Lógica Computacional e Algoritmos

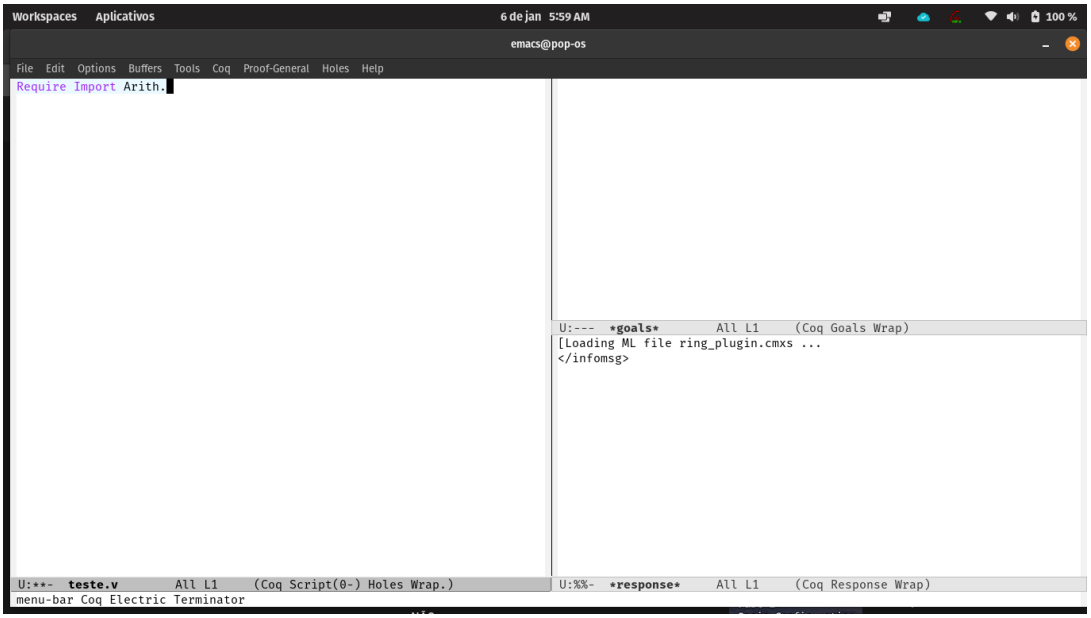
Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília[1]

19 de janeiro de 2022

[1]flaviomoura@unb.br

Notação

Utilizamos algumas notações para facilitar a identificação de diferentes contextos, principalmente no que se refere ao assistente de provas Coq[20]. Os códigos do Coq são escritos em `verbatim`, mas uma sessão típica do Coq possui três janelas:



A janela da esquerda é onde escrevemos as definições, lemas e as táticas de prova, a do canto superior direito, vai nos mostrando o *status* da prova a medida em que a prova é construída, e a do canto inferior direito, nos mostra mensagens do sistema. Os textos da janela da esquerda aparecem em `verbatim`, enquanto que os da janela superior direita, isto é, o *status* das provas aparecem em

```
verbatim e dentro de uma caixa para facilitar a identificação.
```

Assumimos que o Coq está instalado no seu computador.

Introdução

Este material está sendo desenvolvido para dar suporte aos alunos de graduação da Universidade de Brasília nas disciplinas de Lógica Computacional 1 e Projeto e Análise de Algoritmos. Normalmente, o público que cursa estas disciplinas pertence aos cursos de Computação, Matemática, Engenharias e áreas afins, mas acreditamos que este material seja útil para todos que tenham interesse no tema de lógica e algoritmos. O foco deste material está na construção de provas matemáticas tanto em papel e lápis (provas informais) quanto em computador (provas formais). Construção de provas é um tema que costuma ser espinhoso para os estudantes, de forma que aqui tentaremos explorar diversas situações para facilitar este processo. Provaremos propriedades dos números naturais, propriedades de algoritmos, etc. Novas atividades são incorporadas com regularidade, normalmente a cada novo semestre letivo, e portanto este material está em constante atualização.

Ao longo deste estudo utilizaremos diferentes linguagens para a construção de provas. 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). A LPO nos permitirá expressar propriedades de algoritmos de forma mais natural, mas suas provas serão mais complexas. 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 explicar o que queremos dizer com provas feitas por computador. Esta explicação é 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 [17, 8, 10]. Um assistente de provas por outro lado, consiste em um programa que requer a orientação 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 pelo usuário está correto. São exemplos de assistentes de prova o PVS[14], o Isabelle/HOL[12], o Lean[11] e o Coq[20]. 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, e até mesmo ser executado via browser[1].

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 diversos parâmetros distintos 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 possíveis. Caso alguma resposta esteja fora do esperado, 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 de inteiros, 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 em sistemas críticos (aviação, medicina, sistemas bancários, etc), por menores que sejam as chances, podem existir entradas específicas que não foram testadas, e falhas não são toleradas em sistemas críticos principalmente. O que fazer então para garantir que o programa é 100% correto? A resposta é utilizar a lógica para **provar** a correção do programa! Uma prova matemática 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[6, 5, 2, 3] e Computação[9, 15, 13]. Para concluir esta seção e começarmos a colocar a mão na massa, listamos a seguir 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.

A *Lógica Computacional* (LC) tem por objetivo utilizar a lógica para raciocinar sobre Computação, ou seja, consiste na utilização da lógica para a resolução de problemas computacionais. Este conceito pode ser utilizado de diversas formas, por exemplo, é comum a associação da LC com programação em lógica. Neste caso, a lógica é utilizada como uma linguagem de programação [19]. Uma outra abordagem possível é a simples mecanização do raciocínio lógico, de forma a permitir a resolução de exercícios de lógica no computador, ao invés da resolução usual em papel e lápis [7]. A abordagem que utilizaremos difere das anteriores, mas possui um pouco de cada uma delas como veremos a seguir.

Para explicar a nossa abordagem, suponha que você tenha um grande banco de dados com informações de uma determinada população, e que por alguma razão precise ordenar estas informações por idade em determinado momento; em outro, a ordenação que precisa ser feita é por nome ou outro critério qualquer. O que você faz? Uma alternativa é utilizar alguma implementação já feita e resolver o problema. Uma pergunta que pode ser feita é: será que a implementação utilizada gera a resposta correta? Outra alternativa seria construir/implementar um algoritmo de ordenação, mas a questão sobre a correção ainda continua válida: a implementação construída é correta? A abordagem que utilizamos neste curso fornece as ferramentas necessárias para responder a estas perguntas. Em particular, estudaremos sistemas dedutivos que nos permitirão provar propriedades de programas[4].

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"[18]. O problema deste argumento é que pessoas diferentes podem ter compreensões distintas sobre o argumento, de forma que o argumento seja uma prova para uma, 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.

Agradecimentos Este material foi escrito com o apoio do programa Aprendizagem para o Terceiro Milênio (A3M) coordenado pelo CEAD/UnB, que viabilizou o trabalho do estudante Rafael Monteiro Rodrigues na elaboração das soluções das atividades propostas. O estudante Gabriel Silva também fez contribuições importantes na elaboração das atividades, e em particular, na formalização do algoritmo *mergesort*.

A Lógica Proposicional

Iniciaremos nosso estudo com a *lógica proposicional* (LP), que é uma lógica baseada na noção de **proposição**. Uma proposição, por sua vez, é simplesmente uma sentença que pode ser qualificada como verdadeira ou falsa. São exemplos de proposição:

- 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!"não pode ser qualificada como verdadeira ou falsa, e portanto não é uma proposição. Algumas proposições podem ser divididas em proposições menores. Por exemplo, a proposição "João tem 20 anos e Maria tem 22 anos"é composta da proposição "João tem 20 anos"e da proposição "Maria tem 22 anos", que por sua vez não podem mais serem divididas porque os pedaços menores não são mais qualificáveis como verdadeiro ou falso. Uma proposição que não pode ser dividida é um elemento básico utilizado na construção de proposições mais complexas, que chamaremos de *fórmula atômica*. Utilizaremos letras latinas minúsculas para representar fórmulas atômicas. Por exemplo, podemos utilizar a letra q para representar a proposição "Maria tem 22 anos", e a letra p para "João tem 20 anos". A proposição do exemplo acima é construída com a utilização do conectivo "E"(conjunção), que será representado pelo símbolo $\wedge$. Com esta simbologia, podemos codificar a proposição "João tem 20 anos e Maria tem 22 anos"pela fórmula $p \wedge q$. Vejamos então a gramática utilizada na construção das fórmulas da LP, que serão representadas por letras gregas minúsculas:

$$\varphi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \tag{1}$$

A gramática (1) define como as fórmulas da LP são construídas. Ela possui 6 construtores:

1. O primeiro denota uma variável proposicional, e caracteriza uma fórmula atômica, i.e. uma fórmula que não pode ser subdividida em fórmulas menores.
2. O segundo construtor é uma constante que denota o absurdo ($\bot$), que também é uma fórmula atômica. O absurdo será utilizado quando tivermos informações contraditórias em nossas provas. Isto ficará mais claro nos exemplos.
3. O terceiro construtor denota a negação.

4. O quarto construtor denota a conjunção.

5. O quinto construtor denota a disjunção.

6. O sexto construtor é a implicação.

Uma gramática como (1) 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, e pode ser escrito em função dos outros conectivos: $\varphi \leftrightarrow \psi$ é o mesmo que $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. Na verdade, a gramática apresentada possui redundâncias, isto é, conectivos que podem ser escritos em função de outros, mas veremos isto posteriormente.

O sistema conhecido como *dedução natural* será utilizado para a construção das provas. 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. Como veremos, estas regras nos permitem derivar novos fatos a partir das premissas. Os fatos a serem provados são representados por meio de fórmulas da LP. Neste contexto, o primeiro conceito importante que aparece é o de *sequente*. Formalmente, um sequente é um par cujo primeiro elemento é um conjunto finito de fórmulas (hipóteses), e o segundo elemento é uma fórmula (conclusão). Assim, se $\varphi_1, \varphi_2, \ldots, \varphi_n$ são as hipóteses de um dado problema, e ψ é a conclusão, escrevemos $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ para representar o sequente que nos permite provar ψ a partir das hipóteses $\varphi_1, \varphi_2, \ldots, \varphi_n$. 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 será 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 não deixar a notação sobrecarregada. Assim, ao invés de $\Gamma \cup \{\varphi\} \vdash \psi$, escreveremos simplesmente $\Gamma, \varphi \vdash \psi$, onde Γ, φ deve então ser lido como o conjunto que contém a fórmula φ e todas as fórmulas de Γ. O conceito de prova agora será definido de forma mais precisa. Concretamente, uma prova de um sequente da forma $\Gamma \vdash \psi$ consiste em uma árvore cujos nós são anotados com sequentes, a raiz da árvore está anotada com o sequente que queremos provar, isto é, $\Gamma \vdash \psi$, e as folhas da árvore estão anotadas com axiomas. Um axioma é uma regra da forma:

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

Ou seja, um axioma é um sequente que tem como conclusão uma fórmula que está no contexto.

Como veremos, a construção desta árvore deve obedecer alguns critérios que detalharemos ao longo deste capítulo, mas em linhas gerais, o principal critério consiste em obedecer as regras que definem o sistema de dedução natural. As regras são divididas em dois tipos: *regras de introdução* e *regras de eliminação* dos conectivos. As regras de introdução são bastante intuitivas e, em certo sentido, podem ser vistas como uma definição do conectivo que estão introduzindo. Por exemplo, a primeira regra que veremos consiste na *regra de introdução da conjunção*, denotada por $(\wedge_i)$. Esta regra 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:

$$\frac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i) \tag{2}$$

Esta regra nos diz que 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 $\Gamma \vdash \varphi_2$.

Como exemplo de utilização desta regra, construiremos uma prova para o sequente $\varphi, \psi \vdash \varphi \wedge \psi$. Podemos aplicar a regra acima instanciando Γ, φ_1 e φ_2, respectivamente com o conjunto $\{\varphi, \psi\}$, e com

as fórmulas φ e ψ. Como resultado temos a árvore abaixo onde os dois ramos gerados por $(\wedge_i)$ são axiomas:

$$\dfrac{(\text{Ax})\ \dfrac{}{\varphi, \psi \vdash \varphi} \qquad \dfrac{}{\varphi, \psi \vdash \psi}\ (\text{Ax})}{\varphi, \psi \vdash \varphi \wedge \psi}\ (\wedge_i)$$

Apresentaremos as regras do sistema de dedução natural em paralelo com o assistente de provas Coq. Esta é uma forma de aprendermos a utilizar o sistema de uma forma progressiva e suave. O Coq implementa uma lógica de ordem superior baseado em dedução natural. Isto quer dizer que será possível fazer uma analogia entre o sistema de dedução natural que apresentamos aqui e o Coq, 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 de uma vez porque isto facilita o processo de construção de provas em sistemas mais complexos. Iniciaremos com a prova do exemplo anterior, que nos permitirá construir a prova de uma conjunção em Coq. Para simularmos a regra de introdução da conjunção 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 prova é construída entre as palavras reservadas `Proof` (que denota o início da prova) e `Qed` (que indica que a prova foi finalizada). Utilizamos a tática `split` para dividirmos a prova da conjunção em 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`:

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

Neste caso, temos uma correspondência direta entre uma regra do sistema de dedução natural e o Coq, mas este não é sempre o caso. .

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

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

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

$$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i \in \{1,2\}}}\ (\wedge_e) \tag{3}$$

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. Como vimos, as provas em Coq são construídas de baixo para cima, isto é, partimos da raiz da árvore de dedução que é o sequente que queremos provar, e subimos até as folhas que são os axiomas. Em provas simples conseguimos seguir este caminho sem dificuldades, mas em provas mais complexas precisamos ter mais flexibilidade. Em papel e lápis, as provas podem ser construídas tanto de baixo para cima (da raiz para as folhas) quanto de cima para baixo, e na prática usamos as duas estratégias porque dependendo do contexto, pode ser melhor uma estratégia ou outra. Veremos diversos exemplos para facilitar a compreensão desta ideia, e em Coq o mesmo acontece: 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 deste sequente, 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 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[1]. Existem outras táticas que podemos usar no lugar de `inversion` no exemplo anterior, como `destruct` e `elim`, mas elas não serão abordadas agora. No entanto, táticas diferentes podem ser usadas para construir provas diferentes de um mesmo sequente, assim como ocorre em papel e lápis.

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. Vamos construir esta prova passo a passo. A construção da prova é feita inicialmente de baixo para cima, isto é, da raiz para as folhas. Iniciamos observando que a raiz da nossa árvore de prova possui o sequente $\varphi \wedge \psi \vdash \psi \wedge \varphi$:

$$\frac{?}{\varphi \wedge \psi \vdash \psi \wedge \varphi}$$

Considerando as únicas regras que temos até o momento, a saber (2) e (3), é natural imaginarmos que a conclusão será obtida via a regra $(\wedge_i)$:

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

Agora podemos concluir com a regra de eliminação da conjunção e o axioma:

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

Exercício 1. *Utilize os seus conhecimentos de Coq para provar que a conjunção é comutativa.*

Exercício 2. *Em papel e lápis, prove que a conjunção é associativa, isto é, prove o sequente* $(\varphi \wedge \psi) \wedge \rho \vdash \varphi \wedge (\psi \wedge \rho)$. *Agora prove a associatividade da conjunção no Coq.*

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

Observe que no exercício anterior, solicitamos primeiro uma solução em papel e lápis para, somente depois, solicitar a prova no Coq. Este é um detalhe importante porque os assistentes de prova não são ferramentas para nos ajudar a construir provas, 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.

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 alguma das suas componentes:

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

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

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

As regras $(\vee_{i_1})$ e $(\vee_{i_2})$ são implementadas pelas táticas `left` e `right` do Coq, respectivamente. As árvores de prova abaixo mostram esta correspondência considerando o sequente `phi1` $\vdash$ `phi1` $\vee$ `phi2` para o caso $(\vee_{i_1})$, e `phi2` $\vdash$ `phi1` $\vee$ `phi2` para o caso $(\vee_{i_2})$:

$$\dfrac{\dfrac{}{\texttt{phi1} \vdash \texttt{phi1}}\ (\text{Ax})}{\texttt{phi1} \vdash \texttt{phi1} \vee \texttt{phi2}}\ (\vee_{i_1}) \qquad\qquad \dfrac{\dfrac{}{\texttt{phi1} \vdash \texttt{phi1}}\ \texttt{assumption}}{\texttt{phi1} \vdash \texttt{phi1} \vee \texttt{phi2}}\ \texttt{left}$$

$$\dfrac{\dfrac{}{\texttt{phi2} \vdash \texttt{phi2}}\ (\text{Ax})}{\texttt{phi2} \vdash \texttt{phi1} \vee \texttt{phi2}}\ (\vee_{i_2}) \qquad\qquad \dfrac{\dfrac{}{\texttt{phi2} \vdash \texttt{phi2}}\ \texttt{assumption}}{\texttt{phi2} \vdash \texttt{phi1} \vee \texttt{phi2}}\ \texttt{right}$$

A regra de eliminação da disjunção é um pouco mais sofisticada do que as que vimos até aqui. A ideia é que para provarmos algo, digamos γ, a partir de uma disjunção, precisamos provar γ a partir de cada uma das componentes da disjunção: :

$$\frac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\ (\vee_{\mathrm{e}})$$

Assim, para que tenhamos uma prova de γ a partir das fórmulas em Γ (sequente $\Gamma \vdash \gamma$) precisamos de uma prova de γ a partir de φ_1 e das fórmulas de Γ (sequente $\Gamma, \varphi_1 \vdash \gamma$) e de outra prova de γ a partir de φ_2 e das fórmulas de Γ (sequente $\Gamma, \varphi_2 \vdash \gamma$). Observe como os contextos mudam em cada um dos sequentes que compõem esta regra. Este é um detalhe importante que não ocorreu nas regras anteriores.

Exemplo 3. *Vamos mostrar que a disjunção é comutativa, ou seja, queremos construir uma prova para o sequente $\varphi \vee \psi \vdash \psi \vee \varphi$. A ideia aqui é utilizarmos a regra $(\vee_{\mathrm{e}})$. Para isto podemos instanciar Γ com o conjunto unitário contendo a fórmula $\varphi \vee \psi$. Em função da estrutura da regra $(\vee_{\mathrm{e}})$, precisamos construir duas provas distintas de $\psi \vee \varphi$: uma a partir de φ, e outra a partir de ψ. Podemos fazer isto com a*

ajuda da regra $(\vee_i)$*:*

$$\dfrac{(\text{Ax})\ \dfrac{}{\varphi \vee \psi \vdash \varphi \vee \psi} \qquad \dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}\,(\text{Ax})}{\varphi \vdash \psi \vee \varphi}\,(\vee_i) \quad \dfrac{\dfrac{}{\psi \vdash \psi}\,(\text{Ax})}{\psi \vdash \psi \vee \varphi}\,(\vee_i)}{\varphi \vee \psi \vdash \psi \vee \varphi}\,(\vee_e)$$

Em Coq, o sequente deste exemplo pode ser escrito declarando duas variáveis `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. A primeira nos pede para provar `psi \/ phi` a partir de `phi`:

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

que consiste em usar a tática `right` seguida de `assumption`, enquanto que na segunda subprova precisamos provar `psi \/ phi` a partir de `psi`:

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

que consiste em uma aplicação da tática `left` seguida de `assumption`.

A regra de *introdução da implicação*, denotada por $(\rightarrow_i)$, possui alguns detalhes importantes. Para construirmos uma prova de uma implicação, digamos do sequente $\Gamma \vdash \varphi \rightarrow \psi$, precisamos conseguir construir uma prova de ψ tendo φ como hipótese adicional ao contexto Γ. Em outras palavras, na leitura de baixo para cima, reduzimos o problema de $\Gamma \vdash \varphi \rightarrow \psi$ ao novo problema (possivelmente mais simples) de provar o sequente $\Gamma, \varphi \vdash \psi$:

$$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \rightarrow \psi}\,(\rightarrow_i)$$

Também podemos observar esta regra de cima para baixo. Neste caso, ela nos permite passar uma fórmula do conjunto de hipóteses para o consequente como antecedente de uma implicação. Assim, a fórmula φ que era uma das hipóteses necessárias para provar ψ, deixa de ser hipótese, e passa a ser antecedente de uma implicação no consequente. Posteriormente, esta regra será explorada em mais detalhes. Em Coq, esta regra é simulada por meio da tática `intro`.

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

A tática `intro` vai mover o antecedente `phi` da implicação para as hipóteses:

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

Portanto a tática `intro` corresponde a uma aplicação da regra de introdução da implicação. A regra de *eliminação da implicação* é a mais famosa das regras que veremos, a ponto de possuir um nome próprio, a saber *modus ponens*. Esta regra nos diz como podemos usar a prova de uma implicação juntamente com uma prova do antecedente desta implicação:

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

Quando lida de baixo para cima, esta regra corresponde a uma aplicação da regra do corte, que em Coq corresponde à tática `cut`. Assim, aplicando a tática `cut phi`, a prova se divide em dois ramos, ou em dois subobjetivos.

```
2 subgoals (ID 2)

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

subgoal 2 (ID 3) is:
 phi
```

Exemplo 4. *Considere o sequente* $\Gamma, \varphi \to \psi, \varphi \vdash \psi$. *Podemos prová-lo usando a regra* $(\to_e)$ *da seguinte forma:*

$$\frac{\dfrac{}{\Gamma, \varphi \to \psi, \varphi \vdash \varphi \to \psi}\ (\text{Ax}) \quad \dfrac{}{\Gamma, \varphi \to \psi, \varphi \vdash \varphi}\ (\text{Ax})}{\Gamma, \varphi \to \psi, \varphi \vdash \psi}\ (\to_e)$$

Agora faremos a prova acima em Coq considerando o conjunto Γ *como sendo o conjunto vazio. A declaração do sequente é feita como a seguir:*

```
Variables phi psi: Prop.

Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e: psi.
 Proof.
```

e o ambiente de prova corresponde ao sequente abaixo:

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

Agora podemos aplicar a tática `cut phi` *como explicado acima, e concluir a prova com* `assumption`*.*

```
Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e: psi.
 Proof.
   cut phi.
   - assumption.
   - assumption.
 Qed.
End imp_e.
```

Um outro caminho possível para provar este sequente é por meio da tática `apply H1` *seguida de* `assumption`*. Neste caso, o consequente da hipótese* `H1` *é confrontado com o objetivo a ser provado. Como eles coincidem, o novo objetivo gerado passa a ser* `phi`*, ou seja, o antecedente da hipótese* `H1`*.*

```
Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e2: psi.
 Proof.
  apply H1.
  assumption.
Qed.
End imp_e.
```

Por fim, podemos também usar a tática `apply` *nas hipóteses. Neste caso, o antecedente da hipótese* `H1` *é confrontado com a fórmula* `phi` *da hipótese* `H2`*. Como estas fórmulas coincidem, a hipótese* `H2` *é convertida na fórmula* `psi` *e podemos concluir a prova com* `assumption`*.*

```
Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e3: psi.
 Proof.
  apply H1 in H2.
  assumption.
 Qed.
End imp_e.
```

Exercício 5. *Prove que a disjunção é associativa, isto é, prove o sequente* $(\varphi \vee \psi) \vee \rho \vdash \varphi \vee (\psi \vee \rho)$*. Em seguida, prove a associatividade da disjunção no Coq.*

As regras para a negação são muito similares às regras da implicação, e isto não ocorre por acaso. De fato, uma negação é uma implicação particular porque $\neg\varphi$ é definida como $\varphi \to \bot$. Considerando este

	Dedução Natural	Tática Coq
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	`split`
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	`destruct`
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	`left`/`right`
4	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma, \varphi_1 \vee \varphi_2 \vdash \gamma}\ (\vee_e)$	`destruct`
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	`intro`
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	`cut/apply`
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	`intro`
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	`cut/apply`

Tabela 1: Regras da Lógica Minimal

fato, a analogia com as regras da implicação é direta.

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

A Tabela 1 apresenta as regras vistas até aqui, assim como algumas táticas do Coq associadas a estas regras. Estas regras formam a chamada *lógica proposicional minimal*.

Agora vamos resolver alguns exercícios na lógica minimal.

Exemplo 6. *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:*

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

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 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 utilizando a ideia do Exemplo 4:

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

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.

Exercício 7. *O símbolo da negação em Coq é* $\sim$*. Sabendo disto, refaça a prova acima no Coq.*

O sequente que acabamos de provar ocorre com certa frequência em outras provas, de forma que ele é comumente utilizado em outras provas assim como o Exemplo 4 foi utilizado aqui. As regras que são obtidas a partir das regras da Tabela 1 são chamadas de *regras derivadas*. Este é o caso da regra conhecida como *modus tollens* (MT):

$$\frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \neg\psi}{\Gamma \vdash \neg\varphi}\ (\mathrm{MT})$$

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:*

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

Exercício 9. *Reproduza a prova do exemplo anterior no Coq.*

O sequente que acabamos de provar é outro caso que aparece com frequência em provas, e corresponde a uma regra conhecida como *contrapositiva*:

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

Este é um bom momento para simplificarmos a notação que estamos usando, e tentaremos deixar clara a vantagem de nossa abordagem com a mudança de notação neste momento. Vamos retomar o Exercício 2 que consiste em provar que a conjunção é um conectivo que satisfaz a propriedade associativa. Neste ponto acreditamos que você já resolveu este exercício. Em caso negativo, resolva o exercício antes de prosseguir. Em seguida, compare sua solução com a que apresentamos a seguir, ok? Tentar resolver os exercícios antes de olhar qualquer solução é um passo muito importante para a sua evolução nos estudos de lógica. Considere a prova a seguir:

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

Observe que o contexto, isto é, o antecedente de cada um dos sequentes desta prova é o mesmo. De fato, o contexto em cada nó da árvore acima é o conjunto unitário contendo a fórmula $(\phi \wedge \psi) \wedge \varphi$. Como o que muda ao longo da prova é o consequente dos sequentes, é natural considerar que o foco, ou que

a parte principal, desta prova é o consequente de cada sequente. Sabendo com qual contexto estamos trabalhando, podemos removê-lo da prova deixando-a mais limpa e compacta. Veja como fica a prova sem os contextos:

$$\dfrac{\dfrac{\dfrac{(\phi \wedge \psi) \wedge \varphi}{(\phi \wedge \psi)}\,(\wedge_e)}{\phi}\,(\wedge_e) \qquad \dfrac{\dfrac{\dfrac{(\phi \wedge \psi) \wedge \varphi}{\phi \wedge \psi}\,(\wedge_e)}{\psi} \qquad \dfrac{(\phi \wedge \psi) \wedge \varphi}{\varphi}\,(\wedge_e)}{(\psi \wedge \varphi)}\,(\wedge_i)}{\phi \wedge (\psi \wedge \varphi)}\,(\wedge_i)$$

Será que é possível sempre remover os contextos das provas? Sim, e neste exemplo em particular, a situação é simples porque o contexto é o mesmo em toda a prova, mas este não será o caso em geral. Ainda considerando o exemplo anterior, se não soubéssemos qual o contexto que foi apagado, seria possível descobrí-lo? Sim, esta informação vem das folhas da árvore, que são as hipóteses do problema. Neste caso, a única hipótese é a fórmula $(\phi \wedge \psi) \wedge \varphi$ já que todas as folhas são iguais. Agora podemos consultar a Tabela 1 e observar que os contextos da regra $(\wedge_i)$ são os mesmos antes e depois de aplicar a regra. Portanto, o contexto das regras da segunda linha (de baixo para cima) também é o conjunto unitário contendo a fórmula $(\phi \wedge \psi) \wedge \varphi$. A mesma observação vale para a regra $(\wedge_e)$, e portanto o mesmo contexto é utilizado em toda a prova. Outro detalhe importante é que as folhas desta nova árvore não correspondem mais à regra (Ax), e portanto as folhas têm que ser fórmulas pertencentes ao contexto. As árvores onde os contexto ficam implícitos são as árvores normalmente apresentadas na bibliografia que trata de dedução natural.

No Exemplo 6 construímos uma prova do sequente $\varphi \to \psi, \neg\psi \vdash \neg\varphi$, e removendo os contextos obtemos a seguinte árvore de derivação:

$$\dfrac{\dfrac{\dfrac{\varphi \to \psi \qquad \varphi}{\psi} \qquad \neg\psi}{\bot}\,(\neg_e)}{\neg\varphi}\,(\neg_{\mathrm{i}})$$

Agora observe que as fórmulas $\varphi \to \psi$ e $\neg\psi$, que estão nas folhas da árvore, também são fórmulas do contexto. No entanto, a fórmula φ está em uma folha da árvore, mas não é uma fórmula do contexto, e portanto as coisas ficam um pouco mais interessantes aqui. Vamos novamente tentar reconstruir os contextos da raiz para as folhas da árvore. Esta árvore de derivação se refere ao sequente $\varphi \to \psi, \neg\psi \vdash \neg\varphi$, e portanto o contexto da fórmula que está na raiz da árvore é o conjunto $\{\varphi \to \psi, \neg\psi\}$. De acordo com a Tabela 1, a regra $(\neg_i)$ adiciona uma fórmula ao contexto, que neste caso corresponde à fórmula φ, e portanto o contexto da fórmula $\bot$ (segunda linha de baixo para cima) é o conjunto $\{\varphi \to \psi, \neg\psi, \varphi\}$. Como as regras $(\neg_e)$ e $(\to_e)$ não alteram o contexto, o conjunto $\{\varphi \to \psi, \neg\psi, \varphi\}$ também é o contexto das fórmulas que estão nas folhas desta árvore, e isto é o que permite a fórmula φ, que não faz parte do contexto original do problema, ser uma folha desta árvore. De fato, se o contexto fosse o mesmo em toda a árvore, a folha marcada com φ corresponderia ao sequente $\varphi \to \psi, \neg\psi \vdash \varphi$ que não corresponde a um axioma. Para diferenciarmos as fórmulas que fazem parte do contexto inicial, colocaremos as outras fórmulas entre colchetes para nos lembrarmos deste fato:

$$\dfrac{\dfrac{\dfrac{\varphi \to \psi \qquad [\varphi]}{\psi} \qquad \neg\psi}{\bot}\,(\neg_e)}{\neg\varphi}\,(\neg_{\mathrm{i}})$$

e agora fica claro que a fórmula φ não faz parte do contexto inicial. Note que os colchetes são colocados **apenas nas folhas** que contêm fórmulas que não fazem parte do contexto dado pelo problema. Adicionalmente, como o contexto da raiz tem que ser o contexto dado pelo problema, caso contrário a prova não é uma prova do problema proposto, precisamos de um mecanismo para nos informar quando as fórmulas marcadas com os colchetes são **removidas** (ou **descartadas**) do contexto. No exemplo acima,

isto ocorre ao aplicarmos a regra ($\neg_{\mathrm{i}}$). Então, utilizaremos uma letra para registrar este fato:

$$\dfrac{\dfrac{\dfrac{\varphi \to \psi \qquad [\varphi]^u}{\psi} \qquad \neg\psi}{\bot}\,(\neg_e)}{\neg\varphi}\,(\neg_i)\ u\ (\to_e)$$

Agora sabemos em que momento a fórmula φ foi introduzida, e em que momento foi descartada na árvore de derivação. Observe como o Coq faz este trabalho de modificar o contexto da mesma forma que acabamos de descrever:

```
Variables phi psi: Prop.

Section mt.
Hypothesis H1: phi -> psi.
Hypothesis H2: ~psi.
Lemma mt: ~phi.
Proof.
```

Ao iniciarmos a prova do lema `mt`, temos a seguinte configuração:

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

e aplicando a tática `intro`, a fórmula `phi` é introduzida no contexto com a marca `H`:

```
   H1 : phi -> psi
   H2 : ~ psi
   H : phi
   ============================
   False
```

Note que a marca `H` foi criada automaticamente pelo Coq, mas você pode colocar outra marca informando-a como parâmetro da tática `intro`, como por exemplo, `intro u`:

```
  H1 : phi -> psi
  H2 : ~ psi
  u : phi
  ============================
  False
```

Esta prova corresponde ao Exercício 7, cuja solução é dada a seguir:

```
Variables phi psi: Prop.

Section mt.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: ~psi.
 Lemma mt: ~phi.
 Proof.
   intro u.
   apply H2.
```

	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	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma, \varphi_1 \vee \varphi_2 \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
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	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$

Tabela 2: Regras da Lógica Minimal

```
  apply H1.
  assumption.
Qed.

End mt.
```

A seguir, veremos exemplos mais complexos onde fórmulas idênticas podem exigir marcas distintas, mas antes disto comparece as as regras de dedução natural para a lógica proposicional minimal com o contexto explícito e com o contexto implícito na Tabela 2.

Exemplo 10. *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 $\psi \vdash \varphi \to \psi$. A ideia da prova neste caso é simples. Vamos assumir uma prova de φ, e transformá-la em uma prova de ψ que já temos como hipótese. Para isto basta introduzirmos e em seguida eliminarmos uma conjunção contendo ψ:*

$$\dfrac{\dfrac{\dfrac{[\varphi]^u \qquad \psi}{\varphi \wedge \psi}\ (\wedge_i)}{\psi}\ (\wedge_e)}{\varphi \to \psi}\ (\to_i)\ u$$

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

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

Exercício 11. *Refaça a prova do exemplo anterior no Coq.*

Exercício 12. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $\varphi, \neg\varphi \vdash \neg\gamma$ na lógica proposicional minimal. Lembre que a negação é o mesmo que uma implicação no absurdo, ou seja, $\neg\varphi$ é o mesmo que $\varphi \to \bot$, qualquer que seja a fórmula φ.*

Também podemos introduzir a dupla negação de uma fórmula qualquer, como solicitado no exercício a seguir:

Exercício 13. *Seja φ uma fórmula da lógica proposicional. Prove o sequente $\varphi \vdash \neg\neg\varphi$.*

O exercício anterior nos dá mais uma regra derivada na lógica proposicional minimal:

$$\frac{\varphi}{\neg\neg\varphi}\ (\neg\neg_i)$$

Como veremos posteriormente, a eliminação da dupla negação de uma fórmula qualquer não pode ser provada na lógica proposicional minimal, mas a dupla eliminação de uma fórmula negada, sim:

Exercício 14. *Seja φ uma fórmula da lógica proposicional. Prove o sequente $\neg\neg\neg\varphi \vdash \neg\varphi$ na lógica proposicional minimal.*

Exercício 15. *Sejam φ e γ fórmulas da lógica proposicional. 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 lógica proposicional minimal.*

Exercício 16. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\varphi) \vee (\neg\gamma) \vdash \neg(\varphi \wedge \gamma)$ na lógica proposicional minimal.*

Exercício 17. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $\neg\neg(\varphi \wedge \gamma) \vdash (\neg\neg\varphi) \wedge (\neg\neg\gamma)$ na lógica proposicional minimal.*

Exercício 18. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\neg\varphi) \wedge (\neg\neg\gamma) \vdash \neg\neg(\varphi \wedge \gamma)$ na lógica proposicional minimal.*

Exercício 19. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $\neg\neg(\varphi \to \gamma) \vdash (\neg\neg\varphi) \to (\neg\neg\gamma)$ na lógica proposicional minimal.*

Exercício 20. *Seja φ uma fórmula da lógica proposicional. Prove o sequente $\vdash \neg\neg(\varphi \vee \neg\varphi)$ na lógica proposicional minimal.*

Os exercícios anteriores incluem diversos resultados importantes que podem ser provados na lógica proposicional minimal, e por isto, é importante que você resolva todos eles em papel e lápis e posteriormente no Coq para verificar se sua solução está correta.

Exercício 21. *Refaça os exercícios anteriores no Coq.*

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] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2008.

[6] 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.

[7] Cezary Kaliszyk. Web Interfaces for Proof Assistants. *Electronic Notes in Theoretical Computer Science*, 174(2):49–61, 2007.

[8] 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.

[9] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107, 2009.

[10] W. McCune. Prover9 and mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005.

[11] 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.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lncs*. Springer, 2002.

[13] 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.

[14] 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.

[15] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J Autom Reasoning*, 55(1):1–37, 2015.

[16] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catvalin Hriatcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.

[17] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[18] Raymond Smullyan. *Logical Labyrinths*. AK Peters, 2009.

[19] Leon Sterling and Ehud Y Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT press, 1994.

[20] The Coq Development Team. The Coq Proof Assistant. Zenodo, October 2021.

	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	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma, \varphi_1 \vee \varphi_2 \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{matrix}[\varphi_1]^u \\ \vdots \\ \gamma\end{matrix} \qquad \begin{matrix}[\varphi_2]^v \\ \vdots \\ \gamma\end{matrix}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{matrix}[\varphi]^u \\ \vdots \\ \psi\end{matrix}}{\varphi \to \psi}\ (\to_i)\ u$
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	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\begin{matrix}[\varphi]^u \\ \vdots \\ \bot\end{matrix}}{\neg\varphi}\ (\neg_i)\ u$
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 3: Regras da Lógica 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:

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

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 (veja o Exercício 12). 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 $\vdash \varphi \vee \neg\varphi$ não tem prova, quando φ é uma fórmula arbitrária.

2.

Vejamos um exemplo de prova na lógica proposicional intuicionista:

Exemplo 22. *Considere o seguinte sequente $\neg\varphi \vee \gamma \vdash \varphi \to \gamma$. Iniciando esta prova de baixo para cima, isto é, partindo do consequente, podemos aplicar a regra de introdução da implicação:*

$$\neg\varphi \vee \gamma \qquad [\varphi]^u$$

$$?$$

$$\frac{\gamma}{\varphi \to \gamma}\ (\to_i)\ u$$

Agora precisamos construir uma prova de γ tendo as fórmulas $\neg\varphi \vee \gamma$ 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:

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

Agora vamos refazer esta prova no Coq. 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 $(\to_i)$, 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.
```

	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	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma, \varphi_1 \vee \varphi_2 \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{matrix}[\varphi_1]^u \\ \vdots \\ \gamma\end{matrix} \qquad \begin{matrix}[\varphi_2]^v \\ \vdots \\ \gamma\end{matrix}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{matrix}[\varphi]^u \\ \vdots \\ \psi\end{matrix}}{\varphi \to \psi}\ (\to_i)\ u$
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	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\begin{matrix}[\varphi]^u \\ \vdots \\ \bot\end{matrix}}{\neg\varphi}\ (\neg_i)\ u$
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	$\dfrac{}{\vdash \varphi \vee \neg\varphi}\ (\text{LEM})$	$\dfrac{}{\varphi \vee \neg\varphi}\ (\text{LEM})$

Tabela 4: Regras da Lógica Clássica

Exercício 23. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\neg\varphi) \to (\neg\neg\psi) \vdash \neg\neg(\varphi \to \psi)$ na lógica proposicional intuicionista.*

Comparando o exercício anterior com o Exercício 19, podemos concluir que as fórmulas $(\neg\neg\varphi) \to (\neg\neg\psi)$ e $\neg\neg(\varphi \to \psi)$ podem ser provadas uma a partir da outra. Isto nos dá uma noção de equivalência que representaremos por $(\neg\neg\varphi) \to (\neg\neg\psi) \dashv\vdash \neg\neg(\varphi \to \psi)$. Podemos ainda escrever $(\neg\neg\varphi) \to (\neg\neg\psi) \dashv\vdash_i \neg\neg(\varphi \to \psi)$ para enfatizar que esta equivalência se dá na lógica intuicionista.

Exercício 24. *Seja φ uma fórmula da lógica proposicional. Construa uma prova para o sequente $\vdash \neg\neg(\neg\neg\varphi \to \varphi)$ na lógica proposicional intuicionista.*

Exercício 25. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $\vdash \neg\neg(((\varphi \to \psi) \to \varphi) \to \varphi)$ na lógica proposicional intuicionista.*

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 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.

Exemplo 26. *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:

Contexto explícito	*Contexto implícito*
$\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}$ (PBC)	$\begin{array}{c}[\neg\varphi]^u \\ \vdots \\ \dfrac{\bot}{\varphi}\ (PBC)\ u\end{array}$

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 27. *Acabamos de caracterizar a lógica proposicional clássica como sendo a lógica proposicional intuicionista juntamente com a lei do terceiro excluído (ver Tabela 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 5 nos dá outra caracterização da lógica*

	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	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma, \varphi_1 \vee \varphi_2 \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
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	$\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}$ (PBC)	$\dfrac{\begin{array}{c}[\neg\varphi]^u \\ \vdots \\ \bot\end{array}}{\varphi}\ (PBC)\ u$

Tabela 5: Regras da Lógica Clássica (versão 2)

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 5. Sendo assim, prove os sequentes a seguir utilizando as regras da Tabela 5:

1. $\bot \vdash \varphi$ *(regra da explosão)*
2. $\vdash \varphi \vee \neg\varphi$ *(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:

Contexto explícito	Contexto implícito
$\dfrac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}\ (\neg\neg_e)$	$\dfrac{\neg\neg\varphi}{\varphi}\ (\neg\neg_e)$

Exercício 28. *Substitua a regra 9 (PBC) na Tabela 5 pela regra* $(\neg\neg_e)$*, e prove os seguintes sequentes:*

1. $\bot \vdash \varphi$ *(regra da explosão)*
2. $\vdash \varphi \vee \neg\varphi$ *(lei do terceiro excluído)*

3. $\neg\varphi \to \bot \vdash \varphi$ *(prova por contradição)*

4. Refaça estas três provas no Coq.

Considerando que uma 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
$\dfrac{}{\vdash ((\varphi \to \psi) \to \varphi) \to \varphi}$ (LP)	$\dfrac{}{((\varphi \to \psi) \to \varphi) \to \varphi}$ (LP)

Exercício 29. *Assuma a regra (LP) acima, e prove o sequente* $\vdash \varphi \vee \neg\varphi$ *utilizando as regras da Tabela 3*

Exercício 30. $\psi_1 \wedge \psi_2 \dashv\vdash \neg(\neg\psi_1 \vee \neg\psi_2)$

Exercício 31. $\psi_1 \to \psi_2 \dashv\vdash (\neg\psi_1) \vee \psi_2$

Exemplo 32. *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* $x \leftrightarrow Y$*, onde a bi-implicação é apenas uma notação compacta*

para $(x \to Y) \wedge (Y \to x)$*. 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 33. *Prove o sequente* $b \leftrightarrow (a \leftrightarrow (\neg a)), c \leftrightarrow (\neg b) \vdash c$ *construído no exemplo anterior. Em seguida refaça a prova em Coq.*

Exercício 34. *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 dois de nós somos desonestos". Neste momento C se manifesta: "Não acredito nisto! Isto é uma mentira!". Questão: C é honesto ou desonesto? Justifique sua resposta, e depois represente a sua solução via um sequente na lógica proposicional, e por fim construa uma prova para este sequente em dedução natural e refaça a prova em Coq.*

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 (1) 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 ($\bot$), 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 1 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 1 é: dadas duas fórmulas (não necessariamente iguais!), podemos construir a sua conjunção. Alternativamente, poderíamos ter escrito a gramática 1 da seguinte forma equivalente:

$$\varphi, \psi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \psi) \mid (\varphi \vee \psi) \mid (\varphi \to \psi) \tag{4}$$

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 $(\varphi_1 \to \varphi_2)$ com a semântica dada na tabela abaixo.

φ_1	φ_2	$(\varphi_1 \to \varphi_2)$
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 1 (ou 4) 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 um exemplo anterior, pode ser construída a partir da implicação e da conjunção: $\varphi \leftrightarrow \psi$ é o mesmo que $(\varphi \to \psi) \wedge (\psi \to \varphi)$. 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 sempre verdadeira.

Uma nomenclatura importante que classifica as fórmulas da LP é dada a seguir:

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 1 e 4 já vimos que a constante $\bot$ 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 35 (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 36 (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.

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:

$$t ::= x \mid f(t, \ldots, t) \tag{5}$$

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

$$\varphi ::= p(t, \ldots, t) \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \mid \exists_x \varphi \mid \forall_x \varphi \tag{6}$$

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, $\ldots$, 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.

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:

$$\frac{\exists_x \varphi(x) \qquad \begin{array}{c}[\varphi(x_0)]^u \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\exists_e)\ u$$

onde x_0 é uma variável nova que não ocorre em γ.

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 37. *Prove o sequente $\forall_x \neg\varphi \vdash \neg\exists_x \varphi$ na lógica intuicionista.*

Assim como a lógica proposicional, a lógica de primeira ordem é correta e completa, mas estes resultados não serão provados aqui (Veja [4]).

Indução Estrutural

A gramática 1 nos diz como as fórmulas da LP podem ser construídas. Observe em particular os construtores recursivos destas gramáticas: por exemplo, a negação de uma fórmula é construída a partir de outra fórmula já construída; a conjunção, a disjunção ou a implicação se constroem a partir de duas fórmulas já construídas. As árvores de derivação das provas anteriormente também são definidas a partir de uma gramática recursiva: uma árvore é um nó, ou construímos uma nova árvore a partir de uma ou mais árvores já construídas. Nesta seção estudaremos como provar propriedades de elementos de conjuntos definidos recursivamente, como as fórmulas da LP ou as árvores de derivação citadas anteriormente. Vejamos inicialmente um exemplo bem familiar, o conjunto dos números naturais $\mathbb{N}$, que pode ser definido pela gramática a seguir:

$$n ::= 0 \mid S\ n \tag{7}$$

A gramática (7) 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)[2] (passo indutivo) então podemos concluir que P vale para todos os números naturais. Esquematicamente, podemos apresentar este princípio como a seguir:

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

Vejamos o que ocorre no Coq. Inicialmente, com o comando `Print nat`, podemos ver como são definidos os números naturais:

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

A linha acima nos diz que o tipo `nat` dos números naturais é definido indutivamente (palavra reservada `Inductive`), e que esta definição possui dois construtores: o `O` que tem tipo `nat`, ou seja, `O` é um número natural, e o `S` que é uma função que recebe um número natural como argumento e retorna outro natural como resultado. Esta é essencialmente a mesma informação dada pela gramática (7). Toda definição indutiva em Coq é capaz de gerar um princípio de indução de forma automática. 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
                         | O => 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 essencial da resposta acima está na última linha, onde `P` é uma propriedade qualquer dos números naturais. 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`.

Podemos agora, resolver o problema acima da seguinte forma:

Exemplo 38. *a propriedade que diz que 'a soma dos n primeiros números ímpares é igual a n^2' vale trivialmente para 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.*

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

- ***(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: $\sum_{i=1}^{k+1}(2.i-1) = \sum_{i=1}^{k}(2.i-1)+(2.k+1) \stackrel{h.i.}{=} k^2+(2.k+1) = (k+1)^2$.*

Por fim, apresentamos esta prova na forma de árvore, para em seguida reproduzirmos a prova em Coq.

$$\dfrac{\dfrac{}{\sum_{i=1}^{0}(2.i-1) = 0^2}\,0=0 \qquad \dfrac{\dfrac{\dfrac{[\sum_{i=1}^{k}(2.i-1) = k^2]^u}{\sum_{i=1}^{k}(2.i-1)+(2.k+1) = (k+1)^2}}{\sum_{i=1}^{k+1}(2.i-1) = (k+1)^2}(\rightarrow_i)\,u}{\sum_{i=1}^{k}(2.i-1) = k^2 \rightarrow \sum_{i=1}^{k+1}(2.i-1) = (k+1)^2}}{\sum_{i=1}^{n}(2.i-1) = n^2}\,(\text{Ind. em } n)$$

Em Coq, precisamos inicialmente definir a função 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).
Eval compute in (msum 2).
Eval compute in (msum 3).
Eval compute in (msum 4).
```

A primeira linha retorna 1, que é igual ao primeiro número ímpar. A segunda linha retorna 4, que corresponde a soma dos dois primeiros números ímpares, 1+3. De acordo com estes testes, nossa definição de somatório está funcionando como esperado, e portanto podemos definir 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.
Proof.
```

A prova segue a mesma ideia da árvore acima. Iniciamos a prova por indução em `n` com a tática `induction n`. Teremos então dois casos para analisar. 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. O segundo caso corresponde ao passo indutivo, onde `n` é um sucessor, digamos `(S k)`. Aplicamos a tática `simpl` para simplificar a expressão `msum (S k)` para que possamos usar a hipótese de indução via o comando `rewrite IHn`. A tática `rewrite` nos permite substituir o lado esquerdo pelo lado direito de uma igualdade, ou vice-versa com `rewrite <-`. A expressão resultante é uma igualdade envolvendo soma e multiplicação de números naturais. As simplificações algébricas necessárias para que possamos concluir que os lados esquerdo e direito da igualdade coincidem são feitas pela tática `lia`. Segue o código da prova completa:

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

Exercício 39. *Prove (em papel e lápis e no Coq) que a soma dos n primeiros números naturais é igual a* $\frac{n.(n+1)}{2}$*, ou seja, que* $1 + 2 + \ldots + n = \frac{n.(n+1)}{2}$.

Exercício 40. *Prove (em papel e lápis e no Coq) que a soma dos n primeiros cubos é igual ao quadrado da soma de 1 até n, ou seja, que* $1^3 + 2^3 + \ldots + n^3 = (1 + 2 + \ldots + n)^2$.

Como seria o princípio indutivo associado à gramática 1? Este princípio é análogo ao apresentado acima para os naturais considerando que temos uma base de indução para cada construtor não recursivo, e um passo indutivo para cada construtor recursivo. Como os naturais têm apenas um construtor não recursivo (zero), e um recursivo (sucessor), o princípio de indução tem apenas uma base de indução e um passo indutivo. Já a gramática 1 que define as fórmulas da LP, possui dois construtores não recursivos (variáveis proposicionais e a constante $\bot$) e quatro construtores recursivos (negação, conjunção, disjunção e implicação), e portanto o princípio indutivo correspondente terá a seguinte forma, considerando uma propriedade Q qualquer das fórmulas da LP:

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

onde $\star \in \{\wedge, \vee, \rightarrow\}$. Chamamos o princípio de indução construído a partir de uma gramática recursiva de *indução estrutural*.

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 41. *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 **??** 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 $\varphi = \psi_1 \to \psi_2$ 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 **??** da lista sabemos que $\psi_1 \to \psi_2 \dashv\vdash (\neg\psi_1) \vee \psi_2$. Então basta tomar $\varphi' = (\neg\psi_1') \vee \psi_2'$ e estamos prontos.*

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

Exercício 42. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\to$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Exercício 43. *Baseado no que foi estudado sobre indução estrutural na LP, sabemos como gerar princípios de indução para gramáticas recursivas como 6. De fato, no caso dos números naturais temos a gramática:*

$$n ::= 0 \mid S\ n$$

e o princípio de indução:

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

Para a gramática da LP:

$$\varphi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi)$$

o princípio gerado foi:

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

onde $\star \in \{\wedge, \vee, \rightarrow\}$.

Considerando a gramática da LPO

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

escreva o princípio de indução correspondente.

No próximo capítulo estudaremos diversos algoritmos que utilizam a estrutura de lista, 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.

Exercício 44. *Escreva o princípio de indução para listas, e em seguida compare sua resposta com o princípio gerado em Coq via o comando* `Print list_ind`.

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 refletem diversas propriedades envolvendo estas operações. Resolva estes exercícios manualmente, e em seguida, no Coq.

Exercício 45. *Prove que* $|l_1 \circ l_2| = |l_1| + |l_2|$, *quaisquer que sejam as listas* l_1, l_2.

Exercício 46. *Prove que* $l \circ nil = l$, *qualquer que seja a lista* l.

Exercício 47. *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 48. *Prove que* $|rev(l)| = |l|$, *qualquer que seja a lista* l.

Exercício 49. *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 50. *Prove que* $rev(rev(l)) = l$, *qualquer que seja a lista* l.

Algoritmos

O algoritmo de ordenação por inserção

Nesta seção estudaremos o algoritmo de ordenação por inserção. 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 51. *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 52. *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:

$ord\_insercao\ (3 :: 2 :: 1 :: nil) =$ (def. $ord\_insercao$)
$insere\ 3\ (ord\_insercao\ (2 :: 1 :: nil)) =$ (def. $ord\_insercao$)
$insere\ 3\ (insere\ 2\ (ord\_insercao\ (1 :: nil))) =$ (def. $ord\_insercao$)
$insere\ 3\ (insere\ 2\ (insere\ 1\ (ord\_insercao\ nil))) =$ (def. $ord\_insercao$)
$insere\ 3\ (insere\ 2\ (insere\ 1\ nil)) =$ (def. $insere$)
$insere\ 3\ (insere\ 2\ (1 :: nil)) =$ (def. $insere$)
$insere\ 3\ (1 :: (insere\ 2\ nil)) =$ (def. $insere$)
$insere\ 3\ (1 :: 2 :: nil) =$ (def. $insere$)
$1 :: (insere\ 3\ (2 :: nil)) =$ (def. $insere$)
$1 :: 2 :: (insere\ 3\ nil) =$ (def. $insere$)
$1 :: 2 :: 3 :: nil$

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 53. *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:*

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

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

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 53. Segundo esta definição, a lista $(1 :: 2 :: 3 :: nil)$ está ordenada. De fato, a prova deste fato é dada pela seguinte árvore de derivação:

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

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

Lema 54. *Sejam x um número natural, e l uma lista de números naturais. Se l está ordenada então $(insere\ x\ l)$ também está ordenada.*

Demonstração. A prova é por indução na estrutura da lista l. Se l for a lista vazia então $(insere\ x\ l)$ é a lista unitária $x :: nil$ 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, $insere\ x\ (h :: tl)$ retorna a lista $x :: h :: tl$ que está ordenada já que $x \leq h$ e, por hipótese, a lista $h :: tl$ está ordenada:

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

- $x > h$: Neste caso, x será inserido na cauda tl, e por hipótese de indução temos que a lista $(insere\ x\ tl)$ 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 $(insere\ x\ tl)$, e portanto a lista $h :: (insere\ x\ tl)$ 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 51: 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 53 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 54 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 54, 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 54, 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, $\texttt{x} \leq \texttt{h}$ e $\texttt{x} > \texttt{h}$. 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 `x` $>$ `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 54, 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 55. *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 56. *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 57. *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 58. *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 59. *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 54. 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 54.

Exercício 60. *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 61. *O algoritmo de ordenação por inserção da Definição 52 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 54 concluímos que a lista $insereh(ord\_insercao\ tl)$ está ordenada, e portanto $ord\_insercao\ (h :: tl)$ está ordenada.

□

Exercício 62. *Refaça a prova acima utilizando a estrutura de árvore. Em outras palavras, prove o sequente* $\vdash sorted(ord\_insercao\ l)$.

Agora prove o Lemma 61 em Coq:

Exercício 63. `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 64. *Seja x um número natural, e l uma lista de números naturais. Definimos recursivamente o número de ocorrências de x em l por:*

$$num\_oc\ x\ l = \begin{cases} 0, & se\ l = nil \\ 1 + num\_oc\ x\ tl, & se\ l = x :: tl \\ num\_oc\ x\ tl, & caso\ contrário. \end{cases}$$

O predicado $perm$, que define quando duas lista, digamos l e l' são permutações uma da outra.

Definição 65. *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 66. *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 (veja o Exercício 44):

$$\dfrac{(def.)\ \dfrac{(\text{Refl})\ \dfrac{}{perm\ nil\ nil}}{perm\ nil\ (ord\_insercao\ nil)} \qquad \dfrac{\dfrac{(*)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{perm\ (h :: tl)\ (ord\_insercao\ (h :: tl))}}{perm\ l\ (ord\_insercao\ l)}\ (ind.\ em\ l)$$

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:

$$\dfrac{}{t = t}\ (\text{Refl})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\ (h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\ (def.)}{num\_oc\ h\ tl = num\_oc\ h\ (ord\_insercao\ tl)}\ (\forall_e)}{S(num\_oc\ h\ tl) = S(num\_oc\ h\ (ord\_insercao\ tl))}}{num\_oc\ h\ (h :: tl) = num\_oc\ h\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{(*)}$$

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 64, podemos construir a versão em Coq da definição 65:

```
Definition perm l l' := forall n:nat, num_oc n l = num_oc n l'.
```

Agora vamos refazer a prova do Teorema 66 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 67. *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 68. *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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\,(h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\,(def.)}{num\_oc\ h\ tl = num\_oc\ h\ (ord\_insercao\ tl)}\,(\forall_e)}{S(num\_oc\ h\ tl) = S(num\_oc\ h\ (ord\_insercao\ tl))}}{num\_oc\ h\ (h :: tl) = num\_oc\ h\ (insere\ h\ (ord\_insercao\ tl))}\,(h = x)\quad \dfrac{}{(h = x) \vee (h \neq x)}\,(**)}{num\_oc\ x\ (h :: tl) = num\_oc\ x\ (insere\ h\ (ord\_insercao\ tl))}\,(def.)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}}{(*)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\,(h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\,(def.)}{num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\,(\forall_e)}{num\_oc\ x\ (h :: tl) = num\_oc\ x\ (insere\ h\ (ord\_insercao\ tl))}\,(h \neq x)}{(**)}$$

Os lemas *ord_insercao_ordena* e *ord_insercao*$_p$*erm* 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 69. *Mostre que o predicado perm da Definição 65 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 70. *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 71. *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*:

$$\dfrac{\dfrac{(*)}{permutation\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}}{permutation\ (h :: tl)\ (ord\_insercao\ (h :: tl))}\ (def.)$$

□

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 72. *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 71:

$$\dfrac{\dfrac{\dfrac{}{permutation\ tl\ (ord\_insercao\ tl)}\ (h.i.)}{permutation\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (\textsc{Lema}\ 70)}{(*)}$$

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 73. `Theorem ord_insercao_permutation: forall l, permutation l (ord_insercao l).`

Agora vamos fazer a prova do Lema 72:

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 70.

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 $a > h$.

$$\dfrac{\begin{array}{r}(permutation\_skip)\ \dfrac{\begin{array}{r}(permutation\_skip)\ \dfrac{(hip.)\ \dfrac{}{permutation\ tl\ tl'}}{permutation\ (h :: tl)\ (h :: tl')}\end{array}}{permutation\ (a :: h :: tl)\ (a :: h :: tl')}\end{array}\qquad (*)}{permutation\ (a :: h :: tl)\ (insere\ a\ (h :: tl'))}\ (def.)$$

   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.

$$\dfrac{\dfrac{}{permutation\ (a :: h :: tl)\ (h :: a :: tl)}\ \Delta \qquad \dfrac{\dfrac{}{permutation\ (a :: tl)\ (insere\ a\ tl')}\ (h.i.)}{permutation\ (h :: a :: tl)\ (h :: (insere\ a\ tl'))}\ \clubsuit}{permutation\ (a :: h :: tl)\ (h :: (insere\ a\ tl'))}\ \nabla$$
$$(*)$$

   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 $(a :: y :: x :: tl)$:

$$\clubsuit\ \dfrac{\Delta\ \dfrac{}{permutation\ (x :: y : tl)\ (y :: x :: tl)}}{\dfrac{permutation\ (a :: x :: y : tl)\ (a :: y :: x :: tl)}{permutation\ (a :: x :: y :: tl)\ (insere\ a\ (y :: x :: tl))}\ (*)}\ \nabla$$

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:

$$\dfrac{\dfrac{?}{permutation\ (a :: y :: x :: tl)\ (insere\ a\ (y :: x :: tl))}}{(*)}$$

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 74. *Seja x um número natural, e l uma lista de naturais. Então $permutation\ (x :: l)\ (insere\ x\ l)$.*

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.

$$\dfrac{\dfrac{}{permutation\ (a :: y :: x :: tl)\ (insere\ a\ (y :: x :: tl))}\ (Lema\ 72)}{(*)}$$

4. A quarta e última regra é $(permutation\_trans)$, e considerando que a hipótese $(permutation\ l\ l')$ tenha sido gerada por esta regra, temos por hipótese que $(permutation\ l\ l0)$ e $(permutation\ l0\ l')$ para alguma lista $l0$. Além disto, temos duas hipóteses de indução:

   (a) Se $permutation\ l\ l0$ então $permutation\ (a :: l)\ (insere\ a\ l0)$;

   (b) Se $permutation\ l0\ l'$ então $permutation\ (a :: l0)\ (insere\ a\ l')$.

   A prova de $permutation\ (a :: l)\ (insere\ a\ l')$ é como a seguir:

$$\clubsuit\ \dfrac{(hip.)\ \dfrac{}{permutation\ l\ l0}}{permutation\ (a :: l)\ (a :: l0)}\qquad \dfrac{(**)}{permutation\ (a :: l0)\ (insere\ a\ l')}\ \nabla$$

   onde

- ♣ corresponde à regra (*permutation_skip*);
- ∇ corresponde à regra (*permutation_trans*).

E o ramo da direita é concluído com a hipótese de indução:

$$\dfrac{(hip.)\ \dfrac{}{permutation\ l0\ l'} \qquad \dfrac{}{permutation\ l0\ l' \to permutation\ (a :: l0)\ (insere\ a\ l')}\ (h.i.)}{(**)}\ (\to_e)$$

□

Exercício 75. *Prove o Lema 74 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 76. *Prove o Lema 72 no Coq.*

```
Lemma permutation_insere_diff: forall l l' a, permutation l l' ->
                               permutation (a :: l) (insere a l').
```

Exercício 77. *Mostre que o predicado permutatio da Definição 70 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 70. 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 65 e 70 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 78. *Sejam l e l' duas listas de números naturais. Se permutation l l' então perm l l'.*

Teorema 79. *Sejam l e l' duas listas de números naturais. Se perm l l' então permutation l l'.*

A prova do Teorema 78 segue a mesma estrutura da prova do Lema 72, isto é, indução na hipótese (*permutation* l l') e será deixado como exercício:

Exercício 80. *Prove o Teorema 78.*

Exercício 81. *Prove o Teorema 78 em Coq:*

```
Lemma permutation_to_perm: forall l l', permutation l l' -> perm l l'.
```

A prova do Teorema 79 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 82. *Seja l uma lista de números naturais. Se perm nil l então $l = nil$.*

Exercício 83. *Prove o Lema 82, 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 84.
```
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 85.
```
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 86. `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 79 em Coq:

Exercício 87. `Theorem perm_to_permutation: forall l l', perm l l' -> permutation l l'.`

Complexidade do algoritmo de ordenação por inserção

Um outro aspecto que precisa ser analisado além da correção é a eficiência do algoritmos. O que significa dizer que um algoritmo é eficiente? Podemos analisar a eficiência de um algoritmo de duas formas: eficiência temporal e eficiência espacial. No primeiro caso, estamos interessados no tempo de execução do algoritmo, enquanto que no segundo caso, queremos analisar a quantidade de espaço extra (memória) que é utilizado pelo algoritmo durante sua execução. A forma de determinar a eficiência de um algoritmo deve permitir a comparação de algoritmos distintos que resolvam o mesmo problema de forma a permitir determinar qual dos dois é o mais eficiente. Inicialmente, poderíamos pensar em utilizar o tempo de execução de um programa que implementa um algoritmo, mas esta não é uma boa medida porque depende do computador utilizado e da implementação feita. Precisamos de um método que nos informe sobre a eficiência do algoritmo independentemente do computador em que ele venha a ser implementado, da linguagem de programação e do estilo de programação utilizados. O método deve ser preciso e geral de forma que possa ser utilizado para diversos algoritmos e aplicações. Esta independência será obtida via o conceito de *operação básica*, que é a operação que mais contribui para o tempo total de execução do algoritmo. A estratégia utilizada será contar o número de operações básicas executadas em uma entrada de tamanho n. No caso de 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 51). Note que quando l é a lista vazia, a lista unitária $x :: nil$ é retornada e nenhuma operação é realizada. Quando l é uma lista da forma $h :: tl$ então é feita a operação de comparação entre x e h. Quando $x \leq h$ a lista $x :: h :: tl$ é retornada e o algoritmo termina. Quando $x > h$, o algoritmo continua buscando recursivamente a posição correta para inserir x. A operação básica no caso do algoritmo de ordenação por inserção é a comparação. Alguns exemplos de operações básicas para diferentes problemas:

Problema	**Operação básica**
Ordenação de um vetor	Comparação de duas entradas do vetor
Busca em um vetor	Comparação com elementos do vetor
Multiplicação de duas matrizes	Multiplicação de dois números reais

O parâmetro n que denota o tamanho da entrada também precisa ser fornecido a partir de uma medida adequada para que possamos apresentar uma análise concisa. Para o caso de ordenação de um vetor, vimos que o número de elementos do vetor representa uma medida adequada. Vejamos mais exemplos:

Problema	**Tamanho da entrada**
Ordenação de um vetor	Tamanho do vetor
Busca em um vetor	Tamanho do vetor
Multiplicação de duas matrizes	Dimensão das matrizes

Retornando para o algoritmo de ordenação por inserção, denotaremos por $T_{insere}(n)$, o número de operações básicas realizadas pela função $insere$ para uma entrada da tamanho n. Note que o número de comparações pode ser diferente para listas de mesmo tamanho. De fato, $0 \leq T_{insere}(n) \leq n$. Por exemplo, o número de comparações realizadas para inserir o número 1 na lista $1 :: 2 :: 3 :: nil$ é 1, enquanto que para inserir o número 10 na lista $1 :: 2 :: 3 :: nil$ são realizadas 3 comparações. Esta contagem é modelada pela função T_{insere} a seguir:

$$T_{insere}\ x\ l = \begin{cases} 0, & \text{se } l = nil \\ 1, & \text{se } l = h :: tl \text{ e } x \leq h \\ 1 + (T_{insere}\ x\ tl), & \text{se } l = h :: tl \text{ e } x > h \end{cases}$$

De qualquer forma, o número de comparações não pode ser maior do que o tamanho da lista onde o elemento será inserido. Este limite superior dá origem à noção de *análise do pior caso*, isto é, a análise do pior caso fornece a pior situação possível. Assim, podemos definir a função $T^w_{insere}(n)$ que recebe como argumento um natural (que corresponde ao tamanho da lista a ser ordenada) e faz o número máximo de comparações possível:

$$T^w_{insere}(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1 + T_{winsere(n-1),} & \text{se } n > 0 \end{cases}$$

Assim, a relação entre as funções T_{insere} e T^w_{insere} é dada pelo lema a seguir:

Exercício 88. *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.*

Exercício 89. *Prove que $T^w_{insere}(n) = n$, para todo n.*

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 90. `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:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + (n-1), & \text{se } n > 0 \end{cases}$$

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, $T^w_{is}(n) = T^w_{is}(n-1) + (n-1) \stackrel{h.i.}{=} \frac{(n-1)(n-2)}{2} + (n-1) = \frac{n(n-1)}{2}$.

Agora prove este lema em Coq:

Exercício 91.
```
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.
```

A conclusão desta seção é de que o algoritmo de ordenação por inserção é correto, e sua complexidade é quadrática em relação ao tamanho da entrada. Na próxima seção estudaremos um algoritmo mais eficiente do que a ordenação por inserção.

O algoritmo *mergesort*

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 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 $(1 :: 2 :: 3 :: 4 :: nil)$.

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.

Para provarmos a correção do algoritmo `mergesort` precisamos inicialmente mostrar que a função `merge` retorna uma lista ordenada, caso cada uma das listas do par dado como argumento também estejam ordenadas:

Exercício 92. `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 93. `Theorem mergesort_sorts: forall l, sorted (mergesort l).`

Exercício 94. `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 95. `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 96. `Lemma T_mergesort_complexity: forall l k,`
`length l = 2^k -> T_mergesort l <= k * 2^k.`

Árvores binárias

Árvores de busca binária

Esta seção foi adaptada de [16].