponto de vista prático, mas pode fazer em relação à simplicidade ou complexidade das provas envolvidas. Vamos mostrar que o algoritmo de ordenação por inserção gera como saída uma lista que é uma permutação da lista de entrada segundo esta nova definição. Como a definição de *permutation* é feita via regras de inferência, é mais natural que a prova seja feita na forma de árvore:

Lema 58. *Seja l uma lista de números naturais. Então o sequente $\vdash permutation\ l\ (ord\_insercao\ l)$ é válido.*

Demonstração. A prova é por indução na estrutura da lista l. A base de indução é simples:

$$\dfrac{\dfrac{}{permutation\ nil\ nil}\ (permutation\_nil)}{permutation\ nil\ (ord\_insercao\ nil)}\ (def.)$$

Agora suponha que l tenha a forma $h :: tl$. Queremos construir uma prova para o sequente $\vdash permutation\ (h :: tl)\ (ord\_insercao\ (h :: tl))$. O primeiro passo é aplicar a definição de *ord_insercao*:

$$\dfrac{\dfrac{(*)}{permutation\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}}{permutation\ (h :: tl)\ (ord\_insercao\ (h :: tl))}\ (def.)$$

$\square$

E neste ponto precisamos de um resultado auxiliar porque nenhuma das regras pode ser aplicada já que não sabemos quem é(são) o(s) primeiro(s) elemento(s) da lista $(insere\ h\ (ord\_insercao\ tl))$. Na verdade, a utilização da regra *permutation_trans* é possível, mas precisaríamos de uma lista intermediária que nos permitisse avançar na prova. Vamos seguir o caminho do resultado auxiliar e provar a propriedade que corresponde ao objetivo atual:

Lema 59. *Sejam x um número natural, l e l' listas de números naturais. Se $(permutation\ l\ l')$ então $permutation\ (a :: l)\ (insere\ a\ l')$. Ou seja, se l' é uma permutação de l então $(insere\ a\ l')$ é uma permutação de $(a :: l)$.*

A prova deste lema é, sem dúvida a prova mais bonita que apresentaremos aqui, mas antes observe que com ele concluímos de forma imediata a prova do Lema 58:

$$\dfrac{\dfrac{\dfrac{}{permutation\ tl\ (ord\_insercao\ tl)}\ (h.i.)}{permutation\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (\text{Lema } 70)}{(*)}$$

Observe que a aplicação do Lema 70 foi feita instanciando a com h, l com tl, e l' com $(ord\_insercao\ tl)$.

Como exercício, reproduza esta prova em Coq:

Exercício 115. `Theorem ord_insercao_permutation: forall l, permutation l (ord_insercao l).`

Agora vamos fazer a prova do Lema 59:

Demonstração. Observe que o lema consiste em uma implicação: temos como hipótese $(permutation\ l\ l')$ e queremos provar $permutation\ (a :: l)\ (insere\ a\ l')$. Adicionalmente, o predicado *permutation* é indutivo (assim como os números naturais), e portanto podemos fazer a prova por indução na hipótese $(permutation\ l\ l')$. Isto significa que teremos um caso para cada regra da Definição 57.

1. O primeiro caso é o da regra (*permutation_nil*): para que a hipótese $(permutation\ l\ l')$ tenha sido gerada por esta regra é preciso que tanto l quanto l' sejam a lista vazia. Nesta situação, o que queremos provar é $permutation\ (a :: nil)\ (insere\ a\ nil)$. Esta prova pode ser feita como a seguir:

$$\dfrac{\dfrac{\dfrac{}{permutation\ nil\ nil}\ (permutation\_nil)}{permutation\ (a :: nil)\ (a :: nil)}\ (permutation\_skip)}{permutation\ (a :: nil)\ (insere\ a\ nil)}\ (def.)$$

2. A segunda regra é ($permutation\_skip$), e para que a hipótese ($permutation\ l\ l'$) tenha sido gerada por esta regra é preciso que as listas l e l' tenha a mesma cabeça. Assim, considerando que l (resp. l') tenha a forma $h :: tl$ (resp. $h :: tl'$) temos como hipótese $permutation\ tl\ tl'$ (que corresponde ao antecedente da regra neste caso), e temos que provar $permutation\ (a :: h :: tl)\ (insere\ a\ (h :: tl'))$. Adicionalmente, temos como hipótese de indução $permutation\ (a :: tl)\ (insere\ a\ tl')$. Iniciamos a prova aplicando a definição de *insere* que divide a prova em dois subcasos: o da esquerda se dá quando $a \leq h$, e o da direita quando $a > h$.

$$\dfrac{(permutation\_skip)\,\dfrac{(permutation\_skip)\,\dfrac{(hip.)\,\dfrac{}{permutation\ tl\ tl'}}{permutation\ (h :: tl)\ (h :: tl')}}{permutation\ (a :: h :: tl)\ (a :: h :: tl')} \qquad (*)}{permutation\ (a :: h :: tl)\ (insere\ a\ (h :: tl'))}\,(def.)$$

No caso em que $a > h$ usamos a regra da transitividade com a lista $(h :: a :: tl)$ para poder permutar a e h no primeiro argumento de *permutation* (lista da esquerda), e então poder aplicar ($permutation\_skip$) para concluir com a hipótese de indução.

$$\dfrac{\dfrac{\dfrac{}{permutation\ (a :: h :: tl)\ (h :: a :: tl)}\,\Delta \quad \dfrac{\dfrac{}{permutation\ (a :: tl)\ (insere\ a\ tl')}\,(h.i.)}{permutation\ (h :: a :: tl)\ (h :: (insere\ a\ tl'))}\,\clubsuit}{permutation\ (a :: h :: tl)\ (h :: (insere\ a\ tl'))}\,\nabla}{(*)}$$

onde

- Δ corresponde à regra ($permutation\_swap$);
- $\clubsuit$ corresponde à regra ($permutation\_skip$);
- ∇ corresponde à regra ($permutation\_trans$).

3. A terceira regra é ($permutation\_swap$), e para que a hipótese ($permutation\ l\ l'$) tenha sido gerada por esta regra é preciso que as listas l e l' tenham a forma $x :: y :: tl$ e $y :: x :: tl$, respectivamente. Neste caso, precisamos provar $permutation\ (a :: x :: y :: tl)\ (insere\ a\ (y :: x :: tl))$. Note que não existe hipótese de indução neste caso porque a regra ($permutation\_swap$) (assim como a regra ($permutation\_nil$)) é um axioma. O ponto chave aqui é utilizar a transitividade de *permutation* com a lista $(a :: y :: x :: tl)$:

$$\dfrac{\clubsuit\,\dfrac{\Delta\,\dfrac{}{permutation\ (x :: y : tl)\ (y :: x :: tl)}}{permutation\ (a :: x :: y : tl)\ (a :: y :: x :: tl)} \qquad (*)}{permutation\ (a :: x :: y :: tl)\ (insere\ a\ (y :: x :: tl))}\,\nabla$$

onde

- Δ corresponde à regra ($permutation\_swap$);
- $\clubsuit$ corresponde à regra ($permutation\_skip$);
- ∇ corresponde à regra ($permutation\_trans$).

e o ramo da direita é como a seguir:

$$\dfrac{\dfrac{?}{\mathit{permutation}\ (a :: y :: x :: tl)\ (\mathit{insere}\ a\ (y :: x :: tl))}}{(*)}$$

Este ponto da prova é semelhante ao que ocorreu no caso 2, e foi resolvido com a hipótese de indução. Mas neste caso não temos hipótese de indução, já que a regra *permutation_swap* é um axioma! Nossa alternativa será utilizar um novo resultado auxiliar:

Lema 60. *Seja* x *um número natural, e* l *uma lista de naturais. Então* $\mathit{permutation}\ (x :: l)\ (\mathit{insere}\ x\ l)$.

Este lema nos permite fechar o ramo de prova atual de forma imediata. Ele pode ser provado por indução na estrutura da lista l, e será deixado como exercício.

$$\dfrac{\dfrac{}{\mathit{permutation}\ (a :: y :: x :: tl)\ (\mathit{insere}\ a\ (y :: x :: tl))}\ (\mathit{Lema}\ 72)}{(*)}$$

4. A quarta e última regra é ($\mathit{permutation\_trans}$), e considerando que a hipótese ($\mathit{permutation}\ l\ l'$) tenha sido gerada por esta regra, temos por hipótese que ($\mathit{permutation}\ l\ l0$) e ($\mathit{permutation}\ l0\ l'$) para alguma lista $l0$. Além disto, temos duas hipóteses de indução:

   (a) Se $\mathit{permutation}\ l\ l0$ então $\mathit{permutation}\ (a :: l)\ (\mathit{insere}\ a\ l0)$;

   (b) Se $\mathit{permutation}\ l0\ l'$ então $\mathit{permutation}\ (a :: l0)\ (\mathit{insere}\ a\ l')$.

   A prova de $\mathit{permutation}\ (a :: l)\ (\mathit{insere}\ a\ l')$ é como a seguir:

$$\dfrac{\clubsuit\ \dfrac{(\mathit{hip.})\ \dfrac{}{\mathit{permutation}\ l\ l0}}{\mathit{permutation}\ (a :: l)\ (a :: l0)} \qquad \dfrac{(**)}{\mathit{permutation}\ (a :: l0)\ (\mathit{insere}\ a\ l')}}{\mathit{permutation}\ (a :: l)\ (\mathit{insere}\ a\ l')}\ \nabla$$

   onde

   - $\clubsuit$ corresponde à regra ($\mathit{permutation\_skip}$);
   - ∇ corresponde à regra ($\mathit{permutation\_trans}$).

E o ramo da direita é concluído com a hipótese de indução:

$$\dfrac{(\mathit{hip.})\ \dfrac{}{\mathit{permutation}\ l0\ l'} \qquad \dfrac{}{\mathit{permutation}\ l0\ l' \rightarrow \mathit{permutation}\ (a :: l0)\ (\mathit{insere}\ a\ l')}\ (\mathit{h.i.})}{(**)}\ (\rightarrow_e)$$

□

Exercício 116. *Prove o Lema 60 em papel e lápis, e em seguida reproduza a sua prova no Coq.*

```
Lemma permutation_insere: forall l a, permutation (a :: l) (insere a l).
```

Exercício 117. *Prove o Lema 59 no Coq.*

```
Lemma permutation_insere_diff: forall l l' a, permutation l l' ->
                               permutation (a :: l) (insere a l').
```

Exercício 118. *Mostre que o predicado permutatio da Definição 57 é uma relação de equivalência sobre a estrutura de listas, isto é, mostre:*

1. *permutation l l, para qualquer lista l (reflexividade)*
2. *Se permutation l l' então permutation l' l, quaisquer que sejam as listas l e l' (simetria)*
3. *Se permutation l l' e permutation l' l'' então permutation l l'', quaisquer que sejam as listas l, l' e l'' (transitividade)*
4. *Refaça suas provas no Coq:*

```
Lemma permutation_refl: forall l, permutation l l.

Lemma permutation_sym: forall l l', permutation l l' -> permutation l' l.

Lemma permutation_trans: forall l l' l'', permutation l l' ->
                         permutation l' l'' -> permutation l l''.
```

Temos duas provas distintas de que o algoritmo de ordenação por inserção gera uma permutação da lista de entrada, mas veja que a prova foi muito mais simples e elegante com a Definição 57. Em geral, definições indutivas facilitam o processo de construção de provas porque podemos usar o princípio de indução para estas definições. Concluiremos esta seção com a prova de que as definições 55 e 57 são equivalentes, isto é, *perm* l l' se, e somente se, *permutation* l l' quaisquer que sejam as listas l e l'. Esta prova será dividida em duas etapas, isto é, em dois teoremas:

Teorema 61. *Sejam l e l' duas listas de números naturais. Se permutation l l' então perm l l'.*

Teorema 62. *Sejam l e l' duas listas de números naturais. Se perm l l' então permutation l l'.*

A prova do Teorema 61 segue a mesma estrutura da prova do Lema 59, isto é, indução na hipótese (*permutation* l l') e será deixado como exercício:

Exercício 119. *Prove o Teorema 61.*

Exercício 120. *Prove o Teorema 61 em Coq:*

```
Lemma permutation_to_perm: forall l l', permutation l l' -> perm l l'.
```

A prova do Teorema 62 é mais desafiadora porque a definição *perm* não é indutiva, e portanto, não podemos utilizar a mesma estratégia do lema anterior.

Demonstração. A prova é por indução na estrutura da lista `l`. Na base de indução, precisamos provar que, se $perm\ nil\ l'$ então $permutation\ nil\ l'$. A ideia é concluir da hipótese $perm\ nil\ l'$ que l' é a lista vazia, e daí, fecharmos este ramo da prova com a regra (*permutation_nil*). Para isto vamos utilizar o seguinte lema auxiliar, cuja prova é deixada como exercício:

Lema 63. *Seja l uma lista de números naturais. Se $perm\ nil\ l$ então $l = nil$.*

Exercício 121. *Prove o Lema 63, e em seguida refaça esta prova em Coq.*

```
Lemma perm_nil: forall l, perm nil l -> l = nil.
```

No passo indutivo, vamos supor que l tem a forma $(h :: tl)$. Então precisamos provar que, se $perm\ (h :: tl)\ l'$ então $permutation\ (h :: tl)\ l'$. Como l' é uma lista arbitrária, precisamos analisar sua estrutura. Se l' for a lista vazia então a hipótese $perm\ (h :: tl)\ nil$ corresponde ao absurdo, e concluímos este ramo da prova já que podemos provar qualquer coisa a partir do absurdo (regra da explosão). Agora suponha que l' tem a forma $h' :: tl'$. Então temos que provar que, se $perm\ (h :: tl)\ (h' :: tl')$ então $permutation\ (h :: tl)\ (h' :: tl')$, e como hipótese de indução temos que se $(perm\ tl\ l0)$ então $(permutation\ tl\ l0)$, qualquer que seja a lista $l0$. Agora podemos comparar h e h', pois se eles forem iguais então podemos concluir este ramo da prova com a regra (*permutation_skip*) e com a hipótese de indução. Se $h \neq h'$ então, pela hipótese $perm\ (h :: tl)\ (h' :: tl')$, sabemos que h ocorre na lista tl', ou seja, existem listas $l1$ e $l2$ tais que $tl' = l1 + +(h :: l2)$, onde ++ representa a operação de concatenação de listas. Podemos usar esta igualdade para substituir tl' na implicação que temos que provar: $perm\ (h :: tl)\ (h' :: l1 + +(h :: l2))$ então $permutation\ (h :: tl)\ (h' :: l1 + +(h :: l2))$. Agora podemos remover h destas listas, e concluir a prova utilizando a hipótese de indução.

□

Repetir esta prova em Coq exige alguns detalhes adicionais que aparecem nos exercícios a seguir. Por exemplo, o lema `perm_cons_num_oc` do exercício a seguir nos fornece uma forma de dizer que `n` ocorre na lista `l'`:

Exercício 122.
```
Lemma perm_cons_num_oc: forall n l l', perm (n :: l) l' ->
                             exists x, num_oc  n l' = S x.
```

O exercício a seguir, nos permite reescrever a lista `l` sabendo que o elemento `x` ocorre pelo menos uma vez em `l`:

Exercício 123.
```
Lemma num_occ_cons: forall l x n, num_oc x l = S n ->
exists l1 l2, l = l1 ++ x :: l2 /\ num_oc x (l1 ++ l2) = n.
```

A reorganização de elementos em uma lista pode ser feita com um lema como o do exercício a seguir:

Exercício 124.
```
Lemma permutation_app_cons: forall l1 l2 a,
permutation (a :: l1 ++ l2) (l1 ++ a :: l2).
```

Utilizando estes exercícios como dica, refaça a prova do Teorema 62 em Coq:

Exercício 125.
```
Theorem perm_to_permutation: forall l l', perm l l' ->
                                       permutation l l'.
```

A complexidade do algoritmo de ordenação por inserção

Em algoritmos de ordenação sobre listas, o tamanho da entrada consiste no tamanho da lista a ser ordenada. Vamos iniciar nossa análise com a função *insere x l* (Definição 49). A operação básica no caso do algoritmo de ordenação por inserção é a comparação entre chaves. Note que quando l é a lista vazia, a lista unitária $x :: nil$ é retornada, e nenhuma comparação é feita. Quando l é uma lista da forma $h :: tl$ então comparamos x com h, e quando $x \leq h$ a lista $x :: h :: tl$ é retornada e o algoritmo termina. Por outro lado, se $x > h$, o algoritmo continua buscando recursivamente a posição correta para inserir x. Denotaremos por $T_{insere}\ x\ l$ a função que computa o número de operações básicas realizadas pela função *insere* para inserir o elemento x na lista l:

$$T_{insere}\ x\ l = \begin{cases} 0, & \text{se } l = nil \\ 1, & \text{se } l = h :: tl \text{ e } x \leq h \\ 1 + (T_{insere}\ x\ tl), & \text{se } l = h :: tl \text{ e } x > h \end{cases}$$

Normalmente estamos interessados na análise do pior caso, e a função acima não computa necessariamente o número de comparações do pior caso para inserir um elemento qualquer em uma lista com n elementos. De fato, considerando $n = 3$, temos que $T_{insere}\ 1\ 2 :: 3 :: 4 :: nil = 1$, enquanto que $T_{insere}\ 4\ 1 :: 2 :: 3 :: nil = 3$. Assim, para inserirmos um elemento em uma lista com n elementos, no pior caso, precisamos comparar o elemento a ser inserido com todos os elementos da lista. A função $T^w_{insere}(n)$ modela esta situação ao receber o natural n como argumento (que corresponde ao tamanho da lista a ser ordenada) e faz o número máximo de comparações possíveis:

$$T^w_{insere}(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1 + T^w_{insere}(n-1), & \text{se } n > 0 \end{cases}$$

Assim, se a lista não possui elementos, nenhuma comparação é feita ($n = 0$), e caso contrário, uma comparação é feita para cada elemento da lista ($n > 0$). Observe que o número de comparações feitas pela função $T^w_{insere}(n)$ não pode ser maior do que o tamanho da lista:

Exercício 126. *Prove que* $T^w_{insere}(n) = n$*, para todo* n*.*

Assim, a relação entre as funções T_{insere} e T^w_{insere} é dada pelo lema a seguir:

Exercício 127. *Sejam* x *um número natural, e* l *uma lista de números naturais. Prove que* $T_{insere}\ x\ l \leq T^w_{insere}(|l|)$*, onde* $|l|$ *denota o tamanho da lista* l*.*

Os dois últimos exercícios nos permitem concluir que $T_{insere}\ x\ l \leq |l|$, ou seja, que a função *insere* tem complexidade linear. Vamos formalizar este resultado em Coq. A função recursiva *T_insere* é definida por:

```
Fixpoint T_insere (x: nat) (l: list nat) : nat :=
 match l with
 | nil => 0
 | h :: tl => if (x <=? h) then 1 else S (T_insere x tl)
 end.
```

Como exercício, prove que a função `T_insere` tem complexidade linear:

Exercício 128. `Lemma T_insere_linear: forall l x,  T_insere x l <= length l.`

Qual é o número de comparações realizadas pelo algoritmo de ordenação por inserção, isto é, pela função *ord_insercao*, para ordenar uma lista l? Vamos denotar por $T_{is}()$ a função que faz esta contagem. Se l for a lista vazia então nenhuma comparação é feita, ou seja, $T_{is}(nil) = 0$. Se $l = h :: tl$ então é feita uma chamada à função *insere*, além da chamada recursiva à função *ord_insercao*:

$$T_{is}(l) = \begin{cases} 0, & \text{se } l = nil \\ T_{is}(tl) + T_{insere}\ h\ (ord\_insercao\ tl), & \text{se } l = h :: tl \end{cases}$$

Observe que, $T_{is}(1 :: 2 :: 3 :: nil) = 2$, $T_{is}(3 :: 2 :: 1 :: nil) = 3$, $T_{is}(1 :: 2 :: 3 :: 4 :: nil) = 3$ e $T_{is}(4 :: 3 :: 2 :: 1 :: nil) = 6$, etc. Portanto o número de comparações pode ser diferente para listas de mesmo tamanho, o que é esperado pelas chamadas feitas à função *insere*. Como então definir a função $T^w_{is}(n)$ que nos dá um limite superior para o número de comparações feitas pelo algoritmo de ordenação por inserção para uma lista qualquer de tamanho n. Em outras palavras, qual a complexidade do pior caso para o algoritmo de ordenação por inserção? Sabemos que quando $n = 0$, nenhuma comparação é feita. Quando $n > 0$, o algoritmo é aplicado recursivamente na cauda da lista, isto é, em uma lista de tamanho $n - 1$, e é feita uma chamada à função *insere* cuja complexidade já conhecemos. Isto nos permite escrever a função $T^w_{is}(n)$ como a seguir:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + T^w_{insere}(n-1), & \text{se } n > 0 \end{cases}$$

que pode ser simplificada como a seguir, já que $T^w_{insere}(n) = n$:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + (n-1), & \text{se } n > 0 \end{cases}$$

Podemos usar o método da substituição para encontrarmos uma solução para esta recorrência, e em seguida utilizar indução para verificarmos se a solução está correta. Pelo método da substituição, podemos ir aplicando a definição da recorrência, assumindo que $n > 0$:

$$\begin{aligned} T^w_{is}(n) &= T^w_{is}(n-1) + (n-1) \\ &= T^w_{is}(n-2) + (n-2) + (n-1) \\ &= T^w_{is}(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \end{aligned}$$

Podemos continuar este processo de substituição até chegarmos em $T^w_{is}(1)$ que é igual a 0:

$$\begin{aligned} T^w_{is}(n) &= T^w_{is}(n-1) + (n-1) \\ &= T^w_{is}(n-2) + (n-2) + (n-1) \\ &= T^w_{is}(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \\ &= T^w_{is}(1) + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \end{aligned}$$

Para finalizar, precisamos utilizar indução em n para provar que $T^w_{is}(n) = \frac{n(n-1)}{2}$. Se $n = 0$, o resultado é trivial. Se $n > 0$ então, por definição, $T^w_{is}(n) = T^w_{is}(n-1) + (n-1)$. A hipótese de indução, nos dá que $T^w_{is}(n-1) = \frac{(n-1)(n-2)}{2}$, e portanto, $T^w_{is}(n) = T^w_{is}(n-1) + (n-1) \overset{h.i.}{=} \frac{(n-1)(n-2)}{2} + (n-1) = \frac{n(n-1)}{2}$.

Agora prove este lema em Coq:

Exercício 129.

```
Fixpoint T_is_w (n: nat) : nat :=
 match n with
 | 0 => 0
 | S k => (T_is_w k) + (T_insere_w k)
 end.
```

```
Lemma T_ord_insercao_w_teste: forall n, T_is_w (S n) = n * (S n)/2.
```

Nossa conclusão, portanto, é que o algoritmo de ordenação por inserção recursivo é correto, e sua complexidade no pior caso é quadrática, assim como na versão não-recursiva. Na próxima seção estudaremos um algoritmo mais eficiente do que a ordenação por inserção, mas antes vejamos um exercício clássico envolvendo recursão: a função de Fibonacci.

Exercício 130. *A função de Fibonacci é definida pela relação de recorrência*

$$\begin{aligned} &F(0) = 0 \\ &F(1) = 1 \\ &F(n) = F(n-1) + F(n-2) \end{aligned}$$

Seja $\mathcal{F}$ uma função geradora definida por

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F(i) z^i$$

Siga os seguintes passos para resolver a relação de recorrência da função de Fibonacci.

1. *Demonstre que*

   $$\mathcal{F}(z) = \frac{z}{1 - z - z^2} = \frac{z}{(1 - \phi z)(1 - \tilde{\phi} z)} = \frac{1}{\sqrt{5}}(\frac{1}{1 - \phi z} - \frac{1}{1 - \tilde{\phi} z}),$$

   onde

   $$\phi = \frac{1 + \sqrt{5}}{2} \quad e \quad \tilde{\phi} = \frac{1 - \sqrt{5}}{2}$$

2. *Demonstre que*

   $$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \tilde{\phi}^i) z^i$$

3. *Prove que $F(i) = \phi^i / \sqrt{5}$ para $i > 0$, aproximado ao inteiro mais próximo.*

4. *Demonstre que $F(i + 2) \geq \phi^i$, para $i \geq 0$.*

5. *Conclua que $F(n) = \Theta(\phi^n)$; i.e., a função de Fibonacci é de complexidade exponencial.*

3.2.4 O algoritmo *mergesort* (versão com *merge* iterativo)

Algoritmos recursivos desempenham um papel fundamental em Computação. O algoritmo de ordenação *mergesort* é um exemplo de algoritmo recursivo, que se caracteriza por dividir o problema original em subproblemas que, por sua vez, são resolvidos recursivamente. As soluções dos subproblema são então combinadas para gerar uma solução para o problema original. Este paradigma de projeto de algoritmo é conhecido com *divisão e conquista.* Este algoritmo foi inventado por J. von Newmann em 1945.

O algoritmo *mergesort* é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, que consiste das seguintes etapas:

1. **Divisão**: O algoritmo divide a lista (ou vetor) l recebida como argumento ao meio, obtendo as listas l_1 e l_2;

2. **Conquista**: O algoritmo é aplicado recursivamente às listas l_1 e l_2 gerando, respectivamente, as listas ordenadas l'_1 e l'_2;

3. **Combinação**: O algoritmo combina as listas l'_1 e l'_2 através da função *merge* que então gera a saída do algoritmo.

Por exemplo, ao receber a lista $(4 :: 2 :: 1 :: 3 :: nil)$, este algoritmo inicialmente divide esta lista em duas sublistas, a saber $(4 :: 2 :: nil)$ e $(1 :: 3 :: nil)$. O algoritmo é aplicado recursivamente às duas sublistas para ordená-las, e ao final deste processo, teremos duas listas ordenadas $(2 :: 4 :: nil)$ e $(1 :: 3 :: nil)$. Estas listas são, então, combinadas para gerar a lista de saída $(1 :: 2 :: 3 :: 4 :: nil)$.

```
if $p < r$ then
    $q = \lfloor \frac{p+r}{2} \rfloor$;
    mergesort$(A, p, q)$;
    mergesort$(A, q + 1, r)$;
    merge$(A, p, q, r)$;
end
```

Algoritmo 8: mergesort(A, p, r)

A etapa de combinar dois vetores ordenados (algoritmo *merge*) é a etapa principal do algoritmo *mergesort*. O procedimento *merge*(A, p, q, r) descrito a seguir recebe como argumentos o vetor A, e os índices p, q e r tais que $p \leq q < r$. O procedimento assume que os subvetores $A[p..q]$ e $A[q+1..r]$ estão ordenados.

```
$n_1 = q - p + 1$ ;                                      // Qtd. de elementos em $A[p..q]$
$n_2 = r - q$ ;                                        // Qtd. de elementos em $A[q+1..r]$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays;
for $i = 1$ to $n_1$ do
    $L[i] = A[p + i - 1]$;
end
for $j = 1$ to $n_2$ do
    $R[j] = A[q + j]$;
end
$L[n_1 + 1] = \infty$;
$R[n_2 + 1] = \infty$;
$i = 1$;
$j = 1$;
for $k = p$ to $r$ do
    if $L[i] \leq R[j]$ then
        $A[k] = L[i]$;
        $i = i + 1$;
    end
    else
        $A[k] = R[j]$;
        $j = j + 1$;
    end
end
```

Algoritmo 9: merge(A, p, q, r)

Exercício 131. *Prove que o algoritmo merge é correto.*

Exercício 132. *Prove que o algoritmo mergesort é correto.*

Exercício 133. *Faça a análise assintótica do algoritmo merge.*

Exercício 134. *Faça a análise assintótica do algoritmo mergesort.*

3.2.5 O algoritmo *mergesort* (versão com *merge* recursivo)

A seguir apresentamos uma descrição do algoritmo *mergesort* diretamente em Coq:

```
Function mergesort (l: list nat) {measure length l}:=
 match l with
 | nil => nil
 | h::nil => l
 | h::tl =>
   let n := length(l) / 2 in
    let l1 := firstn n l in
    let l2 := skipn n l in
    let sorted_l1 := mergesort(l1) in
    let sorted_l2 := mergesort(l2) in
    merge (sorted_l1, sorted_l2)
 end.
```

A definição é baseada na estrutura da lista `l`, de forma que se `l` for a lista vazia ou uma lista unitária, o algoritmo retorna a própria lista `l`. Caso contrário `l` é dividida nas listas `l1` (contendo os elementos da primeira metade de `l`), e `l2` (contendo os elementos restantes de `l`). Recursivamente, as listas `l1` e `l2` são ordenada para depois serem combinadas pela função `merge`:

```
Function merge (p: list nat * list nat) {measure len p} :=
 match p with
 | (nil, l2) => l2
 | (l1, nil) => l1
 | ((hd1 :: tl1) as l1, (hd2 :: tl2) as l2) =>
 if hd1 <=? hd2 then hd1 :: merge (tl1, l2)
 else hd2 :: merge (l1, tl2)
end.
```

A função `merge` recebe um par de listas ordenadas, e recursivamente gera uma lista ordenada contendo todos os elementos das listas dadas como argumento.

Exercício 135. *Prove que o algoritmo merge é correto.*

Exercício 136. *Prove que o algoritmo mergesort é correto.*

Exercício 137. *Faça a análise assintótica do algoritmo merge.*

Exercício 138. *Faça a análise assintótica do algoritmo mergesort.*

Agora provaremos a correção do algoritmo `mergesort` no Coq. Inicialmente mostrar que a função `merge` retorna uma lista ordenada, caso cada uma das listas do par dado como argumento também estejam ordenadas:

Exercício 139.
```
Lemma merge_sorts: forall p, (sorted (fst p) /\ sorted (snd p)) ->
         sorted (merge p).
```

Em seguida, podemos provar que a função `mergesort` efetivamente ordena e que gera uma permutação de qualquer lista recebida como argumento:

Exercício 140. `Theorem mergesort_sorts: forall l, sorted (mergesort l).`

Exercício 141. `Theorem mergesort_is_perm: forall l, perm l (mergesort l).`

Observe que podemos utilizar tanto `perm` como `permutation` no exercício anterior, já que estas definições são equivalentes. Feito isto, temos o teorema da correção do algoritmo `mergesort`:

```
Theorem mergesort_is_correct: forall l, perm l (mergesort l) /\ sorted (mergesort l).
Proof.
intro. split.
 - apply mergesort_is_perm.
 - apply mergesort_sorts.
Qed.
```

Assim, como fizemos para o algoritmo de ordenação por inserção, analisaremos a complexidade do algoritmo `mergesort` considerando o número de comparações realizadas pelo algoritmo durante o processo de ordenação. Para a função `merge`, chamaremos de `T_merge` a função que faz esta contagem:

```
Function T_merge (p: list nat * list nat) {measure len p} : nat :=
 match p with
 | (nil, l2) => 0
 | (l1, nil) => 0
 | ((hd1 :: tl1) as l1, (hd2 :: tl2) as l2) =>
   if hd1 <=? hd2 then S(T_merge (tl1, l2))
   else S(T_merge (l1, tl2))
 end.
```

Quando alguma das listas do par é a lista vazia, nenhuma comparação é feita, e portanto a função `T_merge` retorna 0. Caso contrário, o contador é incrementado e uma nova chamada de `T_merge` é feita. No exercício a seguir, prove que a função `merge` tem complexidade linear:

Exercício 142. `Lemma T_merge_is_linear: forall l1 l2,`
`T_merge (l1,l2) <= (length l1 + length l2).`

Agora vamos contar o número de comparações feitas pela função `mergesort`. Quando a lista `l` tem no máximo um elemento, nenhuma comparação é feita. Quando `l` tem pelo menos dois elementos, a lista é divida em duas listas `l1` e `l2` e recursivamente contamos as comparações necessárias para ordená-las e também o número de comparações necessárias para juntar as versões ordenadas de `l1` e `l2`. Esta contagem está implementada na função `T_mergesort` a seguir:

```
Function T_mergesort (l: list nat) {measure length l} : nat :=
 match l with
 | nil => 0
 | hd :: nil => 0
 | hd :: tl =>
   let n := length(l) / 2 in
   let l1 := firstn n l in
   let l2 := skipn n l in
   T_mergesort(l1) + T_mergesort(l2) + T_merge (mergesort l1, mergesort l2)
 end.
```

Por fim, resolva o exercício a seguir que mostra que a complexidade do algoritmo `mergesort` é $O(\log_2 n)$, onde n é o tamanho da lista a ser ordenada.

Exercício 143. `Theorem T_mergesort_complexity: forall l k,`
`length l = 2^k -> T_mergesort l <= k * 2^k.`

3.2.6 Equações de recorrência

Nesta seção estudaremos as equações de recorrência utilizadas no paradigma de divisão de conquista [20]:

Definição 64. *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* eventualmente não-decrescente *se existir um número inteiro n_0 tal que $f(n)$ é não-decrescente no intervalo $[n_0, \infty)$, ou seja,*

$$f(n_1) \leq f(n_2), \forall n_2 > n_1 \geq n_0.$$

Definição 65. *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* suave *se for eventualmente não-decrescente e*

$$f(2.n) = \Theta(f(n))$$

Teorema 66. *Sejam $f(n)$ uma função suave, e c e n_0 constantes positivas. Se $f(2n) \leq c.f(n), \forall n \geq n_0$ então $f(2^k n) \leq c^k.f(n), \forall n \geq n_0$ e $k \geq 1$.*

Teorema 67. *Seja $f(n)$ uma função suave. Então para qualquer $b \geq 2$ fixado,*

$$f(b.n) = \Theta(f(n))$$

O teorema a seguir é conhecido como *regra da suavização*

Teorema 68. *Seja $T(n)$ uma função eventualmente não-decrescente, e $f(n)$ uma função suave. Se $T(n) = \Theta(f(n))$ para valores de n que são potências de b ($b \geq 2$), então*

$$T(n) = \Theta(f(n)), \forall n.$$

A regra da suavização nos permite expandir a informação sobre a ordem de crescimento estabelecida para $T(n)$ de um subconjunto de valores (potências de b) para o domínio inteiro. O teorema a seguir é um resultado muito útil nesta direção conhecido como *teorema mestre*:

Teorema 69. *Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência*
$T(n) = a.T(n/b) + f(n), \quad$ *para* $n = b^k, k = 1, 2, 3, \ldots$
$T(1) = c$
onde $a \geq 1, b \geq 2$ e $c \geq 0$. Se $f(n) = \Theta(n^d)$, onde $d \geq 0$, então

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & se\ a > b^d \\ \Theta(n^d . \lg n), & se\ a = b^d \\ \Theta(n^d), & se\ a < b^d \end{cases}$$

Demonstração. Considere que $f(n) = n^d$. Aplicando o método da substituição para a recorrência do teorema, obtemos:

$$T(b^k) = a^k.[T(1) + \sum_{j=1}^{k} f(b^j)/a^j]$$

Como $a^k = a^{\log_b n} = n^{\log_b a}$, podemos reescrever a equação acima como:

$$T(n) = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} f(b^j)/a^j]$$

e para $f(n) = n^d$, temos:

$$T(n) = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} (b^j)^d/a^j] = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} (b^d/a)^j]$$

A soma acima forma uma série geométrica, e portanto:

$$\sum_{j=1}^{\log_b n} (b^d/a)^j = (b^d/a)\frac{(b^d/a)^{\log_b n} - 1}{(b^d/a) - 1}, \text{ se } b^d \neq a.$$

Quando $b^d \neq a$, temos que $\sum_{j=1}^{\log_b n} (b^d/a)^j = \log_b n$. Agora basta analisarmos cada um dos casos: $a < b^d$, $a > b^d$ e $a = b^d$.

□

Apresentaremos agora uma versão um pouco mais geral do teorema mestre[8]. Consideraremos como anteriormente uma recorrência da forma:

$$T(n) = a.T(n/b) + f(n)$$

on $a \geq 1$ e $b > 1$ são constantes, e $f(n)$ é uma função assintoticamente positiva.

Teorema 70. *Sejam $a \geq 1$ e $b \geq 2$ constantes, $f(n)$ uma função assintoticamente positiva, e $T(n)$ definida nos inteiros não-negativos pela recorrência $T(n) = a.T(n/b) + f(n)$, onde n/b deve ser interpretado como $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então $T(n)$ tem as seguintes cotas assintóticas:*

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;*

2. *Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a}. \lg n)$;*

3. *Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$, então para todo n suficientemente grande, temos que $T(n) = \Theta(f(n))$.*

A prova será dividida em três lemas, onde inicialmente consideraremos que n é potência de b.

Lema 71. *Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função não-negativa definida para potências de b. Defina $T(n)$ para potências de b pela recorrência:*

$$T(n) = \begin{cases} \Theta(1), & \textit{se } n = 1; \\ a.T(n/b) + f(n), & \textit{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j.f(n/b^j).$$

Demonstração. Analise a árvore de recorrência da equação dada.

□

Em termos da árvore de recorrência, os três casos do teorema mestre correspondem aos casos onde o custo total da árvore é:

1. dominado pelo custo das folhas;

2. uniformemente distribuído ao longo da árvore;

3. dominado pelo custo da raiz.

Lema 72. *Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função não-negativa definida para potências de b. A função $g(n)$ definida para potências de b por:*

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j.f(n/b^j).$$

tem as seguintes cotas assintóticas para potências de b:

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $g(n) = O(n^{\log_b a})$;*

2. *Se $f(n) = \Theta(n^{\log_b a})$, então $g(n) = \Theta(n^{\log_b a}.\lg n)$;*

3. *Se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então $g(n) = \Theta(f(n))$.*

Demonstração. Exercício.

□

Exercício 144. *Resolva as seguintes relações de recorrência:*

1. $T(1) = 1,\ T(n) = 3T(n/2) + n^2,\ n \geq 2$
2. $T(1) = 1,\ T(n) = 2T(n/2) + n,\ n \geq 2$
3. $T(1) \in \Theta(1),\ T(n) = 3T(n/3 + 5) + n/2$
4. $T(1) = 1,\ T(n) = 2T(n - 1) + 1,\ n \geq 2$
5. $T(1) \in \Theta(1),\ T(n) = 9T(n/3) + n$
6. $T(1) \in \Theta(1),\ T(n) = T(2n/3) + 1$
7. $T(1) \in \Theta(1),\ T(n) = 2T(n/4) + 1$
8. $T(1) \in \Theta(1),\ T(n) = 2T(n/4) + \sqrt{n}$
9. $T(1) \in \Theta(1),\ T(n) = 2T(n/4) + \sqrt{n}\lg^2 n$
10. $T(1) \in \Theta(1),\ T(n) = 2T(n/4) + n$
11. $T(1) \in \Theta(1),\ T(n) = 2T(n/4) + n^2$
12. $T(1) \in \Theta(1),\ T(n) = 3T(n/2) + nln(n)$
13. $T(1) \in \Theta(1),\ T(n) = 3T(n/4) + nln(n)$
14. $T(1) \in \Theta(1),\ T(n) = 2T(n/2) + nln(n)$
15. $T(1) \in \Theta(1),\ T(n) = 2T(n/2) + n/ln(n)$
16. $T(1) \in \Theta(1),\ T(n) = T(n - 1) + 1/n$
17. $T(1) \in \Theta(1),\ T(n) = T(n - 1) + ln(n)$
18. $T(1) \in \Theta(1),\ T(n) = \sqrt{n}T(\sqrt{n}) + n$
19. $T(n) = 8T(n/2) + \Theta(n^2)$
20. $T(n) = 8T(n/2) + \Theta(1)$
21. $T(n) = 7T(n/2) + \Theta(n^2)$

Índice Remissivo

Referências Bibliográficas

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science*, 239:15–27, January 2017.

[2] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2–es, December 2007.

[3] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, April 2014.

[4] M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs.* UTCS. Springer, 2017.

[5] A. Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2017.

[6] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[7] A. Church. A Formulation of the Simple Theory of Types. *journal of Symbolic Logic*, 5:56–68, 1940.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 4 edition, April 2022.

[9] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 625–635, Cham, 2021. Springer International Publishing.

[10] H. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[11] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2008.

[12] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics : A Foundation for Computer Science.* Addison-Wesley, 2004.

[13] T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *ArXiv e-prints*, January 2015.

[14] Thomas Hales. A Proof of the Kepler Conjecture. *Annals of Mathematics*, 162(3):1065–1185, 2005.

[15] J. Roger Hindley. *Basic Simple Type Theory.* Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[16] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, New York, NY, USA, 2004.

[17] Thomas Jech. *Set Theory.* Springer Monographs in Mathematics. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[18] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System Description: E.T. 0.1. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 389–398. Springer International Publishing, Cham, 2015.

[19] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107, 2009.

[20] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.

[21] W. McCune. Prover9 and mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005.

[22] C.A. Mortari. *Introdução à Lógica.* Ed. UNESP, 2001.

[23] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lncs.* Springer, 2002.

[24] R. B. Nogueira, A. C. A. Nascimento, F. L. C. de Moura, and M. Ayala-Rincón. Formalization of Security Proofs Using PVS in the Dolev-Yao Model. In *Booklet Proc. Computability in Europe - CiE*, 2010.

[25] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *CADE*, volume 607 of *Lnai*, pages 748–752. sv, 1992.

[26] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. 2014.

[27] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J Autom Reasoning*, 55(1):1–37, 2015.

[28] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catvalin Hriatcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations.* Electronic textbook, 2014.

[29] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[30] Raymond Smullyan. *Logical Labyrinths.* AK Peters, 2009.

[31] The Coq Development Team. The Coq Proof Assistant. Zenodo, October 2021.

[32] D. van Dalen. *Logic and Structure.* Universitext. Springer London, 2013.