Matemática Discreta em Lean

many

26 de Novembro de 2019

Conteúdo

Capítulo 1

Introdução

Há evidências do uso da matemática já por volta de 3000 a.c, onde povos antigos como os mesopotâmicos, egípcios e moradores de Ebla utilizavam a aritimética, álgebra e a geometria para taxação, comércio, trocas, astronomia e na formulação de calendários e marcação do tempo corrente. Os registros textuais mais antigos desse tipo de prática vem da Mesopotâmia e do Egito, e são datados aproximadamente entre 2000 a.C e 1800 a.C, mas muitos estudiosos consideram a Grécia Antiga como o berço da matemática, onde, por volta de 600 a.c, os Pitagóricos iniciaram o estudo da matemática como uma "disciplina demonstrativa"e introduziram o raciocínio dedutivo e o rigor matemático em provas, sendo que a forma como se escrevia matemática era bem diferente da usual atualmente, com uso extensivo de símbolos para abstrair grandezas e representar variáveis.

Paralelo ao desenvolvimento da Matemática, Aristóteles, estudando Retórica, percebeu padrões na formulação frasal humana e cognitiva da linguagem. De fato, o clássico exemplo

> Todos os homens são mortais.
> Socrates é homem.
> Logo, Sócrates é mortal.

Ilustra um mecanismo dedutivo (modus ponens, que é visto mais à frente) que fazemos naturalmente e é estudado a fundo na sua obra, que é o marco zero dá lógica clássica, que está diretamente relacionada à oposição do verdadeiro e do falso. Essa visão imperou até o fim do século XIX, quando percebeu-se limitações abstratas disto e surgiram esforços no sentido da formalização da Matemática com Frege, Gentzen e outros (esse campo ficou denominado metamatemática).

Apenas quando surge o questionamento de *como sabemos*, e não apenas *o que sabemos* as bases para desenvolver um conhecimento sólido são devidamente colocadas na Matemática. Aqui, a lógica formal entra exatamente ao questionar *como sabemos*, ou melhor, como temos certeza do que concluimos partindo

de um contexto inicial. A formalização do processo de raciocínio garante que *construimos um castelo sólido*.

Não devemos, no entanto, nos limitar a discutir lógica como ferramenta relacionada a teoremas ou metafísica. Esse ramo é presente no estudo da linguagem (lógica informal), processos industriais e computação pela verificação de hardware e de software (métodos formais), estudos em representação do conhecimento por exemplo.

1.1 Lógica formal e linguagem natural

A linguagem humana é conhecidamente um emaranhado gigantesco de simbolos, gestos, palavras, sentidos, interpretações. Essa é fruto de lapidação através de milênios, do que se iniciou com uma linguagem básica animalesca, chegando a complexidade atual em um processo evolutivo constante; assim ocorreu com a fala e escrita.

É impossível afirmar, portanto, que uma língua, digamos o português, é ineficiente na transmissão de significados, sendo fruto de evolução pura e constante. De fato, os linguistas se dedicam em grande parte a desvendar os mecanismos segundo os quais a lingua se molda e se torna tão eficiente, transmitindo significados profundos, inclusive muitos difíceis de explicados formalmente. A linguagem natural consegue representar diversas noções como medo, incertezas e possibilidades; sendo estas formalmente imprecisas.

Em contraparte, linguagens formais buscam firmar o processo de pensamento sob regras bem definidas. Muitas vezes, essas linguagens não captam as nuances da linguagem natural em sua completude, no entanto permitem abstrair alguns importantes conceitos que acabam sendo muito complexos de analisar numa linguagem não formal pela variedade e ambiguidade de várias construções. Por exemplo, podemos afirmar que *toda vaca voadora come gente*, e isso não fará o menor sentido a não ser que estabeleçamos essa como uma afirmativa lógica ao abstrair a realidade. Esse é um exemplo caricato que esclarece como a lógica nos permite discutir conclusões acertadas, que seriam pouco claras apenas utilizando de um arcabouço de linguagem natural e da cognição humana.

[Acho que aqui podemos introduzir para discutir um problema de lógica usando linguagem natural. Mostrar como ficaria extensivo e pouco claro "Procure argumentar uma solução para o problema: "]

Mais uma vez, é importante reforçar que a linguagem natural é um meio extremamente eficiente de transmissão de significaos. Há um ramo da lógica chamado lógica informal que lida com o modo como expressamos o raciocínio através da lingua: argumentação e falácias, por exemplo. Vale a discussão de se *um ser humano sem linguagem enlouqueceria*.

1.2 Sobre Sistemas Dedutivos

Para motivação, considere o seguinte problema: *Você acorda em uma sala com duas portas. Uma dá para a liberdade, e a outra leva a morte, mas você não sabe qual é cada uma. Junto a cada porta há um guarda. Um dos guardas é sempre sincero, enquanto o outro é sempre mentirozo. Você tem direito a uma única pergunta para decidir qual porta tomar. O que você faz?*

Esse é claramente um problema envolvendo um raciocínio complexo, e, se tratando da sua vida em jogo, exige plena certeza da solução antes de uma resposta definitiva. De fato, seremos capazes de formalizar problemas desse tipo, e obter métodos de derivação para obter um prova para a resposta.

1.2.1 Dedução Natural

Uma das maneiras de formalizar e demonstrar problemas lógicos é utilizar sistemas dedutivos como a Dedução Natural, que consiste em um grupo de regras e axiomas que nos permitem manipular expressões formalizadas, ou seja, traduzidas da forma linguística para uma forma lógica matemática de notação, estabelecendo a validade dos argumentos, derivando a conclusão do argumento a partir das premissas usando esse sistema de regras em questão. Por meio dessas regras de inferência, podemos demonstrar a validade de uma infinidade de fórmulas e argumentos sem a necesidade de considerar os valores que cada fórmula ou subfórmula recebe, ou seja, não estamos mais lidando com a semântica, mas com a sintaxe.

Uma outra característica da Dedução Natural é que ela possui um certo formato para suas demonstrações, onde as provas são apresentadas de forma que cada linha corresponda a um passo da prova ou demonstração, as colunas correspondam as premissas e abaixo dessa linha teremos a nossa conclusão. Como o leitor poderá perceber mais à frente, as provas em Dedução Natural pode variar muito de tamanho dependendo das nossas premissas e do que nós queremos provar.

1.2.2 Cálculo de Sequentes

O Cálculo de Sequentes é um estilo de apresentação de provas que foi proposto por Gentzen, em 1934, com o objetivo de servir como uma ferramenta auxiliar para o estudo de dedução natural em lógica de primeira ordem. Assim como outros sistemas dedutivos, o Cálculo de Sequentes também possui suas regras e propriedades que funcionam de maneira harmônica e mantém o aspecto de árvore para as provas em si.

1.2.3 Resolução

O método da resolução foi introduzido por John Alan Robinson que foi um filósofo, matemático e cientista da computação que foi um dos pioneiros em lógica de programação e fez importantes descobertas e grandes contribuições

para a área de fundamentos de provadores de teoremas automáticos, os ATPs (Automated theorem proving). O uso destes provadores automáticos será abordado na próxima seção.

O sistema dedutivo de resolução faz uso de regras de inferência afim de demonstrar por refutação sentenças e inferências da lógica proposicional e da lógica de primeira ordem. Este sistema dedutivo usa a linguagem proposicional no formato CNF (Conjunctive Normal Form) e nâo possui axiomas, mas apenas uma regra de inferência que opera com cláusulas e origina uma nova implicada por elas.

1.3 Provadores, ATP e ITP

Verificação formal envolve o uso de lógica e, mais recentemente, de aparato computacional na descrição de um contexto em termos matemáticos suficientemente precisos, e estabelecimento de assertivas sobre essas definições.

A partir disso, passamos utilizar teoremas matemáticos para verificar assertivas sobre os elementos abstraídos, que podem bem representar todo tipo de objeto de interesse. De fato, somos capazes de descrever matematicamente *softwares*, processos industriais, sistemas de gerenciamento de cargas, ou teoremas sobre conjuntos, por exemplo, utilizando dos mesmos aparatos. Isso significa que na prática, não há distinção entre ferramentas provando teoremas, ou verificando assertivas sobre um objeto qualquer de interesse. Reduzimos os problemas de verificação formal a problemas de provas de teoremas e, mais relevante, podemos utilizar computadores nessa tarefa.

Esse tipo de discussão se tornou possível após os recentes avanços no campo de estudo da lógica. Reduzimos regras de inferencia e derivação presentes na maior parte das provas a um conjunto pequeno de axiomas fundamentais. Provadores desse tipo se tornaram viáveis com o desenvolvimento por Frege de uma base formal adequada a descrição da matemática, em sua *Begriffsschrift*. Após Frege, outros nomes, como Alas, Russeau, Hilbert, e Gentzen se esforlaram a desenvolver bases sólidas e compactas para o pensamento matemático que viriam a ser implementadas pelos computadores.

A compactação do conjunto de regras de demonstração viabilizaria computadores implementando ferramentas para auxílio a prova de teoremas, divididas em basicamente duas classes: auxílio a obtebção de uma prova, ou verificação de uma assertiva dada. Esses são os *ATPs* e o *ITPs*.

1.3.1 Provadores de Teorema Automáticos

Os provadores automáticos (*Automated theorem proving*) são uma primeira classe de provadores assistentes, e visam a obtenção ou verificação de um teorema. Portanto, pertencem a uma classe voltada a obtenção de um termo que representa uma prova, ou a verificação de um teorema, que igualmente se reduz a tarefa de obtenção de uma prova.

Dessa forma, um *ATP* pode tomar como entrada um teorema, e deveria ser capaz de dar um retorno do tipo *verdadeiro* ou *falso*, uma prova ou contraexemplo para a assertiva. Note que nesse processo, a participação do ser humano está restrita apenas a aplicação da entrada, ou a descrição do problema para o computador. Isso expressa o sentido em que esses provadores são *Automáticos*.

De fato, algumas provadores implementam sistemas dedutivos que lidam internamente com o problema, desenvolvendo soluções pouco inteligíveis (resolução, p.ex.). Pode-se dizer que o processo de prova nesses casos é pouco acessível. Exemplos de sietmas, ou algoritmos populares são:

- **Algoritmo de Presburguer** : permite concluir se uma assertiva na *aritmética de presburguer* para números naturais é válida. Em 1954, um aluno da universidade de Princeton implementou o algoritmo de Presburguer em um computador JOHNNIAC que foi capaz de provar que o produto entre números pares é par.
- **Vampire**[1]: é um poderoso provador automático implementado em *C++*, para lógica de primeira ordem. Seu desenvolvimento se iniciou em 1994, e já venceu a *copa mundial de provadores de teoremas*, além de diversas premiações menores.

Em 1976, o computador IBM360 participou da prova dada por Apel e Haken para o *Teorema das quatro cores*, mostrando que o uso de computadores poderia ser relevante no desenvolvimento da matemática.

1.3.2 Provadores de Teorema Interativos

Os provadores interativos (*Interactive theorem proving*) representam uma classe de ferramentas que aproximan a noção de máquina e humano trabalhando juntos no desenvolvimento de uma prova formal. Nesse contexto, a ferramenta é comumente usada na verificação do processo de prova desenvolvido por um indivíduo.

Nesse processo, a intervenção da máquina pode ser superficial, por exemplo na checagem dos teoremas utilizados em um processo de prova, ou mais intenso, em que lacunas são preenchidas ou completadas por um provador quase automático.

A vantagem nesse tipo de método está presente justamente quando trabalhamos provas que se tornam extremamanete complicadas, e um provador automático se torna incapaz de solucionar, ou quando desejamos ter certeza de um resultado, pela dificuldade em garantir a validade do resultado dado por um *ATP*. Problemas desse tipo abrem espaço para o paradigma que garante provas desenvolvidas pela intervenção humana, com assistência computacional.

Alguns *ITPs* populares são:

- **Coq Proof Assistant**[2] : sistema de gestão de provas que permite exprimir assertivas matemáticas, checagem de provas, além de executa algumas

[1] Vampire: www.vprover.org
[2] Coq: coq.inria.fr

provas automáticas. Embora Permita essa forte automação, é ainda considerado sistema de provas interativo, oferecendo um ambiente para provas checadas por máquina.

- **Isabelle**[3]: desenvolvido em parceria entre a Universidade de Cambridge e a Universidade Tecnica de Munique, Isabelle foi pensada para servir de assistente de provas geral, implementando lógica de alta ordem. Teve sua verão inicial lançada em 1986.

Mais a frente discutioms como o Lean implementa um plataforma entre os dois paradigmas: permite o desenvolvimento de provas interativas, sem abrir mão da automação. Podemos discutir *em que ponto o sistema deixa de agir como assistente, e passa a ser considerado autônomo*, ou ainda, *o que é automação.*

1.4 Resumo

[3]Isabelle: isabelle.in.tum.de

Capítulo 2

Introdução ao Lean

Ao final do último capítulo, definimos duas classes fundamentais de ferramentas de auxílio a prova de teoremas, os *ATPs* e *ITPs*.

Essas familias de ferramentas, em geral, são muito bem definidos, distantes; isso gera um espaço entre os métodos automáticos e interativos que pode ser desvantajoso: um matemático pode desejar algo entre uma prova automatizada ou verificada, ou o acesso a ambos os recursos em uma mesma ferramenta.

O provador de teoremas Lean[1] busca preencher essa lacuna: uma única ferramenta contendo o melhor de ambos os mundos acessível. Nesse sentido, age como uma *ponte entre os dois paradigmas*.

O ambiente de Lean suporta interação, construção de termos, e verificação passo-a-passo de expressões. Lean implementa o chamado *calculus of constructions*, ou cálculo de contruções, um sistema dedutivo bastante poderoso, que *contém* demais sistemas a serem discutidos ao longo do livro, tais como Lógica de Proposições, ou Lógica de Primeira Ordem.

Um usuário iniciante pode se deparar com algumas dúvidas pertinentes: *porque contruir um termo (o que é termo e o que não é?) é uma prova?*, *consigo desenvolver provas pensando de trás pra frente e de frente pra trás?*, ou *como conhecer bibliotecas e as função delas que o Lean já tenha implementadas?* Algumas dessas dúvidas são esclarecidas logo mais, enquanto outras serão deixadas para a pŕatica ao longo dos proximos capítulos.

2.1 Teoria dos tipos

Teoria dos Tipos é uma classe de linguagens poderosa, que permite desenvolver assertivas bem definidas sobre definições matemáticas, ou exprimir conceitos desde os abstratos aos mais paupáveis. De fato, essa é uma linguagem bastante expressiva, e útil para definição de elementos formais; até certo ponto, se assemelha a noção comum de teoria dos conjuntos.

[1] documentação, ambiente e tutoriais estão disponiveis em: leanprover.github.io

No entanto, em muitos contextos, é razoável utilizar uma linguagem com ferramental teórico capaz de lidar com uma gama de elementos distitos, pertencentes a diversas classes, *tipos distintos*. Isso ocorre frequentemente quando tratamos de objetos matemáticos, e formalismos: o elemento 1 pode ter o tipo natural ou real, assim como $[1, 2, 3, ...]$ pode expressar o tipo de uma lista de inteiros, dependendo da interpretação e contexto dados. Claramente lidamos constantemente com elementos de *tipos* distintos, objetos que se relacionam, e tipos que se relacionam. Pra isso, *teoria dos tipos* se mostra extremamante útil embasando essas discussões.

Lean, em especial, implementa uma versão da linguagem chamada *Calculo das Construções*, ou *Calculus of Constructions*. A partir dos próximos exemplos, o leitor é convidado a acompanhar fazendo a intelação da ferramenta, ou através do *Lean Web Editor*[2].

2.1.1 Noções Fundamentais

Em teoria dos tipos, tudo tem um tipo definido. Essa é uma exigência razoável; de fato, somos capazes de *tipar* qualquer elemento dentro de um contexto, em uma linguagem adequada. Vejamos alguns exemplos em Lean:

```
1  -- abaixo declaramos algumas variaveis e usamos o
2  -- comando #check para imprimir o tipo do elemento
3  variable b : bool
4  variable n : ℕ
5  variable f : ℕ → ℕ
6  variable g : ℕ × ℕ → ℕ
7
8  #check b            -- b : bool
9  #check 1            -- 1 : ℕ
10 #check n + 1        -- n + 1 : ℕ
11
12 #check f n          -- f n : ℕ
13 #check g (1,n)      -- g (1, n) : ℕ
```

O comando *variable* declara uma nova variável no contexto. A partir disso, somos capazes de checar os tipo.

Note nas linhas 5 e 6 que f e g podem ser interpretadas como funções de naturais em naturais, então pudemos fazer as aplicações em 12 e 13, recebendo fn e $g(1, n)$ como naturais. Essa saída faz sentido quanto a de tipagem: sabemos que a aplicação tem tipo natural, embora não saibamos *qual* natural.

Um leitor curioso pode questionar *se tudo tem um tipo, qual tipo dos tipos?* ou ainda ir além, e *qual o tipo do tipo dos tipos?* E essas perguntas podem ser respondidas novamente através do Lean:

[2] leanprover.github.io/live/latest/

```
#check ℕ            -- ℕ : Type
#check bool         -- bool : Type

-- considere Type = Type 0
#check Type 0       -- Type : Type 1
#check Type 1       -- Type : Type 2
#check Type 2       -- Type : Type 3
```

Essa pode ser uma resposta *frustrante* ou *brilhante*. Os tipos básicos em Lean tem o tipo *Type* (o mesmo que *Type 0*), enquanto o proprio tipo *Type 0* tem tipo *Type 1* que tem tipo *Type 2* e assim por diante.

Essa noção de hierarquia de tipos é uma solução proposta inicialmente por Russel, e posteriormente desenvolvida por Chwistek e Ramsey, em sua *teoria simples dos tipos*, para evitar o conflito entre a teoria dos conjuntos de Frege, e o paradoxo de Russel. De forma simples, a hierarquia de tipos evita o autorreferenciamento, em que um elemento tem o próprio tipo.

O que o Lean se propõe a fazer, portanto, é oferecer um ambiente completo e poderoso para tratar da álgebra de tipos, implementando conceitos como *abstração* e *aplicação*, ou *tipos dependentes*. Exemplos utilizando essas construções são frequentes ao longo do livro em noções como *implicação lógica*, *aplicação de implicações* ou *famílias de conjuntos indexadas*.

2.1.2 Proposições como Tipos

Entramos em uma das discussões mais fundamentais sobre o *como Lean prova coisas*, e para isso, introduzimos a discussão sobre *Proposições como Tipos*, através do tipo *Prop*.

Considere um tipo *Prop* cujos habitantes (elementos daquele tipo) representam uma *prova para essa proposição*. Em outras palavras, dada uma proposição, basta construirmos um termo *habitante daquela proposição* através de aplicações, abstrações ou outras ténicas formais para lidar com os tipos disponibilizadas na teoria, e pelo Lean. Vejamos um exemplo:

```
variables A B : Prop    -- A e B sao proposicoes

variable h1 : A         -- h1 uma prova para A
variable h2 : A → B     -- h2 uma prova para A → B

example : B := h2 h1    -- h2 h1 prova para B
#check h2 h1            -- h2 h1 : B
```

Acima, introduzimos sem maiores explicações algumas noções sobre lógica proposicional em Lean que serão discutidas nos próximos capítulos. No contexto

dado, fomos capazes de construir um termo *h2 h1* que tem tipo B, então construimos uma prova para B, na linha 6; conferimos que tal aplicação tem o tipo desejado, em 7.

Observe ainda, na linha 4: o elemento do tipo $A \to B$, que anteriormente citamos como uma função que leva tipo A em um tipo B. Aqui, introduzimos um exemplo do que será discutido como uma *implicação lógica* no capítulo sobre lógica proposicional.

2.1.3 Outras Proposições

Portanto, introduzimos a noção de proposição: uma assertiva. E essa será base para o conjunto de provas que se seguem o longo do livro; mesmo algumas que não parecem ser *um assertiva do tipo Prop*.

Até aqui, usamos exemplos de proposições como simbolos abstratos, como A e B que reprentam afirmativas, que podem ser afirmaçẽs sobre o mundo. Esse uso faz sentido pela simplicidade; escrevemos A no lugar de $EstaChovendo$ e B ao invés de $SaposPulam$.

No entanto, ao longo do livro nos deparamos com outras expressões, outras *afirmativas/assertivas*; e um leitor pouco atento poderá ignorar o fato de que a noção de *proposição como tipo*, e *prova por obter um termo* ainda está presente. Alguns exemplos dessas proposições:

```
1  variables A B : set ℕ
2  variables a b : bool
3  variables m n : ℕ
4  variables f g : ℕ → ℕ
5
6  #check a ≠ b         -- a ≠ b : Prop
7  #check m = 1         -- m = 1 : Prop
8  #check m = n         -- m = n : Prop
9  #check m = 2*n       -- m = 2 * n : Prop
10 #check A ⊆ B         -- A ⊆ B : Prop
11 #check A = B         -- A = B : Prop
12 #check n ∈ A         -- n ∈ A : Prop
13 #check f m = f n     -- f m = f n : Prop
14 #check f m ≤ g m     -- f m ≤ g m : Prop
```

Lean entende todas essas assertivas sobre Conjuntos, Inteiros, Booleanos, Desigualdades e Relacões como *proposições*, portanto, do tipo $Prop$. Nesse sentido podemos lidar com provas para o tipo daquela assertiva, e todo paradigma de *Proposições Como Tipos*:

```
1  variables A B : set ℕ
2  variables m n : ℕ
```

```
3
4 variable h1 : m = n          -- h1 prova para m = n
5 variable h2 : A ⊆ B          -- h2 prova para A ⊆ B
6 variable h3 : n ∈ A          -- h3 prova para n ∈ A
```

Exemplos como esses serão trabalhados posteriormente, e por isso essa discussão está presente ainda no inicio: o leitor está sendo impelido a obserservar como formalizamos todos os conceitos desejados através de um único paradigma.

2.2 Provas usando Lean

Abriremos discussão sobre o processo de prova utilizando Lean; não esperamos que o leitor a essa altura entenda amplamente as demonstrações dadas, e sim, compreenda como ocorre o processo de prova.

Quando desejamos iniciar a prova explicita para um assertiva, podemos abrir contextos usando as palavras-reservadas *example*, *lemma* ou *theorem*. Cada uma dessas tem sua utilidade distinta, embora semelhantes:

```
1 -- uma prova para A e B
2 example (A B : Prop) (h1 : A) (h2 : A → B): A ∧ B :=
3   and.intro h1 (h2 h1)
4 -- prova para o lemma A e B
5 lemma lema_A_B (A B : Prop) (h1 : A) (h2 : A → B): A ∧ B :=
6   and.intro h1 (h2 h1)
7 -- prova para o teorema A e B
8 theorem teorema_A_B (A B : Prop) (h1 : A) (h2 : A → B): A ∧ B :=
9   and.intro h1 (h2 h1)
```

Para cada contexto, foram dados argumentos *(h : entre prentesis)* considerados na prova; *hipóteses*. Na linha 2, por exemplo, A e B são proposições, $h1$ e $h2$ são provas que deveriamos considerar no contexto. Esse é o contexto que devemos considerar para demonstrar o resultado desejado.

Observe que as provas são identicas; a diferença está exatamente nos contextos. O *theorem*, assim como *lemma*, permite fazer a prova de uma assertiva, e esse resultado poderá ser utilizado eventualmente; naturalmente, isso exige um titulo para o lema ou teorema. Já *example* funciona como contexto fechado, ou rascunho: podemos realizar a prova desejada, mas não nos referir a ela posteriormente.

Veja no exemplo abaixo como podemos utilizar o resultado de um teorema que foi previamente demonstrado em Lean:

```
1 variables A B : Prop     -- A e B sao proposicoes
```

```
2  variable h1 : A          -- h1 uma prova para A
3  variable h2 : A → B      -- h2 uma prova para A → B
4
5  -- prova para o teorema A e B
6  theorem teorema_A_B : A ∧ B := and.intro h1 (h2 h1)
7  -- reutilizando o teorema_A_B
8  example : A ∧ B := (teorema_A_B A B h1 h2)
```

Basta fazer uma chamada ao teorema (ou lema) através do título dado, passando provas para os argumentos esperados. Ao fazermos uso do teorema, linha 8, passamos explicitamente (aplicamos) as proposições, e as provas que temos.

Observe que dessa vez não explicitamos os argumentos que deveriam ser recebidos no *theorem*, ou no *example*. Lean identificou no arquivo quais hipoteses foram utilizadas, e os considerou como argumentos.

Lean é inteligente o bastante para livrar o usuario de parte do trabalho repetitivo: se assinamos as proposições como argumentos implicitos no teorema, o programa será capaz de identificar a partir das hipoteses, quais as proposições:

```
1  -- prova para o teorema U e V, generico
2  theorem teo_U_e_V {U V : Prop} (h1 : U) (h2 : U → V) : U ∧ V :=
3    and.intro h1 (h2 h1)
4
5  variables A B : Prop     -- A e B sao proposicoes
6  variable h1 : A          -- h1 prova para A
7  variable h2 : A → B      -- h2 prova para A → B
8
9  -- utilizando o terema_A_B
10 example : A ∧ B := teo_U_e_V h1 h2
```

Esse uso tambem permite a generalização do resultado obtido. De fato, o *teorema_U_V* vale para quaisquer proposições U e V que se deseje. Quando pusemos chaves, $\{U\ V : Prop\}$ na declaração na linha 2, o próprio programa considerou esses como argumentos que poderiam ser compreendidos no contexto em que fosse utilizado.

Naturalmente, diferentes tipos de proposição podem demandar por uma construção distinta, e esse é um fato inerente ao que se deseja discutir. Nesse sentido, Lean disponibiliza diferentes modos de construção para prova. Esses modos serão abordados em uma gama de exemplos nos próximos capítulos; abordamos aqui, apenas existencia, aspectos e vantagens em cada uso:

2.2.1 Modo Termo

É o modo base ao iniciarmos uma contexto de prova com *theorem*, por exemplo. Nesse modo, somos capazes de construir termos partindo de termos, explicitando cada passo tomado: ao assumir uma hipótese, eliminar uma conjunção, ou introduzir estruturas, exigimos a explicitação do passo. Aqui, as demonstrações são contruidas semelhantes a programação em uma linguagem.

```
1 variables A B C D : Prop
2
3 example (h : ¬ A ∧ ¬ B) : ¬ (A ∨ B) :=
4   -- estamos no Modo Termo
5   show ¬ (A ∨ B), from
6   assume s: A ∨ B,
7     show false, from or.elim s
8       (assume h1: A, show false, from h.left h1)
9       (assume h1: B, show false, from h.right h1)
```

Note que construimos passo-a-passo as várias etapas na prova acima, abrindo pouco esáço pra automação. Outro exemplo:

```
1 variables A B : Prop
2
3 example (h : ¬ A ∨ ¬ B) : ¬ (A ∧ B) :=
4   show ¬ (A ∧ B), from or.elim h
5     (assume h1: ¬ A, show ¬ (A ∧ B), from
6       assume h2: (A ∧ B), show false, from (h1 h2.left))
7     (assume h1 : ¬ B, show ¬ (A ∧ B), from
8       assume h2 : (A ∧ B), show false, from (h1 h2.right))
```

A vantagem nesse modo está no controle direto da prova, e pouca automação, em que o usuário é plenamente consciente sobre o processo. Aqui, Lean atua fortemente na verificação da construção dos termos.

2.2.2 Modo Tática

No Modo Tática a linguagem utilizada na contrução da prova se aproxima da linguagem humana; e exatamente por isso, exige forte automação por parte do sistema. Esse modo é iniciado dentro de um bloco *begin ... end*.

```
1 variables A B C D : Prop
2
3 example (h : ¬ A ∧ ¬ B) : ¬ (A ∨ B) :=
```

```
4  begin
5    intro,           -- A B : Prop, h : ¬A ∧ ¬B, a : A ∨ B ⊢ false
6    cases a,         -- 2 goals case or.inl
7      apply h.left,  -- A B : Prop, h : ¬A ∧ ¬B, a : A ⊢ A
8        assumption,  -- lean conclui que temos a prova "a" para A
9
10     apply h.right, -- A B : Prop, h : ¬A ∧ ¬B, a : B ⊢ B
11       assumption   -- lean conclui que temos a prova "a" para B
12 end
```

Quando abrimos táticas, Lean automatiza várias etapas; quando pousamos o cursor em uma linha, é impresso no console quais as variáveis temos, e quais objetivos, *goals*, precisamos provar. Acima, essas saídas estão comentadas, mas o leitor é incentivado a abrir o console com o exemplo; o *goal* é a proposição após o simbolo "$\vdash$".

Além disso, usamos uma série de palavras-reservadas, como *intro*, *cases*, *assumption*, que indicam comandos para o Lean lidar com esse contexto. O que esses comandos realizam internamente, por exemplo, são:

- **intro/intros**: pede ao Lean que introduza as hipóteses razoáveis dado o problema; no exemplo, essa foi a hipótese do absurdo.

- **cases**: para cada caso *A ou B*, vamos construir a prova desejada; pede ao Lean que introduza os objetivos.

- **assumption**: se a solução já puder ser deduzida no contexto, pede ao Lean que conclua a prova.

Cada uma dessas tarefas exige que o provador resolva problemas de forma autonoma, retirando essa obrigação do usuário. Pequenas automações como essas agilizam grandemente a resolução, mas retiram parte da compreensão ao usuário sobre a demonstração dada.

2.3 Cálculos em Lean

Não confunda com derivadas, integrais ou somatórios; o cálculo essencial desde a matemática básica, como uma sequência de operações para provar que certa relação entre duas expressões. Considere o seguinte exemplo, em que desejamos mostrar a validade de uma iguldade:

$$\begin{aligned}
\frac{(a+b)^2+(a-b)^2}{2} - 2b^2 &=^? (a-b)(a+b) \\
\frac{a^2+2ab+b^2+a^2-2ab+b^2}{2} - 2b^2 &=^? a^2+ab-ab-b^2 \\
\frac{2a^2+2b^2}{2} - 2b^2 &=^? a^2-b^2 \\
a^2+b^2-2b^2 &=^? a^2-b^2 \\
a^2-b^2 &= a^2-b^2
\end{aligned}$$

A impressão é a de que começamos com uma identidade complexa, e conseguimos provar que $x = x$. O melhor a se fazer é justamente partir de uma igualdade válida $x = x$ e construir a igualdade desejada.

A ideia que tentamos transmitir é a de que cada equação é equivalente a seguinte. A primeira equação é verdadeira se, e só se, a próxima for verdadeira. Logo, podemos utilizar o símbolo $\iff$ entre cada uma delas.

$$\begin{aligned}
\frac{(a+b)^2+(a-b)^2}{2} - 2b^2 = (a-b)(a+b) &\iff \\
* \; ** & \\
a^2+b^2-2b^2 = a^2-b^2 &\iff \\
a^2-b^2 = a^2-b^2 & \quad \square
\end{aligned}$$

Ou ainda, frases como *basta provar* entre uma igualdade e outra, destacando a passagem de uma prova como equivalente a próxima prova.

$$\begin{aligned}
\frac{(a+b)^2+(a-b)^2}{2} - 2b^2 = (a-b)(a+b) &\quad \textit{para isso, basta que} \\
* \; ** & \\
a^2+b^2-2b^2 = a^2-b^2 &\quad \textit{para isso, mostro que} \\
a^2-b^2 = a^2-b^2 &\quad \textit{o que é verdadeiro. } \square
\end{aligned}$$

Ambas as provas são válidas, apesar da última possuir uma narrativa repetitiva que nos obriga a encontrar sinônimos para *basta provar*. Ainda assim, construções desse tipo são extremamente comuns.

2.3.1 O modo Calc

Lean disponibiliza um ambiente para cálculos desse tipo, através do identificador *calc*, seguido por um cadeia de passagens, separadas por ":", conforme abaixo:

```
1 -- exemplo de extrutura --
2 show expressaoX < expressaoY, from
```

```
3 calc
4   expressaoX = expressao1 : justificativa1
5          ... = expressao2 : justificativa2
6          ... < expressaoY : justificativa3
```

A cadeia pode se extender quanto quisermos, até chegar na expressão desejada. E cada justificativa é composta por afirmações ou lemas/teoremas. Um exemplo do uso do `calc` em conjuntos é o seguinte:

```
1 variables a b : bool
2
3 example (h : a = b) : b = a :=
4   calc
5     b = a : eq.symm h
```

Esse ambiente representa uma grande utilidade quando desejamos provas procedurais desse tipo.

2.4 Lean na prática

Apresentamos alguns fatos sobre uso do Lean relevantes, que não puderam ser apresentados ao longo do capítulo.

- Lean está disponível no *GitHub*, onde está publicada a sua página[3], contendo tutoriais completos para *download*, documentação, tutoriais sobre *Theorem Proving*, além do histórico de colaboradores.
- Atualmente, a quarta geração da plataforma está sendo desenvolvida, já em fase final; breve estará sendo disponibilizado Lean 4, que pretende servir como uma linguagem de propósito geral.
- Existe uma comunidade extremamente ativa mantendo o desenvolvimento de um repositório de teoremas matemáticos fundamentais em Lean, na *mathlib*[4]. Os próprios usuários desenvolvem provas para os teoremas disponibilizados.
- Existe um fórum[5] ativo sobre Lean, onde dúvidas ou discussões são frequentes. Há ainda uma página para novos membros.

No futuro, alguns desses fatos podem ser alterados, mas vale abrir a discussão sobre esses tópicos, em especial incentivamos o contato de novos usuários de Lean com a ampla comunidade de desenvolvedores.

[3] Lean Theorem Prover: leanprover.github.io
[4] mathlib: github.com/leanprover-community/mathlib
[5] Forum: leanprover.zulipchat.com/login/

Capítulo 3

Lógica Proposicional

3.1 Introdução

A lógica proposicional é uma vertente da lógica na qual representamos proposições como fórmulas (que podem ser falsas ou verdadeiras) e trabalhamos com argumentação desenvolvida por meio de letras, como A e B, que representam essas proposições. Essas fórmulas podem ser proposições atômicas (que não podem ser divididas em outras sentenças mais simples) ou um conjunto de proposições atômicas combinadas por conectivos lógicos, o que nos permite perceber que, em lógica proposicional, a definição de fórmula é recursiva.

De forma diferente de outras vertentes, a lógica proposicional não trabalha com quantificadores ou predicados sobre objetos. Isso a torna, de certo modo, menos geralista e mais limitada em descrever determinadas situações.

Para exemplificar o tipo de problema que será possível resolver utilizando lógica proposicional, considere o seguinte cenário:

Três irmãs - Ana, Maria e Cláudia - foram a uma festa com vestidos de cores diferentes. Uma vestia azul, a outra branco e a terceira preto.

Chegando à festa, o anfitrião perguntou quem era cada uma delas. A de azul respondeu: "Ana é a que está de branco". A de branco falou: "Eu sou Maria". Por fim, a de preto disse: "Cláudia é quem está de branco".

O anfitrião foi capaz de identificar corretamente quem era cada pessoa considerando que: (i) Ana sempre diz a verdade; (ii) Maria às vezes diz a verdade; e (iii) Cláudia nunca diz a verdade.

Pense um pouco sobre o problema e determine qual vestido cada uma das irmãs estava usando.

É possível concluir que Ana estava com o vestido preto, Cláudia com o branco e Maria com o azul. Contudo, utilizamos um raciocínio informal para chegar a esse resultado. A lógica proposicional permite que o método utilizado para chegar a essa conclusão seja formalizado, de forma que chega-se a uma *prova* de qual cor era o vestido de cada irmã.

Para apresentar o assunto, este capítulo foi dividido em outras três seções. Na primeira, são descritas as regras de inferência utilizadas nas provas de lógica proposicional. Na segunda, são apresentadas e discutidas provas mais complexas que utilizam essas mesmas regras. A terceira seção consiste em uma série de exercícios que contam também com um gabarito. Por fim, vale notar que, ao longo dessas seções, optou-se por abordar a lógica proposicional sob três pontos de vista: a partir da dedução natural, das tabelas verdade e do uso do Lean.

3.2 Regras de Inferência

Quando nos comunicamos, é habitual utilizar diferentes estruturas de linguagem que nos aproximem do sentido que queremos dar a alguma sentença. Na lógica proposicional, esses padrões são também amplamente utilizados e importantes para a construção de fórmulas e serão melhor aprofundados a seguir.

3.2.1 Implicação

A implicação é essencial para o condicionamento de sentenças. Se na língua falada utilizamos a estrutura "Se ... então"para nos referirmos a acontecimentos que dependem de outros, na lógica a ideia é muito semelhante.

Considere os seguintes exemplos:

Se Ana viajar para o Chile, comprará pesos chilenos
Se a família real não tivesse vindo ao Brasil, então o território se desintegraria

Apesar das duas sentenças terem a mesma estrutura "Se... então", podemos perceber que as duas tem suas diferenças. Em particular, chamamos a segunda proposição de implicação contrafactual, pois ela afirma como o mundo possivelmente seria, caso a realidade fosse diferentes do que ela realmente é. Esse assunto é discutido por filósofos há seculos e Spinoza e Saul Kripke são nomes de destaque nesses estudos. Entretanto, a lógica matemática se debruça mais especialmente na primeira sentença.

Dessa forma, tomando a primeira sentença como objeto de estudo, como podemos valorar essa implicação? Inicialmente, podemos atribuir a letra A ao evento "Ana viaja para o Chile"e B ao evento "Ana compra pesos chilenos". Temos, então, dois casos e uma tabela verdade correpondente:

Caso 1: Ana viaja para o Chile
Caso 2: Ana não viaja para o Chile

No primeiro caso, Ana viaja para o Chile e A é verdadeiro. Dessa forma, para a implicação ser verdadeira, B precisa ver também verdadeiro.

No segundo caso, Ana não viaja para o Chile e A é falso. Dessa forma, nada podemos dizer sobre o evento B e qualquer valor atribuído a ele torna a implicação verdadeira. A esse tipo de afirmação, chamamos de **verdadeira por**

A	**B**	**A** $\rightarrow$ **B**
V	V	V
V	F	F
F	V	V
F	F	V

vacuidade.

Vamos agora analisar a implicação contrária:

Se Ana comprar pesos chilenos, viajará para o Chile

Ainda analisando o evento A como "Ana viaja para o Chile"e B como "Ana compra pesos chilenos", temos a seguinte tabela verdade:

B	**A**	**B** $\rightarrow$ **A**
V	V	V
V	F	F
F	V	V
F	F	V

A segunda e a terceira linhas das duas tabelas evidenciam que, apesar de A e B terem o mesmo valor em ambas, a implicação tem uma valoração diferente. Ou seja, $A \rightarrow B \neq B \rightarrow A$. Esse resultado condiz com a nossa intuição, pois embora saibamos que se Ana viaja para o Chile, ela comprará pesos chilenos, não podemos afirmar com certeza que uma vez que Ana comprou pesos chilenos, ela viajará para o Chile. Quem sabe Ana seja uma colecionadora de moedas internacionais?

Nesse contexto, a implicação é protagonista da chamada "regra da exclusão da implicação"ou *Modus Ponens* ("A maneira que afirma afirmando"), ou seja:

Se Ana viajar para o Chile, comprará pesos chilenos
Ana viajou para o Chile
Ana comprou pesos chilenos

Escrito em dedução natural como:

$$\frac{A \rightarrow B \qquad \text{A}}{\text{B}}$$

E com seu correspondente no Lean, sendo:

```
1 variables A B: Prop
2 section
3 variable h1 : A → B
4 variable h2 : A
```

```
5
6 example : B := h1 h2
7 end
```

Por outro lado, existe também a "regra de inclusão da implicação". Uma vez que temos uma variável A e conseguimos derivar uma variável B, dizemos que **A implica em B**. Essa regra é utilizada nas árvores de dedução natural da seguinte forma:

$$\dfrac{\begin{array}{c}A\\ \vdots \\ B\end{array}}{A \to B}$$

Enquanto no Lean:

```
1 variables A B : Prop
2
3 example : A → B :=
4 assume h : A,
5 show B, from sorry
```

Note que no caso acima não é necessário utilizar o comando *show* para indicar que se quer chegar em B. O Lean consegue inferir isso a partir do que é escrito no lugar do *sorry* e também a partir do que é colocado no *example*. Contudo, o *show* pode ser um facilitador na hora de escrever provas no Lean, já que evidencia o que se busca provar naquela parte.

3.2.2 Se e somente se

Já vimos anteriormente que $A \to B \neq B \to A$. Entretanto, em muitos casos na Matemática, conseguimos chegar na igualdade dessas duas implicações e precisamos expressar a estrutura da linguagem falada "Se, e somente se". Dessa forma, faz-se uso do símbolo chamado de "bi-implicação"e representado por $\iff$.

Poderíamos também utilizar a formalização $A \to B \wedge B \to A$, mas, por questões ligadas à praticidade da abreviação, é preferível o uso do novo símbolo apresentado.

Para entender melhor a interpretação dessa regra, usaremos o exemplo abaixo:

Ana viajará para o Chile se, e somente se, comprar pesos chilenos

Novamente construíremos a bi-implicação tratando o evento A como sendo "Ana viaja ao Chile"e B como "Ana compra pesos chilenos". A tabela verdade então será:

Para melhor visualização dos resultados, vamos também mostrar uma tabela verdade que utiliza a definição da bi-implicação com o "e":

Dessa forma, temos os casos:

A	**B**	**A** $\iff$ **B**
V	V	V
V	F	F
F	V	F
F	F	V

A	**B**	**A** $\iff$ **B**	**B** $\iff$ **A**	(**B** $\iff$ **A**) $\wedge$ (**A** $\iff$ **B**)
V	V	V	V	V
V	F	F	V	F
F	V	F	V	F
F	F	V	V	V

Caso 1: Ana viaja para o Chile
Caso 2: Ana não viaja para o Chile
Caso 3: Ana compra pesos chilenos
Caso 4: Ana não compra pesos chilenos

Assim, com essa sentença, sabemos que o caso 1 acontece se, e somente se, o caso 3 acontece, e o caso 2 acontece se, e somente se, o caso 4 também acontece.

Em dedução natural a regra da inclusão da bi-implicação evidencia a necessidade de possuirmos duas implicações verdadeiras. Escrevemos essa regra como:

$$\frac{\begin{array}{cc} A & B \\ \vdots & \vdots \\ B & A \end{array}}{A \iff B}$$

No Lean, temos o comando "iff.intro" que introduz o símbolo dado a verdade das duas implicações:

```
1 variables A B: Prop
2
3 example : A ↔ B :=
4 iff.intro
5   (assume h : A,
6     show B, from sorry)
7   (assume h : B,
8     show A, from sorry)
```

Para a exclusão, a regra em dedução natural é muito semelhante à regra da exclusão da implicação:

$$\frac{A \iff B \qquad A}{B}$$

$$\frac{A \iff B \qquad B}{A}$$

O seu correspondente no Lean é feito pelos comandos de eliminação à direita e à esquerda:

```
variables A B: Prop
section
  variable h1 : A ↔ B
  variable h2 : A

  example : B := iff.elim_left h1 h2
end

section
  variable h1 : A ↔ B
  variable h2 : B

  example : A := iff.elim_right h1 h2
end
```

3.2.3 Conjunção

A regra da conjunção se refere ao uso do "e" na linguagem informal, de forma que juntamos duas informações. Podemos ter frases como:

Santiago é a capital do Chile e o deserto do Atacama está localizado no Chile.

Ao mesmo tempo, podemos utilizar o "e" para conectar duas informações que nem mesmo possuem relação entre si. Por exemplo, podemos dizer que:

Santiago é a capital do Chile e Bolsonaro é o presidente do Brasil.

Quando utilizamos o "e", representado pelo símbolo $\wedge$ na lógica, o que de fato importa é estarmos unindo duas informações que são verdadeiras. Isso fica claro na tabela verdade abaixo, na qual é possível ver que quando A ou quando B são falsos, $A \wedge B$ é falso:

A	**B**	**A** $\wedge$ **B**
V	V	V
V	F	F
F	V	F
F	F	F

Dessa forma, na dedução natural, podemos introduzir um "e", quando temos A e também temos B. Assim, se A e B forem verdadeiros, $A \wedge B$ também vai ser.

$$\frac{\text{A} \qquad \text{B}}{A \wedge B}$$

No Lean, representamos essa operação pela função *and.intro*, conforme o exemplo abaixo:

```
1 variables A B : Prop
2
3 example (h1 : A) (h2 : B) : A ∧ B :=
4 and.intro h1 h2
```

Seguindo esse raciocínio, é possível observar que sempre que temos $A \wedge B$, teremos A e B separadamente. Essa é a operação chamada de exclusão do "e".

Para excluir o B de, por exemplo, $A \wedge B$, temos a chamada exclusão pela esquerda, conforme descrita abaixo:

$$\frac{A \wedge B}{\text{A}}$$

No Lean, essa operação pode ser feita utilizando a função $and.left$, conforme o código a seguir:

```
1 variables A B : Prop
2
3 example (h1 : A ∧ B): A :=
4 and.left h1
```

Alternativamente, é possível realizar a mesma operação da seguinte forma:

```
1 variables A B : Prop
2
3 example (h1 : A ∧ B): A :=
4 h1.left
```

Para excluir o A, temos a chamada exclusão pela direita:

$$\frac{A \wedge B}{\text{B}}$$

No Lean, essa operação é realizada utilizando a função $and.right$, de maneira semelhante ao que foi descrito anteriormente para o $and.left$.

3.2.4 Disjunção

A regra da disjunção se refere ao uso do "ou". Na linguagem informal, podemos utilizá-lo para expressar situações excludentes, como da seguinte forma:

Alexandre vai viajar para o Atacama ou Alexandre vai viajar para Santiago.

Nesse caso, é possível que uma pessoa compreenda que somente uma das duas situações vai ocorrer, ou seja, que Alexandre somente vai para um dos dois lugares no Chile. Contudo, quando utilizamos o "ou" na lógica, representado pelo símbolo $\vee$, também estamos levando em consideração situações nas quais

ambas as proposições são verdadeiras. Dessa forma, conforme indicado na tabela verdade a seguir, a sentença formada pelo "ou" será verdadeira quando somente A for verdadeiro, quando somente B for verdadeiro ou quando ambos forem verdadeiros.

A	**B**	**A** $\vee$ **B**
V	V	V
V	F	V
F	V	V
F	F	F

Não só o "ou" formará uma sentença verdadeira quando uma ou ambas as situações forem verdadeiras, como também quando ambas as situações forem verdadeiras, mas não possuírem nenhuma relação entre si. Por exemplo, a seguinte sentença é verdadeira:

Santiago é a capital do Chile ou Bolsonaro é o presidente do Brasil.

É verdadeiro que Santiago é a capital do Chile e que Bolsonaro é o presidente do Brasil. Logo, toda a sentença é verdadeira, apesar de não soar de forma natural na linguagem informal.

Na dedução natural, como o "ou" é verdadeiro mesmo que somente um de seus elementos seja verdadeiro, introduzimos o $\vee$ tendo somente A ou somente B. Logo, pode-se formar $A \vee B$ da seguinte forma:

$$\frac{\text{A}}{A \vee B}$$

No Lean, essa operação de introdução do "ou" é realizada utilizando a função *or.inl*. Essa função indica que queremos adicionar o A do lado esquerdo do $\vee$ (por isso, "or include left"). Dessa forma, teremos o seguinte código:

```
1 variables A B : Prop
2
3 example (h1 : A): A ∨ B :=
4 or.inl h1
```

Alternativamente, é possível realizar a introdução do $\vee$ a partir do B:

$$\frac{\text{B}}{A \vee B}$$

Nesse caso, como o B está sendo adicionado à direita do $\vee$ utilizamos a função *or.inr* no Lean (ou seja, "or include right").

```
1 variables A B : Prop
2
3 example (h1 : B): A ∨ B :=
4 or.inr h1
```

É possível notar que, no caso acima, não foi necessário indicar explicitamente para o Lean que o A seria adicionada à esquerda do B. Isso acontece pois o Lean consegue inferir o que vai ser adicionado, de acordo com o que se quer provar.

Para excluir o "ou" de $A \vee B$ é preciso estruturar uma árvore de dedução natural de forma que A e B resultem em um mesmo resultado C. Chegando nesse C, é possível substituir o $A \vee B$ pelo C. A seguinte árvore de dedução natural demonstra essa estrutura:

$$\dfrac{A \vee B \qquad \begin{array}{c} A \\ \vdots \\ C \end{array} \qquad \begin{array}{c} B \\ \vdots \\ C \end{array}}{C}$$

No Lean, para excluir o "ou", utiliza-se a função *or.elim*. Ao lado do *or.elim* é necessário inserir a hipótese contendo o "ou" que deseja-se eliminar. Em seguida, deve-se abrir dois parênteses. Esses dois parênteses serão equivalentes às duas colunas contendo, respectivamente, A e B na árvore de dedução natural acima. Dessa forma, um parênteses começa com o *assume* de A e o outro com o *assume* de B. O objetivo é que a partir deles chegue-se à hipótese C. O código abaixo mostra a estrutura de como ficaria uma operação de eliminação do "ou" (o "sorry" nas provas abaixo representa a etapa em que se prova que a partir de A é possível chegar em C) :

```
1 variables A B C D : Prop
2
3 example (h1: A ∨ B): C :=
4 or.elim h1
5 (assume h2: A, sorry)
6 (assume h2: B, sorry)
```

Como exemplo mais concreto, sem o uso do "sorry", é possível observar a prova abaixo, na qual utiliza-se as hipóteses $A \to C$ e $B \to C$ para se chegar à C a partir de $A \vee B$:

```
1 variables A B C D : Prop
2
3 example (h1 : A → C) (h2 : B → C) (h3: A ∨ B): C :=
4 or.elim h3
5     (assume h4: A,
6     show C, from h1 h4)
7     (assume h4: B,
8     show C, from h2 h4)
```

3.2.5 Negação

A negação de A é representada em símbolos por $\neg A$. Mostrar que $\neg A$ ocorre, em termos lógicos, é o mesmo que mostrar que A leva a uma contradição.

Em dedução natural, a seguinte estrutura representa a regra de introdução da negação, na qual $\perp$ é o símbolo para falso, contradição ou absurdo:

$$\dfrac{\dfrac{}{\mathrm{A}}\,(1)}{\begin{array}{c}\vdots\\ \perp\end{array}}\,(1)\ \neg\ \mathrm{I}$$

Esta é uma outra forma de raciocínio hipotético, similar a utilizada em "se ... então". Começamos supondo A. Em seguida, continuamos a prova aplicando as regras já apresentadas, representadas pelos três pontinhos na árvore de dedução natural, até chegarmos a uma contradição.

A pergunta a se fazer em seguida é: de que forma uma contradição aparece numa prova, ou seja, como a encontramos? Suponha, por exemplo, que ao construir uma prova temos uma hipótese A que nos leva a concluir que uma outra hipotése B ocorre ao mesmo tempo que $\neg B$ também ocorre. Nesse caso, se temos B e $\neg B$, então temos uma contradição. Em dedução natural, representamos essa ideia (a regra de eliminação da negação) por:

$$\dfrac{\neg B \qquad \mathrm{B}}{\perp}\ \neg\ \mathrm{E}$$

A tabela verdade a seguir reforça o fato de que é impossível que uma proposição e sua negação sejam ambas verdadeiras ao mesmo tempo.

A	$\neg A$
V	F
F	V

Saber negar definições e saber negar sentenças é importante, pois várias ideias matemáticas advêm dessas negações. Abaixo temos as tabelas-verdade para a negação de sentenças conjuntivas e disjuntivas:

P	**Q**	¬ **(P** ∨ **Q)**	¬ **P**	¬ **Q**	¬ **P** ∧¬ **Q**
V	V	F	F	F	F
V	F	F	F	V	F
F	V	F	V	F	F
F	F	V	V	V	V

P	**Q**	¬ **(P** ∧ **Q)**	¬ **P**	¬ **Q**	¬ **P** ∧¬ **Q**
V	V	F	F	F	F
V	F	V	F	V	V
F	V	V	V	F	V
F	F	V	V	V	V

Ao observar as tabelas anteriores, você pode constatar que:
¬ (P ∨ Q) ≡ ¬ P ∧¬ Q e ¬ (P ∧ Q) ≡ ¬P ∨¬ Q

As equivalências acima são chamadas de Leis de De Morgan, e significam que a negação da disjunção (de duas sentenças) é a conjunção das negações e a negação da conjunção (de duas sentenças) é a disjunção das negações (destas sentenças). (Nos exercícios pedimos que você prove algumas equivalências como essas).

Ao utilizarmos o Lean, as regras de inferência de eliminação e introdução da negação podem ser obtidas através dos comandos a seguir:

```
1 variables p q : Prop
2 -- show false, from ...
3
4 example (h1 : P → Q) (h2 : ¬Q) : ¬P :=
5 assume hp : P,
6     show false, from h2 (h1 hp)
```

Note que a hipótese que contém a negação vem primeiro após o $from$, sendo Q o resultado do uso da regra de eliminação da implicação. Além disso, o símbolo $\neg Q$ é escrito como `\not Q`. Na biblioteca padrão do Lean, $\neg P$ é, na verdade, uma abreviação de $P \to false$, ou seja, o fato de que P implica em uma contradição.

Uma vez apresentado o símbolo para *falso*, a seguir apresentamos o símbolo para *verdadeiro*. No entanto, *verdadeiro* ou *true* não possui regra de eliminação, apenas uma regra de introdução: $true.intro : true$, às vezes abreviado como $trivial : true$. Em outras palavras, verdadeiro é simplesmente verdadeiro e tem uma prova canônica, trivial.

$$\frac{}{\top}$$

E para $\bot$, quais regras regem o mundo dos absurdos? A regra de eliminação para $\bot$ possui em latim o nome de *ex falso sequitur quodlibet*, que significa que "a partir de uma contradição, qualquer coisa segue". É especialmente difícil traçar um paralelo na linguagem natural para essa regra de eliminação, já que parece não haver sentido em dizer que podemos concluir qualquer coisa a partir de uma contradição. Contudo, essa regra existe na lógica e é representada da seguinte forma na dedução natural (é a chamada exclusão do absurdo):

$$\frac{\bot}{\mathrm{A}}\ \bot\ \mathrm{E}$$

Uma discussão interessante surge a partir dessa regra, que não abordaremos com profundidade pois fogem do escopo desse livro. Considere a seguinte sentença:

- Para todo número natural, se n é par, então $n + 1$ é ímpar.

Gostaríamos que essa sentença fosse verdadeira para todo n. Sendo ela verdadeira para todos os naturais, podemos tomar um valor particular de n, por exemplo $n = 3$. Nessa sentença condicional, ambos antecedente e sucedente

são falsos. O fato de estarmos comprometidos em dizer que essa afirmação é verdadeira mostra que devemos ser capazes de provar que a afirmação $3 + 1$ é ímpar decorre da afirmação falsa de que 3 é par. O *ex falso* encapsula ordenadamente esse tipo de inferência.

Observe atentamente outra vez a tabela verdade para a implicação, note que em qualquer um dos casos onde o antecedente é falso, a sentença de implicação é verdadeira (por vacuidade).

Como definimos $\neg A$ como $A \to \bot$, então as regras para introdução e eliminação da negação nada mais são do que as regras de introdução e eliminação da implicação, respectivamente.

3.2.6 Prova por contradição

Existe um estilo de fazer matemática conhecido como "matemática construtiva"que nega a equivalência de $\neg\neg A$ e A. Desse modo, uma demonstração de que algo é verdadeiro deveria fornecer evidências explícitas de que uma declaração é falsa, em vez de evidências de que ela não pode ser falsa. Uma prova construtiva é aquela que realmente lhe diz como encontrar o objeto que se afirma existir.

A prova por contradição é informalmente utilizada para se referir a duas regras diferentes de inferência. Sendo elas:

- Para provar $\neg P$ é suficiente assumir P e derivar uma contradição;
- Para provar P, basta assumir $\neg P$ e derivar uma contradição.

É importante que esteja bastante clara a diferença entre as duas regras. No primeiro argumento, uma negação é introduzida na conclusão, enquanto que no segundo, ela é eliminada da hipótese. A regra de inferência que conclui com a sentença positiva P é chamada de redução ao absurdo (em latim, *reductio ad absurdum*). De fato, a regra é equivalente ao princípio $\neg\neg A \leftrightarrow A$.

A primeira vista é difícil perceber claramente porque em uma delas temos uma prova construtiva e na outra uma que utiliza do raciocínio clássico. Na lógica clássica, além de todas as regras de inferência já expostas neste livro, temos o princípio do terceiro excluído que diz que uma proposição A ou é verdadeira ou é falsa, não existindo uma terceira opção. Assim, se é impossível uma proposição P ser falsa, logo ela só pode ser verdadeira.

Na linguagem natural, comumente tomamos a frase "Maria não não estava vestida de azul"como um modo redundante e agramatical de dizer que "Maria estava vestida de azul". Ao mesmo tempo que em "Eu não vi ninguém"as duas palavras negativas não se anulam.

Em dedução natural, a redução ao absurdo é expressa do seguinte modo, no qual a hipótese $\neg A$ é cancelada no final da inferência:

$$\dfrac{\dfrac{}{\neg A}\,(1)}{\dfrac{\vdots}{\dfrac{\bot}{A}\,(1)\ \text{R.A.A}}}$$

As regras de introdução e eliminação que vimos até agora no Lean são todas construtivas, ou seja, refletem um entendimento computacional dos conectivos lógicos com base na correspondência das proposições como tipos. Para utilizar os princípios da lógica clássica, você deve abrir o modo clássico com o comando *openclassical* no início do seu arquivo ou em qualquer lugar antes de usá-lo.

```
1 open classical
2
3 variable P : Prop
4 #check em P
```

Na linha 4, o comando `#check` permite verificar se expressões que escrevemos estão bem formadas e tamnbém qual tipo de objeto eles denotam. Para o exemplo acima obtemos como output:

```
4:0: information: check result
em P : P ∨ ¬P
```

No exemplo abaixo, provamos P a partir de $\neg\neg P$ utilizando a redução ao absurdo. Note que não é preciso escrever explicitamente ao final o que queremos provar, uma vez que o Lean, no caso da prova por contradição, infere que o resultado é $\neg\neg P \to P$ a partir das hipóteses assumidas.

```
1 open classical
2
3 variable P : Prop
4
5 example (h : ¬¬ P) : P :=
6 by_contradiction
7   (assume h1 : ¬ P,
8     show false, from h h1)
```

Em dedução natural, a seguinte árvore corresponderia ao exemplo acima:

$$\dfrac{\dfrac{\dfrac{}{\neg P}\,(1) \qquad \neg\neg P}{\bot}}{\neg\neg P \to P}\,(1)$$

3.2.7 Problema dos vestidos

Temos agora todas as ferramentas necessárias para começar a formalização do primeiro problema apresentado no capítulo. Relembrando o enunciado, temos que:

Três irmãs - Ana, Maria e Cláudia - foram a uma festa com vestidos de cores diferentes. Uma vestia azul, a outra branco e a terceira preto.

Chegando à festa, o anfitrião perguntou quem era cada uma delas. A de azul respondeu: "Ana é a que está de branco". A de branco falou: "Eu sou Maria". Por fim, a de preto disse: "Cláudia é quem está de branco".

O anfitrião foi capaz de identificar corretamente quem era cada pessoa considerando que: (i) Ana sempre diz a verdade; (ii) Maria às vezes diz a verdade; e (iii) Cláudia nunca diz a verdade.

Para começar a análise desse problema, precisamos identificar quais proposições serão representadas pelas letras. Como nossa principal meta é descobrir quem veste cada cor, usaremos: AA (Ana veste azul), AB (Ana veste branco), AP (Ana veste preto), MA (Maria veste azul), MB (Maria veste branco), MP (Maria veste preto), CA (Cláudia veste azul), CB (Cláudia veste branco) e CP (Cláudia veste preto).

Antes de começar a analisar as regras impostas pelo problema, por bom senso, sabemos que nenhuma das irmãs veste mais de uma cor de vestido. Então temos que:

- $AA \vee (AB \vee AP)$
- $MA \vee (MB \vee MP)$
- $CA \vee (CB \vee CP)$
- $(AA \rightarrow (\neg\, AB \wedge \neg\, AP)) \wedge (AB \rightarrow (\neg\, AA \wedge \neg\, AP)) \wedge (AP \rightarrow (\neg\, AB \wedge \neg\, AA))$
- $(MA \rightarrow (\neg\, MB \wedge \neg\, MP)) \wedge (MB \rightarrow (\neg\, MA \wedge \neg\, MP)) \wedge (MP \rightarrow (\neg\, MB \wedge \neg\, MA))$
- $(CA \rightarrow (\neg\, CB \wedge \neg\, CP)) \wedge (CB \rightarrow (\neg\, CA \wedge \neg\, CP)) \wedge (CP \rightarrow (\neg\, CB \wedge \neg\, CA))$

No começo do problema, nos é dito que as três irmãs foram para a festa com cores diferentes de vestido. Logo, sabemos que, se uma irmã foi de determinada cor, as outras duas certamente não vestiram aquela cor. Assim:

- $(AA \rightarrow (\neg\, MA \wedge \neg\, CA)) \wedge (AB \rightarrow (\neg\, MB \wedge \neg\, CB)) \wedge (AP \rightarrow (\neg\, MP \wedge \neg\, CP))$
- $(MA \rightarrow (\neg\, AA \wedge \neg\, CA)) \wedge ((MB \rightarrow (\neg\, AB \wedge \neg\, CB)) \wedge (MP \rightarrow (\neg\, AP \wedge \neg\, CP)))$
- $(CA \rightarrow (\neg\, AA \wedge \neg\, MA)) \wedge (CB \rightarrow (\neg\, AB \wedge \neg\, MB)) \wedge (CP \rightarrow (\neg\, AP \wedge \neg\, MP))$

Vamos agora analisar as sentenças ditas pelas irmãs, considerando as suas características individuais:

Se Ana está de azul, sabemos que a sua sentença é verdadeira e, portanto, Ana está de branco. Escrevemos então:

- AA $\rightarrow$ AB

Se Ana está de branco, sabemos que a sua sentença é verdadeira e, portanto, Maria está de branco. Escrevemos então:

- AB $\rightarrow$ MB

Se Ana está de preto, sabemos que a sua sentença é verdadeira e, portanto, Cláudia é quem está de branco. Escrevemos então:

- AP $\rightarrow$ CB

Se Claudia está de azul, sabemos que a sua sentença é falsa e, portanto, Ana não está de branco. Escrevemos então:

- CA $\rightarrow \neg$ AB

Se Claudia está de branco, sabemos que a sua sentença é falsa e, portanto, Maria não está de branco. Escrevemos então:

- CB $\rightarrow \neg$ MB

Se Claudia está de preto, sabemos que a sua sentença é falsa e, portanto, Cláudia não está de branco. Escrevemos então:

- CP $\rightarrow \neg$ CB

Por fim, sabemos que Ana não pode estar de branco, pois, caso estivesse, estaria mentindo em sua sentença. Escrevemos então:

- $\neg$ AB

Por meio das fórmulas descritas acima, convidamos o leitor, nos exercícios, a chegar na prova da resposta encontrada no início deste capítulo.

3.3 Exemplos adicionais de Dedução Natural

Nesta seção serão apresentados exemplos adicionais de provas de dedução natural e seus equivalentes no Lean. Também será explicado como construir essas árvores do zero, partindo somente do que se quer provar.

1. **Prova de A $\rightarrow$ C a partir de A $\rightarrow$ B e B $\rightarrow$ C:**

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{A}}\,(1) \qquad A \rightarrow B}{\mathrm{B}} \qquad B \rightarrow C}{\mathrm{C}}}{A \rightarrow C}\,(1)$$

Como construir essa prova? Primeiro, começamos pelo que se quer provar: $A \to C$. Podemos escrever isso na última linha, já que é onde queremos chegar. Como a prova é de uma implicação, sabemos que na linha logo acima dessa vamos chegar no C:

$$\frac{\mathrm{C}}{A \to C}$$

Agora podemos considerar as hipóteses. Como a prova é de $A \to C$, vamos utilizar o A em algum momento na árvore. Além disso, pelo enunciado, sabemos que vamos também utilizar o $A \to B$ e $B \to C$. Então teremos uma estrutura razoavelmente parecida com essa:

$$\dfrac{\dfrac{\overline{\mathrm{A}}^{\,(1)} \qquad A \to B \qquad B \to C}{C}}{A \to C}^{\,(1)}$$

A partir disso, é possível observar com mais facilidade quais regras de dedução natural podem ser aplicadas ao problema. No caso, observamos que temos A e $A \to B$. Logo, é possível obter B.

$$\dfrac{\dfrac{\dfrac{\overline{\mathrm{A}}^{\,(1)} \qquad A \to B}{B} \qquad B \to C}{C}}{A \to C}^{\,(1)}$$

Por fim, obtemos C a partir de B e de $B \to C$, formando a árvore apresentada de início.

No Lean, essa prova poderia ser feita da seguinte forma:

```
1 variables A B C: Prop
2 example (h1: A → B) (h2: B → C): A → C :=
3 assume h3: A,
4 have h4: B, from h1 h3,
5 h2 h4
```

Para escrevê-la, adicionamos ao lado de *example* as hipóteses que não vão ser descartadas. No caso, como o enunciado diz que vamos utilizar $A \to B$ e $B \to C$ para realizar a prova, adicionamos os dois depois de *example*. Em seguida, adicionamos : e o que queremos provar, seguido de :=.

Para o teor da prova, devemos considerar que a ordem no qual escrevemos no Lean importa. Logo, devemos seguir razoavelmente a ordem da árvore de dedução natural. Na árvore desse problema, é possível observar que a prova começa com o A (do $A \to C$). Como essa é uma hipótese que vai ser descartada, escrevemos ela com o *assume*. Como a hipótese $A \to B$ já está escrita no *example*, podemos utilizá-la para encontrar B. Nesse caso, utilizamos o *have* para atribuir o nome de uma variável ao B. Isso permite uma organização maior

e evita que, em provas longas, tenha que se repetir muitas vezes como encontrar uma determinada variável. Contudo, não é necessário utilizar o *have*. Nesse caso, por exemplo, somente utilizamos o B uma vez, para aplicá-lo ao $B \to C$. Logo, poderíamos ter escrito a prova acima no modo termo do Lean da seguinte forma:

```
1 variables A B C: Prop
2 example (h1: A → B) (h2: B → C): A → C :=
3 assume h3: A,
4 h2 (h1 h3)
```

Ou ainda no modo táticas como:

```
1 variables A B C: Prop
2 example (h1: A→B) (h2: B→C): A→C :=
3 begin
4 intros,
5 exact h2 (h1 a)
6 end
```

A prova acima também pode ser intuitivamente pensada como (A → B) ∧ (B → C) → (A → C). Nesse caso, teria-se a seguinte árvore:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{A}}\,(1) \quad \dfrac{\dfrac{}{(A \to B) \wedge (B \to C)}\,(2)}{A \to B}}{\text{B}} \quad \dfrac{\dfrac{}{(A \to B) \wedge (B \to C)}\,(2)}{B \to C}}{C}}{\dfrac{A \to C}{(A \to B) \wedge (B \to C) \to (A \to C)}\,(2)}\,(1)$$

Para construir a árvore acima, poderia-se adotar essa estratégia: primeiro, começamos pelo que se quer provar $((A \to B) \wedge (B \to C) \to (A \to C))$. Como vamos chegar em $A \to C$, podemos colocá-lo logo acima do que queremos provar. Ainda assim, continuamos com uma implicação, então podemos inserir o C antes do $A \to C$.

Como hipótese, teremos o o A (de $A \to C$) e o $(A \to B) \wedge (B \to C)$ (vindo do resultado final: $(A \to B) \wedge (B \to C) \to (A \to C)$). Assim, teremos uma estrutura dessa forma:

$$\dfrac{\dfrac{\dfrac{}{\text{A}}\,(1) \quad \dfrac{}{(A \to B) \wedge (B \to C)}\,(2)}{\vdots \qquad \vdots}}{\dfrac{C}{A \to C}\,(1)}}{(A \to B) \wedge (B \to C) \to (A \to C)}\,(2)$$

Como sabemos que queremos chegar em um C, podemos utilizar o $B \to C$ para isso ($B \to C$ pode ser obtido a partir da operação de exclusão do $\wedge$ na hipótese de número 2) . Contudo, é necessário ter B para realizar essa operação.

O B pode ser obtido a partir do A e do $A \to B$. Logo, é possível construir a árvore.

No Lean essa prova poderia ser escrita no modo termo da seguinte forma:

```
variables A B C: Prop
example: ((A → B) ∧ (B → C)) → (A → C) :=
assume h1: (A → B) ∧ (B → C),
assume h2: A,
have h3: B, from (and.left h1) h2,
(and.right h1) h3
```

Ou no modo táticas:

```
variables A B C: Prop
example: ((A → B) ∧ (B → C)) → (A → C) :=
begin
intros,
have h1: A → B, from a.left,
have h2: B → C, from a.right,
exact h2 (h1 a_1)
end
```

Nas duas árvores de deduçao natural acima, podemos observar a utilização de números em determinadas hipóteses. Utilizamos esses para evidenciar onde descartamos as hipóteses marcadas.

2. Prova de $Q \wedge S$ a partir de $(P \wedge Q) \wedge R$ e $S \wedge T$:

$$\dfrac{\dfrac{\dfrac{(P \wedge Q) \wedge S}{P \wedge Q}}{\mathrm{Q}} \qquad \dfrac{S \wedge T}{\mathrm{S}}}{Q \wedge S}$$

Nesse caso, não descartamos nenhuma hipótese pois assumimos $(P \wedge Q) \wedge R$ e $S \wedge T$ como verdade e apenas derivamos a prova.

Para construir essa prova, partimos também de onde queremos chegar: $Q \wedge S$. Para formar um $\wedge$, precisamos de Q e de S separadamente. Logo, podemos escrevê-los acima do $Q \wedge S$. Pelo enunciado, sabemos que vamos usar como hipótese: $(P \wedge Q) \wedge R$ e $S \wedge T$. Assim, teremos a seguinte estrutura:

$$\dfrac{\begin{array}{cc} (P \wedge Q) \wedge S & S \wedge T \\ \vdots & \vdots \\ Q & \mathrm{S} \end{array}}{Q \wedge S}$$

É possível obter o S a partir de qualquer uma das hipóteses. O Q só é possível obter a partir de $(P \wedge Q) \wedge S$. Logo, a partir de uma série de operações de exclusão do $\wedge$, constrói-se a parte restante da prova.

No Lean, essa prova poderia ser feita no modo termo da seguinte forma:

```
variables P Q R S T: Prop
example (h1: (P ∧ Q) ∧ R) (h2: S ∧ T): Q ∧ S :=
have h3: Q, from and.right (and.left h1),
have h4: S, from and.left h2,
and.intro h3 h4
```

E no modo táticas:

```
variables P Q R S T: Prop
example (h1: (P ∧ Q) ∧ R) (h2: S ∧ T): Q ∧ S :=
begin
intros,
have h1: Q, from (h1.left).right,
have h2: S, from h2.left,
exact and.intro h1 h2
end
```

3. Prova de $(A \to (B \to C)) \to (A \wedge B \to C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \to (B \to C)}\,(2) \quad \dfrac{\dfrac{}{A \wedge B}\,(1)}{\mathrm{A}}}{B \to C} \qquad \dfrac{\dfrac{}{A \wedge B}\,(1)}{\mathrm{B}}}{\mathrm{C}}}{\dfrac{A \wedge B \to C}{(A \to (B \to C)) \to (A \wedge B \to C)}\,(2)}\,(1)$$

A construção dessa prova em dedução natural pode ser realizada de modo semelhante às provas descritas anteriormente. No Lean, ela pode ser escrita no modo termo da seguinte forma:

```
variables A B C: Prop
example: (A → (B → C)) → (A ∧ B → C) :=
assume h₁: A → (B → C),
assume h₂: A ∧ B,
show C, from h₁ (h₂.left) h₂.right
```

Note que para rotular as hipóteses é possível utilizar números subscritos, que são digitados a partir de uma barra invertida. Como exemplo, é possível escrever h_1 digitando `h \ 1`.

No modo táticas, a prova acima pode ser escrita como:

```
variables A B C: Prop
example: (A → (B → C)) → (A ∧ B → C) :=
begin
intros,
have h₁ :A, from a_1.left,
have h₂ :B, from a_1.right,
exact (a h₁) h₂
end
```

4. Prova de $A \wedge B \iff B \wedge A$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B}\,(1)}{\mathrm{B}} \quad \dfrac{\dfrac{}{A \wedge B}\,(1)}{\mathrm{A}}}{B \wedge A} \quad \dfrac{\dfrac{\dfrac{}{B \wedge A}\,(2)}{\mathrm{A}} \quad \dfrac{\dfrac{}{B \wedge A}\,(2)}{\mathrm{B}}}{A \wedge B}}{A \wedge B \iff B \wedge A}\,(1,2)$$

Para construir essa prova, partimos do $A \wedge B \iff B \wedge A$ ao final da prova. Para obtê-lo, precisamos partir de $B \wedge A$ e chegar no $A \wedge B$ e também partir de $A \wedge B$ e chegar em $B \wedge A$. Sabendo que chegaremos nos dois, podemos escrevê-los na linha acima de $A \wedge B \iff B \wedge A$. Sabendo que partiremos também dos dois, podemos escrevê-los como hipóteses. Teremos, assim, uma estrutura como essa:

$$\dfrac{\dfrac{\dfrac{}{A \wedge B}\,(1)}{\vdots}}{B \wedge A} \quad \dfrac{\dfrac{\dfrac{}{B \wedge A}\,(2)}{\vdots}}{A \wedge B}}{A \wedge B \iff B \wedge A}\,(1,2)$$

Para formar $A \wedge B$ e $B \wedge A$ precisamos de A e B separadamente. Estes podem ser obtidos a partir das hipóteses. Logo, repetindo as hipóteses e realizando a exclusão do $\wedge$, formamos a árvore final.

No Lean, essa prova poderia ser construída no modo termo da seguinte forma:

```
1 variables A B C: Prop
2 example: A ∧ B ↔ B ∧ A :=
3 iff.intro
4     (assume h1: A ∧ B,
5     and.intro (and.right h1) (and.left h1))
6     (assume h3: B ∧ A,
7     and.intro (and.right h3) (and.left h3))
```

E no modo táticas:

```
1 variables A B C: Prop
2 example: A ∧ B ↔ B ∧ A :=
3 begin
4 apply iff.intro
5      (assume h₁: A ∧ B,
6     and.intro (and.right h₁) (and.left h₁))
7     (assume h₂: B ∧ A,
8     and.intro (and.right h₂) (and.left h₂))
9 end
```

5. Prova de $A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}\,(2)}{B \vee C}}{\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}\,(2)}{\mathrm{A}}\quad\dfrac{}{\mathrm{B}}\,(1)}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)}}\quad\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge (B \vee C)}\,(2)}{\mathrm{A}}\quad\dfrac{}{\mathrm{C}}\,(1)}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{(A \wedge B) \vee (A \wedge C)}\,(1)}{A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)}\,(2)$$

Para escrever essa prova, podemos partir, como nas outras, do objetivo final: $A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)$. Como estamos provando uma implicação, vamos chegar em $(A \wedge B) \vee (A \wedge C)$ na linha anterior. Observamos também que $A \wedge (B \vee C)$ será uma hipótese. Teremos, então, uma estrutura como essa:

$$\dfrac{\dfrac{}{A \wedge (B \vee C)}\,(2)}{\begin{array}{c}\vdots\\ (A \wedge B) \vee (A \wedge C)\end{array}}{A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)}\,(2)$$

Como proceder a partir dessa estrutura? Para formar o $(A \wedge B) \vee (A \wedge C)$ será necessário o $A \wedge B$ isoladamente ou o $A \wedge C$ (a partir de qualquer um dos dois é possível realizar a introdução do $\vee$ e assim chegar em $(A \wedge B) \vee (A \wedge C)$). Independente de qual dos dois serão utilizados, nota-se que, para formar o "e", será necessário: o A e também o B ou o C. O A pode ser facilmente obtido a partir da hipótese. Para obter B ou C isoladamente, será necessário realizar a exclusão do $B \vee C$ contido no "e" da hipótese. Nota-se que, para realizar a exclusão do "ou", é necessário considerar B e C como hipótese. Assim, podemos formar tanto $A \wedge B$, quanto $A \wedge C$. Dessa forma, é possível chegar no resultado final descrito acima.

No Lean, essa prova poderia ser construída no modo termo da seguinte forma:

```
1  variables A B C: Prop
2  example: (A ∧ (B ∨ C)) → ((A ∧ B) ∨ (A ∧ C)) :=
3  assume h1: A ∧ (B ∨ C),
4  have h2: B ∨ C, from and.right h1,
5  have h4: A, from and.left h1,
6  or.elim h2
7      (assume h3: B,
8      or.inl (and.intro h4 h3))
9      (assume h3: C,
10     or.inr (and.intro h4 h3))
```

E no modo táticas:

```
1  variables A B C: Prop
2  example: (A ∧ (B ∨ C)) → ((A ∧ B) ∨ (A ∧ C)) :=
3  begin
4  intros,
```

```
5  have h₁: A, from a.left,
6  have h₂: B ∨ C, from a.right,
7  cases h₂ with ha hb,
8      exact or.inl (and.intro h₁ ha),
9      exact or.inr (and.intro h₁ hb)
10 end
```

6. Prova de $A \to \neg(\neg A \wedge B)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A \wedge B}\,(1)}{\neg A} \qquad \dfrac{}{\mathrm{A}}\,(2)}{\bot}}{\neg(\neg A \wedge B)}\,(1)}{A \to \neg(\neg A \wedge B)}\,(2)$$

Como ja visto anteriormente, quando precisamos provar uma implicação partimos do objetivo final $A \to \neg(\neg A \wedge B)$, de forma que devemos chegar em $\neg(\neg A \wedge B)$ tendo A como hipótese. Pela regra da introdução da negação, sabemos que para chegar em $\neg(\neg A \wedge B)$, precisamos supor $(\neg A \wedge B)$ como hipótese para chegar em um absurdo e obter o resultado desejado. Com as hipóteses descritas, conseguimos chegar em $\neg A$ e A, o que resulta em absurdo, que por sua vez nos possibilita chegar no objetivo final.

No Lean, essa prova poderia ser construída no modo termo da seguinte forma:

```
1 variables A B : Prop
2 example : A → ¬ (¬ A ∧ B) :=
3 assume h₁: A,
4 assume h₂: ¬A ∧ B,
5 show false, from h₂.left h₁
```

E no modo táticas:

```
1 variables A B : Prop
2 example : A → ¬ (¬ A ∧ B) :=
3 begin
4 intros h₁ h₂,
5 have h₃: ¬ A, from h₂.left,
6 contradiction
7 end
```

7. Prova de ¬ (A ↔ ¬ A)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{A}\leftrightarrow\neg\mathrm{A}}\,(1)}{\mathrm{A}\to\neg\mathrm{A}} \qquad \dfrac{}{\mathrm{A}}\,(2)}{\neg\mathrm{A}} \qquad \dfrac{\dfrac{\dfrac{}{\mathrm{A}\leftrightarrow\neg\mathrm{A}}\,(1)}{\mathrm{A}\leftarrow\neg\mathrm{A}} \qquad \dfrac{}{\neg\,\mathrm{A}}\,(3)}{\mathrm{A}}}{\bot}\,(2,3)}{\neg(A \leftrightarrow \neg A)}\,(1)$$

Para construir a prova acima podemos começar deduzindo o passo anterior ao resultado final que é a negação de $(A \leftrightarrow \neg A)$. Essa será nossa hipótese inicial e com ela queremos chegar à uma contradição. Assim, termemos a seguinte estrutura para a prova:

$$\dfrac{\dfrac{\dfrac{}{A \leftrightarrow \neg A}\,(1) \qquad \dfrac{}{A \leftrightarrow \neg A}\,(1)}{\begin{array}{cc}\vdots & \vdots \\ \bot & \end{array}}}{\neg(A \leftrightarrow \neg A)}\,(1)$$

Observe que a dupla implicação aparece duas vezes como hipótese, pois utilizaremos a implicação nas direções esquerda e direita para chegar a um absurdo. Com isso, em cada coluna da árvore (ou *branch*), podemos chegar em $\neg A$ e A, obtendo assim uma contradição.

Temos, no código a seguir, um modo de construir a prova acima no Lean, com o auxílio do *have*. É importante ressaltar que existem diferentes modos de escrever no Lean uma mesma prova em dedução natural. Assim, aconselhamos que você tente reproduzir a seu modo as demonstrações aqui expostas!

```
variable A : Prop

example : ¬ (A ↔ ¬ A) :=
assume h₁ : A ↔ ¬ A,
show false, from
    have h₃ : ¬ A, from
        assume h₂ : A,
         show false, from
            (iff.elim_left h₁ h₂) h₂,
            h₃ (iff.elim_right h₁ h₃)
```

Modo táticas:

```
variable A: Prop

example : ¬ (A ↔ ¬ A) :=
begin
intro h,
have h₁: ¬ A, from
    assume ha: A,
    have hb:¬ A, from iff.elim_left h ha,
    show false, from hb ha,

have h₂: A, from iff.elim_right h h₁,
contradiction
end
```

8. Prova de $\neg A \vee \neg B$ a partir de $\neg(A \wedge B)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A}\,(1) \quad \dfrac{}{B}\,(2)}{A \wedge B} \qquad \neg(A \wedge B)}{\dfrac{\dfrac{\bot}{\neg B}\,(2)}{\neg A \vee \neg B} \qquad \dfrac{}{\neg(\neg A \vee \neg B)}\,(3)}}{\dfrac{\dfrac{\bot}{\neg A}\,(1)}{\neg A \vee \neg B} \qquad \dfrac{}{\neg(\neg A \vee \neg B)}\,(3)}}{\bot}\,(3)}{\neg A \vee \neg B}$$

Essa prova pode ser construída de forma semelhante às provas descritas anteriormente. No Lean, ela poderia ser escrita no modo termo da seguinte forma:

```
1  variables A B : Prop
2  open classical
3
4  example (h: ¬ (A ∧ B)): ¬ A ∨ ¬ B :=
5      by_contradiction
6      (assume h₁ : ¬ (¬ A ∨ ¬ B),
7      have h₂ : ¬ A, from
8          assume h₃ : A,
9          have h₄ : ¬ B, from
10             (assume h₅ : B, show false, from h (and.intro h₃ h₅)),
11         have h₅ : ¬ A ∨ ¬ B, from or.inr h₄,
12         show false, from h₁ h₅,
13     have h₆ : ¬ A ∨ ¬ B, from or.inl h₂,
14     show false, from h₁ h₆)
```

Um dos recursos sintáticos extras do Lean que costumam ser convenientes é o uso do *this*. O *this* pode ser utilizado quando se omite o rótulo (ou seja, o nome) de uma hipótese assumida. Assim, ao escrevê-lo, o Lean busca a última hipótese que não foi nomeada e a utiliza no local da prova onde o *this* foi escrito.

Dessa forma, o mesmo exemplo acima poderia ser escrito da seguinte forma:

```
1  example (h: ¬ (A ∧ B)): ¬ A ∨ ¬ B :=
2      by_contradiction
3      (assume h₁ : ¬ (¬ A ∨ ¬ B),
4      have ¬ A, from
5          assume  h₃ : A,
6          have ¬ B, from
7              (assume  h₅ : B, show false, from h (and.intro h₃ h₅))
       ,
8          have h₅ : ¬ A ∨ ¬ B, from or.inr this,
```

```
9          show false, from h₁ h₅,
10     have ¬ A ∨ ¬ B, from or.inl this,
11     show false, from h₁ this)
```

Também pode ser escrito no modo táticas como:

```
1  variables A B: Prop
2
3  example (h: ¬ (A ∧ B)): ¬ A ∨ ¬ B :=
4  by_contradiction
5  begin
6  intro,
7  have h₁: ¬ A, from
8      assume h₂:A,
9      have h₃: ¬ B, from
10         assume h₄: B, show false, from h (and.intro h₂ h₄),
11     have h₅: ¬A ∨ ¬ B, from or.inr h₃,
12     show false, from a h₅,
13
14 have h₆: ¬ A ∨ ¬ B, from or.inl h₁,
15 show false, from a h₆
16 end
```

No entanto, ao passo que possa parecer mais conveniente escrever a prova assim, note que o entendimento imediato de alguns passos é dificultado, por isso é necessário cuidado ao utilizar esse recurso.

9. Prova do Princípio da Casa dos Pombos (PHP-3): O princípio da casa dos pombos afirma que se n pombos devem ser postos em m casas, e se $n > m$, então pelo menos uma casa irá conter mais de um pombo. Apesar desse princípio generalizado ser muito amplo para ser provado por árvores dedutivas, conseguimos provar o caso de três pombos e duas casas de pombo utilizando a dedução natural. A árvore de solução desse problema é muito grande para ser mostrada aqui, contudo vamos explicar como seria a estrutura dela e como solucionar esse problema no Lean.

Para pensar nessa árvore, vamos analisar como estruturar o problema em lógica proposicional. Se temos três pombos e duas casas, podemos descrever essa situação por P_{ij}, sendo i o número representativo do pombo e j o número representativo da casa. Cada pombo vai ficar somente em uma casa, logo temos: $(P_{11} \vee P_{12}) \wedge (P_{21} \vee P_{22}) \wedge (P_{31} \vee P_{32})$. Queremos provar que isso implica em ao menos dois pombos em uma mesma casa, o que pode ser representado por: $(P_{22} \wedge P_{32}) \vee (P_{11} \wedge P_{31}) \vee (P_{12} \wedge P_{22}) \vee (P_{11} \wedge P_{21}) \vee (P_{12} \wedge P_{32}) \vee (P_{21} \wedge P_{31})$. Assim, queremos provar que: $(P_{11} \vee P_{12}) \wedge (P_{21} \vee P_{22}) \wedge (P_{31} \vee P_{32}) \rightarrow (P_{22} \wedge P_{32}) \vee (P_{11} \wedge P_{31}) \vee (P_{12} \wedge P_{22}) \vee (P_{11} \wedge P_{21}) \vee (P_{12} \wedge P_{32}) \vee (P_{21} \wedge P_{31})$.

De início, podemos notar que queremos chegar em um "ou". Logo, provando apenas um dos "e" podemos chegar em todo esse "ou". Como provar essas conjunções? É possível observar que todos os pares formando o $\wedge$ da conclusão

estão em pares diferentes de ∨ na hipótese. Logo, é necessário abrir esses "ou". O que será feito é abrir cada uma das dinjunções da hipótese dentro das outras, de modo que teremos todos os termos necessários para formar as conjunções da conclusão.

Para tornar a ideia acima mais concreta, é possível observar como essa prova poderia ser construída no modo termo do Lean:

```
1  variables P11 P12 P21 P22 P31 P32 : Prop
2
3  example: ((P11 ∨ P12) ∧ (P21 ∨ P22)) ∧ (P31 ∨ P32) → (P22 ∧ P32
       ) ∨ ((P11 ∧ P31) ∨ ((P12 ∧ P22) ∨ ((P11 ∧ P21) ∨ ((P12 ∧
       P32) ∨ (P21 ∧ P31))))) :=
4
5  assume h: ((P11 ∨ P12) ∧ (P21 ∨ P22)) ∧ (P31 ∨ P32),
6  have ha: P11 ∨ P12, from and.left (and.left h),
7  or.elim ha
8      (assume h1: P11,
9          have hb: P21 ∨ P22, from and.right (and.left h),
10             or.elim hb
11                 (assume h2: P21, or.inr (or.inr (or.inr (or.inl (
       and.intro h1 h2)))))
12                 (assume h2: P22,
13                     have hc: P31 ∨ P32, from and.right h,
14                         or.elim hc
15                             (assume h3: P31, or.inr (or.inl (and.
       intro h1 h3)))
16                             (assume h3: P32, or.inl (and.intro h2
        h3))))
17     (assume h1: P12,
18         have hb: P21 ∨ P22, from and.right (and.left h),
19             or.elim hb
20                 (assume h2: P21,
21                     have hc: P31 ∨ P32, from and.right h,
22                         or.elim hc
23                             (assume h3: P31, or.inr (or.inr (or.
       inr (or.inr (or.inr (and.intro h2 h3))))))
24                             (assume h3: P32, or.inr (or.inr (or.
       inr (or.inr (or.inl (and.intro h1 h3)))))))
25                 (assume h2: P22, or.inr (or.inr (or.inl (and.
       intro h1 h2)))))
```

No modo tática essa prova poderia ser desenvolvida da seguinte forma:

```
1  variables P11 P12 P21 P22 P31 P32 : Prop
2  example: ((P11 ∨ P12) ∧ (P21 ∨ P22)) ∧ (P31 ∨ P32) → (P22 ∧ P32
       ) ∨ ((P11 ∧ P31) ∨ ((P12 ∧ P22) ∨ ((P11 ∧ P21) ∨ ((P12 ∧
       P32) ∨ (P21 ∧ P31))))) :=
```

```
3
4  begin
5  intros,
6  have h₁:P11 ∨ P12, from (a.left).left,
7  have h₂: P21 ∨ P22, from (a.left).right,
8  have h₃: P31 ∨ P32, from a.right,
9
10 cases h₂ with ha hb,
11     cases h₃ with hc hd,
12     exact or.inr (or.inr (or.inr (or.inr (or.inr (and.intro ha hc
       )))),
13         cases h₁ with he hf,
14         exact or.inr (or.inr (or.inr (or.inl (and.intro he ha))))
       ,
15         exact or.inr (or.inr (or.inr (or.inr (or.inl (and.intro
       hf hd))))),
16             cases h₃ with hg hh,
17                 cases h₁ with hi hj,
18                 exact or.inr (or.inl (and.intro hi hg)),
19                 exact or.inr (or.inr (or.inl (and.intro hj hb))),
20                 exact or.inl (and.intro hb hh)
21 end
```

Os códigos acima também mostram como utilizar parênteses para separar os termos, mesmo quando não são totalmente necessários, pode facilitar o desenvolvimento da prova. Isso acontece pois o Lean "atribui" parênteses para as conjunções e implicações (ou seja, ele lê essas proposições) da direita para a esquerda. Como exemplo, no código abaixo, é possível ver que quando aplica-se a função *and.right* em $A \wedge B \wedge C$ o Lean responde com $B \wedge C$.

```
1  variables A B C: Prop
2
3  example: A ∧ B ∧ C → B ∧ C :=
4  assume h: A ∧ B ∧ C,
5  show B ∧ C, from and.right h
```

Contudo, não é tão simples obter $A \wedge B$ a partir dessa mesma hipótese. Apesar de A e B estarem lado a lado, assim como B e C estavam, é necessário redigir uma prova mais extensa:

```
1  variables A B C: Prop
2
3  example: A ∧ B ∧ C → A ∧ B :=
4  assume h: A ∧ B ∧ C,
5  have h1: A, from and.left h,
6  have h2: B, from and.left (and.right h),
7  show A ∧ B, from and.intro h1 h2
```

O problema acima seria resolvido caso tivesse sido explicitamente atribuído um parênteses ao $A \wedge B$. Dessa forma, seria possível obtê-lo a partir da função $and.left$, conforme mostra o código abaixo:

```
variables A B C: Prop

example: (A ∧ B) ∧ C → A ∧ B :=
assume h: (A ∧ B) ∧ C,
show A ∧ B, from and.left h
```

10. Prova de $\neg(A \wedge B) \to (A \to \neg B)$: A árvore de dedução natural seria:

$$\dfrac{\dfrac{\neg(A \wedge B) \qquad \dfrac{\overset{\text{A}}{} \qquad \overset{\text{B}}{\overline{}}\,(1)}{A \wedge B}}{\dfrac{\bot}{\neg B}\,(1)}}{A \to \neg B}}{\neg(A \wedge B) \to (A \to \neg B)}$$

No Lean, é possível escrever essa prova no modo termo da seguinte forma:

```
variables {A B: Prop}
example : ¬ (A ∧ B) → (A → ¬ B) :=
assume h1: ¬ (A ∧ B),
assume h2: A,
assume h3: B,
have h4: A ∧ B, from and.intro h2 h3,
false.elim (h1 h4)
```

E no modo táticas:

```
variables A B: Prop
example : ¬ (A ∧ B) → (A → ¬ B) :=
begin
intros,
assume h: B,
have h_1: A ∧ B, from and.intro a_1 h,
contradiction
end
```

11. Prova de $\neg(A \vee B)$ a partir de $\neg A \wedge \neg B$:
A árvore de dedução natural seria:

$$\dfrac{\dfrac{\overset{(1)}{\overline{}}}{A \vee B} \qquad \dfrac{\dfrac{\neg A \wedge \neg B}{\neg A} \qquad \dfrac{\overset{\text{A}}{}}{}\,(1)}{\bot} \qquad \dfrac{\dfrac{\neg A \wedge \neg B}{\neg B} \qquad \dfrac{\overset{\text{B}}{}}{}\,(1)}{\bot}}{\dfrac{\bot}{\neg(A \vee B)}\,(1)}$$

No Lean, é possível escrever essa prova no modo termo da seguinte forma:

```
variables A B: Prop
example (h : ¬ A ∧ ¬ B) : ¬ (A ∨ B) :=
assume h1: A ∨ B,
or.elim h1
    (assume h2: A, false.elim ((and.left h) h2))
    (assume h2: B, false.elim((and.right h) h2))
```

E no modo táticas como:

```
variables A B: Prop
example (h : ¬ A ∧ ¬ B) : ¬ (A ∨ B) :=
assume h₁: A ∨ B,
begin
cases h₁ with ha hb,
have h₂: ¬ A, from h.left,
contradiction,
have h₃: ¬ B, from h.right,
contradiction
end
```

12. Prova de $\neg A \vee B$ a partir de $A \to B$: A árvore de dedução natural seria:

$$\dfrac{\dfrac{}{A \vee \neg A}\,(1) \qquad \dfrac{\dfrac{A \to B \qquad \dfrac{}{A}\,(1)}{B}}{\neg A \vee B} \qquad \dfrac{\dfrac{}{\neg A}\,(1)}{\neg A \vee B}}{\neg A \vee B}$$

No Lean, é possível escrever essa prova no modo termo da seguinte forma:

```
variables A B: Prop
example (h8:A → B):¬ A ∨ B:=
or.elim(em A)
    (assume he: A,
    show (¬ A ∨ B),from or.inr (h8 he) )
    (assume hi: ¬ A,
    show(¬ A ∨ B),from or.inl hi)
```

E modo táticas como:

```
variables A B: Prop
example (h8:A → B):¬ A ∨ B:=
begin
have h: A ∨ ¬ A, from em A,
cases h with ha hb,
exact or.inr (h8 ha),
exact or.inl hb
end
```

3.4 Exercícios

1. Construa a árvore de dedução natural e prove no Lean que $A \wedge (A \to B) \to B$.

   Gabarito da árvore de dedução natural:

$$\dfrac{\dfrac{\dfrac{\overline{A \wedge (A \to B)}^{(1)}}{A \to B} \qquad \dfrac{\overline{A \wedge (A \to B)}^{(1)}}{\mathrm{A}}}{\mathrm{B}}}{A \wedge (A \to B) \to B}^{(1)}$$

   Gabarito no Lean:

```
variables {A B : Prop}

example : A ∧ (A → B) → B :=
assume h: A ∧ (A → B),
have h2: A → B, from and.right h,
have h3: A, from and.left h,
h2 h3
```

2. Construa a árvore de dedução natural e prove no Lean $C \vee D$, a partir de $A \vee B$, $A \to C$ e $B \to D$.

   Gabarito da árvore de dedução natural:

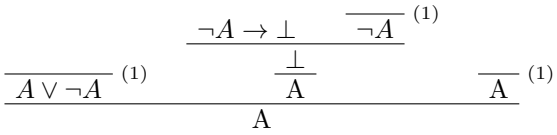
$$\dfrac{\overline{A \vee B}^{(1)} \qquad \dfrac{\dfrac{A \to C \qquad \overline{\mathrm{A}}^{(1)}}{\mathrm{C}}}{C \vee D} \qquad \dfrac{\dfrac{B \to D \qquad \overline{\mathrm{B}}^{(1)}}{\mathrm{D}}}{C \vee D}}{C \vee D}$$

   Gabarito no Lean:

```
variables {A B C D: Prop}
example (h1 : A ∨ B) (h2 : A → C) (h3 : B → D) : C ∨ D :=
or.elim h1
    (assume h: A, show C ∨ D, from or.inl (h2 h))
    (assume h: B, show C ∨ D, from or.inr (h3 h))
```

3. Construa a árvore de dedução natural e prove no Lean A, a partir de $A \vee \neg A$ e $\neg A \to false$.

Gabarito da árvore de dedução natural:

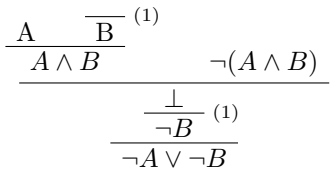


Gabarito no Lean:

```
variables {A: Prop}
example (h1 : A ∨ ¬ A) (h2: ¬ A → false) : A :=
or.elim h1
    (assume h: A,
    show A, from h)
    (assume h: ¬ A,
    show A, from false.elim (h2 h))
```

4. Construa a árvore de dedução natural e prove no Lean $\neg A \vee \neg B$, a partir de $\neg(A \wedge B)$ e A.

Gabarito da árvore de dedução natural:



Gabarito no Lean:

```
open classical
variables {A B: Prop}
lemma stepA (h₁ : ¬ (A ∧ B)) (h₂ : A) : ¬ A ∨ ¬ B :=
have ¬ B, from (assume hk:B,
show false, from h₁ (and.intro h₂ hk)),
show ¬ A ∨ ¬ B, from or.inr this

lemma stepB (h₁ : ¬ (A ∧ B)) (h₂ : ¬ (¬ A ∨ ¬ B)) : false
    :=
have ¬ A, from
  assume : A,
  have ¬ A ∨ ¬ B, from stepA h₁ <>A,
  show false, from h₂ this,
show false, from (h₂ (or.inl this) )
```

```
14
15 theorem stepC (h : ¬ (A ∧ B)) : ¬ A ∨ ¬ B :=
16 by_contradiction
17   (assume h' : ¬ (¬ A ∨ ¬ B),
18     show false, from stepB h h')
```

5. Construa a árvore de dedução natural e prove no Lean P, a partir de $\neg P \to (Q \vee R)$, $\neg Q$ e $\neg R$.

   Gabarito da árvore de dedução natural:

$$\dfrac{\dfrac{\neg P \to (Q \vee R) \qquad \dfrac{}{\neg P}\,(1)}{Q \vee R} \qquad \dfrac{\neg Q \qquad \dfrac{}{\mathrm{Q}}\,(1)}{\bot} \qquad \dfrac{\neg R \qquad \dfrac{}{\mathrm{R}}\,(1)}{\bot}}{\dfrac{\bot}{\mathrm{P}}\,(1)}$$

   Gabarito no Lean:

```
1  open classical
2  variables {P Q R: Prop}
3  example (h5: ¬P →(Q ∨ R)) (h6:¬ Q) (h7:¬ R) :P:=
4  by_contradiction(
5  assume hp: ¬ P,
6  have hqr: Q ∨ R , from h5 hp,
7  show false, from
8      or.elim hqr
9      (assume hi: Q,
10     show false, from h6 hi)
11     (assume hii: R,
12     show false, from h7 hii))
```

6. Construa a árvore de dedução natural e prove no Lean $A \to (A \wedge B) \vee (A \wedge \neg B)$.

   Gabarito da árvore de dedução natural:

$$\dfrac{\dfrac{}{B \vee \neg B}\,(1) \qquad \dfrac{\dfrac{\dfrac{}{\mathrm{A}}\,(2) \qquad \dfrac{}{\mathrm{B}}\,(1)}{A \wedge B}}{(A \wedge B) \vee (A \wedge \neg B)} \qquad \dfrac{\dfrac{\dfrac{}{\mathrm{A}}\,(2) \qquad \dfrac{}{\neg B}\,(1)}{A \wedge \neg B}}{(A \wedge B) \vee (A \wedge \neg B)}}{\dfrac{(A \wedge B) \vee (A \wedge \neg B)}{A \to (A \wedge B) \vee (A \wedge \neg B)}\,(2)}$$

Gabarito no Lean:

```
open classical
variables {A B: Prop}
example :A → (A ∧ B) ∨ (A ∧ ¬ B):=
assume h9: A,
show (A ∧ B) ∨  (A ∧ ¬ B), from
or.elim(em B)
    (assume hr:B,
    show (A ∧ B) ∨  (A ∧ ¬ B), from or.inl(and.intro h9 hr)
     )
    (assume hw: ¬ B,
    show (A ∧ B) ∨  (A ∧ ¬ B), from or.inr(and.intro h9 hw)
    )
```

7. Prove no Lean que $(A \vee B) \wedge (C \vee D) \wedge (E \vee F) \rightarrow (A \wedge E \wedge C) \vee (F \wedge B \wedge D) \vee (A \wedge F \wedge C) \vee (A \wedge E \wedge D) \vee (A \wedge F \wedge D) \vee (B \wedge E \wedge C) \vee (B \wedge F \wedge C) \vee (B \wedge E \wedge D)$.

Gabarito:

```
variables {A B C D E F: Prop}
example: ((A ∨ B) ∧ (C ∨ D)) ∧ (E ∨ F) →  (((((((((A ∧ E)
    ∧ C) ∨ ((F ∧ B) ∧ D)) ∨ ((A ∧ F) ∧ C)) ∨ ((A ∧ E) ∧ D)
    ) ∨ ((A ∧ F) ∧ D)) ∨ ((B ∧ E) ∧ C)) ∨ ((B ∧ F) ∧ C)) ∨
     ((B ∧ E) ∧ D)) :=
assume h: ((A ∨ B) ∧ (C ∨ D)) ∧ (E ∨ F),
have ha: A ∨ B, from and.left (and.left h),
or.elim ha
    (assume h1: A, have hb: C ∨ D, from and.right (and.left
    h),
    or.elim hb
        (assume h2: C, have hc: E ∨ F, from and.right h,
        or.elim hc
            (assume h3: E, or.inl (or.inl (or.inl (or.inl (
    or.inl (or.inl (or.inl (and.intro (and.intro h1 h3) h2)))
    )))))
            (assume h3: F, or.inl (or.inl (or.inl (or.inl (
    or.inl (or.inr (and.intro (and.intro h1 h3) h2))))))))
        (assume h2: D, have hc: E ∨ F, from and.right h,
        or.elim hc
            (assume h3: E, or.inl (or.inl (or.inl (or.inl (
    or.inr (and.intro (and.intro h1 h3) h2))))))
            (assume h3: F, or.inl (or.inl (or.inl (or.inr (
    and.intro (and.intro h1 h3) h2)))))))
    (assume h1: B, have hb: C ∨ D, from and.right (and.left
    h),
```

```
17      or.elim hb
18          (assume h2: C, have hc: E ∨ F, from and.right h,
19          or.elim hc
20              (assume h3: E, or.inl (or.inl (or.inr (and.intro
         (and.intro h1 h3) h2))))
21              (assume h3: F, or.inl (or.inr (and.intro (and.
        intro h1 h3) h2))))
22          (assume h2: D, have hc: E ∨ F, from and.right h,
23          or.elim hc
24              (assume h3: E, or.inr (and.intro (and.intro h1
        h3) h2))
25              (assume h3: F, or.inl (or.inl (or.inl (or.inl (
        or.inl (or.inl (or.inr (and.intro (and.intro h3 h1) h2)))
        )))))))
```

8. Prove no Lean $A \to B$, a partir de $\neg B \to \neg A$.

   Gabarito:

```
1  open classical
2  variables {A B: Prop}
3  example (hl : ¬ B → ¬ A) : A → B :=
4  assume h10: A, show B,from
5  by_contradiction(
6      assume hnb: ¬ B,
7      show false, from (hl hnb) h10)
```

9. Prove no Lean $\neg A \vee B$, a partir de $A \to B$.

   Gabarito:

```
1  open classical
2  variables {A B: Prop}
3  example (hp : A → B) : ¬ A ∨ B :=
4  show ¬ A ∨ B, from
5  or.elim (em A)
6      (assume ha1: A,
7      show ¬A ∨ B, from or.inr (hp ha1) )
8      (assume ha2: ¬ A,
9      show ¬A ∨ B, from or.inl ha2)
```

10. Prove no Lean o problema dos vestidos descrito na introdução.

    Gabarito em modo termo:

```
variables AA AB AP MA MB MP CA CB CP : Prop
variable h1: AA ∨ (AB ∨ AP)
variable h2: MA ∨ (MB ∨ MP)
variable h3: CA ∨ CB ∨ CP
variable h4: AA → AB
variable h5: CA → ¬ AB
variable h6: AB → MB
variable h7: CB → ¬ MB
variable h8: AP → CB
variable h9: CP → ¬ CB
variable h10: ¬ AB
variable h11: (AA → (¬ AB ∧ ¬ AP)) ∧ (AB → (¬ AA ∧ ¬ AP))
     ∧ (AP → (¬ AB ∧ ¬ AA))
variable h12: (MA → (¬ MB ∧ ¬ MP)) ∧ (MB → (¬ MA ∧ ¬ MP))
     ∧ (MP → (¬ MB ∧ ¬ MA))
variable h13: (CA → (¬ CB ∧ ¬ CP)) ∧ (CB → (¬ CA ∧ ¬ CP))
     ∧ (CP → (¬ CB ∧ ¬ CA))
variable h14: (AA → (¬ MA ∧ ¬ CA)) ∧ (AB → (¬ MB ∧ ¬ CB))
     ∧ (AP → (¬ MP ∧ ¬ CP))
variable h15: (MA → (¬ AA ∧ ¬ CA)) ∧ ((MB → (¬ AB ∧ ¬ CB)
    ) ∧ (MP → (¬ AP ∧ ¬ CP)))
variable h16: (CA → (¬ AA ∧ ¬ MA)) ∧ (CB → (¬ AB ∧ ¬ MB))
     ∧ (CP → (¬ AP ∧ ¬ MP))

example: ((AP ∧ CB) ∧ MA) :=

have h17: AP, from
or.elim h1
    (assume h18: AA, false.elim (h10 (h4 h18)))
    (assume h18: AB ∨ AP,
        or.elim h18
            (assume h19: AB, false.elim (h10 h19))
            (assume h19: AP, h19)),

have h20: CB, from
(h8 h17),

have h21: MA, from
or.elim h2
    (assume h22: MA, h22)
    (assume h22: MB ∨ MP,
    or.elim h22
        (assume h23: MB, false.elim ((and.right ((and.left (
    and.right h15)) h23)) h20))
        (assume h23: MP, false.elim ((and.left ((and.right (
    and.right h15)) h23)) h17))),
```

```
39
40 and.intro (and.intro h17 h20) h21
```

Gabarito em modo táticas:

```
1  variables AA AB AP MA MB MP CA CB CP : Prop
2
3  theorem step1 {AA AB AP :Prop}(h1: AA ∨ (AB ∨ AP))(h4: AA →
       AB) (h10: ¬ AB) :AP:=
4  begin
5  cases h1 with ha hb,
6      exact false.elim(h10 (h4 ha)),
7      cases hb with hc hd,
8          exact false.elim(h10 hc),
9          exact hd,
10 end
11
12 theorem step2 {AA AB AP CB: Prop} (h1: AA ∨ (AB ∨ AP))(h4:
       AA → AB) (h10: ¬ AB)(h8: AP → CB):CB:=
13 begin
14 have h: AP, from step1 h1 h4 h10,
15 exact h8 h,
16 end
17
18 theorem step3 {AA AB AP MA MB MP CA CB CP : Prop}(h1: AA ∨ (
       AB ∨ AP))(h2: MA ∨ (MB ∨ MP))(h4: AA → AB)
19 (h8: AP → CB)(h10: ¬ AB)(h15: (MA → (¬ AA ∧ ¬ CA)) ∧ ((MB
       → (¬ AB ∧ ¬ CB)) ∧ (MP → (¬ AP ∧ ¬ CP))))
20 :MA:=
21 begin
22 cases h2 with ha hb,
23     exact ha,
24     cases hb with hc hd,
25         exact false.elim((h15.right.left hc).right (step2 h1
       h4 h10 h8) ),
26         exact false.elim((h15.right.right hd).left (step1
       h1 h4 h10) ),
27 end
28
29 theorem final (h1: AA ∨ (AB ∨ AP))(h2: MA ∨ (MB ∨ MP))(h4:
       AA → AB)
30 (h8: AP → CB)(h10: ¬ AB)(h15: (MA → (¬ AA ∧ ¬ CA)) ∧ ((MB
       → (¬ AB ∧ ¬ CB)) ∧ (MP → (¬ AP ∧ ¬ CP))))
31 :((AP ∧ CB) ∧ MA):=
32 begin
33 have h_1:AP, from step1 h1 h4 h10,
```

```
have h₂:CB, from step2  h1 h4 h10 h8,
have h₃:MA, from step3 h1 h2 h4 h8 h10 h15,
exact and.intro(and.intro h₁ h₂) h₃
end
```

11. Escolha um grafo qualquer pequeno, e usando a codificação em lógica proposicional, tente provar no Lean que um dado caminho é mesmo um caminho hamiltoniano no grafo.

    Comentários:

    Um caminho hamiltoniano de um grafo é um caminho que visita cada nó do grafo exatamente uma vez.

    xij significa "que a in-ésima posição no caminho hamiltoniano é ocupado pelo nó j". Dado um grafo G, podemos construir um CNF R(G), de modo que R(G) é satisfatível se, e somente se, G possui um caminho hamiltoniano.

    As condições de R(G) são:

    (a) Todos os nós j devem aparecer no caminho.

    - x1j $\vee$ x2j $\vee$ . . . $\vee$ xnj para cada j.

    (b) Nenhum nó j aparece duas vezes no caminho

    - $\neg$ xij $\vee$ $\neg$ xkj para todos os i,j,k tais que i $\neq$ k.

    (c) Cada posição i no caminho deve estar ocupada.

    - xi1 $\vee$ xi2 $\vee$ . . . $\vee$ xin para cada i.

    (d) Nenhum dos dois nós j e k ocupam a mesma posição no caminho, onde j $\neq$ k.

    (e) Nós i e j não adjacentes não podem ser adjacentes no caminho.

    - $\neg$ xki $\vee\neg$ x_{k+1}j para todos (i,j) $\notin$ G e k= 1,2, ..., n-1

    (Arestas que não existem no grafo não podem fazer parte do caminho)

    Escolhido um caminho, verificamos se ele obedece as restrições estabelecidas, se sim, ele é um caminho hamiltoniano. Ao interpretar e começar a formular um modo de resolver esse problema podemos iniciar tentando fazer algo parecido com o que fizemos para o exemplo dos vestidos e provar que uma dada fórmula xij $\wedge$...$\wedge$ xkn é verdadeira, mas nesse caso observe que não há só uma possibilidade de caminho que satisfaz as restrições, e xij sozinho não nos diz muita coisa já que ele depende do restante da fórmula.

Possível gabarito:

```
open classical
variables {x11 x12 x13 x14 x21 x22 x23 x24 x31 x32 x33 x34
    x41 x42 x43 x44 : Prop}

/-     grafo:               caminho:
     x1 ---- x2          x11 ∧ x22 ∧ x33 ∧ x44
     |        |         e, naturalmente ¬xij para
     x4------x3         qualquer outro par i,j
-/


--- REGRA 1 E 3 ---
lemma rule_1and3 (h1: x11 ∧ x22 ∧ x33 ∧ x44):
(((x11 ∨ x21 ∨ x31 ∨ x41)∧ (x44 ∨ x14 ∨ x24 ∨ x34))∧ (x33
    ∨ x13 ∨ x23 ∨ x43))∧ (x22 ∨ x12 ∨ x32 ∨ x42)
  :=
  show
  (((x11 ∨ x21 ∨ x31 ∨ x41)∧
  (x44 ∨ x14 ∨ x24 ∨ x34))∧
  (x33 ∨ x13 ∨ x23 ∨ x43))∧
  (x22 ∨ x12 ∨ x32 ∨ x42),
  from and.intro
        (and.intro
            (and.intro
                (or.inl h1.left)
                (or.inl h1.right.right.right))
            (or.inl h1.right.right.left))
        (or.inl h1.right.left)

--- REGRA 2 ---

--- para o no 1 ---
lemma rule_2_node1 (h2: x11 ∧ ¬x21 ∧ ¬x31 ∧ ¬x41 ∧ x22 ∧
    x33 ∧ x44):
  (((((¬ x11 ∨ ¬  x21)∧
      (¬ x11 ∨ ¬  x31))∧
      (¬ x11 ∨ ¬  x41))∧
      (¬ x21 ∨ ¬  x31))∧
      (¬ x21 ∨ ¬  x41))∧
      (¬ x31 ∨ ¬  x41)
  :=
  show(((((¬ x11 ∨ ¬  x21)∧
      (¬ x11 ∨ ¬  x31))∧
      (¬ x11 ∨ ¬  x41))∧
```

```
42        (¬ x21 ∨ ¬  x31))∧
43        (¬ x21 ∨ ¬  x41))∧
44        (¬ x31 ∨ ¬  x41),
45    from and.intro
46            (and.intro
47                      (and.intro
48                            (and.intro
49                              (and.intro
50                                ( or.inr h2.right.left )
51                                ( or.inr h2.right.right.left )
      )
52                              (or.inr h2.right.right.right.
      left))
53                            (or.inl h2.right.left))
54                      (or.inl h2.right.left))
55            (or.inl h2.right.right.left)
56
57  --- para o no 2 ---
58  lemma rule_2_node2 (h2: x11 ∧ ¬x12 ∧ ¬x32 ∧ ¬x42 ∧ x22 ∧
      x33 ∧ x44):
59    (((((¬ x12 ∨ ¬  x22)∧
60        (¬ x12 ∨ ¬  x32))∧
61        (¬ x12 ∨ ¬  x42))∧
62        (¬ x22 ∨ ¬  x32))∧
63        (¬ x22 ∨ ¬  x42))∧
64        (¬ x32 ∨ ¬  x42):=
65    show
66      (((((¬ x12 ∨ ¬  x22)∧
67        (¬ x12 ∨ ¬  x32))∧
68        (¬ x12 ∨ ¬  x42))∧
69        (¬ x22 ∨ ¬  x32))∧
70        (¬ x22 ∨ ¬  x42))∧
71        (¬ x32 ∨ ¬  x42),
72    from and.intro
73            (and.intro
74                      (and.intro
75                            (and.intro
76                              (and.intro
77                                ( or.inl h2.right.left )
78                                ( or.inl h2.right.left ))
79                              (or.inl h2.right.left))
80                            (or.inr h2.right.right.left))
81                      (or.inr h2.right.right.right.left))
82            (or.inl h2.right.right.left)
83
84  --- para o no 3 ---
```

```
lemma rule_2_node3 (h2: x11 ∧ ¬x13 ∧ ¬x43 ∧ ¬x23 ∧ x22 ∧
    x33 ∧ x44):
  (((((¬ x13 ∨ ¬  x23)∧
     (¬ x13 ∨ ¬  x33))∧
     (¬ x13 ∨ ¬  x43))∧
     (¬ x23 ∨ ¬  x33))∧
     (¬ x23 ∨ ¬  x43))∧
     (¬ x33 ∨ ¬  x43):=
  show
   (((((¬ x13 ∨ ¬  x23)∧
     (¬ x13 ∨ ¬  x33))∧
     (¬ x13 ∨ ¬  x43))∧
     (¬ x23 ∨ ¬  x33))∧
     (¬ x23 ∨ ¬  x43))∧
     (¬ x33 ∨ ¬  x43),
  from and.intro
          (and.intro
                    (and.intro
                          (and.intro
                            (and.intro
                              ( or.inl h2.right.left )
                              ( or.inl h2.right.left ))
                            (or.inl h2.right.left))
                          (or.inl h2.right.right.right.left)
    )
                    (or.inl h2.right.right.right.left))
          (or.inr h2.right.right.left)

--- para o no 4 ---
lemma rule_2_node4 (h2: x11 ∧ ¬x14 ∧ ¬x34 ∧ ¬x24 ∧ x22 ∧
    x33 ∧ x44):
  (((((¬ x14 ∨ ¬  x24)∧
     (¬ x14 ∨ ¬  x34))∧
     (¬ x14 ∨ ¬  x44))∧
     (¬ x24 ∨ ¬  x34))∧
     (¬ x24 ∨ ¬  x44))∧
     (¬ x34 ∨ ¬  x44):=
  show
   (((((¬ x14 ∨ ¬  x24)∧
     (¬ x14 ∨ ¬  x34))∧
     (¬ x14 ∨ ¬  x44))∧
     (¬ x24 ∨ ¬  x34))∧
     (¬ x24 ∨ ¬  x44))∧
     (¬ x34 ∨ ¬  x44),
  from and.intro
          (and.intro
```

```
128                     (and.intro
129                           (and.intro
130                             (and.intro
131                               ( or.inl h2.right.left )
132                               ( or.inl h2.right.left ))
133                             (or.inl h2.right.left))
134                           (or.inl h2.right.right.right.left)
    )
135                     (or.inl h2.right.right.right.left))
136           (or.inl h2.right.right.left)
137
138 --- REGRA 4 ---
139
140 --- para a posicao 1 ---
141 lemma rule_4_node1 (h2: ¬x12 ∧ ¬x13 ∧ ¬x14 ∧ x11 ∧ x22 ∧
      x33 ∧ x44):
142   (((((¬ x11 ∨ ¬  x12)∧
143       (¬ x11 ∨ ¬  x13))∧
144       (¬ x11 ∨ ¬  x14))∧
145       (¬ x12 ∨ ¬  x13))∧
146       (¬ x12 ∨ ¬  x14))∧
147       (¬ x13 ∨ ¬  x14):=
148   show
149    (((((¬ x11 ∨ ¬  x12)∧
150       (¬ x11 ∨ ¬  x13))∧
151       (¬ x11 ∨ ¬  x14))∧
152       (¬ x12 ∨ ¬  x13))∧
153       (¬ x12 ∨ ¬  x14))∧
154       (¬ x13 ∨ ¬  x14),
155   from and.intro
156           (and.intro
157                     (and.intro
158                           (and.intro
159                             (and.intro
160                               ( or.inr h2.left )
161                               ( or.inr h2.right.left ))
162                             (or.inr h2.right.right.left))
163                           (or.inl h2.left))
164                     (or.inl h2.left))
165           (or.inl h2.right.left)
166
167 --- para a posicao 2 ---
168 lemma rule_4_node2 (h2: ¬x21 ∧ ¬x23 ∧ ¬x24 ∧ x11 ∧ x22 ∧
      x33 ∧ x44):
169   (((((¬ x21 ∨ ¬  x22)∧
170       (¬ x21 ∨ ¬  x23))∧
```

```
171       (¬ x21 ∨ ¬  x24))∧
172       (¬ x22 ∨ ¬  x23))∧
173       (¬ x22 ∨ ¬  x24))∧
174       (¬ x23 ∨ ¬  x24):=
175   show
176   (((((¬ x21 ∨ ¬  x22)∧
177       (¬ x21 ∨ ¬  x23))∧
178       (¬ x21 ∨ ¬  x24))∧
179       (¬ x22 ∨ ¬  x23))∧
180       (¬ x22 ∨ ¬  x24))∧
181       (¬ x23 ∨ ¬  x24),
182   from and.intro
183           (and.intro
184                     (and.intro
185                           (and.intro
186                             (and.intro
187                               ( or.inl h2.left )
188                               ( or.inl h2.left ))
189                             (or.inl h2.left))
190                           (or.inr h2.right.left))
191                     (or.inr h2.right.right.left))
192           (or.inl h2.right.left)
193
194 --- para a posicaoo 3 ---
195 lemma rule_4_node3 (h2: ¬x31 ∧ ¬x32 ∧ ¬x34 ∧ x11 ∧ x22  ∧
        x33 ∧ x44):
196   (((((¬ x31 ∨ ¬  x32)∧
197       (¬ x31 ∨ ¬  x33))∧
198       (¬ x31 ∨ ¬  x34))∧
199       (¬ x32 ∨ ¬  x33))∧
200       (¬ x32 ∨ ¬  x34))∧
201       (¬ x33 ∨ ¬  x34):=
202   show
203   (((((¬ x31 ∨ ¬  x32)∧
204       (¬ x31 ∨ ¬  x33))∧
205       (¬ x31 ∨ ¬  x34))∧
206       (¬ x32 ∨ ¬  x33))∧
207       (¬ x32 ∨ ¬  x34))∧
208       (¬ x33 ∨ ¬  x34),
209   from and.intro
210           (and.intro
211                     (and.intro
212                           (and.intro
213                             (and.intro
214                               ( or.inl h2.left )
215                               ( or.inl h2.left))
```

```
                              (or.inl h2.left))
                            (or.inl h2.right.left))
                      (or.inl h2.right.left))
            (or.inr h2.right.right.left)

--- para a posicao 4 ---
lemma rule_4_node4 (h2: ¬x41 ∧ ¬x42 ∧ ¬x43 ∧  x11 ∧ x22  ∧
     x33 ∧ x44):
  (((((¬ x41 ∨ ¬  x42)∧
      (¬ x41 ∨ ¬  x43))∧
      (¬ x41 ∨ ¬  x44))∧
      (¬ x42 ∨ ¬  x43))∧
      (¬ x42 ∨ ¬  x44))∧
      (¬ x43 ∨ ¬  x44):=
  show
  (((((¬ x41 ∨ ¬  x42)∧
      (¬ x41 ∨ ¬  x43))∧
      (¬ x41 ∨ ¬  x44))∧
      (¬ x42 ∨ ¬  x43))∧
      (¬ x42 ∨ ¬  x44))∧
      (¬ x43 ∨ ¬  x44),
  from and.intro
          (and.intro
                    (and.intro
                          (and.intro
                            (and.intro
                              ( or.inl h2.left )
                              ( or.inl h2.left))
                            (or.inl h2.left))
                          (or.inl h2.right.left))
                    (or.inl h2.right.left))
          (or.inl h2.right.right.left)

--- REGRA 5 ---

--- para os nos 1 e 3 ---
lemma rule_5_edge13 (h2: ¬x23 ∧ ¬x21 ∧ ¬x43 ∧ ¬x41 ∧  x11
    ∧ x22  ∧ x33 ∧ x44):
  (((((¬ x11 ∨ ¬  x23)∧
      (¬ x21 ∨ ¬  x33))∧
      (¬ x31 ∨ ¬  x43))∧
      (¬ x13 ∨ ¬  x21))∧
      (¬ x23 ∨ ¬  x31))∧
      (¬ x33 ∨ ¬  x41):=
  show
  (((((¬ x11 ∨ ¬  x23)∧
```

```
260       (¬ x21 ∨ ¬  x33))∧
261       (¬ x31 ∨ ¬  x43))∧
262       (¬ x13 ∨ ¬  x21))∧
263       (¬ x23 ∨ ¬  x31))∧
264       (¬ x33 ∨ ¬  x41),
265   from and.intro
266           (and.intro
267                     (and.intro
268                           (and.intro
269                             (and.intro
270                               ( or.inr h2.left )
271                               ( or.inl h2.right.left))
272                             (or.inr h2.right.right.left))
273                           (or.inr h2.right.left))
274                     (or.inl h2.left))
275           (or.inr h2.right.right.right.left)
276
277 --- para os nos 3 e 4 ---
278 lemma rule_5_edge24 (h2: ¬x12 ∧ ¬x24 ∧ ¬x34 ∧ ¬x32 ∧  x11
      ∧ x22  ∧ x33 ∧ x44):
279   (((((¬ x12 ∨ ¬  x24)∧
280       (¬ x22 ∨ ¬  x34))∧
281       (¬ x32 ∨ ¬  x44))∧
282       (¬ x14 ∨ ¬  x24))∧
283       (¬ x24 ∨ ¬  x34))∧
284       (¬ x34 ∨ ¬  x44):=
285   show
286   (((((¬ x12 ∨ ¬  x24)∧
287       (¬ x22 ∨ ¬  x34))∧
288       (¬ x32 ∨ ¬  x44))∧
289       (¬ x14 ∨ ¬  x24))∧
290       (¬ x24 ∨ ¬  x34))∧
291       (¬ x34 ∨ ¬  x44),
292   from and.intro
293           (and.intro
294                     (and.intro
295                           (and.intro
296                             (and.intro
297                               ( or.inl h2.left )
298                               ( or.inr h2.right.right.left))
299                             (or.inl h2.right.right.right.
        left))
300                           (or.inr h2.right.left))
301                     (or.inl h2.right.left))
302           (or.inl h2.right.right.left)
```

Capítulo 4

Lógica de Primeira Ordem

Até agora vimos a Lógica Proposicional, em que variáveis proposicionais podem ser valoradas como verdadeiras ou falsas. Esse sistema lógico, porém, contém limitações. Considere a afirmação de que todo natural é maior do que ou igual a zero. Ou a afirmação de que existe número natural que é primo e é par. Como representar isso em Lógica Propocional? Precisamos de um sistema lógico que saiba lidar com objetos e suas propriedades e relações.

Talvez esses exemplos não tenham demonstrado a necessidade de tal sistema lógico. Porém, acredite: ao final deste capítulo, essa necessidade ficará evidente.

4.1 Sintaxe

A sintaxe de um sistema lógico aborda basicamente os símbolos que são utilizados para representá-lo. Portanto, nesta seção, serão abordados funções, predicados, relações e quantificadores, dentre eles $\forall$ e $\exists$.

4.1.1 Funções, Predicados e Relações

Dentro de Lógica de Primeira Ordem, funções, predicados e relações são mapeamentos que, dado algum elemento do domínio, retornam uma proposição ou outro elemento do domínio. Parece confuso? Vamos olhar um exemplo.

$$x \text{ natural é par ou ímpar.}$$

Nosso domínio, neste caso, são os números naturais ($\mathbb{N}$). Dizemos que *par* é "algo" que recebe um número e retorna V ou F. Chamamos isso de **predicado**. Sendo assim, podemos escrever:

$$par(x) \vee impar(x).$$

A partir deste exemplo, podemos extrair alguns símbolos para exemplificar a construção de um sistema lógico de primeira ordem.

- O domínio são os naturais;
- Os objetos são os números 0, 1, 2 etc.;
- Existem **funções**, como *adição* e *subtração*, que recebem (zero ou mais) números e retornam outros números;
- Existem **predicados**, como *par* e *ímpar*, que recebem um número e retornam V ou F;
- Existem **relações**, como *igual* e *menor*, que recebem dois números e retornam V ou F.

Os objetos pertencentes ao domínio, chamados constantes, como o 1 e o 4, no exemplo, podem ser considerados funções que tomam zero elementos. Além disso, podemos considerar predicados que tomam zero elementos como os valores lógicos $\top$ e $\bot$.

Expressões que representam elementos do domínio(incluindo funções de elementos) são chamados de **termos**. Alguns exemplos:

- 5
- $sucessor(10)$ (função sucessora, retorna o elemento acrescido de 1)
- $33 + 44$

Observe que o símbolo para a função de adição está "infixo"; poderíamos ter representado como $+(33, 44)$ ou $adicao(33, 44)$.

Expressões que retornam V ou F são chamadas **fórmulas**:

- $maiorDoQue(1, 2)$ $(1 > 2)$
- $par(10) \vee impar(5)$
- $2 = 2$

O Lean é muito eficiente em expressar Lógica de Primeira Ordem. Vejamos nosso exemplo:

```
1 constant U : Type
2 constant zero : U
3 constant par : U → Prop
4 constant primo : U → Prop
5 constant igual : U → U → Prop
6 constant adicao : U → U → U
```

Pelo fato de que o Lean é baseado em *Teoria dos Tipos*, declaramos um novo tipo `U`. Podemos intuitivamente considerá-lo como "universo" ou "domínio". Por exemplo, o conjunto dos naturais.

Foi declarado um objeto chamado `zero` do tipo `U` (nossa analogia com o zero natural). `par` é um predicado, pois toma um elemento do tipo `U` e retona um elemento do tipo proposição (`Prop`).

`adicao` é uma função que toma dois elementos do tipo `U` e retorna outro do mesmo tipo. Ora, podemos constatar:

```
1 #check par zero
2 #check adicao zero zero
3 #check par (adicao zero zero)
```

O `#check` da linha 1 informa que a expressão tem tipo `Prop`; na linha 2, tipo `U`; e, na linha 3, tipo `Prop`. Importante observar o papel dos parênteses acima, para que `par` receba apenas um elemento.

Uma função (ou relação) que recebe mais de um elemento tem notação `U → U → U`. A notação para predicados (`U → Prop`) e relações (`U → U → Prop`) funciona como se ambas fossem funções, porém retornassem `Prop`.

Vários conjuntos estão nas bibliotecas padrão do Lean, como os naturais, utilizados nos exemplos anteriores. O comando para o símbolo ℕ é `\nat` ou `\N`.

```
1 constant zero : ℕ
2 #check zero + zero
```

O `#check` da linha 2 retorna algo do tipo ℕ.

Podemos misturar Lógica Proposicional com Lógica de Primeira Ordem:

```
1 constant U : Type
2 constant zero : U
3 constant par : U → Prop
4 constant primo : U → Prop
5 constant igual : U → U → Prop
6 constant adicao : U → U → U
7
8 #check ¬ (par zero ∨ par (adicao zero zero)) ∧ primo zero
```

E o `#check` nos retorna algo do tipo `Prop`.

4.1.2 Quantificador Universal

Grande parte do poder da Lógica Proposicional se deve aos quantificadores. O símbolo ∀ é o quantificador universal, que representa "para todo". Quando ele é seguido de uma variável e de uma expressão, ele indica que aquela expressão é verdadeira para toda variável do domínio. Por exemplo:

- $\forall x\ (par(x) \vee impar(x))$
- $\forall y\ (par(y) \rightarrow impar(y+1))$

A primeira expressão nos diz que todo número é par ou ímpar (no caso dos naturais). A segunda diz que, para todo número, o fato dele ser par implica que seu sucessor é ímpar.

No Lean, é possível obter o símbolo ∀ digitando `\all`. Os exemplos anteriores ficam assim:

```
1 constant U : Type
2 constant par : U → Prop
3 constant impar : U → Prop
4 constant sucessor : U → U
5
6 #check ∀ x : U, par x ∨ impar x
7 #check ∀ y : U, par y → impar (sucessor y)
```

Os dois `#check`'s nos retornam `Prop`. Quando é utilizado o quantificador universal, por padrão devemos dizer o tipo da variável que o acompanha. No exemplo, `x : U`. Porém, o Lean é esperto o suficiente pra inferir o tipo da variável sozinho, então na maioria dos casos não será um problema omiti-lo.

Observe as três sentenças:

- $\forall x\ (par(x) \vee impar(x))$
- $\forall x\ par(x) \vee impar(x)$
- $\forall x\ (par(x)) \vee impar(x)$

Por uma questão de convenção, as duas últimas sentenças são equivalentes, enquanto a primeira é diferente. Neste contexto, estamos lidando com o **escopo** da variável x. A convenção diz, portanto, que o escopo da variável é o menor possível.

Curiosamente, o modo como o Lean lida com escopo é diferente: `∀ x : U, par x ∨ impar x` equivale a `∀ x : U, (par x ∨ impar x)`. Ou seja, o Lean busca o maior escopo possível.

Quando estamos lidando com quantificadores, a variável que o acompanha é dita **limitada** (*bound*, em inglês). Na expressão $\forall x\ A(x)$, a variável x é limitada. Isso significa que o x não representa um valor em si, mas apenas um "espaço reservado" para qualquer outra variável. Observe que a expressão $\forall y\ A(y)$ representa exatamente a mesma coisa.

Uma variável que não é limitada é chamada **livre**. Por exemplo, considere a expressão $\forall x\ y \leq x$. Ora, sabemos que x representa uma "espaço reservado" para toda variável do domínio. O que representa y, portanto? Um elemento específico do domínio. Digamos que o domínio é $\mathbb{N}$ e y representa o elemento zero. Então a expressão é verdadeira. Sendo assim, se trocarmos y por z (que representa, neste caso, outro elemento), então não vale a expressão $\forall x\ z \leq x$. Observe como trocar y por z fez toda a diferença!

No exemplo anterior, tanto y quanto z são variáveis livres. Quando uma expressão **não** contém variáveis livres, é chamada sentença.

Quantificadores também possuem regras de introdução e eliminação. Vejamos:

$$\frac{A(y)}{\forall x\ A(x)}\ \forall I \qquad \frac{\forall x\ A(x)}{A(t)}\ \forall E$$

A regra da esquerda demonstra a **introdução** do quantificador universal. Ela vale quando y não é livre em nenhuma hipótese não cancelada. A intuição neste caso é que, dado que vale $A(y)$ para algum y qualquer, então podemos dizer que vale para todo y. Mudamos o nome da variável apenas pra que a operação fique mais evidente e intuitiva.

A regra da direita demonstra a **eliminação** do quantificador universal. A intuição, neste caso, é que, se $A(x)$ vale para todo x, então vale para algum t qualquer do domínio.

Observe a semelhança dessas regras com aquelas relativas à implicação. No caso da introdução da implicação, assumimos A e provamos B, então $A \to B$. No caso da introdução do quantificador universal, assumimos x e mostramos $A(x)$, então $\forall x\ A(x)$. Para a eliminação da implicação, temos $A \to B$ e A, então temos B. Já no caso da eliminação do quantificador, temos $\forall x\ A(x)$ e y, então $A(y)$.

Vejamos um exemplo aplicando essas regras. Observe como derivar $\forall x\ A(x) \wedge B(x)$ a partir de $\forall x\ A(x)$ e $\forall x\ B(x)$:

$$\dfrac{\dfrac{\dfrac{\forall x\ A(x)}{A(y)}\,\forall E \qquad \dfrac{\forall x\ B(x)}{B(y)}\,\forall E}{A(y) \wedge B(y)}\,\wedge I}{\forall x\ A(x) \wedge B(x)}\,\forall I$$

Ora, e como representar isso em Lean? Vejamos a introdução:

```
1 constant U : Type
2 constant A : U → Prop
3
4 example : ∀ x, A x :=
5     assume y,
6     show A y, from sorry
```

Estamos mostrando $\forall x\ A(x)$ da seguinte forma: assumimos um y qualquer e provamos $A(y)$. Neste caso, necessitamos de uma prova de $A(y)$, o que justifica o uso do `sorry`.

Observe, agora, a regra da eliminação:

```
1 constant U : Type
2 constant A : U → Prop
3 constant h1 : ∀ x, A x
4 constant t : U
5
6 example : A t :=
7     h1 t
```

Dado que sabemos que $\forall x\ A(x)$ (por `h1`), podemos "aplicar" t e obter $A(t)$.

Vejamos agora nosso exemplo que deriva $\forall x\ A(x) \wedge B(x)$ a partir de $\forall x\ A(x)$ e $\forall x\ B(x)$:

```
constant U : Type
constants A B : U → Prop
constant h1 : ∀ x, A x
constant h2 : ∀ x, B x

example : ∀ x, A x ∧ B x :=
    assume y,
        have h3 : A y, from h1 y,
        have h4 : B y, from h2 y,
    show A y ∧ B y, from and.intro h3 h4
```

4.1.3 Quantificador Existencial

O quantificador existencial é representado pelo símbolo $\exists$ e representa "existe algum". Seguido de uma variável e uma expressão, ele significa que existe algum elemento no domínio tal que a expressão seja verdadeira. Alguns exemplos para o domínio dos naturais:

- $\exists x\ x \times x = x$
- $\exists y\ y \leq 0$
- $\forall x\ \exists y\ par(y) \wedge y > x$

Ora, as expressões significam:

- Existe número natural que é igual ao seu quadrado (0 e 1)
- Existe número menor do que ou igual a zero (o próprio zero)
- Para todo natural, existe um número que é par e maior do que ele

Em Lean, pode-se inserir $\exists$ digitando `\ex`. Vejamos os exemplos:

```
constant U : Type
constant par : U → Prop
constant maiorDoQue : U → U → Prop

#check ∃ x, x * x = x
#check ∃ y, y ≤ 0
#check ∀ x, ∃ y, par y ∧ maiorDoQue y x
```

Como no caso do quantificador universal, a variável que acompanha o quantificador é dita **limitada**. Por exemplo, x em $\exists x\ A(x)$. Variáveis não limitadas são **livres**.

Vejamos as regras de introdução e eliminação desse quantificador:

$$\frac{A(t)}{\exists x\ A(x)}\ \exists I \qquad \frac{\exists x\ A(x) \qquad \begin{array}{c}\overline{A(y)}\\ \vdots \\ B\end{array}}{B}\ \exists E$$

A primeira regra é a **introdução** do quantificador existencial. A intuição que segue é a de que, para mostrar que existe x tal que $A(x)$, basta mostrar que, para algum t, vale $A(t)$.

A segunda regra é a **eliminação** e vale quando y não é livre em B e em nenhuma hipótese não cancelada. É uma regra um pouco menos trivial de se entender. Vamos lá:

- Digo que existe algum x para o qual vale $A(x)$;
- Sei que, independentemente do y que eu escolher, de $A(y)$ é possível derivar B;
- Concluo que vale B.

Outra forma de se interpretar essa regra é a seguinte: sei que existe x tal que vale $A(x)$. A única maneira de eu ter certeza de que vale B é mostrando que, pra qualquer y, $A(y)$ implica em B. Tenho que assumir um y qualquer, pois não sei para qual y especificamente vale $A(y)$.

Tecnicamente, o que estamos fazendo é: dado que vale $\exists x\ A(x)$ e vale $\forall y\ (A(y) \to B)$, concluímos B, desde que B não contenha y livre. Para algumas pessoas, a utilização do quantificador universal ($\forall$) pode ser muito conveniente. Porém, para outras, pode ser um pouco mais confuso.

Assim como as regras do quantificador universal se assemelham às da implicação, as regras mostradas acima se assemelham àquelas da introdução e eliminação do operador lógico ou. Observe:

- Assim como de A eu derivo $A \vee B$, de $A(t)$ derivo $\exists x\ A(x)$. O que estamos derivando diz que pelo menos um dos argumentos é válido ou existe.
- Para mostrar C de $A \vee B$, assumo A e mostro C e depois assumo B e mostro C. No caso do nosso quantificador, para saírmos de $\exists x\ A(x)$ e chegarmos em B, assumo qualquer valor de y e mostro que de $A(y)$ chego em B. Nos dois casos, estamos verificando se, em todas as possibilidades, conseguimos concluir o que queremos.

Vejamos um exemplo que utiliza as duas regras, mostrando $(\exists x\ A(x)) \wedge (\exists x\ B(x))$ a partir de $\exists x\ (A(x) \wedge B(x))$:

$$\frac{\exists x\ (A(x) \wedge B(x)) \qquad \dfrac{\dfrac{\dfrac{\overline{A(y) \wedge B(y)}}{A(y)}\ \wedge E}{\exists x\ A(x)}\ \exists I \qquad \dfrac{\dfrac{\overline{A(y) \wedge B(y)}}{B(y)}\ \wedge E}{\exists x\ B(x)}\ \exists I}{(\exists x\ A(x)) \wedge (\exists x\ B(x))}\ \wedge I}{(\exists x\ A(x)) \wedge (\exists x\ B(x))}\ \exists E$$

Essas regras também podem ser utilizadas em Lean. Vejamos a introdução:

```
1 constant U : Type
2 constant A : U → Prop
3 constant t : U
4 constant h1 : A t
5
6 example : ∃ x, A x :=
7     exists.intro t h1
```

Para a introdução do quantificador existencial, utilizamos `exists.intro`, que requer dois argumentos: um termo e uma prova de que a expressão vale para esse termo. É o que está sendo feito no código acima. Na linha 4, sabemos que vale $A(t)$. Passando t e $A(t)$ como argumentos para `exists.intro`, temos $\exists x\ A(x)$.

Vejamos a regra da eliminação:

```
1 constant U : Type
2 constant A : U → Prop
3 constant h1 : ∃ x, A x
4 constant B : Prop
5
6 example : B :=
7     exists.elim h1
8         (assume y (h2 : A y),
9         show B, from sorry)
```

Utilizamos `exists.elim`, que toma dois argumentos: a expressão $\exists x\ A(x)$ e uma expressão que prova B a partir de y qualquer e $A(y)$.

Observe como fica mais clara a consideração feita anteriormente com relação ao quantificador universal ($\forall$): estamos dando para `exists.elim` as expressões $\exists x\ A(x)$ e $\forall y\ (A(y) \to B)$, e ela nos retorna B. Observe como o `show` nos ajuda a mostrar isso:

```
1 example : B :=
2     exists.elim h1
3         (show ∀ y, A y → B, from
4             assume y (h2 : A y),
5             show B, from sorry)
```

Vejamos nosso exemplo anterior, agora implementado em Lean, que mostra $(\exists x\ A(x)) \wedge (\exists x\ B(x))$ a partir de $\exists x\ (A(x) \wedge B(x))$:

```
1 constant U : Type
2 constants A B : U → Prop
3 constant h1 : ∃ x, A x ∧ B x
4
5 example : (∃ x, A x) ∧ (∃ y, B y) :=
6     exists.elim h1
```

```
7         (assume y (h2 : A y ∧ B y),
8             have h3 : A y, from h2.left,
9             have h4 : B y, from h2.right,
10            have h5 : ∃ x, A x, from exists.intro y h3,
11            have h6 : ∃ x, B x, from exists.intro y h4,
12        show (∃ x, A x) ∧ (∃ y, B y), from and.intro h5 h6)
```

4.1.4 Relativização

Os quantificadores $\forall$ e $\exists$ apresentados até agora variam sobre todos os elementos do domínio. Por exemplo, se eu especificar um domínio como sendo as espécies do reino animal, posso dizer que todas são fascinantes da seguinte forma: estabeleço um predicado $fascinante(x)$ que retorna V se uma espécie x é fascinante e escrevo: $\forall x\ fascinante(x)$.

Alguém, porém, que goste muito de borboletas, poderia dizer que apenas as borboletas são incríveis. Então ela poderia escrever: $\forall x\ (borboleta(x) \rightarrow incrivel(x))$. O que ela está dizendo, então, é: dentre todos os animais, se x é uma borboleta, então x é incrível. Como o quantificador universal varia sobre todo o domínio, utilizar a implicação como no exemplo é um truque para restringir o domínio apenas às borboletas. Isso é chamado **relativização**.

E se, agora, quiséssemos dizer que existe algum cachorro feliz? No caso do quantificador existencial, podemos **relativizá-lo** utilizando a conjunção: $\exists x\ (cachorro(x) \wedge feliz(x))$. Estamos dizendo que existe algum animal que é um cachorro e é feliz.

Talvez soe contraintuitivo utilizar a conjunção em detrimento da implicação, como no exemplo anterior. Porém, observe o que acontece se utilizarmos a implicação: $\exists x\ (cachorro(x) \rightarrow feliz(x))$. Isso equivale a dizer $\exists x\ (\neg cachorro(x) \vee feliz(x))$. Ora, estamos dizendo que existe algo que não é cachorro ou é feliz, ou seja, não é bem o que queríamos dizer.

4.1.5 Igualdade

Na lógica de primeira ordem também é utilizado o símbolo = para expressar o fato de que duas expressões se referem ao mesmo objeto, como em "s = t", em que definimos que ambos s e t estão se referindo ao mesmo objeto do universo. A igualdade é utilizada em expressões como "1 + 1 = 2"ou "seu professor é o meu pai", note que neste segundo exemplo não utilizamos o símbolo de forma explícita, no entanto, com as funções da linguagem afirmamos que "seu professor"e "meu pai"são expressões que se referem ao mesmo objeto.
A definição de igualdade e identidade ao longo da história foram frequentemente debatidas entre filósofos. Um exemplo é o paradoxo do Navio de Teseu, inspirado na história do herói grego Teseu que era navegava de Atena para Creta para ser sacrificado em um tributo. No paradoxo, temos A = navio que Teseu começou a viagem; e B = navio que Teseu terminou a viagem; ao longo da viagem o navio foi aos poucos necessitando de reparos, troca de peças, até o momento

que ao final da viagem, todas as peças do navio já haviam sido trocadas, com isso podemos afirmar que A = B? Em uma extensão do paradoxo, existe um outro barco, o Carniceiro, que pega as peças que Teseu joga ao mar e substitui em seu barco, ao final da viagem o Carniceiro possui todas as peças que o barco de Teseu tinha ao sair para navegar. Qual dos dois barcos é o de Teseu? Este paradoxo foi discutido por filósofos como Heráclito, Sócrates, Hobbes, John Locke e Leibniz. No entanto, em lógica ao usarmos o símbolo de igualdade é pressuposto a nossa interpretação do mundo, e escrever que "s = t"afirma que s e t representam exatamente o mesmo objeto.

Definição: Igualdade Duas expressões são iguais se referem-se a exatamente o mesmo objeto do universo e é utilizado o símbolo =, chamado de igualdade, para representar essa relação. Possui as seguintes propriedades:

- Reflexão: $t = t$, para qualquer termo t.
- Simetria: se $s = t$, então $t = s$.
- Transitividade: se $s = t$ e $t = v$, então $s = v$.

$$\frac{}{t = t}\ \text{refl} \qquad \frac{s = t}{t = s}\ \text{sim} \qquad \frac{s = t \qquad t = v}{s = v}\ \text{trans}$$

Exemplo: No universo $\mathbb{N}$ com constantes $\{1, 2, 3, 5, 7\}$ e funções de adição e multiplicação, temos duas hipóteses, $2 + 5 = 3 \cdot 2 + 1$ e $7 = 3 \cdot 2 + 1$. Pela propriedade da simetria, $3 \cdot 2 + 1 = 7$, e por fim com a proprieade da transividade podemos afirmar que $2 + 5 = 7$. Representando esta mesma demonstração em uma dedução natural, temos:

$$\dfrac{\dfrac{2 + 5 = 3 \cdot 2 + 1 \qquad \dfrac{7 = 3 \cdot 2 + 1}{3 \cdot 2 + 1 = 7}\ \text{sim}}{2 + 7 = 7}\ \text{trans}$$

Na dedução, utilizamos apenas a regra de simetria e em sequência a regra de transitividade. Como expressariamos esta simples dedução em Lean? A primeira necessidade é definir nosso universo.

```
1        variable N : Type
2        variable add : N → N → N
3        variable mul: N → N → N
4        variables um dois tres cinco sete: N
5
```

] Definimos a nossa linguagem lógica, com as funções de adição e multiplicação e algumas contantes, números. As primeiras propriedades da igualdade em Lean são utilizadas da forma `eq.refl` para a propriedade reflexiva, `eq.symm` para a propriedade da simetria e `eq.trans` para a propriedade da transividade. Neste exemplo iremos apenas utilizar as duas últimas. A dedução termina assim:

```
1        example (h1 : add dois cinco = add (mul tres dois) um)
2                (h2 : sete =  add (mul tres dois) um):
3        (add dois cinco = sete) :=
```

```
4         have h3 : (add (mul tres dois) um = sete), from eq.symm
      h2,
5         show add dois cinco = sete, from eq.trans h1 h3
6
```

Na linha 4 utilizamos a primeira regra, a `eq.symm` com a nossa hipótese `h2`, e terminamos a prova com a regra `eq.trans` utilizando das hipóteses `h1` e `h3` . Note que apesar de ser uma prova com apenas dois passos, foi uma prova extremamente verbosa, isto ocorreu pela maneira que definimos a nossa linguagem, para torná-la mais curta podemos utilizar de símbolox infixados da adição e da multiplicação e poderiamos utilizar os algorimos no lugar do nome dos números, no entanto, essa segunda sugestão é mais complicada de se realizar. A nova dedudção fica:

```
1         namespace hidden
2         constant N : Type
3         constant add : N → N → N
4         constant mul: N → N → N
5         constants um dois tres cinco sete: N
6
7         infix + := add
8         infix * := mul
9
10        example (h1 :  dois + cinco = (tres * dois) + um)
11                (h2 : sete =  (tres * dois) + um):
12                (dois + cinco = sete) :=
13            have h3 : (tres * dois) + um = sete, from eq.symm h2,
14            show dois + cinco = sete, from eq.trans h1 h3
15        end hidden
16
```

A primeira necessidade foi criar um `namespace hidden`, pois para definir um `infix`, precisamos que a nossa função seja uma `costant` e apenas podemos declarar `constant` dentro de um `namespace`. Dessa forma, alteramos todas as nossas variáveis para constantes e utilizamos na linha 7 e 8 a definição do `infix`, a sintaxe é o símbolo que será usado, `:=` e em seguida a função que será infixada. Essas primeiras propriedades da adição não são suficientes, por exemplo, em uma linguagem com o predicado par, a função da adição e os números 1 e 2, com as hipóteses $1+1=2$ e $par(2)$, como provaríamos que $par(1+1)$ também vale? É para isso que também são definidas propriedades da subsituição em predicados e funções.

Definição: Seja s e t constantes, P um predicado unário e r uma função unária. Para a relação de igualdade entre dois objetos também vale as propriedades:

- Subsituição em função: se $s = t$, então $r(s) = r(t)$.
- Subsituição em predicado: se $s = t$ e $P(s)$, então $P(t)$.

$$\frac{s = t}{r(s) = r(t)}\ \text{sub} \qquad \frac{s = t \qquad P(s)}{P(t)}\ \text{sub}$$

Exemplo: Agora com um universo que abrange os números reais, temos o predicado unário $irracional$, a função unária $raiz$, a função binária mul e as contantes $dois, tres, dezoito$. Com as hipóteses $raiz(dezoito) = tres * (raiz(dois))$ e $\forall x, irracional(x \cdot (raiz(dois)))$ queremos provar $irracional(raiz(dezoito))$. O primeiro passo é a exclusão do universal na segunda hipótese, utilizando a constante $tres$ obtemos $irracional(tres \cdot raizdois)$, para aplicar a substituição do predicado, precisamos invertar nossa igualdade utilizando a simetria, e por fim utilizamos a regra de substituição. A dedução natural fica:

$$\frac{\dfrac{raiz(dezoito) = tres \cdot raiz(dois)}{tres \cdot raiz(dois) = raiz(dezoito)}\ \text{sim} \qquad \dfrac{\forall x, irracional(x \cdot raiz(dois))}{irracional(tres \cdot raiz(dois))}}{irracional(raiz(dezoito))}\ \text{subs}$$

A mesma demosntração utilizando o Lean:

```
1         namespace hidden
2
3         constant N : Type
4         constant mul : N → N → N
5         constant raiz : N → N
6         constant irracional : N → Prop
7         constants dois tres dezoito : N
8
9         infix * := mul
10
11        example (h1 : raiz dezoito = tres * raiz dois)
12                (h2 : ∀ x : N, irracional(x * raiz dois)) :
13                irracional (raiz dezoito):=
14        have h3 : irracional(tres * raiz dois), from h2 tres,
15        have h4 : tres * raiz dois = raiz dezoito, from eq.symm
      h1,
16        show irracional (raiz dezoito), from eq.subst h4 h3
17
18        end hidden
19
```

Para a regra de substituição da igualdade utilizamos `eq.subst`, como utilizamos na linha 16. Note que além dessa regra, apenas utilizamos a exclusão do universal utilizando a constante `tres` na linha 14 e a regra de simetria da igualdade na linha 15.

4.1.6 Exercícios

1. Lembra-se dos exemplos, na introdução deste capítulo, que esperávamos que esclarecesse a necessidade da Lógica de Primeira Ordem? Formalize-os utilizando funções e predicados que forem convenientes.

(a) Todo natural é maior do que ou igual a zero.

(b) Existe número natural que é primo e é par.

2. Formalize as afirmações abaixo em Lógica de Primeira Ordem, utilizando os seguintes predicados e relações:

   $primo(x)$, $par(x)$, $impar(x)$, $perfeito(x)$ e $amigos(x, y)$.

   (a) Todo número primo maior do que 2 é ímpar.

   (b) A soma de dois números pares é par.

   (c) Existe um número perfeito menor do que 10.

   (d) 220 e 284 são números amigos.

3. Considere que os quantificadores variam sobre estudantes de determinada escola. Formalize as afirmações abaixo, utilizando os seguintes predicados e relações: $amigos(x, y)$, $roqueiro(x)$, $estudioso(x)$, $estudamJuntos(x, y)$ e $irmaos(x, y)$.

   Experimente formalizar também em Lean.

   (a) Roqueiros são amigos de todos.

   (b) Existem dois irmãos.

   (c) Existe um roqueiro estudioso.

   (d) Se duas pessoas estudiosas não estudam juntas, então elas não são irmãs.

Soluções:

1. (a) $\forall x\ x \geq 0$.

   (b) $\exists x\ primo(x) \wedge par(x)$.

2. (a) $\forall x\ ((primo(x) \wedge x > 2) \rightarrow impar(x))$.

   (b) $\forall x\ \forall y\ ((par(x) \wedge par(y)) \rightarrow par(x + y))$.

   (c) $\exists x\ (perfeito(x) \wedge x < 10)$.

   (d) $amigos(220, 284)$.

3. (a) $\forall x\ \forall y\ (roqueiro(x) \rightarrow amigos(x, y))$.

   (b) $\exists x\ \exists y\ irmaos(x, y)$.

   (c) $\exists x\ (roqueiro(x) \wedge estudioso(x))$.

   (d) $\forall x\ \forall y\ ((estudioso(x) \wedge estudioso(y) \wedge \neg estudamJuntos(x, y)) \rightarrow \neg irmaos(x, y))$.

4.2 Semântica

A seção anterior abordou a sintaxe da lógica de primeira ordem, ou seja, apresentou os símbolos e estruturas das fórmulas deste sistema. A semântica, por outro lado, define o valor-verdade das fórmulas pela **interpretação** e **valoração**.

Analisemos alguns exemplos deste sistema lógico. Sejam:

- $\mathbb{N}$ um domínio escolhido;
- $0, 1, 2, 3$ símbolos constantes;
- *adicao* e *sucessor* os símbolos de função que operam neste domínio;
- *par*, *impar*, *menorDoQue*,*menorIgualDoQue* os simbolos de predicado .

A expressão

$$\forall x \ (\ menorDoQue(0, x)\ )$$

é falsa, uma vez que para x assumindo o valor de 0, a relação *menorDoQue* não é válida. Já a expressão

$$\forall x \ (\ menorIgualDoQue(0, x)\ )$$

é verdadeira no domínio $\mathbb{N}$, porém se o domínio de interesse fosse $\mathbb{Z}$ a expressão se tornaria falsa.

Logo, o valor verdade de uma sentença depende de como os quantificadores, o domínio, funções, predicados e relações são interpretados. Porém, há algumas fórmulas que assumem sempre valor verdadeiro independente de qual for a interpretação, análogo à tautologia da lógica proposicional:

$$\forall x \ (\ par(x) \rightarrow par(x)\ )$$

Sentenças assim são chamadas **válidas**.

Analogamente à valoração em lógica proposicional, há o **modelo** em lógica de primeira ordem. Enquanto a valoração permitia que atribuíssemos valores-verdade (V ou F) à todas as fórmulas da lógica proposicional, a escolha de um modelo permite a atribuição de valores-verdade a todas as sentenças da lógica de primeira ordem.

4.2.1 Interpretações

Usamos anteriormente alguns símbolos para representar predicados e constantes. Alguns deles:

$$0,\ 1,\ par,\ maiorDoQue$$

Estes símbolos são autodescritivos considerando o domínio $\mathbb{N}$, e se torna natural sua valoração. Agora, sejam os seguintes predicados no mesmo domínio:

$$ligeiro, contente, facil$$

Quando o predicado *ligeiro* é verdadeiro?
Não conseguimos responder, uma vez que não foram dadas informações suficientes. Se *ligeiro* são os números ímpares, então $2, 4, 6, ...$ são *ligeiro* e $1, 3, 5, ...$ não são. Seja *contente* os múltiplos de 3. Então 6 é *ligeiro* e *contente*. Se *ligeiro* fossem os números da sequência de Fibonacci, então 6 seria *contente*, mas não seria *ligeiro*. Cada explicação dada (ímpar, múltiplos de 3, números da sequência de Fibonacci) são **interpretações**. E vemos que é necessário a interpretação para a valoração.

Assim como os predicados, podemos interpretar as funções, relações e constantes:

- A interpretação de um predicado unário P é um conjunto de elementos do domínio os quais P é verdadeiro.
- Para uma relação R com aridade n, a interpretação é o conjunto de todas as tuplas com n elementos para as quais R é verdadeiro.
- E por fim, a interpretaçãode de uma função f com aridade n, é uma função que relaciona n elementos do domínio a outro elemento também do domínio.

É importante ressaltar a diferença entre símbolo sintático e semântica do predicado, função, relação e constante. Veja que não faz sentido escrevermos a relação $sucessorDe(4, 3)$, pois $sucessorDe$ é um símbolo sintático sem significado por si só.
Outra distinção importante é entre os objetos dos domínio e os símbolos constantes. Se considerarmos o domínio U de todas as cores, conhecemos os objetos deste domínio, mas podemos escrever o símbolo constante *verde* e podemos interpretá-lo com verde ou como rosa, objetos do domínio. De maneira análoga, podemos definir os símbolos 0, 1, 2 e interpretálos como os objetos do domínio 0, 1 e 2.

4.2.2 Verdade em modelos

Na lógica proposicional a valoração dizia quais elementos deveriam ser interpretados como falsos e quais como verdadeiros. Já na lógica de primeira ordem serão avaliados cada termo e em seguida a interpretação é aplicada na estrutura. Mais adiante veremos exemplos que deixarão a ideia mais clara.

Suponhamos um domínio D, em linguagem de primeira ordem, e uma interpretação em D para cada símbolo da linguagem. Um **modelo** é esta estrutura formada por um domínio D e a interpretação relativa a este domínio. Um modelo fornece as informações necessárias para avaliarmos todas as sentenças da linguagem em verdadeiro ou falso.

Vamos relembrar a diferença entre termo e sentença e descrever as respectivas interpretações:

- *Termos* representam objetos e não possuem valor verdade. São termos: $a + b, f(x), c$. E a sua interpretação é definida pelo próprio modelo e são elementos do domínio. Para interpretar $a + b$, verificamos a interpretação do modelo para cada termo a e b e em seguida aplicamos a interpretação de $+$ à estes termos. Da mesma forma, para $f(x)$, analizamos a interpretação do termo x, dada pelo modelo e plicamos a interpretação de f ao termo x.

- *Sentenças* são relações ou predicados que assumem valor verdade. Alguns exemplos: $R(a, b), x + y < x, P(a)$. Para interpretar um predicado ou uma relação primeiro se interpreta os termos como objetos do domínio, e em seguida verificar se a interpretação do símbolo da relação é verdadeira para estes objetos do domínio.

Seja A uma sentença e $\mathbb{M}$ um modelo da linguagem de A. Por questão de praticidade, vamos adotar as notações $\mathbb{M} \vDash A$ (pode ser lido como *modela*, *satisfaz* ou *valida*) para quando o modelo $\mathbb{M}$ avalia a sentença A como verdadeira (**T**) e $\mathbb{M} \nvDash A$ para quando $\mathbb{M}$ avalia A como falsa (**F**).

4.2.3 Exemplos

4.2.4 Validação e consequência lógica

4.2.5 Correção e completude

G	G	B	b
b	b	G	G
g	B	b	b
G	g	G	B

4.3 Dedução Natural

Devemos estabelecer as regras de inclusão e exclusão de dedução para os quantificadores e para a igualdade. Listamos elas inicialmente:
Quantificador universal:

$$\frac{A(y)}{\forall x A(x)}\ \forall I \qquad \frac{\forall x A(x)}{A(t)}\ \forall E$$

Para inserir o quantificador universal devemos ter que a variável y não deve estar atrelada em nenhuma hipótese, isto é, em todas as hipóteses não canceladasela não pode ser livre.
Quantificador existencial:

$$\frac{A(t)}{\exists x A(x)}\,\exists I \qquad \frac{\exists x A(x) \qquad \begin{array}{c}\overline{A(t)}\\ \vdots \\ B\end{array}}{B}\,\exists E$$

Para retirarmmos o quantificador existencial, a variável t não pode estar livre em B, isto é, ela não deve estar livre em qualquer hipótese não cancelada.

Igualdade:

$$\frac{}{t = t}\,\text{refl} \qquad \frac{t = s}{s = t}\,\text{sim}$$

$$\frac{t = s \qquad s = v}{t = v}\,\text{trans} \qquad \frac{t = s}{r(t) = r(s)}\,\text{subs} \qquad \frac{t = s \qquad P(t)}{P(s)}\,\text{subs}$$

Iremos passar por cada um destas regras para apresentá-las com o auxílio de exemplos.

4.3.1 Quantificador universal

Como primeiro exemplo segue abaixo uma dedução natural de $(\forall x P(x) \to \forall x Q(x)) \to \forall x(P(x) \to Q(x))$. Note que apesar de parecer uma implicação de duas fórmulas idênticas, na premissa a propriedade P recebe uma variável e a propriedade Q pode receber outra variável, enquanto que na conclusão o valor que x assume é o mesmo para P e Q.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\forall x P(x) \to \forall x Q(x)}\,1}{P(t) \to \forall x Q(x)}\,\forall E}{P(t) \to Q(t)}\,\forall E}{\forall x(P(x) \to Q(x))}\,\forall I}{\forall x P(x) \to \forall x Q(x) \to \forall x(P(x) \to Q(x))}\,1$$

O primeiro passo será a exclusão da implicação (a implicação principal da fórmula), assumindo $\forall x P(x) \to \forall x Q(x)$ como nossa hipótese. Aplicamos uma primeira exclusão do universal em $\forall x P(x)$ e novamente aplicamos a exclusão do universal em $\forall x Q(x)$. Note que ao definirmos a variável com o mesmo nome t em ambas as exclusões do universal foi o que permitiu inserir um universal que englobe ambas variáveis, que é o passa seguinte.

Vamos para um outro exemplo, utilizando as hipóteses $\forall x(\neg Q(x) \to R(x))$ e $\forall x(P(x) \wedge \neg Q(x))$ provaremos $\forall x R(x)$:

$$\frac{\dfrac{\dfrac{\dfrac{\forall x(P(x) \wedge \neg Q(x))}{P(t) \wedge \neg Q(t)}\,\forall E}{\neg Q(t)} \qquad \dfrac{\forall x(\neg Q(x) \to R(x))}{\neg Q(t) \to R(t)}\,\forall E}{R(t)}}{\forall x R(x)}\,\forall I$$

Vamos inicialmente utilizar de nossas hipóteses, em $\forall x(P(x) \wedge \neg Q(x))$ nós executamos a primeira exclusão do universal, utilizando a variável t e em seguida executamos a exclusão do $\wedge$ visto que não é necessário o $P(t)$ na dedução. Do outro lado executamos mais uma exclusão do universal em $\forall x(\neg Q(x) \rightarrow R(x))$ e atribuímos novamente a mesma variável t. Com $\neg Q(t)$ e $\neg Q(t) \rightarrow R(t)$ podemos realizar uma exclusão da implicaçã e por fim incluímos o universal em $\forall I$.

4.3.2 Quantificador existencial

Lembrando a regra de exclusão do existencial, o que fazemos é que com $\exists x A(x)$, supomos um y arbitrário que satisfaça $A(y)$, a partir desta premissa chegamos até B, uma fórmula que não contém y ou qualquer outra variável aberta em alguma hipótese não cancelada, e podemos concluir B.
Já a regra da inclusão do existencial, se uma propriedade vale para um y arbitrário, então existe uma váriavel em que ela é válida.
Iniciemos com uma demosntração simples de que se existe x que satisfaça A e B, então existe x que satisfaça A.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A(t) \wedge B(t)}\,2}{A(t)}}{\exists x A(x)} \qquad \dfrac{}{\exists x(A(x) \wedge B(x))}\,1}{\exists x A(x)}\,2}{\exists x(A(x) \wedge B(x)) \rightarrow \exists x A(x)}\,1$$

A primeira etapa foi a retirada da implicação, passando $\exists x(A(x) \wedge B(x))$ como uma hipótese, e utilizando ela aplicamos a regra de exclusão do existencial, criando uma variável t arbitrária em que $A(x) \wedge B(x)$ sejá válido, com isso podemos concluir $A(t)$, note que em 2 não podemos concluir $A(t)$, pois a variável ainda está aberta em $A(t) \wedge B(t)$, por isso inserimos o existencial e concluímos $\exists x A(x)$, resultado que queríamos obter.
O próximo exemplo relaciona os quantificadores universal e existencial, se para todo x A é válido, então existe algum x que A sejá válido.

$$\dfrac{\dfrac{\forall x A(x)}{A(t)}}{\exists x A(x)}$$

Note que se $A(x)$ vale para todo x, também vale para um t específico e no passo seguinte poderíamos tanto utilizar a regra da inclusão do universal ou do existencial. Outro comentário relevante é que não necessariamente precisávamos concluir o existencial utilizando a mesma variável x.
Vamos provar mais uma relação entre os quantificadores, iremos que provar que se para todo x não vale A, então não existe x tal que A valha:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x A(x)}\,2 \quad \dfrac{\dfrac{}{A(t)}\,3 \quad \dfrac{\dfrac{}{\forall x \neg A(x)}\,1}{\neg A(t)}}{\bot}}{\bot}\,3}{\neg \exists x A(x)}\,2}{\forall x \neg A(x) \to \neg \exists x A(x)}\,1$$

Para a nossa dedução a primeira etapa foi desmontar a implicação, passando $\forall x \neg A(x)$ como uma de nossas hipóteses, em seguida, como temos uma negação devemos chegar até ao falso. Utilizamos $\exists x A(x)$ como mais uma de nossas hipóteses e aplicamos a regra de exclusão do existencial, dessa forma obtemos no mesmo ramo $A(t)$ e $\neg A(t)$, obtendo a contradição que estávamos procurando.

4.3.3 Igualdade

TO DO

4.3.4 Dedução natural no LEAN

No Lean a dedução ocorre de forma similar a dedução em primeira ordem, apenas devemos utilizar de novos símbolos e das regras de exclusão e inclusão dos quantificadores. Os símbolos $\exists$ e $\forall$ são escritos no Lean como \exist e \all. Para a regra de exclusão do universal apenas passamos para uma proposição $\forall x A(x)$ uma letra, por exemplo t, para termos $A(t)$. Para a inclusão do universal, devemos assumir uma letra, por exemplo t, utilizando o "assume"e com esta letra livre de qualquer hipótese provamos $A(t)$, dessa forma o Lean é capaz de inferir $\forall x A(x)$.
Vamos utilizando o Lean provar o nosso primeiro exemplo do quantificador universal, $\forall x P(x) \to \forall x Q(x) \to \forall x (P(x) \wedge Q(x))$:

```
1  variable U : Type
2  variables P Q : Type → Prop
3
4  example : (∀x, P x) → (∀x, Q x) → ∀x P x ∧ Q x :=
5  assume h₁ : ∀x, P x,
6  assume h₂ : ∀x, Q x,
7  assume t,
8  have h₃ : P t, from h₁ t,
9  have h₄ : Q t, from h₂ t,
10 show P t ∧ Q t, from and.intro h₃ h₄
```

Note que na linha 4 utilizamos da inclusão do universal, assumimos um t e desejamos provar $P(t) \wedge Q(t)$ sem que t possua qualquer restrição e nas linhas 5 e 6 utilizamos a exclusão do universal, temos fórmulas do tipo $\forall x P(x)$ e passamos a letra t, obtendo $P(t)$. Além disso note nesse exemplo a importância da utilização dos parenteses para delimitar a ação do quantificador, quando definimos o exemplo, caso não tivessemos colocado o parenteses em $(\forall x, Px)$, o primeiro

quantificador $\forall x$ seria interpretado como se valesse para toda a expressão, incluindo as duas implicações.
Vamos também provar o segundo exemplo em Lean:

```
variable U : Type
variables P Q R : U → Prop

example (h₁ : ∀ x, P x ∧¬ Q x) (h₂ : ∀ x, ¬ Q x → R x) :
∀ x, R x :=
assume t,
have h₃ : P t ∧ ¬ Q t, from h₁ t,
have h₄ : ¬ Q t → R t, from h₂ t,
show R t, from h₄ h₃.right
```

Note que neste simples exemplo, apesar de possuirmos fórmulas maiores, apenas realizamos simples regras de inclusão e exclusão do universal.
Para as regras do existencial, na exclusão utilizamos "exists.elim"seguida por uma proposição do tipo $\forall x A(x)$, para provarmos B, devemos provar $Ay \to B$, ou seja, assumimos um y e Ay e chegamos até B, concluindo assim a exclusão do existencial. Para a inclusão do existencial utilizamos "exists.intro", devemos passar uma letra, por exemplo t, e uma prova de que $A(t)$ vale.
Vamos provar o nosso primeiro exemplo existencial com o Lean, $\exists x(A(x) \wedge B(x)) \to \exists x A(x)$:

```
variable U : Type
variables A B : U → Prop

example : (∃x, A x ∧ B x) → ∃x, A x :=
assume h₁ : ∃ x, A x ∧ B x,
exists.elim h₁
    (assume t (h₂ : A t ∧ B t)
        show ∃ x, A x, from exists.intro t h₂.left)
```

4.3.5 Exercícios

1. Temos que que $A \to A \vee B$, apresente uma dedução natural para $(\forall x A(x)) \to (\forall x A(x) \vee B(x))$. Em seguida, demonstre utilizando Lean.
2. Assim como temos em lógica proposicional que $A \wedge B \to A$, de uma dedução natural para $(\forall x A(x) \wedge B(x)) \to (\forall x A(x))$. Em seguinda, demonstre utilizando Lean.
3. Apresente uma dedução natural e uma prova Lean para a relação $(\forall x A(x) \to \neg B(x)) \iff \neg(\exists x A(x) \wedge B(x))$.
4.Resolve as seguintes deduções naturais com auxílio do Lean.
a) $\forall x A(x) \iff \neg\exists x \neg A(x)$.
b) $\forall x(A(x) \to B(x)) \to (\forall x A(x) \to \forall x B(x))$.
c) Utilizando $\forall y \neg A(y)$ e $\forall x A(x) \vee B(x)$ prove $\forall x B(x)$.
d) Utilizando $\forall x A(x) \wedge B(x)$ e $\forall x A(x) \wedge B(x) \to C(x)$ prove $\forall x A(x) \wedge C(x)$.

e)De uma prova de $\forall y P(y) \to P(f(fy))$ utilizando de $\forall x P(x) \to P(f(x))$.
5. Utilizando dos predicados unários *even* e *odd*, e da função unária *succ*, das hipóteses $\forall x even(x) \vee odd(x)$ e $\forall x odd(x) \to even(succ(x))$, prove que $\forall x even(x) \vee even(succ(x))$.
6. Prove que $y = x \to y = z \to x = z$.
7. Retornando ao capítulo anterior com o exercício de Ana, Cláudia e Maria e seus três vestidos agora podemos resolver o problema utilizando de proposição de primeira ordem. Na situação, temos:
Três irmãs - Ana, Maria e Cláudia - foram a uma festa com vestidos de cores diferentes. Uma vestia azul, a outra branco e a Terceira preto. Chegando à festa, o anfitrião perguntou quem era cada uma delas. As respostas foram:
- A de azul respondeu: "Ana é a que está de branco"
- A de branco falou: "Eu sou Maria"
- A de preto disse: "Cláudia é quem está de branco"
O anfitrião foi capaz de identificar corretamente quem era cada pessoa considerando que:
- Ana sempre diz a verdade
- Maria às vezes diz a verdade
- Cláudia nunca diz a verdade
Pensando um pouco sobre o problema, pode-se concluir que a Ana estava com o vestido preto, a Cláudia com o branco e a Maria com o azul.
a) Escreva as fórmulas de primeira ordem necessárias para o problema.
b) Utilizando da ferramente Lean, apresente uma deduração natural que prove que Ana veste preto, Cláudia branco e Maria azul.

4.3.6 Gabarito

1.

```
variable U : Type
variables A B : U → Prop
--term mode
example : (∀ x, A x) → ∀ x, A x ∨ B x :=
assume h1: ∀ x, A x,
assume t,
have h2 : A t, from h1 t,
or.inl h2

--tatics
example : (∀ x, A x) → ∀ x, A x ∨ B x :=
begin
intros h1 t,
exact or.inl (h1 t)
end
```

2.

```
variable U : Type
```

```
variables A B : U → Prop
--term mode
example : (∃ x, A x ∧ B x) → ∃ x, A x :=
assume h1 : ∃ x, A x ∧ B x,
exists.elim h1
    (assume (t) (h2 : A t ∧ B t),
    have h3 : A t, from h2.left,
    show ∃ x, A x, from exists.intro t h3)

--tatics
example : (∃ x, A x ∧ B x) → ∃ x, A x :=
begin
intros h1,
cases h1 with t h2,
apply exists.intro t,
exact h2.left
end
```

3.

```
variable U : Type
variables A B : U → Prop

--term mode
example : (∀ x, A x → ¬ B x) ↔ ¬ ∃ x, A x ∧ B x :=
iff.intro
  (assume h1 : ∀ x, A x → ¬ B x,
  assume h2 : ∃ x, A x ∧ B x,
  exists.elim h2
    (assume (t) (h3 : A t ∧ B t),
    have h4 : A t → ¬ B t, from h1 t,
    have h5 : ¬ B t, from h4 h3.left,
    show false, from h5 h3.right))
  (assume h2 : ¬ ∃ x, A x ∧ B x,
  assume t,
  assume h3 : A t,
  assume h4 : B t,
  have h5 : A t ∧ B t, from and.intro h3 h4,
  have h6 : ∃ x, A x ∧ B x, from exists.intro t h5,
  show false, from h2 h6)

--observe que no term mode podemos passar mais de uma ́hiptse
--para o comando assume
example : (∀ x, A x → ¬ B x) ↔ ¬ ∃ x, A x ∧ B x :=
iff.intro
  (assume (h1 : ∀ x, A x → ¬ B x) (h2 : ∃ x, A x ∧ B x),
  exists.elim h2
```

```
    (assume (t) (h3 : A t ∧ B t),
    have h4 : A t → ¬ B t, from h1 t,
    have h5 : ¬ B t, from h4 h3.left,
    show false, from h5 h3.right))
  (assume (h2 : ¬ ∃ x, A x ∧ B x) t (h3 : A t) (h4 : B t),
   have h5 : A t ∧ B t, from and.intro h3 h4,
   have h6 : ∃ x, A x ∧ B x, from exists.intro t h5,
   show false, from h2 h6)

--tatics
example : (∀ x, A x → ¬ B x) ↔ ¬ ∃ x, A x ∧ B x :=
begin
apply iff.intro,
intros h1 h2,
cases h2 with t h3,
exact ((h1 t) h3.left) h3.right,
intros h1 t h2 h3,
exact h1 (exists.intro t (and.intro h2 h3))
end
```

4)a):

```
variable U : Type
variables A : U → Prop

--ida
--term mode
example : (∀ x, A x) → (¬ ∃ x, ¬ A x) :=
assume h1 : ∀ x, A x,
assume h2 : ∃ x, ¬ A x, show false, from
    (exists.elim h2
        (assume t (h3 : ¬ P t),
            have h4 : A t, from h1 t, h3 h4))

--tatics
example : (∀ x, A x) → (¬ ∃ x, ¬ A x) :=
begin
intros h1,
intro h2,
apply exists.elim h2,
intro t,
intro h3,
exact h3 (h1 t)
end

--volta
--term mode
```

```
example : (∃ x, ¬ A x) → ¬ ∀ x, A x :=
assume h1 : ∃ x, ¬ A x,
assume h2 : ∀ x, A x,
show false, from
    (exists.elim h1
        (assume t (h3 : ¬ A t),
            have h4 : A t, from h2 t,
            h3 h4))

--tatics

example : (∃ x, ¬ A x) → ¬ ∀ x, A x :=
begin
intro h1,
intro h2,
apply exists.elim h1,
intro t,
intro h3,
exact h3 (h2 t)
end
```

4.b)

```
variable U : Type
variables A B : U → Prop

--term mode
example : (∀ x, A x → B x) → ((∀ x, A x) → ∀ x, B x) :=
assume h1 : ∀ x, A x → B x,
assume h2 : ∀ x, A x,
assume t,
have h3 : A t → B t, from h1 t,
have h4 :  A t, from h2 t,
show B t, from h3 h4


--tatics
example : (∀ x, A x → B x) → ((∀ x, A x) → ∀ x, B x) :=
begin
intros h1 h2 t,
exact (h1 t) (h2 t)
end
```

4.c)

```
open classical
variable U : Type
variables A B : U → Prop
```

```
4  --term mode
5  example (h1 : ∀ y, ¬ A y) (h2: ∀ x, A x ∨ B x) : ∀ x, B x :=
6  assume t,
7  have h3 : ¬ A t, from h1 t,
8  have h4 : A t ∨ B t, from h2 t,
9  or.elim h4
10     (assume h5 : A t,
11     by_contradiction
12     (assume h6 : ¬ B t, show false, from h3 h5))
13     (assume h6 : B t, show B t, from h6)
14
15 --tatics
16 example (h1 : ∀ y, ¬ A y) (h2: ∀ x, A x ∨ B x) : ∀ x, B x :=
17 begin
18 intro t,
19 cases (h2 t),
20 apply classical.by_contradiction,
21 intro h3,
22 exact (h1 t) h,
23 exact h
24 end
```

4.d)

```
1  variable U : Type
2  variables A B C : U → Prop
3
4  --term mode
5  example (h1 : ∃ x, A x ∧ B x) (h2 : ∀ x, (A x ∧ B x → C x)) : ∃
       x, A x ∧ C x :=
6  exists.elim h1
7      (assume t (h3 : A t ∧ B t),
8      have h4 : A t ∧ B t → C t, from h2 t,
9      have h5 : C t, from h4 h3,
10     have h6 : A t ∧ C t, from and.intro  (h3.left) h5,
11     show ∃ x, A x ∧ C x, from exists.intro t h6)
12
13 --tatics
14 example (h1 : ∃ x, A x ∧ B x) (h2 : ∀ x, (A x ∧ B x → C x)) : ∃
       x, A x ∧ C x :=
15 begin
16 cases h1 with t h3,
17 apply exists.intro t,
18 apply and.intro,
19 exact h3.left,
20 exact (h2 t) h3
21 end
```

4.e)

```
variable U : Type
variable f : U → U
variable P : U → Prop
example  (h1 : ∀ x, P x → P (f x)) : ∀ y, P y → P ( f (f y)) :=
begin
intros t h2,
exact (h1 (f t))((h1 t) h2)
end
```

5.

```
variables even odd: U → Prop
variable s : U → U
example (h1 : ∀ x, even(x) ∨ odd(x)) (h2: ∀ x, odd(x) → even(s(x
    )))
: (∀ x, even(x) ∨ even(s(x))) :=
assume t,
have h3 : even(t) ∨ odd(t), from h1 t,
have h4 : odd(t) → even(s(t)), from h2 t,
or.elim h3
    (assume h5 : even(t), show even(t) ∨ even(s(t)),
    from or.inl h5)
    (assume h5 : odd(t),
    have h6: even(s(t)), from h4 h5,
    show even(t) ∨ even(s(t)), from or.inr h6)

example (h1 : ∀ x, even(x) ∨ odd(x)) (h2: ∀ x, odd(x) → even(s(x
    )))
: (∀ x, even(x) ∨ even(s(x))) :=
begin
intro t,
apply or.elim (h1 t),
intro h3,
apply or.inl,
exact h3,
intro h3,
apply or.inr,
exact (h2 t) h3
end
```

6.

```
variable N : Type
variables x y z : N

example : y = x → y = z → x = z :=
```

```
assume h1 : y = x,
assume h2 : y = z,
show x = z, from eq.trans (eq.symm h1) h2

example : y = x → y = z → x = z :=
begin
intros h1 h2,
apply eq.trans,
apply eq.symm,
exact h1,
exact h2
end
```

7.

```
section
variable {pessoa : Type}
variable {cor : Type}
variables {Ana Maria Claudia : pessoa}
variables {preto branco azul : cor}
variable {veste : pessoa → cor → Prop}

-- Restricoes:

-- Se uma usa uma cor, as outras nao podem usar essa cor
variable r1 : ∀c, ∀x, ∀y, (veste x c) ∧ ¬(x = y) → ¬(veste y c)
-- Se uma pessoa usa tal cor, nao pode usar as outras cores
variable r2 : ∀x, ∀c, ∀d, (veste x c) ∧ ¬(c = d) → ¬(veste x d)

-- Raciocinios basicos:

--Se Ana for a de azul, ela esta de branco
variable h1 : (veste Ana azul) → (veste Ana branco)
--Se Ana for a de branco, entao Maria esta de branco
variable h2 : (veste Ana branco) → (veste Maria branco)
--Se ana for de preto, entao Claudia esta de branco
variable h3 : (veste Ana preto) → (veste Claudia branco)
--Se Ana ãno usa azul e nem branco, ela esta de preto
variable h4 : ¬ (veste Ana azul) ∧ ¬ (veste Ana branco)  →  (
    veste Ana preto)
--Se Ana veste preto e Claudia veste branco, entao Maria esta de
    azul
variable h5 : (veste Ana preto) ∧ (veste Claudia branco) →  (
    veste Maria azul)

variable d1 : ¬ (azul = branco)
variable d2 : ¬ (Ana = Maria)
```

```
30 -- prova de que Ana veste preto
31 lemma Ana_p : (veste Ana preto) :=
32 have n1 : ¬ (veste Ana azul), from
33 assume m1 : veste Ana azul, show false, from
34     have q1 : veste Ana branco, from h1 m1,
35     have q2 : ¬ veste Ana branco, from (r2 Ana azul branco (
36         and.intro m1 d1)),
37 q2 q1,
38 have n2 : ¬ (veste Ana branco), from
39 assume m2 : veste Ana branco, show false, from
40     have q1 : veste Maria branco, from h2 m2,
41     have q2 : ¬ veste Maria branco, from (r1 branco Ana Maria (
       and.intro m2 d2)),
42 q2 q1,
43 show veste Ana preto, from h4 (and.intro n1 n2)
44
45 theorem solution : (veste Ana preto) ∧ (veste Claudia branco) ∧ (
       veste Maria azul) :=
46 have n1 : veste Ana preto, from (Ana_p r1 r2 h1 h2 h4 d1 d2),
47 have n2 : veste Claudia branco, from h3 (Ana_p r1 r2 h1 h2 h4 d1
       d2),
48 have n3 : veste Maria azul, from h5 (and.intro n1 n2),
49 and.intro n1 (and.intro n2 n3)
50
51 end
```

Capítulo 5

Conjuntos

5.1 Introdução

Para tratar de conjuntos iremos tentar uma abordagem que não deixe de lado o rigor matemático necessário, mas que ao mesmo tempo permita utilizar o Lean e contextualizar exemplos do dia a dia.

Caro leitor, como você definiria conjunto? Vamos, pense um pouco. No século XIX, o matemático Georg Cantor, no jornal acadêmico Mathematische Annalen, descreveu um conjunto (ou utilizando sua terminologia, *Menge*) como, em tradução livre: "Por conjunto nós entendemos qualquer coleção M de objetos deteminados e distintos (chamados de elementos de M) da nossa intuição ou do nosso pensamento em um todo."

Ou seja, mesmo um conjunto podendo ser algo tão abstrato quanto os números naturais ($\mathbb{N}$), os números reais ($\mathbb{R}$) e o Conjunto de Cantor, podemos ter coisas menos abstratas como o conjunto das palavras desse texto, o dos planetas do Sistema Solar ou até dos alunos de sua turma. A questão é, todos esses conjuntos podem ser mais intuitivos ou abstratos (além de como são definidos) dependendo da pessoa que irá interpretá-los, por exemplo, o conjunto dos naturais pode ser definido como $\mathbb{N} = \{1, 2, 3, ...\}$ ou $\mathbb{N} = \{0, 1, 2, 3, ...\}$ (para não desagradar ninguém), o de planetas do Sistema Solar $P = \{Mercurio, Venus, Terra, Marte, Jupiter, Saturno, Urano, Netuno\}$ (lembramos que se este livro fosse escrito há uns 15 anos, Plutão ainda seria considerado como um planeta, isto é, estaria no conjunto), por consequência, vemos que a interpretação do que determinado conjunto representa varia de pessoa para pessoa, mesmo que a ideia principal continue a mesma.

Nosso intuito, durante essa aventura pelo mundo dos conjuntos será entender melhor certos conceitos e definições (como o fato do conjunto vazio estar contido em todos os conjuntos), através de demonstrações, exemplos e exercícios, aumentando sua capacidade de abstração e a nossa também, já que para escrever esse capítulo nós teremos que ir além, pois não queremos apenas entender o que está aqui, mas que você entenda e aprenda também.

5.2 Fundamentações

5.2.1 Notações

Nesta seção iremos começar a introduzir notações matemáticas e algumas definições sobre conjuntos.

Pertence e Não Pertence: Quando um determinado elemento x faz parte de determinado conjunto A, nós dizemos que x pertence a A (denotamos $x \in A$). Caso x não faça parte de A, diz-se que x não pertence a A (denota-se $x \notin A$).

Contém, Contido e similares: Já quando um conjunto B possui todos os elementos que A possui e, B tem, pelo menos, um objeto que A não possua, dizemos que A está contido em B ($A \subset B$) ou que B contém A ($B \supset A$). Também podemos chamar A de subconjunto próprio de B.

Se não sabemos se B possui um objeto que A não possua (e B ainda possui todos os elementos que A), denotamos $A \subseteq B$, que significa $A \subset B$ ou $A = B$, de modo equivalente $B \supseteq A$, significa $B \supset A$ ou $A = B$. Neste caso, dizemos que A é um subconjunto de B.

Se A possui pelo menos um elemento que B não possua, dizemos que A não está contido em B ($A \not\subset B$) ou que B não contém $B \not\supset A$, assim $A \neq B$.

Interseção: Se estamos interessados em conjuntos/elementos que pertencem simultaneamente a dois conjuntos A e B, dizemos que estamos interessados na interseção de A e B (denotada como $A \cap B$).

União: Já se estamos interessados nos conjuntos/elementos que fazem parte de A ou de B dizemos que, nosso objetivo é a união de A e B ($A \cup B$).

Universo: Quando estamos trabalhando com conjuntos é comum definirmos quem é nosso universo ($\mathcal{U}$), isto é, o conjunto que conterá todos os conjuntos/elementos que estaremos trabalhando em um contexto. Por exemplo, na reta real nosso universo é $\mathcal{U} = \mathbb{R}$.

Conjunto Complementar: Sendo A um conjunto, dizemos que o conjunto A complementar ou complemento de A (denotado como $\overline{A}$ ou A^C) contém todos os conjuntos/elementos que não estão contidos/pertencem a A, mas fazem parte de nosso universo ($\mathcal{U}$).

Diferença de Conjuntos: Quando temos dois conjuntos e nosso objetivo são os conjuntos/elementos que pertencem a um destes conjuntos, mas não do outro, dizemos que estamos interessados na diferença destes conjuntos. No caso, se queremos os conjuntos/elementos de B, mas não desejamos pegar os que também pertencem a A, queremos os elementos/conjuntos que pertencem a diferença de B com A (denotamos como $B - A$ ou $B \backslash A$).

5.2.2 Definições

Seja A e B conjuntos quaisquers, $\mathcal{U}$ nosso universo e $\emptyset$ o conjunto vazio. Assim, temos formalmente (e em Linguagem de Primeira Ordem):

- **Conjunto Vazio**: $\emptyset = \{x|false\}$

$$\forall x(x \in \emptyset \iff \bot)$$

- **Universo**: $\mathcal{U} = \{x|true\}$

$$\forall x(x \in \mathcal{U} \iff \top)$$

- **União**: $A \cup B = \{x|x \in A \vee x \in B\}$

$$\forall x(x \in A \cup B \iff x \in A \vee x \in B)$$

- **Interseção**: $A \cap B = \{x|x \in A \wedge x \in B\}$

$$\forall x(x \in A \cap B \iff x \in A \wedge x \in B)$$

- **Diferença**: $A \backslash B = \{x|x \in A \wedge x \notin B\}$

$$\forall x(x \in A \backslash B \iff x \in A \wedge x \notin B)$$

- **Complementar**: $\overline{A} = \mathcal{U} \backslash A = \{x|x \in \mathcal{U} \wedge x \notin A\}$

$$\forall x(x \in \overline{A} \iff x \notin A)$$

5.2.3 Axiomas

Agora iremos apresentar alguns axiomas que servirão como base para todo o desenvolvimento dos conteúdos aqui propostos. Quando utilizarmos a palavra elemento, estaremos utilizando-a com a ideia de que um conjunto que pertença a outro é um elemento do segundo, para evitar repetir o uso excessivo da palavra conjunto.

Axioma da Extensionalidade: Dois conjuntos são iguais, se e somente se, todo elemento que pertence ao primeiro conjunto pertence ao segundo e todo elemento que pertence ao segundo também pertence ao primeiro, ou seja:

$$\forall A\ \forall B\ (A = B) \iff (\forall x\ (x \in A \iff x \in B))$$

Através desse axioma fica mais claro de entender duas propriedades dos conjuntos.

Deste axioma, vem a explicação do motivo de que a ordem dos elementos de um conjunto não importa. Pois dado dois conjuntos com os mesmos elementos, mas em ordem diferente (por exemplo, $X = \{a, b, c, d, e, f\}$ e o conjunto $Y = \{e, c, f, b, a, d\}$) eles ainda satisfazem a propriedade de que se t pertence a um deles implica t pertencer ao outro. Outro ponto interessante é que não importa se um conjunto possui elementos repetidos ele continuará igual ao que possui apenas um elemento, isto é, $X = \{a, b, d, e\}$ é igual ao $Y = \{b, a, d, a, e, b, b\}$. Ou seja, em um conjunto não importa a ordem e nem as repetições de elementos.

Axioma da Existencia do Conjunto Vazio: Diremos que no nosso universo ($\mathcal{U}$), existe um conjunto tal que ele não contém ninguém, ou seja, ele é vazio, daí seu nome, Conjunto Vazio (denotado por $\emptyset$). O axioma é:

$$\exists A\ \forall x\ \neg (x \in A)$$

Unicidade do Conjunto Vazio

Podemos provar a unicidade do conjunto vazio a partir dos dois axiomas acima. Suponhamos que existam dois conjuntos (A e B) com a propriedade do conjunto vazio, assim utilizando o axioma da Extensão concluíremos que eles são iguais, seja t arbitrário:

$$\dfrac{\forall a \forall b (a = b) \iff (\forall x (x \in a \iff x \in b))}{\dfrac{(A = B) \iff (\forall x (x \in A \iff x \in B))}{\text{A=B}}}$$

$$\dfrac{\dfrac{t \in A \qquad \dfrac{\dfrac{\forall x \neg (x \in A)}{\neg (t \in A)}\,{}^{1}}{\bot}}{t \in B} \qquad \dfrac{t \in B \qquad \dfrac{\dfrac{\forall x \neg (x \in B)}{\neg (t \in B)}\,{}^{1}}{\bot}}{t \in A}}{\dfrac{t \in A \iff t \in B}{\forall x (x \in A \iff x \in B)}} \iff_{1}$$

Axioma do Par: Este axioma nos diz que para todos os conjuntos A e B, existe um conjunto conjunto que é $\{A, B\}$. Ou seja,

$$\forall A\ \forall B\ \exists C\ \forall x\ (x \in C \leftrightarrow x = A \vee x = B)$$

Vale ressaltar que se $A = B$, teremos $\{A, B\} = \{A, A\} = \{A\}$. A aplicação sucessiva deste axioma nos permite criar uma infinidade de conjuntos finitos. Por exemplo, $A = \emptyset$ e seja $B = \emptyset$, assim teremos $\{\emptyset\}$, sendo $B = \{\emptyset\}$, teremos agora $\{\emptyset, \{\emptyset\}\}$, agora fazendo $A = \{\emptyset\}$, teremos $\{\{\emptyset\}\}$.

Não pretendemos nos aprofundar mais nos axiomas de conjuntos, dado que eles utilizarão conceitos abordados futuramente, nos capítulos de Relações, Funções e Indução.

Mas como se pode ver em Lean - Logic and Proof, podemos reduzir uma grande gama de coisas a conjuntos, assim podemos tratá-las na Teoria dos Conjuntos. Para saber mais sobre Teoria dos Conjuntos, consulte Teoria dos Conjuntos - USP.

5.3 Diagrama de Venn

A maneira mais simples de entender a Teoria de Conjuntos, talvez seja o Diagrama de Venn. Criado por John Venn em 1880, esse sistema de representar graficamente conjuntos auxilia imensamente quem está começando a aprender esse assunto, principalmente para entender sobre a parte inicial de notações e definições. Basicamente, ele consiste em representar num plano, o universo $\mathcal{U}$ como sendo um retângulo e cada conjunto $A, B, ...$ como uma curva fechada simples (geralmente, círculo).

5.3.1 Para 1 ou 2 conjuntos

Começando com a ideia mais simples, a imagem abaixo representa em vermelho o conjunto A dentro do universo $\mathcal{U}$:

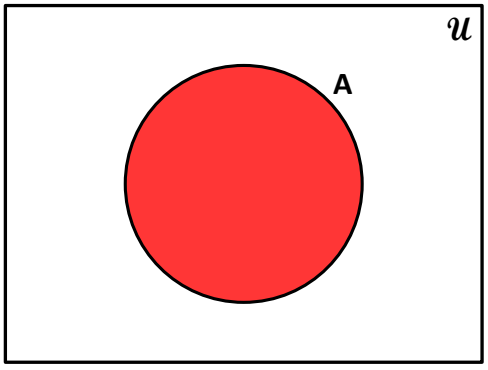


Figura 5.1: Conjunto A dentro de $\mathcal{U}$

Já sobre o conjunto complementar $\overline{A}$, ele simplesmente é a parte que está no retângulo, mas não está no círculo, justamente o que não estava em vermelho na figura anterior.

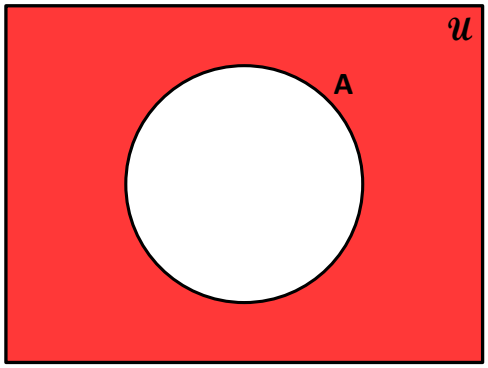


Figura 5.2: Conjunto complementar $\overline{A}$

Para representar que um elemento pertence ao conjunto A, simplesmente colocamos ele dentro do espaço delimitado pelo círculo que representa o conjunto, e para representar que um elemento não pertence ao conjunto A, fazemos o inverso.

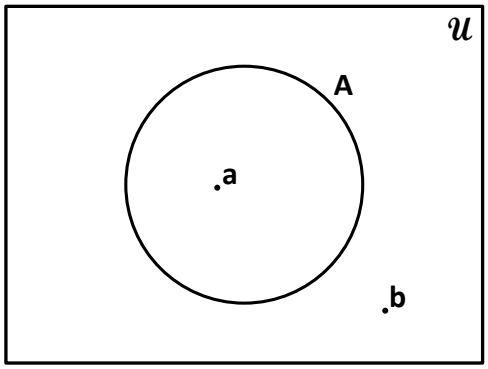


Figura 5.3: $a \in A$ e $b \notin A$

Quando vamos representar mais de um conjunto em um diagrama de Venn, devemos necessariamente ter todas as possíveis relações, mas o que isso significa? Por exemplo, quando temos 2 conjuntos A e B, significa que devemos ter 4 regiões representando respectivamente: elementos que pertencem somente a A, elementos que pertencem somente a B, elementos que pertencem a A e B simultaneamente e elementos que não pertencem a nenhum dos conjuntos. Precisamos disso, para que tudo que provarmos para dois conjuntos A e B possa ser generalizado para dois conjuntos quaisquer, independente do problema e da situação que estamos estudando.

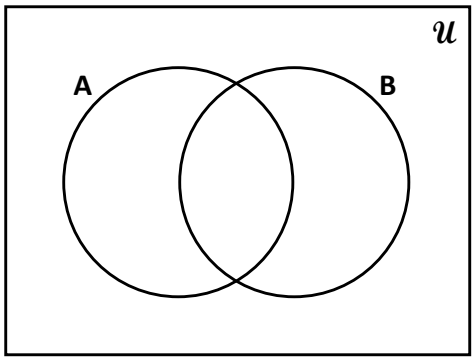


Figura 5.4: Conjuntos A e B

Utilizando esse artifício, podemos representar todas as definições de intersecção, união e diferença de 2 conjuntos, introduzidas na seção anterior. Veja:

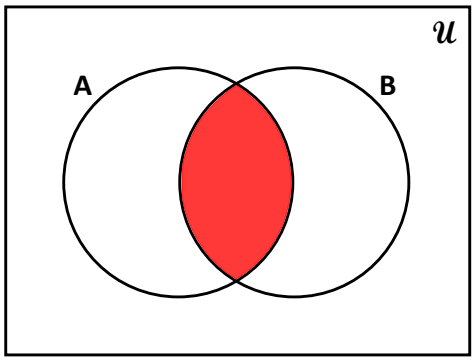


Figura 5.5: Intersecção $A \cap B$

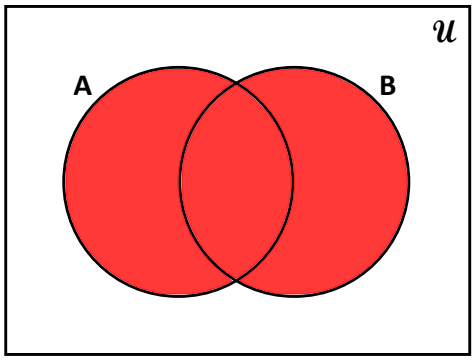


Figura 5.6: União $A \cup B$

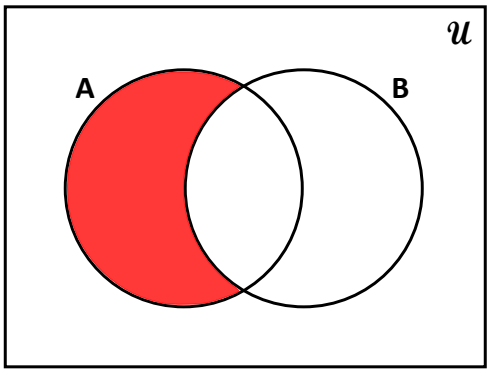


Figura 5.7: Diferença $A \setminus B$

Todavia, isso ainda não permite fazer tudo que desejamos. Se quisermos representar que $A \subseteq B$, a ideia inicial seria colocar o círculo A dentro do círculo B, quebrando o rigor de manter todas as possíveis relações, pois não teremos uma região para representar os elementos que pertecem somente a A. Então, como resolver esse problema? Representamos os conjuntos A e B da mesma forma que anteriormente e também escrevemos o símbolo do conjunto vazio $\emptyset$ na região dos elementos que pertencem somente a A. Assim, só existem elementos no conjunto A que estão na região $A \cap B$, ou seja, se um elemento está em A, como consequência ele está em B, exatamente a definição de $A \subseteq B$.

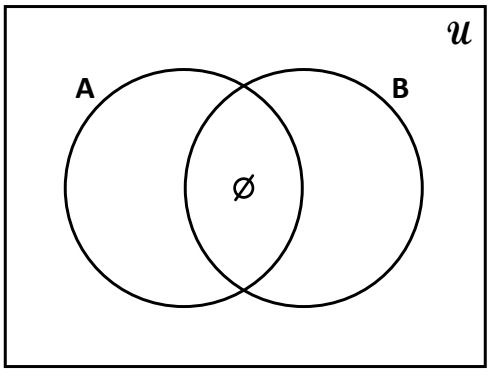


Figura 5.8: Subconjunto $A \subseteq B$

É inegável que para muitos exemplos isso se torna inviável, principalmente quando o único objetivo é fazer uma ilustração matemática do problema, como por exemplo: tomamos o universo $\mathcal{U}$ como a fauna do nosso planeta, nele temos dois conjuntos A de humanos e B de mamíferos. É previamente conhecido que todos os humanos são mamíferos, falando de outra forma, que $A \subseteq B$. Logo, para representar um problema que envolva esses elementos, podemos utilizar o **Diagrama de Euler**, similar ao Diagrama de Venn, com a diferença de que não

é necessário mostrar todas as possíveis relações, mas apenas aquelas específicas do problema retratado. E assim, fazemos exatamente o que tinha sido proposto no parágrafo anterior, colocar o círculo A dentro do círculo B.

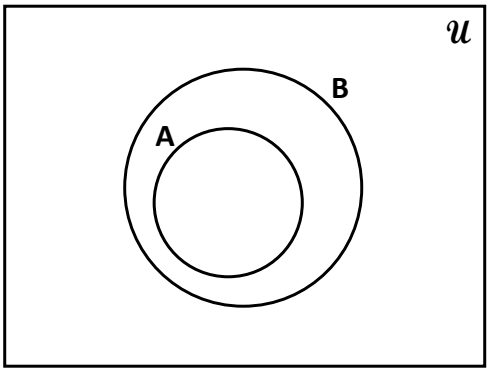


Figura 5.9: Diagrama de Euler $A \subseteq B$

5.3.2 Para 3 conjuntos ou mais

Já foi bastante falado sobre Diagrama de Venn, mas nem chegamos a trabalhar com mais de 2 conjuntos, o que é importante, dado que a Teoria de Conjuntos não se resume a A e B. Mas antes de partirmos para mais conjuntos, vamos pensar numa generalização de quantas regiões diferentes devemos ter para que o diagrama seja um Diagrama de Venn. Dado n conjuntos diferentes, tomamos um elemento qualquer x, e para cada um dos n conjuntos existem duas possibilidades: $x \in$ conjunto e $x \notin$ conjunto. Logo, concluímos que existem $\underbrace{2 \cdot 2 \cdots 2 \cdot 2}_{n} = 2^n$ possibilidades de pertencimento de x nos conjuntos, equivalente à dizer que existem 2^n regiões diferentes. Isso bate perfeitamente com o caso anterior pra $n = 2$, pois vimos que era necessário ter $4 = 2^2$ regiões diferentes.

Agora, com 3 conjuntos A, B e C, o número de regiões diferentes é $2^3 = 8$, mas como iremos representá-las? A primeira ideia que vem a cabeça é adicionar um círculo representando o conjunto C no diagrama da figura 5.4, intersectando as regiões já existentes, resultando na figura 5.10.

Fazendo uma rápida contagem, obtemos 8 regiões diferentes, exatamente como deve ser (lembrete: A região fora dos conjuntos mas dentro do universo $\mathcal{U}$, também é considerada na contagem). E da mesma forma que representamos intersecção, união e diferença de conjuntos anteriormente, também podemos representar com 3 conjuntos, veja nas figuras a seguir:

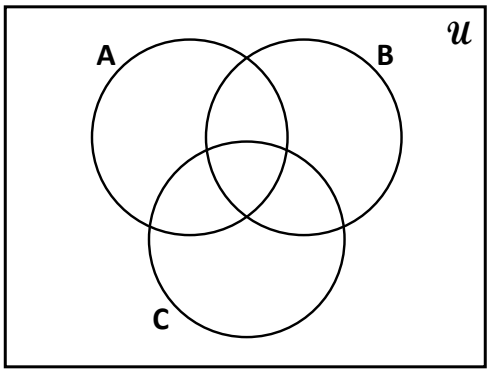


Figura 5.10: Conjuntos A, B e C

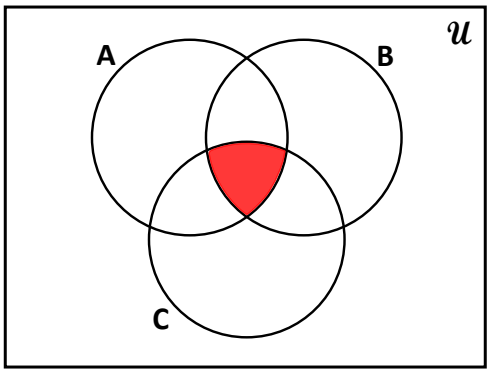


Figura 5.11: Intersecção $A \cap B \cap C$

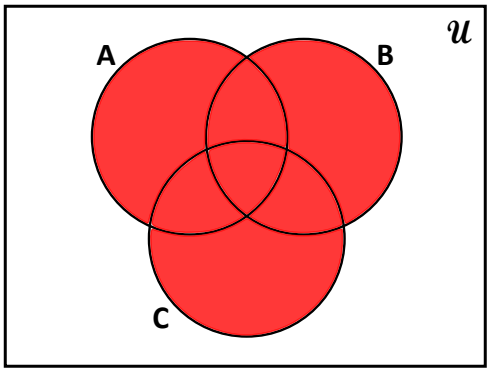


Figura 5.12: União $A \cup B \cup C$

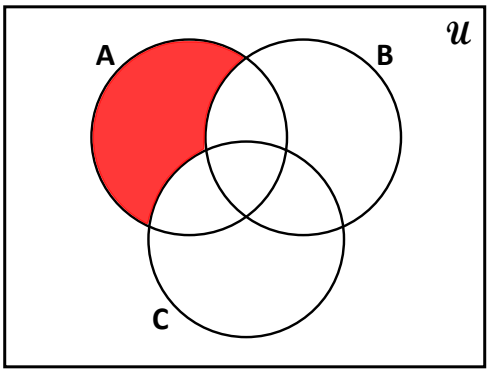


Figura 5.13: Diferença $A \setminus (B \cup C)$

Sendo ganancioso e indo além, podemos querer representar 4 conjuntos, adicionando o conjunto D. E igual o caso anterior, só temos que adicionar mais um círculo representando o conjunto D e, dado que antes obtemos um triângulo de círculos, dessa vez iremos ter uma quadrado de cícurlos, correto?

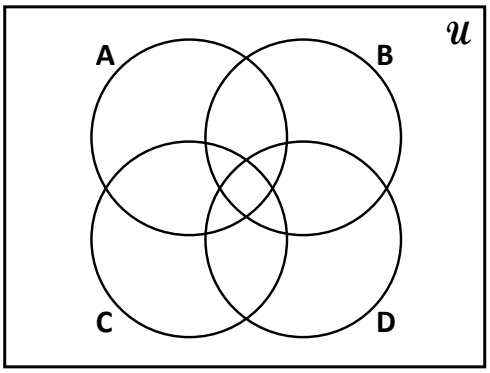


Figura 5.14: 4 Conjuntos

ERRADO! Se você é um leitor observador, já deve ter notado que esse diagrama não possui as $2^4 = 16$ regiões diferentes necessárias, mas somente 14. Não existem regiões que representam os elementos que pertencem ao mesmo tempo aos conjuntos A e D, mas não pertencem ao conjunto B e nem ao C, e o inverso disso, ou seja, os elementos que pertencem ao mesmo tempo aos conjuntos B e C, mas não pertencem ao conjunto A e nem ao D.

É válido ressaltar que é impóssível utilizar 4 círculos pra representar 4 conjuntos em um diagrama de Venn. Contudo, existem diagramas de Venn pra 4 conjuntos, e antes de ler o próximo parágrafo, pegue uma folha e tente encontrar possíveis maneiras dessa representação. Dica: Não se abstenha de utilizar formas bem diferenciadas.

Se você conseguiu, parabéns. Saiba que existem diversas maneiras, como por exemplo: 4 elipses ou 3 círculos e uma forma semelhante à metade de uma rosquinha. Fique a vontade para pesquisar na internet essas e outras maneiras de representação. Ainda assim, é relevante mostrar pelo menos uma delas aqui, e escolhemos a da figura 5.15, onde cada conjunto é representado por um meio círculo junto com um retângulo, resultando em algo muito semelhante a dois corações.

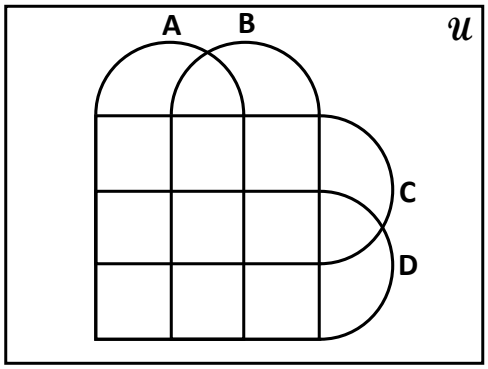


Figura 5.15: Diagrama em formato de 2 corações

Podemos continuar e encontrar representações para 5 ou mais conjuntos, as quais vão ficando cada vez mais complicadas. Porém, como nosso principal objetivo é utilizar visualizações gráficas para facilitar o aprendizado em Teoria de Conjuntos, a partir do momento que isso vem a se tornar complicado, perde totalmente a utilidade. Por esse motivo, vamos parar em 4 conjuntos, mas se a curiosidade for grande, procure aprender mais sobre esse assunto.

5.3.3 Aplicações

A mais importante e útil das aplicações é justamente demonstrar propriedades da Teoria de Conjuntos. Se fosse solicitado para você leitor demonstrar as propriedades a seguir, somente com o que aprendeu até agora, já conseguiria utilizar o Diagrama de Venn para demonstrá-las. Apesar de parecer complicado, é incrivelmente simples, veja:

Exemplo 1: Demonstre que $A \setminus B = A \cap \overline{B}$.

Resposta: Primeiramente, vamos fazer o Diagrama de Venn desses dois conjuntos e enumerar as 4 regiões, conforme a figura 5.16.

Assim, temos que $A = \{1, 2\}$, $B = \{2, 3\}$, $\overline{A} = \{3, 4\}$ e $\overline{B} = \{1, 4\}$. Utilizando esse valores, podemos calcular $A \setminus B$ e $A \cap \overline{B}$, conforme abaixo:

$$A \setminus B = \{1, 2\} \setminus \{2, 3\} = \{1\}$$
$$A \cap \overline{B} = \{1, 2\} \cap \{1, 4\} = \{1\}$$

Portanto, concluímos que $A \setminus B = A \cap \overline{B}$.

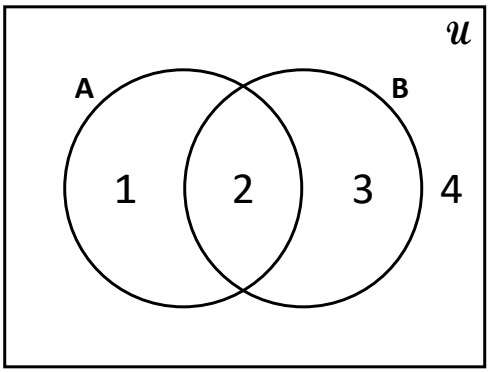


Figura 5.16: Regiões 1, 2, 3 e 4

Exemplo 2: Demonstre que $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.

Resposta: Utilizando o mesmo Diagrama de Venn e a mesma numeração de regiões do exemplo anterior, temos:

$$\begin{aligned} A \setminus B &= \{1,2\} \setminus \{2,3\} = \{1\} \\ B \setminus A &= \{2,3\} \setminus \{1,2\} = \{3\} \\ A \cup B &= \{1,2\} \cup \{2,3\} = \{1,2,3\} \\ A \cap B &= \{1,2\} \cap \{2,3\} = \{2\} \\ (A \setminus B) \cup (B \setminus A) &= \{1\} \cup \{3\} = \{1,3\} \\ (A \cup B) \setminus (A \cap B) &= \{1,2,3\} \setminus \{2\} = \{1,3\} \end{aligned}$$

Portanto, concluímos que $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$.
É possível responder essas questões somente utilizando o diagrama e pintando as regiões, mas a enumeração torna o processo mais fácil.

Outra aplicação interessante, muito recorrente em testes de lógica e vestibulares, é tentar interpretar problemas da vida real com conjuntos, conforme no exemplo a seguir:

Exemplo 3: Em uma sala de aula do Ensino Fundamental com 40 alunos, a professora Ana perguntou quem gostava de matemática e quem gostava de português. Ela obteve os seguintes dados:

- 21 alunos gostam de matemática
- 17 alunos gostam de português
- 9 alunos não gostam de nenhuma das matérias

Quantos alunos gostam de ambas as matérias?

Resposta: Vamos utilizar um Diagrama de Venn para dois conjuntos, onde um conjunto representa os alunos que gostam de matemática e o outro representa os que gostam de português. Dado que 9 alunos não gostam de nenhuma das matérias e temos 40 alunos na sala, então $40 - 9 = 31$ alunos pertencem a pelo menos um dos conjuntos, ou seja, gostam de matemática ou português.

Como 21 gostam de matemática, logo $31 - 21 = 10$ gostam somente de português (região da direita) e, analogamente, como 17 gostam de português, logo $31 - 17 = 14$ gostam somente de matemática (região da esquerda). Utilizando esses dados, vemos que ainda faltam $31 - 10 - 14 = 7$ alunos, justamente os que gostam de ambas as matérias (região do meio). Veja abaixo o diagrama final:

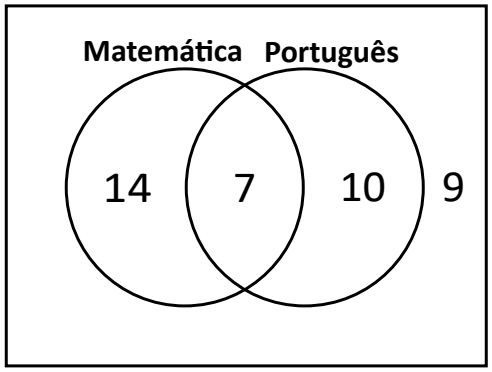


Figura 5.17: Diagrama para Matemática e Português

Apesar do estudo de conjuntos com Diagrama de Venn ser fácil, simples e útil, como já comentado anteriormente, é difícil para trabalhar com muitos conjuntos, além das provas utilizando esse recurso não terem um grau de formalidade muitas vezes requerido pelos matemáticos. Devido a esse fato, iremos estudar nas próximas seções outras maneiras de lidar com Teoria dos Conjuntos, como provas matemáticas formais, dedução natural, cálculo em conjuntos e é claro, utilizando o Lean.

5.4 Prova de Teoremas

Nesta seção iremos tratar de formalizar definições e Teoremas. Ao contrário da seção 5.2, onde o foco era transmitir a notação e uma ideia geral dos conceitos.

Aqui serão apresentadas algumas provas em Lean para o conteúdo apresentado. Não se preocupe, caso não entenda sobre o que o código quer dizer ignore ele por enquanto e após ler as próximas duas seções (que falam de conjuntos em Lean), volte e tente compreender o que foi feito. Já que o foco principal nesta parte são as provas matemáticas tradicionais e as definições mais precisas.

5.4.1 Prova Matemática Formal

Ainda vai ter algo escrito aqui

5.4.2 Dedução Natural

Ainda vai ter algo escrito aqui

5.5 Conjuntos em Lean

Embora na teoria axiomática dos conjuntos se considere conjuntos de objetos distintos, em matemática é mais comum considerar subconjuntos de algum domínio fixo ($\mathcal{U}$). É assim que os conjuntos são tratados em Lean. Para qualquer dado do tipo `U`, Lean nos retorna um novo dado tipo `set U`, que consiste no conjunto dos elementos de `U`. Assim, por exemplo, podemos raciocinar sobre conjuntos de números naturais, conjuntos de números inteiro ou conjuntos de pares de números naturais.

5.5.1 Notações

O lean possui uma biblioteca padrão para lidar com conjuntos chamada `set` e, sempre que formos utilizar um comando que pertence a ela, é necessário escrever `set.comando`. Entretanto, pra facilitar nossa vida, podemos escrever `open set` no início do código, o que permite escrevermos somente `comando`, e o lean já entende que ele pertence a biblioteca set.

Além disso, para trabalharmos com conjuntos, também é essencial definirmos um tipo `U` e sabermos utilizar conjuntos e elementos desse tipo. Para isso, utilizamos o código abaixo:

```
1 open set
2
3 variable {U : Type}
4 variables A B C : set U
5 variable x : U
```

Temos aqui uma pequena lista de como se representa os principais caractéres da parte de conjuntos em Lean:

- $\in$ → \in
- $\notin$ → \notin
- $\subset$ → \subset
- $\subseteq$ → \sub
- $\emptyset$ → \empty
- $\cup$ → \un ou \cup ou \union

- $\cap \rightarrow$ \i ou \cap ou \intersection

Obs[1].: O conjunto universal é denotado `univ`.

Obs[2].: O complementar de um conjunto é denotado com um símbolo de subtração antes dele: $-A$

Podemos ver alguns exemplos abaixo:

```
1  open set
2  variable {U : Type}
3  variables A B C : set U
4  variable x : U
5
6  #check x ∈ A
7  #check A ∪ B
8  #check B \ C
9  #check C ∩ A
10 #check -C
11 #check ∅ ⊆ A
12 #check B ⊆ univ
```

Noções básicas da teoria dos conjuntos são definidas na biblioteca principal do Lean, mas teoremas e notações adicionais que iremos utilizar nesse capítulo, estão disponíveis em uma biblioteca auxiliar que é carregada com o comando `import data.set`, o qual deve aparecer no início do arquivo.

```
1  import data.set
2  open set
3  variable {U : Type}
4  variables A B C : set U
5  variable x : U
```

A partir desse momento, para evitar repetição, iremos omitir as 4 primeiras linhas do código, no entanto **você deve lembrar que elas existem para seu código funcionar.** Já sobre a linha 5, não é necessário escrever nos próximos exemplos, pois sempre iremos nos referenciar a um elemento do tipo `U`, dentro de `example, lemma` ou `theorem`.

5.5.2 Primeiros Passos

Relembrando a definição de subconjunto, podemos utilizar o template abaixo para mostrar que o conjunto A é um subconjunto de B:

```
1  example : A ⊆ B :=
2  assume x : U,
3  assume h : x ∈ A,
4  show x ∈ B, from sorry
```

Obs: Na linha 2 poderíamos ter escrito somente `assume x`, pois já inferiria que `x` é do tipo `U`.

Já para mostrar que A e B são iguais, temos dois comandos diferentes: `eq_of_subset_of_subset` e `ext`.

eq_of_subset_of_subset: Ele funciona interpretando a seguinte expressão $(A \subseteq B \wedge B \subseteq A) \Rightarrow A = B$, ou seja, obtém a equivalência dos conjuntos a partir do fato de que o primeiro é subconjunto do segundo, e vice-vera. Veja o código:

```
example : A = B :=
eq_of_subset_of_subset
(assume x,
  assume h : x ∈ A,
  show x ∈ B, from sorry)
(assume x,
  assume h : x ∈ B,
  show x ∈ A, from sorry)
```

ext: É uma sigla para "extensionality", ou seja, extensionalidade. Matemáticamente, isso representa a expressão $\forall x\ (x \in A \leftrightarrow x \in B) \Rightarrow A = B$. Veja o código:

```
example : A = B :=
ext (assume x, iff.intro
(assume h : x ∈ A,
  show x ∈ B, from sorry)
(assume h : x ∈ B,
  show x ∈ A, from sorry))
```

Além disso, o Lean possui interpretação ambígua para regras de união, interseção e outras operações em conjuntos que são consideradas "definições". Isso significa que as expressões $x \in A \cap B$ e $x \in A \wedge x \in B$ possuem a mesma interpretação no Lean. Isso também é válido para outras construções em conjuntos, como: $x \in A \setminus B$ e $x \in A \wedge \neg\ (x \in B)$. O termo $\neg\ (x \in B)$ é somente outra forma de escrever $x \notin B$. Abaixo são apresentadas algumas aplicações dessas interpretações:

```
example : ∀ x, x ∈ A → x ∈ B → x ∈ A ∩ B :=
assume x,
assume h₁ : x ∈ A,
assume h₂ : x ∈ B,
show x ∈ A ∩ B, from and.intro h₁ h₂

example : A ⊆ A ∪ B :=
assume x,
assume h : x ∈ A,
show x ∈ A ∪ B, from or.inl h

example : ∅ ⊆ A  :=
assume x,
```

```
14 assume h : x ∈ (∅ : set U),
15 show x ∈ A, from false.elim h
```

Observe no último exemplo a necessidade de usar a notação `(∅ : set U)`, dizendo ao nosso provador que o ∅ é um conjunto de `U`. Isso acontece pois ele não consegue inferir que tipo é o conjunto vazio, dado que por definição, esse conjunto existe em qualquer universo, ou seja, pode ser de qualquer tipo.

Opcionalmente, podemos usar alguns teoremas da biblioteca `data.set`, projetados especificamente para uso em conjuntos:

```
1  example : ∀ x, x ∈ A → x ∈ B → x ∈ A ∩ B :=
2  assume x,
3  assume h : x ∈ A,
4  assume g : x ∈ B,
5  show x ∈ A ∩ B, from mem_inter h g
6
7  example : A ⊆ A ∪ B :=
8  assume x,
9  assume h : x ∈ A,
10 show x ∈ A ∪ B, from mem_union_left B h
11
12 example : ∅ ⊆ A  :=
13 assume x,
14 assume h : x ∈ ∅,
15 show x ∈ A, from absurd h (not_mem_empty x)
```

Lembre-se que o comando `absurd` pode ser usado para provar qualquer fato a partir de duas hipóteses contrárias: $h_1 : P$ e $h_2 : \neg P$.

Aqui, o teorema `not_mem_empty x` significa $x \notin \emptyset$. Para ver a declaração de teoremas disponíveis, utilize o comando `#check`:

```
1 #check @mem_inter
2 #check @mem_of_mem_inter_left
3 #check @mem_of_mem_inter_right
4 #check @mem_union_left
5 #check @mem_union_right
6 #check @mem_or_mem_of_mem_union
7 #check @not_mem_empty
```

Neste caso, o símbolo `@` (arroba) impede que ele tente preencher argumentos implícitos automaticamente, forçando-o a exibir a declaração completa do teorema.

Já que podemos relacionar conjuntos com suas definições lógica, isso auxilia a comprovação de certas relações entre conjuntos:

```
1 example : A \ B ⊆ A :=
2 assume x,
3 assume h : x ∈ A \ B,
4 show x ∈ A, from and.left h
```

```
5
6  example : A \ B ⊆ -B :=
7  assume x,
8  assume h : x ∈ A \ B,
9  have g : x ∉ B, from and.right h,
10 show x ∈ -B, from g
```

Novamente, é possível usar versões dos teoremas projetados especificamente para conjuntos:

```
1  example : A \ B ⊆ A :=
2  assume x,
3  assume h : x ∈ A \ B,
4  show x ∈ A, from mem_of_mem_diff h
5
6  example : A \ B ⊆ -B :=
7  assume x,
8  assume h : x ∈ A \ B,
9  have g : x ∉ B, from not_mem_of_mem_diff h,
10 show x ∈ -B, from g
```

Como o Lean tem que desenvolver definições, ele pode acabar se confundindo às vezes. Por exemplo, na prova a seguir, se você subtituir a última linha por `sorry`, ele terá problemas tentando entender que você quer que ele desenvolva o símbolo de subconjunto:

```
1  example : A ∩ B ⊆ B ∩ A :=
2  assume x,
3  assume h : x ∈ A ∩ B,
4  have h₁ : x ∈ A, from and.left h,
5  have h₂ : x ∈ B, from and.right h,
6  and.intro h₂ h₁
```

Uma solução alternativa é usar o comando `show`. Na maioria das vezes, fornecer informações adicionais para o Lean pode ser útil. Outra solução é nomear um teorema, o que leva o nosso provador a usar um método um pouco diferente de processar a prova, corrigindo o problema como um efeito colateral.(Substituindo a parte posterior ao `from` na linha 6 ou a linha 13 inteira por "`sorry`"o provedor não localiza problemas).

```
1  example : A ∩ B ⊆ B ∩ A :=
2  assume x,
3  assume h : x ∈ A ∩ B,
4  have h₁ : x ∈ A, from and.left h,
5  have h₂ : x ∈ B, from and.right h,
6  show x ∈ B ∩ A, from and.intro h₂ h₁
7
8  theorem my_example : A ∩ B ⊆ B ∩ A :=
9  assume x,
```

```
10 assume h : x ∈ A ∩ B,
11 have h₁ : x ∈ A, from and.left h,
12 have h₂ : x ∈ B, from and.right h,
13 show x ∈ B ∩ A, from and.intro h₂ h₁
```

5.6 Propriedades

A seguir teremos algumas propriedades e a prova do porque estão corretas, utilizando várias maneiras vistas anteriormente neste capítulo.

1. **Básicas**
   - $A \cap A = A$
   - $A \cup A = A$
   - $A \cap \mathcal{U} = A$
   - $A \cup \mathcal{U} = \mathcal{U}$
   - $A \cap \emptyset = \emptyset$
   - $A \cup \emptyset = A$
2. **Comutatividade**
   - $A \cap B = B \cap A$ (a prova desta identidade escontra-se na seção de Provas de Teoremas)
   - $A \cup B = B \cup A$
3. **Associatividade**
   - $(A \cap B) \cap C = A \cap (B \cap C)$
   - $(A \cup B) \cup C = A \cup (B \cup C)$
4. **Distributividade**
   - $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
   - $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

   Prova em Lean:

```
1 example : A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C) :=
2 eq_of_subset_of_subset
3     (assume x,
4     assume h : x ∈ A ∪ (B ∩ C),
5     or.elim h
6         (assume h₁ : x ∈ A,
7         have h₂ : x ∈ A ∪ B, from or.inl h₁,
8         have h₃ : x ∈ A ∪ C, from or.inl h₁,
```

```
 9          show x ∈ (A ∪ B) ∩ (A ∪ C), from and.intro h₂ h
       ₃)
10          (assume h₁ : x ∈ B ∩ C,
11          have h₂ : x ∈ B, from and.left h₁,
12          have h₃ : x ∈ C, from and.right h₁,
13          have h₄ : x ∈ A ∪ B, from or.inr h₂,
14          have h₅ : x ∈ A ∪ C, from or.inr h₃,
15          show x ∈ (A ∪ B) ∩ (A ∪ C), from and.intro h₄ h
       ₅))
16      (assume x,
17      assume h : x ∈ (A ∪ B) ∩ (A ∪ C),
18      have h₁ : x ∈ A ∪ B, from and.left h,
19      have h₂ : x ∈ A ∪ C, from and.right h,
20      or.elim h₁
21          (assume h₃ : x ∈ A,
22          show x ∈ A ∪ (B ∩ C), from or.inl h₃)
23          (assume h₃ : x ∈ B,
24          or.elim h₂
25              (assume h₄ : x ∈ A,
26              show x ∈ A ∪ (B ∩ C), from or.inl h₄)
27              (assume h₄ : x ∈ C,
28              have h₅ : x ∈ B ∩ C, from and.intro h₃ h₄,
29              show x ∈ A ∪ (B ∩ C), from or.inr h₅)))
```

5. **Complementar/Lei de Demorgan**

- $\mathcal{U}^c = \emptyset$
- $\emptyset^c = \mathcal{U}$
- $A \cap A^c = \emptyset$

  (i) Provaremos que $\forall x(x \in A \cap \overline{A}) \to \forall x(x \in \emptyset)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x(x \in A \cap \overline{A})}\,1}{t \in A \cap \overline{A}}}{t \in A \wedge t \in \overline{A}}}{t \in A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x(x \in A \cap \overline{A})}\,1}{t \in A \cap \overline{A}}}{t \in A \wedge t \in \overline{A}}}{t \in \overline{A}}}{t \in \mathbb{U} \wedge t \notin A}}{t \notin A}}{\dfrac{\dfrac{\bot}{t \in \emptyset}}{\forall x(x \in \emptyset)}}}{\forall x(x \in A \cap \overline{A}) \to \forall x(x \in \emptyset)}\,1$$

  (ii) Provaremos que $\forall x(x \in \emptyset) \to \forall x(x \in A \cap \overline{A})$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x(x \in \emptyset)}\,1}{\bot}}{t \in A \cap \overline{A}}}{\forall x(x \in A \cap \overline{A})}}{\forall x(x \in \emptyset) \to \forall x(x \in A \cap \overline{A})}\,1$$

Portanto, de (i) e (ii), concluímos que $\forall x(x \in A \cap \overline{A}) \iff \forall x(x \in \emptyset)$, ou seja, $A \cap \overline{A}$.

- $A \cup A^c = \mathcal{U}$
  Prova: Seja x um elemento de $A \cup \overline{A}$, assim temos que

$$x \in (A \cup \overline{A})$$

$$\iff x \in A \vee x \in \overline{A}$$

- $(A^c)^c = A$
- $(A \cap B)^c = A^c \cup B^c$
- $(A \cup B)^c = A^c \cap B^c$

6. **Lei da Absorção**

   - $A \cap (A \cup B) = A$

```
1 lemma inter_subseq (H : Type)(P Q : set H) : P ∩ (P ∪ Q)
      = P :=
2 eq_of_subset_of_subset
3   (assume x,
4     assume h : x ∈ P ∩ (P ∪ Q),
5     show x ∈ P, from h.left)
6   (assume x,
7     assume h : x ∈ P,
8     have h₁ : x ∈ P ∪ Q, from or.inl h,
9     show x ∈ P ∩ (P ∪ Q), from and.intro h h₁)
```

   - $A \cup (A \cap B) = A$

7. **Extras**

   - $A \setminus B = A \cap B^c$

5.7 Cálculo em Conjuntos

Podemos provar que dois conjuntos são iguais argumentando sobre seus elementos, ou podemos ser mais eficientes e utilizar o cálculo juntamente com as propriedades vistas na seção 5.6. Veja alguns exemplos:

Exemplo 1: Demonstre que $(A \setminus B)^c = A^c \cup B$

Prova: Utilizando cálculo:

$$\begin{aligned}(A \setminus B)^c &= (A \cap B^c)^c) \\ &= A^c \cup (B^c)^c \\ &= A^c \cup B\end{aligned}$$

Relembrando um dos exemplos da seção 5.3:
Exemplo 2: Demonstre que $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$
Prova: Utilizando cálculo:

$$\begin{aligned}(A \setminus B) \cup (B \setminus A) &= (A \cap B^c) \cup (B \cap A^c) \\ &= ((A \cap B^c) \cup B) \cap ((A \cap B^c) \cup A^c) \\ &= ((A \cup B) \cap (B^c \cup B)) \cap ((A \cup A^c) \cap (B^c \cup A^c)) \\ &= ((A \cup B) \cap \mathcal{U}) \cap (\mathcal{U} \cap (B^c \cup A^c)) \\ &= (A \cup B) \cap (B^c \cup A^c) \\ &= (A \cup B) \cap (A^c \cup B^c) \\ &= (A \cup B) \cap (A \cap B)^c \\ &= (A \cup B) \setminus (A \cap B)\end{aligned}$$

COLOCAR ALGO AQUI

Nos próximos exemplos, omitimos as quatro primeiras linhas, no entanto, lembre-se que elas existem, e devem ser incluídas no uso do provador.

```
1     import data.set
2     open set
3     variable U : Type
4     variables A B C : set U
5
6     example (h : A = B) : B = A :=
7       calc
8         B = A : eq.symm h
```

Nesse caso, utilizamos `eq.symm h`, que já nos deu a igualdade `B = A` direto. Mas tem outro jeito. Podemos querer somente baseado em `h`, reescrever `B`, e para isso utilizamos a tática `rewrite`.

```
1  example (h : A = B) : B = A :=
2  calc
3    B = A : by rewrite h
```

Sempre utilizaremos essa tática quando quisermos a partir de uma igualdade, reescrever alguma parte da expressão, e pra agilizar o processo, ela pode ser abreviada para somente `rw`. Podemos também escrever isso como lema/teorema e ainda colocar o universo e os conjuntos na definição dele (nesse caso, excluimos as linhas 3 e 4 que definem variáveis, omitidas anteriormente), veja:

```
lemma a_eq_b {U : Type} (A B : set U) (h : A = B) : B = A :=
calc
  B = A : by rw h
```

É evidente que o nosso objetivo não é utilizar `calc` somente para coisas tão simples. Queremos usar diversas propriedades sobre conjuntos que já estão a nossa disposição, para provar cada vez mais novas propriedades, como o exemplo a seguir:

Exemplo 1: Demonstre que $(A \setminus B)^c = A^c \cup B$.

```
example : -(A \ B) = -A ∪ B :=
calc
  -(A \ B) = -(A ∩ -B)  : by rw diff_eq
       ... = -A ∪ -(-B) : by rw compl_inter
       ... = -A ∪ B     : by rw compl_compl
```

Para aquelas pessoas que gostam de ser minimalistas e tentam usar o mínimo de linhas possível, existe como reduzir a prova em `calc` pra somente uma linha, veja:

```
example : -(A \ B) = -A ∪ B :=
calc
  -(A \ B) = -A ∪ B : by rw [diff_eq, compl_inter, compl_compl]
--OU
example : -(A \ B) = -A ∪ B :=
by rw [diff_eq, compl_inter, compl_compl]
```

Todavia, não é factível esperar que todo mundo adivinhe o nome dos lemas que representam cada propriedade para então poder utilizá-los em provas de cálculo. Por esse fato, apresentamos a seguir os nomes dos lemas para todas as propriedades da seção anterior e entre parênteses, a sua versão caso os conjuntos estejam na ordem inversa.

- $A \cap A = A$: `inter_self`
- $A \cup A = A$: `union_self`
- $A \cap \mathcal{U} = A$: `inter_univ` (`univ_inter`)
- $A \cup \mathcal{U} = \mathcal{U}$: não existe
- $A \cap \emptyset = \emptyset$: `inter_empty` (`empty_inter`)
- $A \cup \emptyset = A$: `union_empty` (`empty_union`)
- $\mathcal{U}^c = \emptyset$: `compl_univ`
- $\emptyset^c = \mathcal{U}$: `compl_empty`
- $A \cap A^c = \emptyset$: `inter_compl_self` (`compl_inter_self`)
- $A \cup A^c = \mathcal{U}$: `union_compl_self` (`compl_union_self`)

- $(A^c)^c = A$: `compl_compl`
- $A \cap B = B \cap A$: `inter_comm`
- $A \cup B = B \cup A$: `union_comm`
- $(A \cap B) \cap C = A \cap (B \cap C)$: `inter_assoc`
- $(A \cup B) \cup C = A \cup (B \cup C)$: `union_assoc`
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$: `inter_distrib_left` (`inter_distrib_right`)
- $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$: `union_distrib_left` (`union_distrib_right`)
- $(A \cap B)^c = A^c \cup B^c$: `compl_inter`
- $(A \cup B)^c = A^c \cap B^c$: `compl_union`
- $A \setminus B = A \cap B^c$: `diff_eq`

Com toda essa bagagem já somos capazes de fazer provas bem mais complexas e que envolvem diversos passos, como a seguir:

Exemplo 2: Demonstre que $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$

```
1  example : (A \ B) ∪ (B \ A) = (A ∪ B) \ (A ∩ B) :=
2  calc
3    (A \ B) ∪ (B \ A) = (A ∩ -B) ∪ (B \ A) : by rw diff_eq
4    ... = (A ∩ -B) ∪ (B ∩ -A) : by rw diff_eq
5    ... = ((A ∩ -B) ∪ B) ∩ ((A ∩ -B) ∪ -A) : by rw
       union_distrib_left
6    ... = ((A ∪ B) ∩ (-B ∪ B)) ∩ ((A ∩ -B) ∪ -A) : by rw
       union_distrib_right
7    ... = ((A ∪ B) ∩ univ) ∩ ((A ∩ -B) ∪ -A) : by rw
       compl_union_self
8    ... = (A ∪ B) ∩ ((A ∩ -B) ∪ -A) : by rw inter_univ
9    ... = (A ∪ B) ∩ ((A ∪ -A) ∩ (-B ∪ -A)) : by rw
       union_distrib_right
10   ... = (A ∪ B) ∩ (univ ∩ (-B ∪ -A)) : by rw union_compl_self
11   ... = (A ∪ B) ∩ (-B ∪ -A) : by rw univ_inter
12   ... = (A ∪ B) ∩ -(B ∩ A) : by rw compl_inter
13   ... = (A ∪ B) ∩ -(A ∩ B) : by rw inter_comm B A
14   ... = (A ∪ B) \ (A ∩ B) : by rw diff_eq
```

O leitor bem atento já notou uma pequena diferença na linha 13 do código, não escrevemos `by rw inter_comm`, mas sim `by rw inter_comm B A`. Isso é necessário, porque na expressão dessa linha existem 2 símbolos de intersecção em que poderia ser aplicado o lema, criando uma ambiguidade pro provador decidir em qual deles vai aplicá-lo. Quando isso acontece, o provador sempre escolhe a primeira ocorrência, ou seja, quando aplicamos o lema `inter_comm` na expressão `(A ∪ B) ∩ -(B ∩ A)`, obtemos `-(B ∩ A) ∩ (A ∪ B)`. Entretanto, o objetivo

é `(A ∪ B) ∩ -(A ∩ B)`, e para especificar isso, é só passar depois do lema os parâmetros que queremos que ele seja aplicado, no nosso caso, `B A`.

Outro caso de ambiguidade que também vemos nesse exemplo, está na linha 4. A expressão anterior é `(A ∩ -B) ∪ (B \ A)`, ou seja, temos dois termos possíveis para aplicar o lema `diff_eq`, podemos reescrever `(A ∩ -B)` como `(A \ B)` ou podemos reescrever `(B \ A)` como `(B ∩ -A)`. Nesse caso, o Lean prioriza a ida do lemma, que significa transformar o lado esquerdo da igualdade no lado direito (veja a lista de propriedades novamente pra entender melhor), e demos sorte de ser exatamente o que queríamos. Para fazer o inverso, só precisamos especificar os parâmetros pro lema: `by rw diff_eq A B`.

Em algumas vezes, é necessário colocar cada parâmetro entre parênteses. Por exemplo, se um parâmetro for parecido com `-(A ∩ B) ∩ C`, o que vamos escrever depois do lemma é `(-(A ∩ B) ∩ C)` e os outros parâmetros.

Exemplo 3: Demonstre que $(A \cup B \cup C \cup D)^c = (A \cup B)^c \cap (C \cup D)^c$.

```
1 example : -(A ∪ B ∪ C ∪ D) = -(A ∪ B) ∩ -(C ∪ D) :=
2 calc
3   -(A ∪ B ∪ C ∪ D) = -(A ∪ B ∪ C) ∩ -D : by rw compl_union
4   ... = -(A ∪ B) ∩ -C ∩ -D : by rw compl_union
5   ... = -(A ∪ B) ∩ (-C ∩ -D) : by rw inter_assoc (-(A ∪ B)) (-C)
      (-D)
6   ... = -(A ∪ B) ∩ -(C ∪ D) : by rw compl_union C D
```

Esse é um ótimo exemplo de especificação de parâmetros no lema, entretanto, a 5 linha está confundindo bastante. Após as duas primeiras transformações, o Lean interpreta que `-(A ∪ B) ∩ -C ∩ -D = (-(A ∪ B) ∩ -C) ∩ -D` implicitamente. Logo, precisamos utilizar a associatividade para rearranjar a expressão de uma maneira que podemos continuar trabalhando com ela.

Antes de finalizarmos a seção, saiba que os lemas que usamos dentro do `calc`, não precisam ser já criados dentro do Lean. Você pode criar um lema, prová-lo e depois utilizá-lo para provar outros lemas, o que se torna muito divertido!

5.8 Famílias Indexadas

5.8.1 Definição

Ainda vai ter algo escrito aqui

5.8.2 Em Lean

Ainda vai ter algo escrito aqui

5.9 Conjunto das Partes

5.9.1 Definição

Seja um conjunto A, o conjunto das partes de A, representado por $\wp(A)$, é o conjunto formado por todos os subconjuntos de A.

Os elementos de um conjunto das partes são, na verdade, outros conjuntos, e mais precisamente, são todos os subconjuntos que podem ser formados a partir dos elementos do conjunto de referência.

Exemplo 1: Seja $A = \{a, b, c\}$ o conjunto referência. O conjunto das partes de A é $\wp(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$:

5.9.2 Cardinalidade

Se um conjunto A possui n elementos, então o número de subconjuntos de A é igual 2^A.

De fato, no **Exemplo 1**, o conjunto das partes $\wp(A)$ possui 8 elementos e A possui 3 elementos, como $2^3 = 8$, temos uma prova para a cardinalidade de conjuntos das parte.

Agora, iremos provar no caso de A ter k elementos: $(k \in \mathbb{N})$

A demonstração para esse caso é bastante simples, vamos utilizar o **Princípio Fundamental da Contagem** para contar quantos subconjuntos o conjunto A possui.

Vamos criar um subconjunto qualquer B. Para cada um dos k elementos de A, existem somente duas possibilidades:

- Ou o elemento está no subconjunto B;
- Ou o elemento não está no subconjunto B.

Assim, pelo **PFC**, nós podemos montar o conjunto B de

$$\underbrace{2 \cdot 2 \cdot 2 \cdot (\cdots) \cdot 2}_{k} = 2^k \text{ maneiras.}$$

E, portanto, há todos os 2^k subconjuntos de A em $\wp(A)$.

5.9.3 Conjuntos das Partes em Lean

Em lean, trataremos "Conjuntos das Partes"como `powerset`, e pode ser definido da seguinte forma:

```
1 variable {U : Type}
2
3 def powerset (A : set U) : set (set U) := {B : set U | B ⊆ A}
4
5 example (A B : set U) (h : B ∈ powerset A) : B ⊆ A := h
```

Como está exposto na linha 5 do template, $B \in \wp(A)$ é, por definição, o mesmo que $B \subseteq A$.

De fato, a função `powerset` é definida desta forma em Lean, e fica disponível com os comandos: `import data.set` e `open set`. Abaixo temos o **Exemplo 2** de como utilizar esta função:

```
1  import data.set
2  open set
3
4  variable  {U : Type}
5  variables (A B : set U)
6  --Exemplo 2
7  example : A ∈ powerset (A ∪ B) :=
8  assume x,
9  assume h : x ∈ A,
10 show x ∈ A ∪ B, from or.inl h
11
12 #check powerset A
```

5.10 Exercícios

1. Realize as provas das seguintes propriedades: (Fornecendo uma prova tradicional e uma em Lean.)

   (a) Comutatividade em $\cap$ e $\cup$

```
1  import data.set
2  open set
3
4  variable U : Type
5  variables A B : set U
6
7  example : A ∩ B = B ∩ A :=
8  eq_of_subset_of_subset
9  (assume x,
10     assume h : x ∈ A ∩ B,
11     have h₁ : x ∈ A, from h.left,
12     have h₂ : x ∈ B, from h.right,
13     show x ∈ B ∩ A, from and.intro h₂ h₁)
14 (assume x,
15     assume h : x ∈ B ∩ A,
16     have h₁ : x ∈ B, from h.left,
17     have h₂ : x ∈ A, from h.right,
18     show x ∈ A ∩ B, from and.intro h₂ h₁)
19
20 example : A ∪ B = B ∪ A:=
```

```
eq_of_subset_of_subset
(assume x,
    assume h : x ∈ A ∪ B,
    or.elim h
    (assume h₁ : x ∈ A,
    show x ∈ B ∪ A, from or.inr h₁)
    (assume h₁ : x ∈ B,
    show x ∈ B ∪ A, from or.inl h₁))
(assume x,
    assume h : x ∈ B ∪ A,
    or.elim h
    (assume h₁ : x ∈ B,
    show x ∈ A ∪ B, from or.inr h₁)
    (assume h₁ : x ∈ A,
    show x ∈ A ∪ B, from or.inl h₁))
```

(b) Associatividade em ∩ e ∪

```
import data.set
open set

variable U : Type
variables A B C : set U

example : (A ∩ B) ∩ C = A ∩ (B ∩ C) :=
eq_of_subset_of_subset
(assume x,
    assume h : x ∈ (A ∩ B) ∩ C,
    have h₁ : x ∈ A ∩ B, from h.left,
    have h₂ : x ∈ B ∩ C, from and.intro h₁.right h.right
    ,
    show x ∈ A ∩ (B ∩ C), from and.intro h₁.left h₂)
(assume x,
    assume h : x ∈ A ∩ (B ∩ C),
    have h₁ : x ∈ B ∩ C, from h.right,
    have h₂ : x ∈ A ∩ B, from and.intro h.left h₁.left,
    show x ∈ (A ∩ B) ∩ C, from and.intro h₂ h₁.right)

example : (A ∪ B) ∪ C = A ∪ (B ∪ C) :=
eq_of_subset_of_subset
(assume x,
    assume h : x ∈ (A ∪ B) ∪ C,
    or.elim h
        (assume h₁ : x ∈ A ∪ B,
        or.elim h₁
            (assume h₂ : x ∈ A,
```

```
            show x ∈ A ∪ (B ∪ C), from or.inl h₂)
            (assume h₂ : x ∈ B,
            have h₃ : x ∈ B ∪ C, from or.inl h₂,
            show x ∈ A ∪ (B ∪ C), from or.inr h₃))
        (assume h₁ : x ∈ C,
        have h₂ : x ∈ B ∪ C, from or.inr h₁,
        show x ∈ A ∪ (B ∪ C), from or.inr h₂))
(assume x,
    assume h : x ∈ A ∪ (B ∪ C),
    or.elim h
        (assume h₁ : x ∈ A,
        have h₂ : x ∈ A ∪ B, from or.inl h₁,
        show x ∈ (A ∪ B) ∪ C, from or.inl h₂)
        (assume h₁ : x ∈ B ∪ C,
        or.elim h₁
            (assume h₂ : x ∈ B,
            have h₃ : x ∈ A ∪ B, from or.inr h₂,
            show x ∈ (A ∪ B) ∪ C, from or.inl h₃)
            (assume h₂ : x ∈ C,
            show x ∈ (A ∪ B) ∪ C, from or.inr h₂)))
```

(c) Distribuitividade (Dica: veja o exemplo utilizado na seção "Propriedades")

```
import data.set
open set

variable U : Type
variables A B C : set U

example : A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C) :=
eq_of_subset_of_subset
(assume x,
    assume h : x ∈ A ∩ (B ∪ C),
    have h.r : x ∈ B ∪ C, from h.right,
    or.elim h.r
        (assume h₁ : x ∈ B,
        have h₂ : x ∈ A ∩ B, from and.intro h.left h₁,
        show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inl h₂)
        (assume h₁ : x ∈ C,
        have h₂ : x ∈ A ∩ C, from and.intro h.left h₁,
        show x ∈ (A ∩ B) ∪ (A ∩ C), from or.inr h₂))
(assume x,
    assume h : x ∈ (A ∩ B) ∪ (A ∩ C),
    or.elim h
        (assume h₁ : x ∈ A ∩ B,
```

```
23          have h₂ : x ∈ B ∪ C, from or.inl h₁.right,
24          show x ∈ A ∩ (B ∪ C), from and.intro h₁.left h₂)
25          (assume h₁ : x ∈ A ∩ C,
26          have h₂ : x ∈ B ∪ C, from or.inr h₁.right,
27          have h₃ : x ∈ A, from h₁.left,
28          show x ∈ A ∩ (B ∪ C), from and.intro h₃ h₂))
```

(d) Lei da Absorção (Dica: veja o exemplo utilizado em "Propriedades")

```
1  import data.set
2  open set
3
4  variable U : Type
5  variables A B : set U
6
7  example : A ∩ (A ∪ B) = A :=
8  eq_of_subset_of_subset
9  (assume x,
10     assume h : x ∈ A ∩ (A ∪ B),
11     show x ∈ A, from h.left)
12 (assume x,
13     assume h : x ∈ A,
14     have h₁ : x ∈ A ∪ B, from or.inl h,
15     show x ∈ A ∩ (A ∪ B), from and.intro h h₁)
```

(e) Lei de De Morgan

```
1  import data.set
2  open set
3  open classical
4
5  variable U : Type
6  variables A B : set U
7
8  example : -(A ∩ B) = -A ∪ -B :=
9  ext (assume x, iff.intro
10 (assume h₁ : x ∈ -(A ∩ B),
11     have g₁ : x ∈ (A ∪ -A), from em (x ∈ A),
12     have g₂ : x ∈ (B ∪ -B), from em (x ∈ B),
13     or.elim g₁
14         (assume h₂ : x ∈ A, or.elim g₂
15             (assume h₃ : x ∈ B, show x ∈ -A ∪ -B,
16                 from false.elim (h₁ ⟨h₂,h₃⟩))
17             (assume h₃ : x ∈ -B, show x ∈ -A ∪ -B,
18                 from or.inr h₃))
```

```
19          (assume h₂ : x ∈ -A, show x ∈ -A ∪ -B,
20              from or.inl h₂))
21  (assume h₁ : x ∈ -A ∪ -B,
22      assume h₂ : x ∈ (A ∩ B), show false,
23      from or.elim h₁
24          (assume h₃ : x ∈ -A, h₃ h₂.left )
25          (assume h₃ : x ∈ -B, h₃ h₂.right)))
26
27  example : -(A ∪ B) = -A ∩ -B :=
28  ext (assume x, iff.intro
29  (assume h₁ : x ∈ -(A ∪ B),
30      have g₁ : x ∈ -A, from
31          assume h₂ : x ∈ A,
32          have h₃ : x ∈ A ∪ B, from or.inl h₂, (h₁ h₃),
33      have g₂ : x ∈ -B, from
34          assume h₂ : x ∈ B,
35          have h₃ : x ∈ A ∪ B, from or.inr h₂, (h₁ h₃),
36      show x ∈ -A ∩ -B, from ⟨g₁, g₂⟩)
37  (assume h₁ : x ∈ -A ∩ -B,
38      show x ∈ -(A ∪ B), from
39          assume h₂ : x ∈ A ∪ B,
40          or.elim h₂
41              (assume h3 : x ∈ A, h₁.left h3)
42              (assume h3 : x ∈ B, h₁.right h3)))
```

2. Prove que $A \cup \overline{A} = \mathcal{U}$. (Fornecendo uma prova tradicional e uma em Lean.)

   Resposta

3. Questões com calc

Capítulo 6

Relações

Em capítulos anteriores, discutimos proposições que lidavam com a relação entre objetos matemáticos. Muitas vezes na matemática, e mesmo no contexto em que estamos inseridos, estamos interessados em definir e estudar relações entre objetos distintos. Por exemplo, podemos estar interessados em certas própriedades sobre a relação *é vais velho que*, entre seres vivos, e diremos que essa é uma relação *irreflexiva*, *transitiva*, ou ainda, uma relação de *ordem estrita*.

Nesse capítulo discutimos exatamente essas noções, e definimos certos tipos de relaçõs mais comuns.

6.1 Conceito de Relações

Dados dois conjuntos A e B, uma relação de A em B é qualquer subconjunto de $A \times B$ (Produto cartesiano).

Exemplo: $A = \{1, 2, 3\}$ e $B = \{4, 5, 6\}$
$R_1 = \{(1, 4), (2, 5)\}$
$R_2 = \{(1, 4), (1, 5), (2, 4), (3, 5)\}$

Para efeito de nomenclatura, considere, por exemplo, que o elemento $a \in A$ está relacionado a $b \in B$ por uma relação R, Podemos denotar aRb, ou $R(a, b)$ significando que o par $(a, b) \in R$, está definido como existente no universo daquela relação. Como se pode esperar, a relação pode ter qualquer aridade necessária, e relacionar objetos de tipos distintos.

O domínio de uma relação R de A em B é o conjunto formado pelos elementos de A que se relacionam com alguém de B, no caso da relação R_1, por exemplo, $Dom(R_1) = \{1, 2\}$.

O contradomínio de uma relação R de A em B é o conjunto de elementos de B que podem se relacionar com elementos de A, ou seja, é o própio B, $Cd(R_1) = Cd(R_2) = \{4, 5, 6\}$.

A imagem de uma relação R de A em B é o conjunto de elementos de B que são relacionados por alguém de A, no caso da relação R_2, por exemplo, $Im(R_2) = \{4, 5\}$.

Considere, a partir disso, o universo de objetos $u = \{Ana, Bia, Cid\}$, e a relação *conhece*, definida por $U \times U \supseteq A = \{(Ana, Bia), (Bia, Cid)\}$. Podemos dizer que *Bia conhece Ana*?

Definimos uma série de tipos de relações importantes, frequentemente encontradas na literatura. Note que muitas das definições se aplicam a relações conhecidas como *maior que*, nos naturais, ou *pertence* para conjuntos.

6.2 Relações em apenas um conjunto

Sendo U um conjunto, trata-se de uma relação de U em U.

Ex: $\{1, 2, 3, 4, 5\} \to \{1, 2, 3, 4, 5\}$
$R = \{(1, 2), (2, 3), (3, 2), (4, 5)\}$

6.2.1 Propiedades:

1. Relexiva : Em uma relação relexiva, cada elemento de U relaciona-se com ele mesmo.

   R é reflexiva $\Leftrightarrow (\forall x)(x \in U \to xRx)$

   Ex: $\{1, 2, 3\} \to \{1, 2, 3\}$

   $R = \{(1, 1), (2, 2), (3, 3), (2, 3), (1, 3)\}$

2. Simétrica : Em um relação simétrica, cada par de elementos de U se relacionam mutuamente.

   R é simétrica $\Leftrightarrow (\forall x)(\forall y)(x \in U \wedge y \in U \wedge xRy \to yRx)$

   Ex: $\{1, 2, 3\} \to \{1, 2, 3\}$

   $R = \{(1, 2), (2, 1), (2, 2)\}$

3. Antissimétrica : Em uma relação antissimétrica, os elementos só se relacionam mutuamente se forem iguais.

   R é Antissimétrica $\Leftrightarrow (\forall x)(\forall y)(x \in U \wedge y \in U \wedge (xRy \wedge yRx) \to x = y)$

   Ex: $\{1, 2, 3\} \to \{1, 2, 3\}$

   $R = \{(1, 2), (2, 2), (3, 1)\}$

4. Transitiva : Em uma relação transitiva, quando um elemento $x \in U$ de relaciona com $y \in U$, e este y se relaciona com $z \in U$, então x se relaciona com z.

   R é transitiva $\Leftrightarrow (\forall x)(\forall y)(\forall z)(x \in U \wedge y \in U \wedge z \in U \wedge (xRy \wedge yRz) \to xRz)$

   Ex: $\{1, 2, 3\} \to \{1, 2, 3\}$

   $R = \{(1, 2), (2, 3), (1, 3)\}$

6.3 Relações de Ordem

Discutimos uma classe de relações binárias importantes: as chamadas relações de ordem, que é uma relação ao mesmo tempo, relexiva, antissimétrica e transitiva. Aqui, definimos relações *parciais* ou *estritas*. Usaremos os símbolos $\leq$ e $<$ para nos referir a relações quaisquer entre elementos de alguma estrutura A, e os usamos infixados: $x \leq y$ ou $x < y$.

Definição 6.3.1. Seja $\leq$ uma relação. Dizemos que $\leq$ é de *ordem parcial* se respeita as seguintes propriedades:

- **reflexividade:** para todo $x \in A$, $x \leq x$.
- **transitividade:** para todo $x, y, z \in A$, se $x \leq y$, e $y \leq z$, então $x \leq z$.
- **antissimetria:** para todo $x, y \in A$, se $x \leq y$ e $y \leq x$, então $x = y$.

Note que se entendemos $\leq$ por um predicado binário, as definições acima são facilmente expressos em lógica de primeira ordem. Exemplos desse tipo são: $\leq$ em $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, e $\mathbb{R}$ ou a inclusão $\supseteq$ para a classe dos conjuntos.

Há ainda uma classe especial de relações de ordem vistas a seguir:

Definição 6.3.2. Dizemos que a relação de *ordem parcial* $\leq$ é *total* se:

- para todo $x, y \in A$, $x \leq y$ ou $y \leq x$.

Vale observar que nos exemplos anteriores, apenas $\leq$ é total. De fato, tome $A = \mathcal{P}(\mathbb{N})$, os conjuntos $x = 3$ e $y = 5$ subconjuntos de A; claramente não vale a completude de $\subseteq$ em A.

O que dizer, no entanto, das relações *menor* ou *pertence*? De fato, essas pertencem a classe a seguir, as chamadas relações de *ordem estrita*:

Definição 6.3.3. Considere $<$ relação em um conjunto A. Dizemos que a relação é de *ordem estrita* $\leq$ é *total* se:

- **transitividade:** para todo $x, y, z \in A$, se $x < y$ e $y < z$ então $x < z$.
- **irreflexividade:** para todo $x \in A$, $x \not< x$.

Dizemos, ainda, que essa relação estrita é total em A se:

- **tricotomia:** para todo $x, y \in A$, vale $x < y$, $x > y$ ou $x = y$.

Novamente, é facil ver como formalizar essas noções utilizando proposições em lógica de primiera ordem.

A seguir, discutimos um resultado intuitivo que estabelece uma ligação importante entre as relações de ordem *parciais* e *estritas*:

Proposição 6.3.1. *Considere $\leq$ parcial em A. Podemos definir uma relação estrita $<$ em A, em que $x < y$ significa que $x \leq y$ e $x \neq y$. Ainda, se $\leq$ for total, então $<$ também será total.*

Proposição 6.3.2. *Considere $<$ estrita em A. Podemos definir a relação de ordem parcial $\leq$ em A, em que $x \leq y$ significa que $x < y$ ou $x = y$. Ainda, se $<$ for total, então $\leq$ também será total.*

Demonstração. Exercício para o leitor! □

6.4 Relações de Equivalência

6.4.1 Equivalencia e Igualdade

Apenas discutimos brevemente como e porque Equivalencia e Igualdade são animais completamente diferentes. Toma os exemplos acima pra discutir.

6.5 Relações em Lean

6.6 Exemplos

Os exemplos abaixo serão eventuais exercícios. Mas já são dados com as soluções. Alguns poderão ser retirados e postos no interior do capítulo para ser usados como exemplos.

1. Dada uma relação de ordem estrita R, definimos R', parcial. Dê uma prova para os seguintes resultados sobre R':

```
1  section
2  -- Inicio da Sessao
3
4  parameters {A : Type} {R : A → A → Prop}
5  parameter (irreflR : irreflexive R)
6  parameter (transR : transitive R)
7
8  local infix < := R
9
10 def R' (a b : A) : Prop := R a b ∨ a = b
11 local infix ≤ := R'
12
13 -- Reflexividade de R'
14 example (a : A) : a ≤ a :=
15   have h2 : a = a, from rfl,
16   show a ≤ a, from (or.inr h2)
17
18 -- Transitividade de R'
19 example {a b c : A} (h1 : a ≤ b) (h2 : b ≤ c): a ≤ c :=
20   or.elim h1
```

```
21      (assume s1 : a < b,
22        or.elim h2
23          (assume s2 : b < c,
24            or.inl (transR s1 s2))
25          (assume s2 : b = c,
26            or.inl (eq.subst s2 s1)))
27      (assume s1 : a = b,
28        eq.subst s1.symm h2)
29
30  -- Antissimetria de R'
31  example {a b : A} (h1 : a ≤ b) (h2 : b ≤ a) : a = b :=
32    or.elim h1
33      (assume s1 : a < b,
34        or.elim h2
35          (assume s2 : b < a,
36            have s3: a < a, from transR s1 s2,
37            false.elim (irreflR a s3))
38          (assume s2 : b = a, s2.symm))
39      (assume s1 : a = b, s1)
40
41  -- Fim da Sessao
42  end
```

2. Dada uma relação R, definimos uma relação S supostamente transitiva. Dê uma prova para esse fato.

```
1   section
2   parameters {A : Type} {R : A → A → Prop}
3   parameter (reflR : reflexive R)
4   parameter (transR : transitive R)
5
6   def S (a b : A) : Prop := R a b ∧ R b a
7
8   example : transitive S :=
9     assume a b c,
10    assume h1 : S a b,
11    assume h2 : S b c,
12    show S a c, from
13      have l1 : R a b, from h1.left,
14      have l2 : R b c, from h2.left,
15      have r1 : R b a, from h1.right,
16      have r2 : R c b, from h2.right,
17      ⟨transR l1 l2, transR r2 r1⟩
18  end
```

3. Apenas um dos teoremas é verdadeiro. Defina qual o verdadeiro, e dê uma prova para a sua resposta.

```
section
  parameters {A : Type} {a b c : A} {R : A → A → Prop}
  parameter (Rab : R a b)
  parameter (Rbc : R b c)
  parameter (nRac : ¬ R a c)

  -- R e parcial estrita
  theorem R_strict : irreflexive R ∧ transitive R :=
  sorry

  -- R nao e parcial estrita
  theorem R_not_strict : ¬(irreflexive R ∧ transitive R) :=
    assume h : irreflexive R ∧ transitive R,
    have h1 : transitive R, from h.right,
    have h2 : R a c, from h1 Rab Rbc,
    show false, from nRac h2
end
```

4. Prove o fato a seguir utilizando paradigma *calc*. Tente o mesmo apenas através de *tatic mode* e *term mode*.

```
open nat

-- utilizando modo calc
example : 1 ≤ 4 :=
calc
    1 ≤ 2 : le_succ 1
  ... ≤ 3 : le_succ 2
  ... ≤ 4 : le_succ 3

-- utilizando term mode
example : 1 ≤ 4 :=
  have h1 : 1 ≤ 2, from le_succ 1,
  have h2 : 2 ≤ 3, from le_succ 2,
  have h3 : 3 ≤ 4, from le_succ 3,

  le_trans h1 (le_trans h2 h3)

-- utilizando tatic mode
example : 1 ≤ 4 := sorry
```

Capítulo 7

Funções

Desde o final do século XIX, diversas áreas da matemática estudam consistentemente *conjuntos*, *relações*, já apresentadas nesse texto, e funções. Neste capítulo, debruçaremos nossa atenção nas propriedades dessa terceira área.

Entende-se que uma função f é um mapeamento entre um domínio X e um domínio Y, este que será conhecido como contradomínio, posteriormente. Entretanto, para teóricos de conjuntos, esses domínios são simplesmente considerados conjuntos. Vimos que *Lean* é uma linguagem baseada em Tipos. Dessa forma, estabelece-se uma diferença clara entre o Domínio X, caracterizado pelos Tipos, e o Conjunto A, que é do tipo (sub)conjunto de X. Em Lean:

```
1 variable X : Type
2 variable A : set X
```

Entretanto a visão da função como mapeamento entre conjuntos é comum entre os matemáticos e será considerada nesse texto, fazendo as devidas comparações com a linguagem de referência, o Lean.

7.1 O Conceito de Função

Considere dois conjuntos quaisquer X e Y e um mapeamento f do conjunto X para o conjunto Y. Se f atribui um e apenas um valor para cada elemento de X, dizemos que f é uma função total e escrevemos $f : X \to Y$. Chamamos X de domínio de f, enquanto Y é o contradomínio de f e $\forall x, (x \in X \Rightarrow f(x) \in Y)$. Nesse sentido, $\forall x_1 \in X, \forall x_2 \in X, x_1 = x_2 \Rightarrow f(x_1) = f(x_2)$. Uma função também pode ser parcial quando ela não é definida para alguns valores do domínio. Escrevemos $f : X \nrightarrow Y$ para representar uma função parcial definida em $A \subseteq X$, mas não definida no complementar de A.

Um exemplo amplamente conhecido de funções parciais é $f : \mathbb{R} \nrightarrow \mathbb{R}$, definida por $f(x) = \frac{1}{x}$, que é indefinida em $x = 0$. Assim, na realidade, expressamos f como $f : \mathbb{R} \setminus \{0\} \to \mathbb{R}$. Alguns chamam A de domínio de definição. Como toda função parcial é total ao se restringir o domínio, trateremos nesse texto

das funções totais e a extensão das definições devido à parcialidade são deixadas ao leitor.

A maneira mais simples de se representar uma função é escrevê-la explicitamente para cada elemento do domínio. Por exemplo, podemos escrever as seguintes expressões:

- Seja $f : \mathbb{N} \to \mathbb{N}$ definida por $f(n) = 2 \cdot n + 1$
- Seja $g : \mathbb{N} \to \mathbb{R}$ definida por $g(n) = \frac{n}{n+1}$
- Seja $h : \mathbb{R} \to \{0, 1\}$ definida por

$$\begin{cases} h(x) = 1 \; if \; x \in \mathbb{Q} \\ h(x) = 0 \; if \; x \notin \mathbb{Q} \end{cases}$$

A questão que se levanta é: o que torna uma expressão explícita legítima? Neste momento, deixaremos essa questão de lado e notaremos que a matemática é confortável com diversos tipos de definições, como, por exemplo, a definição de $h(x)$ no exemplo acima. Nesse mesmo exemplo, não fica claro em como podemos definir algoritmicamente se um número real de entrada é um número racional ou não. Porém, esse não é o objetivo do capítulo.

Note que a escolha das variáveis x e n são arbitrárias. Isto nos leva à definição no Capítulo de *FOL* de variável ligada (*bound*), visto que se renomearmos x com y, os valores continuam os mesmos.

Outra forma muito comum, principalmente na ciência da computação de definir funções é utilizando **recorrência**. Esse tipo de escrita é mais comum quando o domínio são os números naturais. Essas função são conhecidas como **sequências**.

Exemplo 7.1.1. Considere $f : \mathbb{N} \to \mathbb{N}$, onde definimos $f(0) = 1$ e $f(n) = n \cdot f(n-1)$, para todo $n \in \mathbb{N}, n > 0$. Essa função é chamada de *fatorial* de n.

Exemplo 7.1.2. Considere $g : \mathbb{N} \to \mathbb{N}$, onde definimos $f(0) = 0$, $f(1) = 1$ e $f(n) = f(n-1) + f(n-2)$, para todo $n \in \mathbb{N}, n > 1$. Essa função é chamada de *fibonacci*. Ela determina uma sequência conhecida, dada por $0, 1, 1, 2, 3, 5, 8, ...$

Lógicos frequentemente utilizam a notação $\lambda x \; e(x)$ para denotar a função que mapeia x para $e(x)$. Essa notação chama-se *notação lambda* e pode ser usada da seguinte forma: $f = \lambda x(x + 1)$, que significa $f(x) = x + 1$. Essa notação é mais interessante para cientistas da computação e lógicos do que propriamente para matemáticos. Em Lean, definimos da seguinte forma:

```
1 variables X Y : Type
2 variable f : X → Y
3
4 def square (n : ℕ) := n*n
5
6 def double (n : ℕ) := 2*n
```

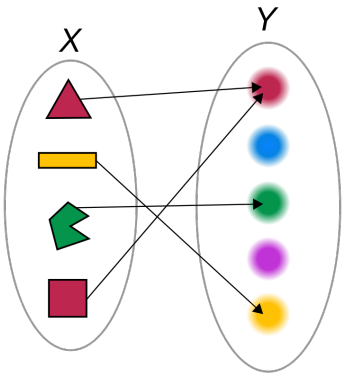


Figura 7.1: Mapeamento entre o Domínio X e o Domínio Y, ambos não numéricos.

```
7
8  variable n: ℕ
9
10 #check square n      -- square n : ℕ
11
12 #reduce square 3     -- 9
13
14 #eval square 12120   -- #reduce 12120 return an error, because the
15                      -- calculation is by deep recursion
16
17 example (n : ℕ) : double 2 = square 2 :=
18 by rw [double, square]
```

Lembre-se que diferenciamos o tipo X do tipo *set* X em Lean.

Outra forma comum de representar funções de forma didática é utilizando gráficos, como podemos observar na Figura 7.1. Essa representação tem uma limitação muito clara que é quando X é um conjunto infinito. Um conjunto é finito quando existe um número natural n tal que exista uma função b bijetiva entre esse conjunto e o conjunto dos números naturais menores ou iguais a n.

A definição de função bijetiva pode ser encontrada na Definiçao 7.3.3, enquanto maiores informações sobre os números naturais podem ser encontradas no Capítulo de *Indução dos Números Naturais*. Conjuntos encontram-se no *Capítulo 5*.

A partir de agora, vamos introduzir uma série de definições sobre a terminologia que de funções, que acabamos de apresentar, e vamos extrair proposições importantes a partir.

7.2 Primeiras Definições

Definição 7.2.1. Seja um conjunto A. A função identidade de A é a função $i_A : A \to A$ definida para todos os valores $x \in A$ tal que $i_A(x) = x$.

Definição 7.2.2. Sejam $f : X \to Y$ e $g : Y \to Z$ funções. Defina $k : X \to Z$ por $k(x) = g(f(x))$. A função k é chamada de composição de f e g ou f composta com g e é escrita $g \circ f$. Desta forma, para cada elemento, primeiro avaliamos $f(x) \in Y$ e depois avaliamos $g(f(x)) \in Z$.

Podemos ver em Lean as definições de composição e de identidade. Note que estamos em um namespace hidden para que não haja conflito de definições.

```
1  namespace hidden
2      variables {X Y Z : Type}
3
4      def comp (f : Y → Z) (g : X → Y) : X → Z :=
5      λx, f (g x)
6
7      infixr  ‘ ∘ ‘ := comp
8
9      #check @comp
10
11     def id (x : X) : X := x
12
13     #check @id
14
15 end hidden
```

Nesse, utilizo o comando λ, na linha 5. Esse símbolo, nesse sentido, tem significado da palavra `assume`, que já foi trabalha ao percorrer do livro.

Definição 7.2.3. Considere $f, g : X \to Y$. Dizemos que essas funções são iguais, quando para todos os valores do domínio X, a correspondência no contradomínio Y é a mesma. Em lógica simbólica, $\forall x(x \in X \to (f(x) = g(x)) \iff f = g$.

Obseve que em termos de lógica formal de tipos, poderíamos reescrever como $\forall x : X(f(x) = g(x)) \iff f = g$. Escrevemos essa equivalência entre lógica e funções utilizando a extensionalidade das funções, muito semelhante à descrita no capítulo de *conjuntos*. Por exemplo, se $f, g : \mathbb{R} \to \mathbb{R}$ definidas por $f(x) = x+1$ e $g(x) = 1 + x$, então $f = g$, pois para cada valor de x, vale a comutatividade da soma.

Em lean, o comando `funext` (de "function extensionality") prova a igualdade de funções.

```
1  variables {X Y : Type}
2
3  example (f g : X → Y) (h : ∀ x, f x = g x) : f = g :=
4      funext h
```

Proposição 7.2.1. *Para todo $f : X \to Y$, $f \circ i_X = f$ e $i_Y \circ f = f$*

Demonstração. Seja x um elemento qualquer de X. Então $(f \circ i_X)(x) = f(i_X(x)) = f(x)$ e $(i_Y \circ f)(x) = i_Y(f(x)) = f(x)$, o que mostra a igualdade. □

No Lean, podemos mostrar essa proposição da seguinte forma. Para algumas funções, será necessário escrevermos `open function`.

```
variables {X Y Z W : Type}

lemma left_id (f : X → Y) : id ∘ f = f := rfl

lemma right_id (f : X → Y) : f ∘ id = f := rfl

theorem comp.assoc (f : Z → W) (g : Y → Z) (h : X → Y) :
    (f ∘ g) ∘ h = f ∘ (g ∘ h) := rfl
```

Definição 7.2.4. Suponha $f : X \to Y$ e $g : Y \to X$ satisfaz $g \circ f = i_X$. Assim, $g(f(x)) = i_X(x) = x$, para todos os elementos de X. Neste caso g é dita inversa à esquerda de f e f é dita inversa à direita de g. Quando g é inversa à direita e é inversa à esquerda de f, então g é dita simplesmente inversa de f.

Exemplo 7.2.1. Defina $f, g : \mathbb{R} \to \mathbb{R}$ por $f(x) = x + 1$ e $g(x) = x - 1$. Então g é inversa à direita e à esquerda de f e vice-versa.

Exemplo 7.2.2. Denote $\mathbb{R}^+$ como os reais não negativos. Defina $f : \mathbb{R} \to \mathbb{R}^+$ por $f(x) = x^2$ e defina $g : \mathbb{R}^+ \to \mathbb{R}$ por $g(x) = \sqrt{x}$. Então $f(g(x)) = (\sqrt{x})^2 = x$, para todo x no domínio de g. Então f é inversa à esquerda de g e g inversa à direita de f. Por outro lado, $g(f(x)) = \sqrt{x^2} = |x|$. Logo g não é inversa à esquerda de f e f não é inversa à direita de g.

Proposição 7.2.2. *Suponha que $f : X \to Y$ tem inversa à esquerda, $h : Y \to X$ e uma inversa à direita, $k : Y \to X$. Então $h = k$.*

Demonstração. Seja $y \in Y$. $h(f(k(y))) = k(y)$, pois h é uma inversa à esquerda de f. Por outro lado, $f(k(y)) = y$ e, portanto $h(f(k(y))) = h(y)$. Assim $k(y) = h(y)$ e as funções são iguais. □

Essa proposição pode também ser vista em Lean. Considerarei a prova utilizando táticas e utilizando termos. O leitor pode estudar aquela que sentir mais confortável. Note que neste exemplo, já foi necessário o `namespace function`, visto que `left_inverse` e `right_inverse` já estão predefinidas.

```
open function

variables {X Y Z: Type}

-- Term and Calc Mode
example (f: X → Y) (h: Y → X) (k: Y → X)
```

```
7     (hf: left_inverse h f) (fk: right_inverse k f): h = k :=
8     have H: ∀ ( x : Y ), h x = k x, from
9         assume x,
10        have h1: h (f (k x)) = k x , from calc
11            h ( f (k x)) = k x: by apply hf,
12        have h2: h (f (k x)) = h x, from calc
13            h (f (k x)) = h x: by rw fk,
14        show h x = k x, from eq.trans (eq.symm h2) h1,
15    show h = k, from funext H
16
17 -- Tatics Mode
18 example (f: X → Y) (h: Y → X) (k: Y → X)
19    (hf: left_inverse h f) (fk: right_inverse k f): h = k :=
20    begin
21        apply funext,
22        assume x,
23        have hf: h (f (k x)) = k x, by apply hf,
24        rw ←hf,
25        rw fk,
26    end
```

Proposição 7.2.3. *Seja $f : X \to Y$. Se a inversa de f existe, então ela é única. Isto é, se $g_1, g_2 : Y \to X$ são inversas de f, então $g_1 = g_2$*

Demonstração. Sabemos que g_1 é inversa à esquerda de f. Então,

$$g_1(f(g_2(x))) = g_2(x), \text{ para todos os valores de } x.$$

Também, g_2 é inversa de f. Então,

$$g_1(f(g_2(x))) = g_1(x), \text{ para todos os valores de } x.$$

Logo $g_1 = g_2$. □

Quando a inversa de f existe, então, podemos escrevê-la como f^{-1}. Dada a Definição 7.2.4, podemos afirmar que $(f^{-1})^{-1} = f$.

Observe que uma função pode possuir mais de uma função inversa à esquerda ou mais de uma função inversa à direita. Ainda, quando uma função possui mais de uma função inversa à esquerda, ela não possui inversa à direita. Se ela possuisse, ela seria igual a todas as funções inversas à esquerda pela Proposição 7.2.2, portanto as funções inversas à esquerda seriam todas iguais, o que é uma contradição,por hipótese. O mesmo vale para inversas à direita.

Proposição 7.2.4. *Seja $f : X \to Y$ e $g : Y \to Z$. Se $h : Y \to X$ e $k : Z \to Y$ são inversas à esqueda de f e g, respectivamente, então $h \circ k$ é inversa à esquerda de $g \circ f$. O mesmo vale quando substituimos esquerda por direita na proposição.*

Demonstração. $(h \circ k) \circ (g \circ f)(x) = h(k(g(f(x)))) = h(f(x)) = x$. A demonstração quando substituimos a proposição de esquerda para direita é análoga e é deixada como exercício ao leitor. □

Corolário 7.2.1. *Se $f : X \to Y$ e $g : Y \to Z$ possuem inversas, então $(f \circ g)^{-1}$ existe e $(f \circ g)^{-1} = f^{-1} \circ g^{-1}$.*

Demonstração. Segue diretamente da Proposição 7.2.4 e Proposição 7.2.2. □

7.3 Funções Injetiva, Sobrejetiva e Bijetiva

Definição 7.3.1 (Função Injetiva)**.** Também conhecida como função injetora, é uma função em que elementos distintos do domínio são mapeados para elementos diferentes do contradomínio. Dessa forma, seja $f : X \to Y$. Se $\forall x_1 \in X, \forall x_2 \in X, x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$, f é injetiva. A definição pode ser feita pela contrapositiva dessa afirmação, também. A definição pela contrapositiva será bastante utilizada nas demonstrações.

Definição 7.3.2 (Função Sobrejetiva)**.** Também conhecida como função bijetora, é uma função em que todos os elementos do contradomínio estão na imagem da função. Dessa forma, seja $f : X \to Y$. Se $\forall y \in Y, \exists x \in X, f(x) = y$, f é sobrejetiva.

Definição 7.3.3 (Função Bijetiva)**.** Também conhecida como bijeção ou correspondência um a um, é uma função simultaneamente injetiva e sobrejetiva.

Em Lean, essas definições são descritas no `namespace function`.

Exemplo 7.3.1. Considere os conjuntos $X = \{1, 2, 3, 4\}$, numéricos e $Y = \{John, Paul, George, Ringo\}$, não numérico.É possível construir uma função bijetiva entre X e Y. Ainda mais, toda função sobrejetiva entre esses conjuntos é também injetiva e consequentemente bijetiva.

```
1 variables {X Y: Type}
2
3 def injective (f : X → Y) : Prop :=
4 ∀ x₁ x₂, f x₁ = f x₂ → x₁ = x₂
5
6 def surjective (f : X → Y) : Prop :=
7 ∀ y, ∃ x, f x = y
8
9 def bijective (f : X → Y) := injective f ∧ surjective f
```

Proposição 7.3.1. *Seja $f : X \to Y$.*

I Se f possui inversa à esquerda, f é injetiva.

II Se f possui inversa à direita, f é sobrejetiva.

III Se f possui inversa, f é bijetiva.

Demonstração. Para provar I), suponha que $f(x_1) = f(x_2)$ e que g é inversa à esquerda de f. Assim $g(f(x_1)) = x_1$ e $g(f(x_2)) = x_2$. Portanto $x_1 = x_2$ e está provado. Para provar II), considere h inversa à direita de f. Seja $y \in Y$ e $x = h(y)$. Então $f(x) = f(h(y)) = y$. A terceira sai diretemente das duas anteriores e da Proposição 7.2.2. □

```
1  open function
2
3  variables {X Y : Type}
4
5  theorem inj_of_left_inverse {g : Y → X} {f : X → Y} :
6  left_inverse g f → injective f :=
7      assume h, assume x₁ x₂, assume feq,
8      calc x₁ = g (f x₁) : by rw h
9          ... = g (f x₂) : by rw feq
10         ... = x₂       : by rw h
11
12 theorem surj_of_right_inverse {g : Y → X} {f : X → Y} :
13 right_inverse g f → surjective f :=
14     assume h, assume y,
15     let  x : X := g y in
16     have f x = y, from calc
17         f x  = (f (g y))    : rfl
18         ... = y            : by rw [h y],
19     show ∃ x, f x = y, from exists.intro x this
```

Proposição 7.3.2. *Seja $f : X \to Y$*

I Se X é não vazio e f é injetiva, então f possui inversa à esquerda.

II Se f é sobrejetiva, então f possui inversa à direita.

III Se X é não vazio e f é bijetiva, então f possui inversa.

Demonstração. Seja $\hat{x} \in X$. Defina uma função $g : Y \to X$ com $g(y) = x$, tal que $f(x) = y$, se y pertence a imagem de f. Caso não pertença, defina $g(y) = \hat{x}$. Agora, assuma $x \in X$. Suponha que $g(f(x)) = x'$. Essa suposição é válida, pois $f(x) \in Y$. Pela definição de g, como $f(x)$ pertence a imagem de f, então $g(f(x)) = x'$, tal que $f(x') = f(x)$. Como f é injetiva, temos que $x = x'$ e $g(f(x)) = x$ e g é inversa à esquerda de f.

Para a segunda afirmação, defina $h : Y \to X$, onde, para cada $y \in Y$, escolha um elemento $x \in X$, tal que $f(x) = y$. Note que a sobrejetividade garante a existência desse elemento, mas não garante a unicidade na escolha. Então $f(h(y)) = f(x) = y$, por definição de h.

A fim de provar a terceira, basta as demonstrações das anteriores e da Proposição 7.2.2. □

Algumas observações sobre essa demonstração são importantes. Ao definir g na primeira parte, precisa-se decidir se $x \in X$ existe tal que $f(x) = y$. Isso pode não ser algoritmicamente feito, logo g poderia não ser computável. Na construção de h, a prova requere que haja uma escolha de valor de x entre os possíveis candidatos. Isto é uma versão do *axioma da escolha*. Este axioma foi muito debatido no século XX, mas hoje já é comum para demonstrações. O paradoxo de Banach-Tarski é um argumento contra o axioma. Um bom vídeo sobre o assunto pode ser encontrado nesse link.

Utilizando conceitos e resultados da seção anterior, podemos provar a seguinte proposição.

Proposição 7.3.3. *Seja $f : X \to Y$ e $g : Y \to Z$.*

I Se f e g são injetivas, $g \circ f$ também será.

II Se f e g são sobrejetivas, $g \circ f$ também será.

Demonstração. Basta aplicarmos a Proposição 7.2.4 e definirmos h e k como inversas à esquerda de f e g respectivamente. Logo $(h \circ k)$ é inversa à esquerda de $(g \circ f)$ e, portanto, ela é injetiva.

O mesmo vale para a segunda afirmação. □

```
1  open function
2
3  namespace hidden
4      variables {X Y Z : Type}
5
6      theorem injective_comp {g : Y → Z} {f : X → Y}
7      (Hg : injective g) (Hf : injective f) :
8      injective (g ∘ f) :=
9          assume x₁ x₂,
10         assume : (g ∘ f) x₁ = (g ∘ f) x₂,
11         have f x₁ = f x₂, from Hg this,
12         show x₁ = x₂, from Hf this
13
14     theorem surjective_comp {g : Y → Z} {f : X → Y}
15         (hg : surjective g) (hf : surjective f) :
16     surjective (g ∘ f) :=
17     begin
18         assume z,
19         apply exists.elim (hg z),
20         assume y (hy: g y = z),
21         apply exists.elim (hf y),
22         assume x (hx: f x = y),
23         rw ←hx at hy,
24         apply exists.intro x hy
25     end
```

```
26
27      theorem bijective_comp {g : Y → Z} {f : X → Y}
28          (hg : bijective g) (hf : bijective f) :
29          bijective (g ∘ f) :=
30      have ginj : injective g, from hg.left,
31      have gsurj : surjective g, from hg.right,
32      have finj : injective f, from hf.left,
33      have fsurj : surjective f, from hf.right,
34      and.intro (injective_comp ginj finj)
35                  (surjective_comp gsurj fsurj)
36  end hidden
```

Exemplo 7.3.2. Considere $f : \mathbb{N} \to Y$, tal que $f(n) = 2n$. Podemos, ter:

- $Y = \mathbb{N}$. f é injetiva, mas não é sobrejetiva.
- $Y = \mathbb{R}$. f é injetiva, mas não é sobrejetiva.
- $Y = \{n \in \mathbb{N} | n \ par\}$. f é bijetiva.

7.4 Funções e Subconjuntos do Domínio

Nós podemos querer saber o comportamento de uma função em algum subconjunto A de X. Por exemplo, podemos dizer que f é injetiva em A se para todo x_1 e x_2 em A, $f(x_1) = f(x_2)$ implicar $x_1 = x_2$, por exemplo. A diferença com relação à função parcial é que nesse caso, a função pode ser definida em seu complementar.

Definição 7.4.1. Se f é função de X e Y, dizemos que $f[A]$ denota a imagem de f em A, definido por $f[A] = \{y \in Y | \ \exists x \in A, y = f(x)\}$.

Proposição 7.4.1. *Seja $f : X \to Y$ e A um subconjunto de X. Então, para todo x em A, $f(x)$ está em $f[A]$.*

Demonstração. Por definição, $f(x) \in f[A]$ se, e somente se, existe x' em A tal que $f(x') = f(x)$. Isto vale para $x' = x$. □

Em Lean, utilizando Táticas, esta prova pode ser apresentada da seguinte forma:

```
1  import data.set
2  open function
3
4  variables {X Y : Type}
5
6  example (f: X → Y) (A : set X) (a : X) (h: a ∈ A) : (f a) ∈ f ''
       A :=
7  begin
```

```
8     apply exists.intro a,
9     exact and.intro h (eq.refl (f a)),
10 end
```

Proposição 7.4.2. *Seja $f : X \to Y$ e $g : Y \to Z$. Seja A subconjunto de X. Então*

$$(g \circ f)[A] = g[f[A]]$$

Demonstração. Seja $z \in (g \circ f)[A]$. Então, para algum $x \in A$, $z = (g \circ f)(x) = g(f(x))$. Pelo que acabamos de provar na Proposição 7.4.2, $f(x) \in f[A]$. Novamente, pelo que acabamos de provar, $g(f(x)) \in g[f[A]]$.

Alternativamente, seja $z \in g[f[A]]$. Então, existe y em $f[A]$ tal que $f(y) = z$. Como $y \in f[A]$, existe $x \in A$, tal que $f(x) = y$. Então $(g \circ f)(x) = g(f(x)) = g(y) = z$, então $z \in (g \circ f)[A]$ □

Uma prova de que a composição de funções sobrejetivas é sobrejetiva é a que descrevemos cima, pois $f : X \to Y$ é sobrejetiva se, e somente se, $f[X] = Y$.

Nós podemos ver f como uma função de A, um subconjunto de X a Y, simplesmente ignorando o comportamento de f nos elementos fora de A.

Definição 7.4.2. Denotamos $f \restriction A$ como restrição de f para A. Isto é, dadas $f : X \to Y$ e $A \subseteq X$, $f \restriction A : A \to Y$ é definida por $(f \restriction A)(x) = x$, para todo x em A.

Agora, f é injetiva em A significa que a restrição de f em A é injetiva.

Definição 7.4.3 (Pré-imagem)**.** Se $f : X \to Y$ e $B \subseteq Y$, então a pré-imagem de B em f, denotado por $f^{-1}[B]$ é definida por $f^{-1}[B] = \{x \in X | f(x) \in B\}$. Ou seja, é o conjunto de elementos de X que são mapeados em B.

Note que essa definição faz sentido mesmo que f não tenha inversa, visto que dado $y \in B$ pode não haver $x \in X$ com a propriedade de $f(x) \in B$, como podem ter vários. Se f tem inversa (f^{-1}), então, para todo y em B, existe exatamente um elemento x em X com $f(x) \in B$. Neste caso, dizemos que $f^{-1}[B]$ é a imagem de B sobre f^{-1} ou a pré-imagem de B sobre f.

Em Lean, a definição de Pré-imagem encontra-se na bilbioteca *Mathlib* e é definida da seguinte forma:

```
1 variables {X Y : Type}
2
3 def preimage (f : X → Y) (B : set Y) : set X := {x : X | f(x) ∈
      B}
4
5 infix ` ⁻¹' `:80 := preimage
```

Proposição 7.4.3. *Seja $f : X \to Y$, $g : Y \to Z$ e $C \subseteq Z$. Então*

$$(g \circ f)^{-1}[C] = f^{-1}[g^{-1}[C]]$$

.

Demonstração. Para qualquer y, $y \in (g \circ f)^{-1}[C]$ se, e somente se, $g(f(y))$ está em C. Isto acontece se, e somente se, $f(y) \in g^{-1}[C]$, o que acontece se, e somente se $y \in f^{-1}[g^{-1}[C]]$. □

Seguem algumas propriedades sobre imagens e pré-imagens. Aqui, f denota uma função arbitrária de X em Y. $A, A_1, A_2, ...$ denotam subconjuntos arbitrários de X, e $B, B_1, B_2, ...$ denotam subconjuntos arbitrários de Y.

1. $A \subseteq f^{-1}[f[A]]$, e se f é injetiva, $A = f^{-1}[f[A]]$.
2. $f[f^{-1}[B]] \subseteq B$, e se f é sobrejetiva, $B = f[f^{-1}[B]]$.
3. Se $A_1 \subseteq A_2$, então $f[A_1] \subseteq f[A_2]$.
4. Se $B_1 \subseteq B_2$, então $f^{-1}[B_1] \subseteq f^{-1}[B_2]$.
5. $f[A_1 \cup A_2] = f[A_1] \cup f[A_2]$.
6. $f^{-1}[B_1 \cup B_2] = f^{-1}[B_1] \cup f^{-1}[B_2]$.
7. $f[A_1 \cap A_2] \subseteq f[A_1] \cap f[A_2]$ e, se f é injetiva, $f[A_1 \cap A_2] = f[A_1] \cap f[A_2]$.
8. $f^{-1}[B_1 \cap B_2] = f^{-1}[B_1] \cap f^{-1}[B_2]$.
9. $f[A] \setminus f[B] \subseteq f[A \setminus B]$.
10. $f[A] \cap B = f[A \cap f^{-1}[B]]$.
11. $f[A \cup f^{-1}[B]] \subset f[A] \cup B$.
12. $A \cap f^{-1}[B] \subseteq f^{-1}[f[A] \cap B]$.
13. $A \cup f^{-1}[B] \subseteq f^{-1}[f[A] \cup B]$.

A partir de agora, vamos demonstrar algumas dessas proporições, ora com descrições da demonstração e ora com provas em Lean. Tambémm sugerimos que essas proposições sejam exercícios para a leitora ou o leitor.

7.4.1 Demonstrações com linguagem natural

Proposição 7.4.4 (Item 7)**.** *Sejam X e Y conjuntos, $A_1, A_2 \in X$ e $f : X \to Y$.*

Demonstração. Se $y \in f[A_1 \cap A_2]$, temos que existe $x \in A_1 \cap A_2$, com $f(x) = y$. Nesse caso, $f(x) \in f[A_1]$, e $f(x) \in f[A_2]$, o que demonstra a primeira parte.

Agora, suponha a injetividade de f. Suponha também que $y \in f[A_1] \cap f[A_2]$. Assim, existem $x_1 \in A_1$ e $x_2 \in A_2$, com as propriedades de $f(x_1) = y = f(x_2)$. Como a função é injetiva, $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$. Assim $x_1 \in A_2$, que implica $x_1 \in A_1 \cap A_2$. Logo, $y \in f[A_1 \cap A_2]$. □

Proposição 7.4.5 (Item 10)**.** *Sejam X e Y conjuntos, $f : X \to Y, A \subseteq X$ e $B \subseteq Y$. Então, $f[A] \cap B = f[A \cap f^{-1}[B]]$*

Demonstração. Suponha, inicialmente, que $y \in f[A] \cap B$. Então $y \in B$ e para algum $x \in A$, $f(x) = y$. Nesse sentido, $x \in f^{-1}[B]$. Assim, $x \in A \cap f^{-1}[B]$ e, portanto, $y \in f[A \cap f^{-1}[B]]$.

Alternativamente, se $y \in f[A \cap f^{-1}[B]]$, existe $x \in A \cap f^{-1}[B]$, com $f(x) = y$. Daqui, concluímos que $y = f(x) \in f[A]$ e $y \in B$, pela definição de pré-imagem. Então $y \in f[A] \cap B$, como queríamos provar. □

Proposição 7.4.6 (Item 12)**.**

Demonstração. Suponha que $x \in Af^{-1}[B]$. Desta maneire, $x \in A$ e $x \in f^{-1}[B]$, que implica que existe $y \in B$, tal que $f(x) = y$. Nesse sentido, $y \in B$ e $y \in f[A]$, que implica $y \in f[A]B$. Como $f(x) = y$, então $x \in f^{-1}[f[A]B]$.

□

7.4.2 Demontrações em Lean

Nós podemos utilizar variáveis limitadas para falar sobre o comportamento de funções em conjuntos particulares.

```
1  import data.set -- inclui o ísmbolo de subconjunto da imagem ''
2  open set function
3
4  variables {X Y : Type}
5  variables (A  : set X) (B : set Y)
6
7  def maps_to (f : X → Y) (A : set X) (B : set Y) :=
8      ∀ x ∈ A, f x ∈ B
9
10 def inj_on (f : X → Y) (A : set X) :=
11     ∀ (x₁ ∈ A) (x₂ ∈ A), f x₁ = f x₂ → x₁ = x₂
12
13 def surj_on (f : X → Y) (A : set X) (B : set Y) :=
14     B ⊆ f '' A
```

A definição de `maps_on` é a ideia de que a imagem de um subconjunto do domínio X está totamente inclusa em um subconjunto específico do contra-domínio. As definições de `inj_on` e `surj_on` são as definições usuais.

Proposição 7.4.7 (Item 1)**.**

```
1  import data.set
2  open function
3  open set
4
5  variables {X Y : Type}
6
7  theorem item1.a (f : X → Y) (A : set X) : A ⊆  f ⁻¹' (f '' A) :=
8  assume a h,
```

```
have h1: a ∈ A ∧ f a = f a, from and.intro h (eq.refl (f a)),
have h2: (f a) ∈ f '' A, from exists.intro a h1,
show a ∈ f ⁻¹' (f '' A), from h2

theorem item1.b (f : X → Y) (A : set X) (h: injective f):
        A =  f ⁻¹' (f '' A) :=
eq_of_subset_of_subset
(item1.a f A)
(assume x h1,
have h2: f x ∈ f '' A, from h1,
have h3: ∃ (a : X), a ∈ A ∧ f a = f x, from h1,
show x ∈ A, from exists.elim h3
    (assume (y : X) (h4: y ∈ A ∧ f y = f x),
    have h5: y = x, from h h4.right,
    show x ∈ A, from eq.subst h5 h4.left)
)
```

Proposição 7.4.8 (Item 3)**.**

```
import data.set
open set function

variables {X Y : Type}
variables (A B A₁ A₂ : set X)

theorem item3 (f: X → Y) : A₁ ⊆ A₂ → f '' A₁ ⊆ f '' A₂ :=
    assume h : A₁ ⊆ A₂,
    assume y,
    assume h₁ : y ∈ f '' A₁,
    have h₂ : ∃ x, x ∈ A₁ ∧ f(x) = y, from h₁,
    show y ∈ f '' A₂, from exists.elim h₂
        (assume (x' : X) (ha: x' ∈ A₁ ∧ f(x') = y ),
        have h₃ : x' ∈ A₂ ∧ f(x') = y, from and.intro (h ha.left)
      ha.right,
        show y ∈ f '' A₂, from exists.intro x' h₃)
```

Proposição 7.4.9 (Item 9)**.**

Observe que para o item 9, trataremos X e Y como conjuntos iguais. Caso não sejam, $f[B]$ pode nem estar definida.

```
import data.set
open set function

variables {X Y : Type}
variables (A B A₁ A₂ : set X)
```

```
6
7  theorem item9 (f: X → X) : f '' A \ f '' B ⊆ f '' (A \ B) :=
8  begin
9      intros y h,
10     have h₁ : y ∈ f '' A, from mem_of_mem_diff h,
11     have h₂ : ¬ (y ∈ f '' B), from not_mem_of_mem_diff h,
12     apply exists.elim h₁,
13     intros x h₃,
14     apply exists.intro x,
15     apply and.intro,
16     apply mem_diff_of_mem,
17         exact h₃.left,
18         assume h₄ : x ∈ B,
19         exact false.elim (h₂ (exists.intro x (and.intro h₄ h₃.
       right))),
20         exact h₃.right
21 end
22
23 #check @mem_of_mem_diff
24 #check @not_mem_of_mem_diff
25 #check @mem_diff_of_mem
```

As funções `mem_of_mem_diff`, `not_mem_of_mem_diff` e `mem_diff_of_mem` tem o objetivo de lidar com a diferença e sua relação com a interseção. Note que usar táticas auxilia o passo a passo, porém dificulta o posterior entendimento. Por isso, é recomendável, nesses exemplos, utilizar alguma ferramenta para Lean. Essas funções e outras já apresentadas, encontram-se na biblioteca do Lean e grande parte delas está disponível quando você abre o namespace `function`.

```
1  open function
2
3  #check @comp
4  #check @has_left_inverse
5  -- A right_inverse tem duas definicoes para Lean. Apenas uma
6  -- esta no namespace function. Por isso e importante especificar.
7  #check @function.right_inverse
```

7.4.3 Definindo a inversa Classicamente

Para definir funções inversas, é necessário que utilizemos o racionínio clássico.

```
1  open classical
2
3  #check @axiom_of_choice
4  #check @some_spec
5
```

```
variables A B : Type
variable P : A → Prop
variable R : A → B → Prop

example : (∀ x, ∃ y, R x y) → ∃ f : A → B, ∀ x, R x (f x) :=
axiom_of_choice

example (h : ∃ x, P x) : P (some h) :=
some_spec h
```

O axioma da escolha fala que se para todo `x : X`, existe `y: Y` com `R x y`, então existe uma função `f: X → Y` que para todo `x`, escolhe-se `y`. Em Lean, é utilizada a função `some` para mostrar este axioma, através da construção clássica. Podemos, portanto, definir a inversa como:

```
open classical function
local attribute [instance] prop_decidable

variables {X Y : Type}

noncomputable def inverse (f : X → Y) (default : X) : Y → X :=
λ y, if h : ∃ x, f x = y then some h else default
```

Observe que a definição é não computacional, visto que como já argumentamos, para uma determinada função, essa hipótese `h` pode não ser possível de validar algoritmicamente e, se a hipótese for válida, pode não ser possível encontrar um valor de `x` adequado, também algoritmicamente. Também observe que essa inversa é definida assumindo `y`, logo ela é definida para todo valor que ele assume e retorna algum valor qye cumpre essa propriedade `f x = y`, quando a hipótese é válida, ou um valor padrão dado inicialmente, caso não exista. O comando `local attribute` fala para a instância que a regra `prop_decidable` vai ser utilizada no arquivo que se segue. Nesse caso, importamos os axiomas clássicos e tornamos disponível a instância genérica da decição.

Podemos, então, demonstrar os seguintes teoremas.

```
open classical function
local attribute [instance] prop_decidable

variables {X Y : Type}

noncomputable def inverse (f : X → Y) (default : X) : Y → X :=
λ y, if h : ∃ x, f x = y then some h else default

theorem inverse_of_exists (f : X → Y) (default : X) (y : Y)
    (h : ∃ x, f x = y) : f (inverse f default y) = y :=
    have h₁ : inverse f default y = some h, from dif_pos h,
    have h₂ : f (some h) = y, from some_spec h,
eq.subst (eq.symm h₁) h₂
```

```
14
15 #check @dite
16 #check @dif_pos
17
18 -- Using Term Mode
19 theorem is_left_inverse_of_injective (f : X → Y) (default : X)
20     (injf : injective f) : left_inverse (inverse f default) f :=
21 let finv := (inverse f default) in
22     assume x,
23     have h1 : ∃ x', f x' = f x, from exists.intro x rfl,
24     have h2 : f (finv (f x)) = f x,
25         from inverse_of_exists f default (f x) h1,
26     show finv (f x) = x, from injf h2
27
28 -- Using Tatic Mode
29 theorem is_left_inverse_of_injective2 (f : X → Y) (default : X)
30     (injf : injective f) : left_inverse (inverse f default) f :=
31 begin
32     intro x,
33     apply injf,
34     apply inverse_of_exists,
35     apply exists.intro x,
36     exact eq.refl (f x),
37 end
```

Observado o significado de `dite`, que expressa a validade de um α `: Sort`, independente de `c: Prop` e o significado de `dif_pos`, que expressa a igualdade entre uma expressão do formato `dite` e outra do mesmo `Sort`, podemos entender o siginificado dessa prova. A versão em Táticas foi inserida como instrução de aplicação em comparação como o modo em termos.

7.5 Lógica de Segunda ou Mais Alta Ordem

Até agora, formalmente, definimos lógica de primeira ordem, onde iniciamos com um estoque fixo de símbolos de funções e relações. Nos últimos tópicos que consideramos, existe a motivação de extender a linguagem para funções e relações. Por exemplo, ao afirmar que $f : X \to Y$ possui inversa à esquerda pode ser dito como:

$$\exists g, \forall x, g(f(x)) = x.$$

Outro exemplo é o seguinte teorema, descrito em linguagem lógica

$$\forall x_1, x_2, (f(x_1) = f(x_2) \to x_1 = x_2) \to \exists g, \forall x, g(f(x)) = x.$$

Isto é, se $f : X \to Y$ é injetiva, então existe inversa à sua esquerda.

Nesse sentido, existe uma quantificação sobre as funções e relações, o que se distancia do que a lógica de primeira ordem cobre. Como forma de resolver esse

impasse, podemos desenvolver uma teoria na linguagem de lógica de primeira ordem no qual o universo contém funções e relações como objetos ou extender essa linguagem para envolver novos tipos de quantificadores e variáveis, que é o caso descrito nessa sessão. Isto é o que Lógica de Ordem mais Alta faz.

Alonzo Church (1903 - 1995) foi um matemático e lógico estadunidense, conhecido pelo cálculo lambda, e foi o responsável por formular a ideia de ordens mais altas na lógica. Isso é algumas vezes descrito como Teoria Simples dos Tipos ou Teoria dos Tipos de Chunch's. Sejam $X, Y, Z, ...$ tipos básicos e $Prop$ um tipo especial das proposições. Temos, então:

- Se X e Y são tipos, então $X \times Y$ também será.Isto denota que o par ordenado (x, y), com $x \in X$ e $y \in Y$, é tipado.
- Se X e Y são tipos, então $X \to Y$ também será.

Para formar expressões, a teoria de Church aficiona os seguintes caminhos:

1. Se x é do tipo X e y é do tipo Y, (x, y) é do tipo $X \times Y$.
2. Se z é do tipo $X \times Y$, então $(p)_1$ é do tipo X e $(p)_2$ é do tipo Y, que denotam o primeiro e segundo elemento do par z, respectivamente.
3. Se x é uma variável do tipo X e y é uma expressão do tipo Y, então λxy é do tipo $X \to Y$. Para lembrar o significado dessa expressão, voltar em 7.1.
4. Se f é do tipo $X \to Y$ e x é do tipo X, $f(x)$ é do tipo Y.

Além desses meios, a teoria simples dos tipos possui conetivos booleanos, quantificadores e a igualdade, oriunda da lógica de primeira ordem, para construir proposições.

A função $f(x, y)$ que toma elementos de X e Y para o tipo Z é vista como um objeto do tipo $X \times Y \to Z$. Note que, nesse sentido, uma relação binária $R(x, y)$ definida em X e Y é vista como um objeto $X \times Y \to Prop$. O que torma essa lógica ser de ordem mais alta é que podemos iterar o tipo função indefinidamente. Por exemplo, se $\mathbb{N}$ é o tipo dos números naturais, $\mathbb{N} \to \mathbb{N}$ denota o tipo das funções dos números naturais aos números naturais, e $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ denota o tipo de funções $F(f)$ que tomam uma função como argumento e retornam um número natural. Vamos ver em Lean:

```
1 open nat
2
3 #check ℕ                -- ℕ : Type
4 #check ℕ → ℕ             -- ℕ → ℕ : Type
5 #check (ℕ → ℕ) → ℕ       -- (ℕ → ℕ) → ℕ : Type
6
7 variable F: (ℕ → ℕ) → ℕ
8 variable f: ℕ → ℕ
9 variable n: ℕ
```

```
10
11 #check f n              -- f n : ℕ
12 #check F f              -- F f : ℕ
13 #check F n              -- error
```

A variavél F espera uma função, por isso retorna erro no último `#check`. Se retirarmos o parênteses, Lean entenderá que `F : ℕ → (ℕ → ℕ)`.

As especificações de síntaxe e regras da lógica de ordem mais alta são descritas mais cuidadosamente em livros-texto mais avançados. Para considerar os tipos de funções e relações, basta o conhecimento de lógica de segunda ordem. É importante ter em mente que já lidamos desde os primeiros capítulos com os tipos e lógica de segunda ordem, em Lean, que é um sistema de lógica ainda mais elaborado e expressivo.

7.6 Funções e Relações

Podemos ver que nem toda relação é uma função. Uma relação binária $R(x, y)$ em A e B é um *funcional* se para todo x em A, existe um valor único y em B tal que $R(x, y)$. Se R é uma relação funcional, podemos definir $f_R : X \to B$ fazendo com que $f_R(x)$ seja igual ao valor de y único em B em que a relação seja verdadeira. A contrapartida é verdadeira, basta definir, para $f : X \to B$, $R_f(x, y)$, uma relação funcional. Essa relação é conhecida como o *grafo de f*.

Podemos extrair dessa ideia que existe uma dualidade entre uma relação funcional R e uma função f. Inclusive, muitas vezes, as definições são intercambiáveis. Em fundamentos tipo-teóricos, como a que vemos em Lean, relações são frequentemente definidas como uma classe de funções, enquanto para fundamentos teóricos em conjuntos definem função como uma relação funcional.

Podemos considerar funções $f(x, y)$ ou $g(x, y, z)$ que tomam múltiplos argumentos. Por exemplo, $f(x, y) = x + y$. Em lean, podemos exemplificar da seguinte forma:

```
1  variables X Y : Type
2  variable f: X → X → X → Y
3  variables x₁ x₂ x₃  : X
4
5  #check f x₁ x₂ x₃ -- tipo Y
6
7  def g (x y : ℤ): ℤ := x + y
8
9  example : g 1 1 = 2 :=
10 calc
11 g 1 1 = 1 + 1 : by rw g
12     ... = 2 :  rfl
```

O número de argumentos de uma função é chamado de *aridade*, que em inglês é *arity*. No exemplo acima f tem aridade 3, enquanto g tem aridade 2. Podemos pensar uma função com múltiplos argumentos como uma função

unária definida em um produto cartesiano. Nesse sentido, quando estamos nos tratando de teoria dos tipos, é conveniente pensar que f é uma função de X que retorna uma função $X \to X \to Y$. Desta maneira, $f(x_1, x_2, x_3)$ abrevia $((f(x_1))(x_2))(x_3)$.

Exemplo 7.6.1 (Teorema de Cantor)**.** Seja f uma função que mapeia elementos do conjunto A para o seu conjunto das partes $P(A)$. Então $f : A \to P(A)$ não é sobrejetiva. Esse teorema é um teorema fundamental sobre conjuntos que foi apresentada nesse capítulo devido às definições importantes de funções. A referência é no capítulo sobre Conjuntos. A demonstração se dá por contradição e é realizada no Lean com o auxílio do `namespace classical`.

```
1   import data.set
2
3   variables X Y: Type
4
5   def surjective {X: Type} {Y: Type} (f : X → Y) : Prop := ∀ y
    , ∃ x, f x = y
6
7   theorem Cantor : ∀ (A: set X), ¬ ∃ (f: A → set A),
    surjective f :=
8
9       begin
10          intro A,
11          intro h,
12          -- Utiliza a regra çãeliminao existe.
13          cases h with f h1,
14          -- Como f é sobrejetiva e esse é um subconjunto de A
15          have h2: ∃ x, f x = {t : A | ¬ (t ∈ f t)},
16              from h1 {t : A | ¬ (t ∈ f t)},
17          -- çãEliminao do existe, novamente.
18          cases h2 with x h3,
19          -- classical.em A := A ∪ ¬A é uma tautologia.
20          apply or.elim (classical.em
21                      (x ∈ {t : A | ¬ (t ∈ f t)})),
22              intro h4,
23          -- Usa a proriedade do conjunto
24                  have h5: ¬ (x ∈ f x), from h4,
25                  rw (eq.symm h3) at h4,
26                  apply h5 h4,
27              intro h4,
28                  rw (eq.symm h3) at h4,
29                  have h5: x ∈ {t : A | ¬ (t ∈ f t)}, from h4,
30                  rw (eq.symm h3) at h5,
31                  apply h4 h5,
32      end
33
```

Note que uma relação binária R de X em Y é um funcional se satisfaz a seguinte proposição: $\forall x, \exists! y R(x, y)$ Um lógico poderia usar a notação *iota*, descrita como $f(x) = \iota y R(x, y)$. Se y não ser necessariamente único, um lógico utiliza a notação de *Helbert epsilon* $f(x) = \epsilon y R(x, y)$ para escolher um valor de y que satisfaça $R(x, y)$.

7.6.1 Representação de Funções

Apesar de não ser o objetivo nesse capítulo, considere a seguinte definição:

Definição 7.6.1. A grosso modo, uma função total $f(x_1, ..., x_k)$ é dita *representável*, se existe uma expressão na linguagem formal, digamos A, que usa variáveis $x_1, ..., x_k, x_{k+1}$ tal que para quaisquer números $m_1, ..., m_k$ e n, temos $f(m_1, ..., m_k) = n$ se, e somente se, podemos provar em nosso sistema formal $A(m_1, ..., m_k, n)$ e não podemos provar $A(m_1, ..., m_k, j)$ para qualquer outro número j.

Essa definição é encontrada no livro [1]. Apesar de estar em termos um pouco diferentes dos que propomos nesse livro, essa definição permite, em um sistema consistente, afirmar que toda função representável é computável. Isso responde parcialmente nossa questão da primeira sessão. Entretanto, em livros mais avançados de lógica, esse conceito é mais discutido.

7.7 Exercícios

1. Quais das seguintes funções são totais:
   (a) $f : \mathbb{N} \to \mathbb{N}$, definida por $f(n) = n - 1$.
   (b) $g : \mathbb{R} \setminus \{1\} \to \mathbb{R}$, definida por $g(x) = \frac{x}{x-1}$.
   (c) A função de Heaviside. Referência.
   (d) $h : \mathbb{R}^+ \to \mathbb{R}$, definida por $h(x) = log(x)$.
2. Seja $f : X \to Y$ e $g : Y \to Z$.
   (a) Mostre que se $g \circ f$ é uma função injetiva, então f é injetiva.
   (b) Mostre um exemplo onde a condição do item anterior ocorra, porém g não seja injetiva.
   (c) Prove que se f é sobrejetiva e $g \circ f$ é injetiva, então g é injetiva.
   (d) Demonstre (a) e (c) utilizando Lean.
3. Uma função $f : X \to Y$ é dita *monótona* se preserva ou inverte a relação de ordem. Se a função é estrita, a relação estabelecida é a de $<$. Isto é, se $\forall x_1, \forall x_2, x_1 < x_2 \Rightarrow f(x_1) < f(x_2)$ mostra que f é estritamente crescente e com o sinal invertido, f será estritamente decrescente. Prove que toda função monótona é injetiva. Defina em Lean e prove esse teorema, considerando $X = Y = \mathbb{N}$.

4. Prove que a função $g : \mathbb{N} \to \mathbb{N}$ definida por $g(n) = \lceil \frac{n}{2} \rceil$ é sobrejetiva. Essa função é injetiva?

5. Prove os item 5 da lista 7.4. Utilize Lean.

7.8 Gabarito

1. b e c. No primeiro caso, a função não é definida para $f(0)$, pois esse não tem antecessor. No quarto caso, a função não é definida para $h(0)$.

2. Seguem as provas:

   (a) Sejam $x_1 \in X$ e $x_2 \in X$, com $f(x_1) = f(x_2)$. Como g é uma função bem definida, $g(f(x_1)) = (g \circ f)(x_1) = (g \circ f)(x_2) = g(f(x_2))$. Pela injetividade da composição, $x_1 = x_2$ está provado.

   (b) Para essa questão, existem vários exemplos. Seja $f : \mathbb{N} \to \mathbb{R}$ definida por $f(n) = n$ e seja $g : \mathbb{R} \to \mathbb{R}$, $g(x) = x^2$. Temos que $(g \circ f) : \mathbb{N} \to \mathbb{R}$ é definida por $g(f(n)) = n^2$. Note que essa função é injetiva, porém g não é.

   (c) Assuma $y_1, y_2 \in Y$. Se $g(y_1) = g(y_2)$, pela sobrejetividade de f, existem $x_1, x_2 \in X$, tal que $f(x_1) = y_1$ e $f(x_2) = y_2$. Pela injetividade da composição, $x_1 = x_2$. Como f é uma função bem definida, $y_1 = f(x_1) = f(x_2) = y_2$, o que demostra a injetividade de g.

   (d) Em lean,

```
1  import data.set
2  open function
3  open set
4
5  variables {X Y Z: Type}
6
7  theorem e2a (f : X → Y) (g: Y → Z) (h: injective (g ∘ f
       )) : injective f :=
8  assume x₁ x₂ h₁,
9  have h₂ : g(f(x₁)) = g(f(x₂)), from eq.subst h₁ (eq.refl
       (g(f x₁))),
10 show x₁ = x₂, from h h₂
11
12 theorem e2c (f : X → Y) (g: Y → Z) (h₁: injective (g ∘
       f)) (h₂: surjective f) :
13         injective g :=
14 assume y₁ y₂ g₁,
15 have g₂ : ∃ x, f(x) = y₁, from h₂ y₁,
16 have g₃ : ∃ x, f(x) = y₂, from h₂ y₂,
17 show y₁ = y₂, from exists.elim g₂
18     (assume x₁ i₁,
```

```
19      show y₁ = y₂, from exists.elim g₃
20          (assume x₂ i₂,
21          have i₃ : g(f(x₁)) = g(y₁), from eq.subst i₁ (eq.
        refl (g(f x₁))),
22          have i₄ : g(f(x₂)) = g(y₂), from eq.subst i₂ (eq.
        refl (g(f x₂))),
23          have i₅ : x₁ = x₂, from h₁ (eq.trans i₃ (eq.trans
         g₁ (eq.symm i₄))),
24          have i₆ : f(x₁) = f(x₂), from eq.subst i₅ (eq.
        refl (f(x₁))),
25          show y₁ = y₂, from eq.trans (eq.trans (eq.symm i₁
        ) i₆) i₂))
```

3. Seja $f : X \to Y$ monótona. Assuma $x_1, x_2 \in X$ e $f(x_1) = f(x_2)$. Se $x_1 < x_2$ ou $x_1 > x_2$, temos um absurdo. Logo $x_1 = x_2$, o que mostra que f é monótona.

   Para lean, eu faço uso de uma função `ne_of_lt`, que significa que dado uma relação de menor ou maior entre dois naturais, eles são diferentes.

```
1  open function
2
3  variables {X Y Z: Type}
4
5  def strict_cresc (f: ℕ → ℕ) :=
6      ∀ x1, ∀ x2, x1 < x2 → f x1 < f x2
7
8  def strict_decresc (f: ℕ → ℕ) :=
9      ∀ x1, ∀ x2, x1 < x2 → f x1 > f x2
10
11 def monotonous (f: ℕ → ℕ) : Prop :=
12     strict_cresc f ∨ strict_decresc f
13
14 lemma inj_of_cresc (f: ℕ → ℕ) (h: strict_cresc f) :
       injective f :=
15 assume x1 x2 h1,
16 nat.lt_by_cases
17     (assume : x1 < x2,
18     have h2: f x1 < f x2, from h x1 x2 this,
19     show x1 = x2, from false.elim (ne_of_lt h2 h1))
20     (assume : x1 = x2, this)
21     (assume : x1 > x2,
22     have h2: f x2 < f x1, from h x2 x1 this,
23     show x1 = x2, from false.elim (ne_of_lt h2 (eq.symm h1))
       )
24
```

```
25 lemma inj_of_decresc (f: ℕ → ℕ) (h: strict_decresc f) :
       injective f :=
26 assume x1 x2 h1,
27 nat.lt_by_cases
28     (assume : x1 < x2,
29     have h2: f x1 > f x2, from h x1 x2 this,
30     show x1 = x2, from false.elim (ne_of_lt h2 (eq.symm h1))
       )
31     (assume : x1 = x2, this)
32     (assume : x1 > x2,
33     have h2: f x2 > f x1, from h x2 x1 this,
34     show x1 = x2, from false.elim (ne_of_lt h2 h1))
35
36 theorem inf_mon (f: ℕ → ℕ)(h: monotonous f):injective f :=
37 begin
38     cases h,
39     apply inj_of_cresc f h,
40     apply inj_of_decresc f h
41 end
42
43 #check @ne_of_lt
```

4. Suponha $y \in \mathbb{N}$. Note que posso escrever $2 \cdot y = n$. De fato, $g(n) = \lceil \frac{2y}{2} \rceil = \lceil y \rceil = y$, pois y é natural. Assim $\exists n \in \mathbb{N}, g(n) = y$. Assim, a função é sobrejetiva.

5. Considere o item 5 Suponha $y \in f[A_1 \cup A_2]$ Assim, existe $x \in A_1 \cup A_2$, tal que $f(x) = y$. Se $x \in A_1$, $f(x) = y \in f[A_1]$ e análogo se $x \in A_2$. Logo $y \in f[A_1] \cup f[A_2]$. Alternativamente, se $y \in f[A_1] \cup f[A_2]$. Suponha $y \in f[A_1]$. Então existe $x \in A_1$, tal que $f(x) = y$ e $x \in A_1 \cup A_2$ que implica $y \in f[A_1 \cup A_2]$. Análogo para $y \in f[A_2]$.

```
1  import data.set
2  open function set
3
4  variables {X Y : Type}
5  variable  f : X → Y
6  variables A1 A2 : set X
7
8  example : f '' (A1 ∪ A2) = f '' A1 ∪ f '' A2 :=
9  eq_of_subset_of_subsetimport data.set
10 open function set
11
12 variables {X Y : Type}
13 variable  f : X → Y
14 variables A1 A2 : set X
```

```
15
16 example : f '' (A1 ∪ A2) = f '' A1 ∪ f '' A2 :=
17 eq_of_subset_of_subset
18     (assume y,
19     assume h1 : y ∈ f '' (A1 ∪ A2),
20     exists.elim h1
21     (assume x h,
22     have h2 : x ∈ A1 ∪ A2, from h.left,
23     have h3 : f x = y, from h.right,
24     or.elim h2
25         (assume h4 : x ∈ A1,
26         have h5 : y ∈ f '' A1, from ⟨x, h4, h3⟩,
27         show y ∈ f '' A1 ∪ f '' A2, from or.inl h5)
28         (assume h4 : x ∈ A2,
29         have h5 : y ∈ f ''  A2, from ⟨x, h4, h3⟩,
30         show y ∈ f '' A1 ∪ f '' A2, from or.inr h5)))
31     (assume y,
32     assume h2 : y ∈ f '' A1 ∪ f '' A2,
33     or.elim h2
34         (assume h3 : y ∈ f '' A1,
35         exists.elim h3
36         (assume x h,
37         have h4 : x ∈ A1, from h.left,
38         have h5 : f x = y, from h.right,
39         have h6 : x ∈ A1 ∪ A2, from or.inl h4,
40         show y ∈ f '' (A1 ∪ A2), from ⟨x, h6, h5⟩))
41         (assume h3 : y ∈ f '' A2,
42         exists.elim h3
43         (assume x h,
44         have h4 : x ∈ A2, from h.left,
45         have h5 : f x = y, from h.right,
46         have h6 : x ∈ A1 ∪ A2, from or.inr h4,
47         show y ∈ f '' (A1 ∪ A2), from ⟨x, h6, h5⟩)))
```

Capítulo 8

Indução nos Números Naturais

Muitos matemáticos como Platão e Bhaskara usavam implicitamente provas por indução em seus diálogos e documentos. O uso de indução mais antigo conhecido foi na prova de Euclides de que os números primos são infinitos. A primeira pessoa a formular explicitamente o princípio de indução foi Pascal em seu livro *Traité du Triangle Arithmétique* (1665). No século XIX, finalmente foi feito um tratamento rigoroso e sistemático pela contribuição de vários nomes importantes como George Boole, Augustus de Morgan, Giuseppe Peano, entre outros.

Enquanto todos os tipos vistos até agora são vitais, seus valores não são ideais para representações mais complexas de objetos. Para compor construções matemáticas ricas podemos usar a indução. Nesse capítulo, veremos como definir esses novos tipos de dados, com ênfase nos naturais. Em particular, queremos provar que todos os objetos de um certo tipo tenham uma certa propriedade, ou como são chamados, princípios indutivos. Todo tipo vem automaticamente acompanhado de uma regra indutiva.

8.1 Construção de Tipos por Indução

Para definir um novo tipo e seus valores em Lean, usamos uma definição indutiva. Essa definição recebe o nome do novo tipo e um conjunto de construtores que introduzem seus valores. Quando o construtor não recebe argumentos, é dito um tipo enumerado, isto é, todos seus elementos são listados. Por exemplo, o tipo interno `bool` é enumerado e possui dois valores, `bool.tt` e `bool.ff`. A seguir temos a construção de um novo tipo chamado período que representa os períodos de um dia:

```
1 inductive periodo : Type
2 | dia : periodo
```

```
3 | tarde : periodo
4 | noite : periodo
```

Com isso, o nosso tipo criado possui três valores possíveis e qualquer variável desse tipo assume um desses três valores distintos. Logo podemos definir funções que recebem e retornam valores desse tipo. Vamos definir a função próximo período, abreviada por `proxPer`. Funções no Lean devem ser totais, isto é, devem cobrir todas suas possíveis aplicações.

O jeito que nossa função trabalha pode ser feito por *case analysis* (análise de casos) e *pattern matching* (casamento de padrão). Da primeiro maneira especificamos o retorno da função para uma ou mais entradas específicas, enquanto no segundo jeito definimos uma recursão usando funções definidas indutivamente no construtor (o nosso tipo periodo não contém funções no construtor, veremos mais a frente que os naturais contém a função sucessor).

Segue a função `proxPer` em Lean:

```
1 open periodo
2
3 def proxPer : periodo → periodo
4 | dia := tarde
5 | tarde := noite
6 | noite := dia
7
8 #reduce proxPer dia
```

De maneira semelhante, podemos definir uma função que retorna se um periodo têm sol ou não:

```
1 def temSol : periodo → bool
2 | dia := tt
3 | tarde := tt
4 | _ := ff
5
6 #reduce temSol dia
```

O símbolo _ (underline) indica que para qualquer entrada diferente de dia e tarde, retorne `ff` (falso). Também podemos provar teoremas sobre o nosso tipo como, por exemplo, a função proxPer aplicada três vezes à qualquer periodo sempre é o próprio período. Traduzindo para FOL temos que `∀ x : periodo, proxPer (proxPer (proxPer x) ) = x`.

Para todo tipo definido indutivamente temos um princípio indutivo associado. Esse princípio é a regra de introdução para as proposições *para todo*. Usamos regras indutivas para construir provas do tipo *forall t : T, P t* onde *P* é um predicado.

Abaixo temos a prova usando táticas:

```
1 example : ∀ x : periodo, proxPer (proxPer (proxPer x) ) = x :=
2 begin
3     intro x,
```

```
4     induction x,
5     repeat { exact rfl }
6 end
```

A tática `induction` separa o nosso objetivo dado o construtor do tipo da variável passada, que no nosso caso é `x`. Como ela é de um tipo enumerado, o princípio indutivo é simplesmente provar para cada possível valor, logo nosso objetivo é dividido em três objetivos diferentes: uma prova para cada valor possível.

O operador `rfl` funciona da seguinte maneira: ele tenta simplificar ambos os lados da igualdade seguindo definições e propriedades. Caso ele consiga chegar na igualdade desejada apenas simplificando, ele considera como trivial o objetivo.

8.2 Os Naturais

Lidaremos com os naturais por simplicidade e também devido à maioria dos conjuntos poderem ser construídos a partir dos naturais com apenas algumas modificações. Muitos autores discutem acerca da inclusão do número 0. Neste livro, consideraremos o 0 como um número natural. O conjunto dos números naturais é o conjunto:

$$\mathbb{N} = \{0, 1, 2, 3, ...\}$$

Ele é construído pela constante 0 e a função $s(n) = n + 1$ chamada função sucessora. Com isso, qualquer número natural pode ser construído aplicando a função sucessora um número finito de vezes à constante 0. De um ponto de vista mais amplo, parece que estamos usando uma definição na própria definição, pois aplicar uma função um número finito de vezes é usar os números naturais.

O princípio da indução, que será explicado na seção 8.3, descreverá essa propriedade dos naturais de uma maneira não circular. Mas por enquanto, nos atentaremos a como o Lean lida com o nosso tipo *natural*.

No Lean, os naturais são definidos internamente como um tipo indutivo:

```
1 namespace hidden
2
3 inductive nat : Type
4 | zero : nat
5 | succ : nat → nat
6
7 end hidden
```

Como não queremos que o nosso novo tipo `nat` entre em conflito com o `nat` do Lean, usa-se o `namespace`. Com ele, `nat` é reconhecido como `hidden.nat` e suas respectivas propriedades também, `hidden.nat.zero` e `hidden.nat.succ`. O símbolo unicode $\mathbb{N}$ pode ser escrito com $\backslash N$ ou $\backslash nat$, e equivale à mesma coisa que `nat`.

Repare que o construtor do `nat` possue duas definições: a constante zero e uma função sucessor, diferente do tipo periodo visto na seção anterior.

Apesar de termos definido o tipo `nat` anteriormente, usaremos o da biblioteca do Lean chamada `nat`. Como este tipo é definido indutivamente, podemos criar funções recursivas. Por exemplo:

```
1  open nat
2
3  def five_mult : ℕ → ℕ
4  | 0        := 0
5  | (succ n) := 5 + five_mult n
6
7  def fact : ℕ → ℕ
8  | 0        := 1
9  | (succ n) := (succ n) * fact n
10
11 def fibonacci : ℕ → ℕ
12 | 0        := 0
13 | 1        := 1
14 | (n + 2)  := fibonacci (n + 1) + fibonacci n
```

No trecho de código acima, definimos três funções de maneira recursiva: $5 * n$, $n!$ e $fib(n)$ (a dedução de uma fórmula explícita para essas funções será melhor explicada na seção 8.5). As funções necessitam saber o que fazer com todas entradas possíveis, que no nosso caso seriam a constante 0 e um outro natural qualquer, isto é, o sucessor de alguém. Quando colocado `(succ n)`, o Lean interpreta como se a entrada fosse `(succ n)`, isto é, n torna-se o valor tal que `(succ n)` é igual à entrada. Com isso, podemos atribuir o valor desejado recursivamente à nossa função. Para $n + 2$ o raciocínio é idêntico: n torna-se o valor tal que $n + 2$ é igual à entrada.

No Lean, $n + 1$ é reconhecido da mesma maneira que `(succ n)`. Todas definições acima funcionariam da mesma maneira caso este fosse substituído.

8.3 O Príncipio de Indução

P(0) e P(n) -¿ P(n+1) e alguns exemplos . . .

8.4 Variantes da Indução

Indução começando em um natural k e indução completa . . .

8.5 Recursão

Na seção 8.2 definimos três exemplos de funções recursivas, a multiplicação por 5, o fatorial e a Fibonacci. Por exemplo, na função `five_mult` definimos:

$$f(0) = 0$$

$$f(n+1) = 5 + f(n)$$

Sabemos que esta função também pode escrita como $f(n) = 5 \cdot n$, no entanto, as duas condições estabelecidas no modo recursivo definem o comportamento de f para todo o conjunto $\mathbb{N}$.

Definição: Princípio de definição por recursão Seja A um conjunto, e $\alpha \in A$, e $g : \mathbb{N} \times A \to A$. Então existe uma unica função f que satisfaça:

$$f(0) = \alpha$$

$$f(n+1) = g(n, f(n))$$

Exemplo Indo no caminho contrário, temos $f : \mathbb{N} \to \mathbb{N}$, $f(n) = 2^n$, vamos tentar expressar a função de forma recursiva. Nosso conjunto é o conjunto dos naturais, e temos $f(0) = 1$, além disso, como podemos expressar $2^{n+1} = 2 \cdot 2^n$, temos que $f(n+1) = g(n, f(n)) = 2 \cdot f(n)$.

O princípio da definição por recursão possui forte relação com o princípio da indução, a partir do primeiro conseguimos obter o segundo e a partir do segundo também conseguimos obter o primeiro. Neste capítulo iremos utilizar o princípio da indução para provar igualdades de funções recursivas.

Proposição Tanto a notação do somatório de uma função de $\mathbb{N}$, quanto a notação do produtório podem ser descritas através de definições recursivas. Ou seja, seja $f : \mathbb{N} \to \mathbb{N}$, $s(n) = \sum_{i=0}^{n} f(i)$, é:

$$s(0) = 0$$

$$s(n+1) = s(n) + f(n)$$

e $p(n) = \prod_{i=0}^{n} f(i)$, é:

$$p(0) = 1$$

$$p(n+1) = p(n) \cdot f(n)$$

Da mesma forma que o princípio da indução completo, o princípio da definição por recursão também pode ser extendido.

Proposição Seja A um conjunto, e $\alpha_0, \alpha_1, \ldots, \alpha_m \in A$, e $g : \mathbb{N}^m \times A \to A$. Então existe uma única função f que satisfaça:

$$f(0) = \alpha_0$$
$$f(1) = \alpha_1$$
$$\dots$$
$$f(m) = \alpha_m$$
$$f(n+1) = g(n, f(n), f(n-1), \dots, f(n-m))$$

Exemplo A ideia de extensão é que podemos definir de forma não recursiva o comportamento da função até um número m e a partir disso a recursão em n pode depender dos m valores anteriores da função. Como já apresentado na seção 8.2, os números de Fibonacci podem ser definidos através de recursão. Temos que $\alpha_0 = 0$ e $\alpha_1 = 1$, e $f(n+1) = g(n, f(n-1), f(n-2)) = f(n-1) + f(n-2)$. Neste exemplo, nosso m é 2.

Exemplo Para ilustrar a relação entre o princípio de indução e o princípio da definição por recursão vamos provar a fórmula explícita da sequência de Fibonacci utilizando de indução. Vamos provar que

$$F(n) = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right\}$$

i) Para o caso $n = 0$ temos que $F(0) = 0$ (definição recursiva) e

$$\frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0\right\} = \frac{1}{\sqrt{5}}(1-1) = 0$$

Para o caso $n = 1$ temos que $F(1) = 1$ (definição recursiva) e

$$\frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)\right\} = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}-1+\sqrt{5}}{2}\right)\right\} = 1$$

ii) Pela hipótese de indução, supomos que a propriedade vale para $F(p-2)$ e para $F(p-1)$, para $F(p)$ teremos:

$$F(p) = F(p-1) + F(p-2) =$$
$$= \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p-1} + \left(\frac{1-\sqrt{5}}{2}\right)^{p-1}\right\} + \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p-2} + \left(\frac{1-\sqrt{5}}{2}\right)^{p-2}\right\}$$
$$= \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p-2}\left(1 + \frac{1+\sqrt{5}}{2}\right) + \left(\frac{1-\sqrt{5}}{2}\right)^{p-2}\left(1 + \frac{1-\sqrt{5}}{2}\right)\right\}$$

$$= \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p-2}\left(\frac{3+\sqrt{5}}{2}\right) + \left(\frac{1-\sqrt{5}}{2}\right)^{p-2}\left(\frac{3-\sqrt{5}}{2}\right)\right\}$$

$$= \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p-2}\left(\frac{1+2\cdot\sqrt{5}+5}{4}\right) + \left(\frac{1-\sqrt{5}}{2}\right)^{p-2}\left(\frac{1-2\cdot\sqrt{5}+5}{4}\right)\right\}$$

$$= \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p-2}\left(\frac{1+\sqrt{5}}{2}\right)^{2} + \left(\frac{1-\sqrt{5}}{2}\right)^{p-2}\left(\frac{1-\sqrt{5}}{2}\right)^{2}\right\}$$

$$= \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^{p} + \left(\frac{1-\sqrt{5}}{2}\right)^{p}\right\}$$

Como queriamos demonstrar. Observe que o principal truque para a demonstração ocorreu na linha 5, em que multiplicamos a fração por 2 no numerador e denominador e separamos $\frac{6}{4}$ em $\frac{1+5}{4}$ e em seguida concluímos que é o formato de um quadrado perfeito.

8.6 Operadores Aritméticos

Operadores aritméticos em um contexto global . . .

8.6.1 Aritmética nos Naturais

Operadores aritméticos nos naturais . . .

Agora que temos a base dos axiomas para cada operador nos naturais, podemos definir alguns deles em Lean e realizar provas por indução. Para criar uma função de potência por exemplo, o seguinte código funciona muito bem e inclusive é a definição da biblioteca `nat`, que pode ser chamada por *nat.pow*.

```
1 def pow : ℕ → ℕ → ℕ
2 | m 0        := 1
3 | m (n + 1)  := pow m n * m
```

Repare que a recursão ocorre no segundo parâmetro. Com esta nova função, podemos provar alguns teoremas:

```
1 open nat
2
3 theorem pow_zero (n : ℕ) : pow n 0 = 1 := rfl
4 theorem pow_succ (m n : ℕ) : pow m (n+1) = pow m n * m := rfl
```

Da mesma maneira que o infixo + foi definido, também é definido o infixo "^". A fórmula `m^n` equivale a escrever `pow m n`. Repare que no teorema `pow_succ` poderiamos ter invertido `m^n * m` por `m * m^n` e usado a comutatividade da multiplicação.

Usando o conceito de indução nos naturais estabelecido nas seções anteriores, podemos realizar provas em Lean:

```
1  open nat
2
3  theorem pow_succ' (m n : ℕ) : ∀ m n : nat, m^(succ n) = m * m^n
       :=
4  assume m n : nat,
5  nat.rec_on n
6    (show m^(succ 0) = m * m^0, from calc
7      m^(succ 0) = m^0 * m : by rw pow_succ
8             ... = 1 * m   : by rw pow_zero
9             ... = m       : by rw one_mul
10            ... = m * 1   : by rw mul_one
11            ... = m * m^0 : by rw pow_zero)
12   (assume n,
13     assume ih : m^(succ n) = m * m^n,
14     show m^(succ (succ n)) = m * m^(succ n), from calc
15       m^(succ (succ n)) = m^(succ n) * m   : by rw pow_succ
16                     ... = (m * m^n) * m    : by rw ih
17                     ... = m * (m^n * m)    : by rw mul_assoc
18                     ... = m * m^(succ n)   : by rw pow_succ)
```

O comando `nat.rec_on` recebe como entrada três parâmetros: a variável que se deseja aplicar indução; uma prova de $P(0)$ e uma prova de `∀n : nat, P(n) → P(n+1)`, onde P seria o nosso objetivo. No exemplo acima, usando `calc` e algumas funções já definidas na biblioteca `nat` e no Lean, conseguimos provar sem usar a comutatividade da multiplicação. Repare que como usamos `open nat`, não há necessidade de escrever `nat.pow`, apenas `pow`.

Uma boa dica para provas usando `calc` é de introduzir o seu resultado desejado e ir trabalhando a equação atual de modo a chegar ao seu objetivo. Como todas as coisas no Lean, provas por `calc` também podem ser simplificadas. Segue abaixo o exemplo anterior com sintaxe reduzida:

```
1  open nat
2
3  theorem pow_succ' (m n : ℕ) : m^(succ n) = m * (m^n) :=
4  nat.rec_on n
5    (show m^(succ 0) = m * m^0,
6      by rw [pow_succ, pow_zero, mul_one, one_mul])
7    (assume n,
8      assume ih : m^(succ n) = m * m^n,
9      show m^(succ (succ n)) = m * m^(succ n),
10       by rw [pow_succ, ih, mul_assoc, mul_comm (m^n)])
```

Você também pode usar o `rw` fora do `calc`, lhe passando uma lista de argumentos, que seriam equivalentes a usá-lo separadamente em cada um desses argumentos. Entretanto, você não consegue controlar qual parte da equação você deseja alterar, algo que seria possível usando `calc`. Segue abaixo um exemplo da identidade `m^(n + k) = m^n * m^k`.

```
open nat

theorem pow_add (m n k : ℕ) : m^(n + k) = m^n * m^k :=
nat.rec_on k
  (show m^(n + 0) = m^n * m^0, from calc
    m^(n + 0) = m^n       : by rw add_zero
          ... = m^n * 1   : by rw mul_one
          ... = m^n * m^0 : by rw pow_zero)
  (assume k,
    assume ih : m^(n + k) = m^n * m^k,
    show m^(n + succ k) = m^n * m^(succ k), from calc
      m^(n + succ k) = m^(succ (n + k)) : by rw nat.add_succ
                 ... = m^(n + k) * m    : by rw pow_succ
                 ... = m^n * m^k * m    : by rw ih
                 ... = m^n * (m^k * m)  : by rw mul_assoc
                 ... = m^n * m^(succ k) : by rw pow_succ)
```

Dessa vez, nossa indução foi em k. Uma dica interessante presente no editor web do Lean é que você pode passar o mouse por cima do nome do teorema e ver o que ele afirma. Usando `calc` para estruturar o objetivo e `rewrite` para completar a justificativa de cada passo, sua prova torna-se muito mais legível e elegante.

Segue abaixo alguns axiomas da biblioteca `nat`, e algumas provas no Lean:

```
open nat

variables m n : ℕ

#check add_zero m
#check add_succ m n
#check @pred_zero
#check pred_succ m
#check mul_zero m
#check mul_succ m n

theorem succ_pred (n : ℕ) : n ≠ 0 → succ (pred n) = n :=
nat.rec_on n
  (assume H : 0 ≠ 0,
    show succ (pred 0) = 0, from absurd rfl H)
  (assume n,
    assume ih,
    assume H : succ n ≠ 0,
     show succ (pred (succ n)) = succ n,
      by rewrite pred_succ)

theorem zero_add' (n : nat) : 0 + n = n :=
nat.rec_on n
```

```
24   (show 0 + 0 = 0, from rfl)
25   (assume n,
26     assume ih : 0 + n = n,
27      show 0 + succ n = succ n, from
28       calc
29        0 + succ n = succ (0 + n) : rfl
30               ... = succ n       : by rw ih)
```

Como vimos no capítulo anterior, a relação $\leq$ nos naturais é uma ordem parcial e possui várias propriedades que estão definidas na biblioteca `nat`. Dentre elas, temos sua reflexividade, antissimetria e transitividade, que estão representadas respectivamente por `le_refl`, `le_antisymm` e `le_trans`. Grande parte dos teoremas em Lean sobre desigualdade começam com "le_"ou "lt_", logo caso você deseje explorar outros teoremas internos escreva essas iniciais e explore as opções fornecidas pelo autocompletamento (função disponível no editor web). Portanto podemos realizar provas por indução. Segue uma prova de $\forall$ `m n k : nat, n + k ≤ m + k → n ≤ m` em Lean.

```
1  open nat
2
3  example : ∀ m n k : nat, n + k ≤ m + k → n ≤ m :=
4  assume m n k,
5   nat.rec_on k
6    (assume h : n + 0 ≤ m + 0,
7      show n ≤ m, from
8       calc
9          n = n + 0 : by rw add_zero
10        ... ≤ m + 0 : h
11        ... = m     : by rw add_zero
12        ... ≤ m     : le_refl m)
13   (assume k,
14    assume ih : n + k ≤ m + k → n ≤ m,
15     assume h1 : n + succ k ≤ m + succ k,
16      have h2 : succ(n + k) ≤ succ(m + k), from
17       calc
18         succ(n + k) = n + succ(k) : by rw add_succ
19                 ... ≤ m + succ(k) : h1
20                 ... = succ(m + k) : by rw add_succ,
21      have h3 : pred(succ(n + k)) ≤ pred(succ(m + k)), from
      pred_le_pred h2,
22      have h4 : n + k ≤ m + k, from
23       calc
24          n + k = pred(succ(n + k)) : by rw pred_succ
25            ... ≤ pred(succ(m + k)) : h3
26            ... = m + k             : by rw pred_succ,
27    show n ≤ m, from ih h4)
```

Repare que o Lean preserva o operador que estabelece a maior restrição quando se está usando `calc` em desigualdades.

8.6.2 Aritmética nos Inteiros

Operadores aritméticos nos inteiros . . .

8.7 Exercícios

Alguns exercícios de indução . . .

Bibliografia

[1] Walter A. Carnielli e Richard L. Epstein. *Computabilidade, funções computáveis, lógica e os fundamentos da Matemática*. Editora unesp, São Paulo - SP, 2005.