Resolução de problemas via SMT e ITPs

Aula Prática — Introdução à Lógica Computacional

Pedro Saccomani

Universidade Federal de Minas Gerais

22 de abril de 2026

Objetivos da aula

- Como podemos resolver problemas vistos em aula utilizando ferramentas computacionais?

Objetivos da aula

- Como podemos resolver problemas vistos em aula utilizando ferramentas computacionais?
  - **Solucionadores SMT** (`cvc5`): Como decidir se uma fórmula com predicados sobre certas teorias é satisfatível?

Objetivos da aula

- Como podemos resolver problemas vistos em aula utilizando ferramentas computacionais?
  - **Solucionadores SMT** (`cvc5`): Como decidir se uma fórmula com predicados sobre certas teorias é satisfatível?

    Mais ainda: Como decidir se uma fórmula é **válida** ou encontrar um **contra-exemplo**.

Objetivos da aula

- Como podemos resolver problemas vistos em aula utilizando ferramentas computacionais?
  - **Solucionadores SMT** (`cvc5`): Como decidir se uma fórmula com predicados sobre certas teorias é satisfatível?

    Mais ainda: Como decidir se uma fórmula é **válida** ou encontrar um **contra-exemplo**.
  - **Assistentes de demonstração** (`Lean 4`): Como podemos escrever demonstrações garantindo sua corretude?

Objetivos da aula

- Como podemos resolver problemas vistos em aula utilizando ferramentas computacionais?
  - **Solucionadores SMT** (`cvc5`): Como decidir se uma fórmula com predicados sobre certas teorias é satisfatível?

    Mais ainda: Como decidir se uma fórmula é **válida** ou encontrar um **contra-exemplo**.
  - **Assistentes de demonstração** (`Lean 4`): Como podemos escrever demonstrações garantindo sua corretude?
- Praticar **modelagem**: Iremos estudar como modelar alguns problemas de relevância nessas ferramentas.

Parte I

Satisfatibilidade Módulo Teorias

Ferramentas: cvc5

O que é um solucionador SMT?

SAT (*Satisfatibilidade Booleana*): Dado φ sobre variáveis booleanas, existe valoração $\mathcal{V}$ tal que $[\![\varphi]\!]^{\mathcal{V}} = T$?

O que é um solucionador SMT?

SAT (*Satisfatibilidade Booleana*): Dado φ sobre variáveis booleanas, existe valoração $\mathcal{V}$ tal que $[\![\varphi]\!]^{\mathcal{V}} = T$?

SMT (*Satisfatibilidade Modulo Teorias*): A ideia é a mesma, mas agora variáveis vivem em *teorias*, como inteiros, reais, vetores de bits, arrays e funções não-interpretadas.

O que é um solucionador SMT?

SAT (*Satisfatibilidade Booleana*): Dado φ sobre variáveis booleanas, existe valoração $\mathcal{V}$ tal que $[\![\varphi]\!]^{\mathcal{V}} = T$?

SMT (*Satisfatibilidade Modulo Teorias*): A ideia é a mesma, mas agora variáveis vivem em *teorias*, como inteiros, reais, vetores de bits, arrays e funções não-interpretadas.

cvc5 é um dos solucionadores SMT de ponta.

- Utilizado amplamente na indústria e academia para verificação de software.
- Desenvolvido principalmente por Stanford, Iowa, Bar Ilan e **UFMG** ☺.
- APIs em **C**, C++, Python, Java, Rust.
- `https://cvc5.github.io`

Instalação

Linux:

```
$ git clone https://github.com/cvc5/cvc5.git
$ cd cvc5
$ ./configure.sh production --auto-download
$ cd build
$ make -j$(nproc)
$ sudo make install
```

Instalação

Linux:

```
$ git clone https://github.com/cvc5/cvc5.git
$ cd cvc5
$ ./configure.sh production --auto-download
$ cd build
$ make -j$(nproc)
$ sudo make install
```

Para utilizar a biblioteca, só é necessário incluir a header `<cvc5/c/cvc5.h>` e a compilar com `-lcvc5`:

```
$ gcc meuprograma.c -lcvc5 -o meuprograma
$ ./meuprograma
```

Primeiro programa: `hello.c`

Para entender a API, iniciaremos com um exemplo simples:
Queremos verificar a satisfatibilidade da fórmula booleana $(x \wedge \neg x)$.

```
#include <cvc5/c/cvc5.h>
#include <stdio.h>

int main(void) {
    Cvc5TermManager *tm = cvc5_term_manager_new();
    Cvc5 *slv           = cvc5_new(tm);
    cvc5_set_logic(slv, "QF_UF");

    Cvc5Sort bool_sort = cvc5_get_boolean_sort(tm);
    Cvc5Term x = cvc5_mk_const(tm, bool_sort, "x");

    Cvc5Term not_args[1] = { x };
    Cvc5Term not_x = cvc5_mk_term(tm, CVC5_KIND_NOT, 1,
        not_args);
    Cvc5Term and_args[2] = { x, not_x };
    Cvc5Term f = cvc5_mk_term(tm, CVC5_KIND_AND, 2,
        and_args);

    cvc5_assert_formula(slv, f);
    Cvc5Result r = cvc5_check_sat(slv);
    printf("cvc5: %s\n", cvc5_result_to_string(r));
```

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).
- `cvc5_new(tm)`: Cria o solucionador.

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).
- `cvc5_new(tm)`: Cria o solucionador.
- `cvc5_mk_const(tm, sort, "x")`: Cria uma **variável** do tipo `sort`.

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).
- `cvc5_new(tm)`: Cria o solucionador.
- `cvc5_mk_const(tm, sort, "x")`: Cria uma **variável** do tipo `sort`.
- `cvc5_mk_term(tm, KIND, aridade, args)`: Constrói termos compostos, operadores (AND, OR, IMPLIES, ...) são *kinds*.

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).
- `cvc5_new(tm)`: Cria o solucionador.
- `cvc5_mk_const(tm, sort, "x")`: Cria uma **variável** do tipo `sort`.
- `cvc5_mk_term(tm, KIND, aridade, args)`: Constrói termos compostos, operadores (AND, OR, IMPLIES, ...) são *kinds*.
- `cvc5_assert_formula(slv, f)`: Adiciona f às hipóteses.

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).
- `cvc5_new(tm)`: Cria o solucionador.
- `cvc5_mk_const(tm, sort, "x")`: Cria uma **variável** do tipo `sort`.
- `cvc5_mk_term(tm, KIND, aridade, args)`: Constrói termos compostos, operadores (AND, OR, IMPLIES, ...) são *kinds*.
- `cvc5_assert_formula(slv, f)`: Adiciona f às hipóteses.
- `cvc5_check_sat(slv)`: Evoca o solucionador.

Exemplo Inicial

- `cvc5_term_manager_new`: Cria um gerenciador de termos (fábrica de fórmulas).
- `cvc5_new(tm)`: Cria o solucionador.
- `cvc5_mk_const(tm, sort, "x")`: Cria uma **variável** do tipo `sort`.
- `cvc5_mk_term(tm, KIND, aridade, args)`: Constrói termos compostos, operadores (AND, OR, IMPLIES, ...) são *kinds*.
- `cvc5_assert_formula(slv, f)`: Adiciona f às hipóteses.
- `cvc5_check_sat(slv)`: Evoca o solucionador.
- Saída:

```
$ gcc hello.c -lcvc5 -o hello && ./hello
cvc5: unsat
```

Codificações equivalentes em predicados

Exercício (Lista 3, Rosen 1.4.11 – item (e))

Sejam, com domínio "todas as pessoas da universidade":

- $S(x)$: "x é estudante"
- $F(x)$: "x é professor"
- $A(x, y)$: "x fez uma pergunta a y"
- m: a constante "Prof. Marcos"

Exercício (Lista 3, Rosen 1.4.11 – item (e))

Sejam, com domínio "todas as pessoas da universidade":

- $S(x)$: "x é estudante"
- $F(x)$: "x é professor"
- $A(x, y)$: "x fez uma pergunta a y"
- m: a constante "Prof. Marcos"

Enunciado:

"Todo professor que já foi perguntado por algum estudante foi questionado pelo Prof. Marcos."

Duas codificações

Dois alunos realizam a questão de diferentes maneiras:

Codificação A: Antecedente em conjunção

$$\forall x.\ \big(F(x) \land \exists y.\ (S(y) \land A(y, x))\big) \rightarrow A(m, x)$$

"Todo x que (é professor E foi perguntado por algum estudante) foi perguntado por Marcos."

Duas codificações

Dois alunos realizam a questão de diferentes maneiras:

Codificação A: Antecedente em conjunção

$$\forall x.\ \big(F(x) \land \exists y.\ (S(y) \land A(y,x))\big) \to A(m,x)$$

"Todo x que (é professor E foi perguntado por algum estudante) foi perguntado por Marcos."

Codificação B: Implicações encadeadas

$$\forall x.\ F(x) \to \big(\exists y.\ (S(y) \land A(y,x)) \to A(m,x)\big)$$

"Todo professor x satisfaz: se algum estudante o perguntou, então Marcos também o perguntou."

Equivalência lógica via SMT

Pergunta: Essas duas codificações são equivalentes?

Equivalência lógica via SMT

Pergunta: Essas duas codificações são equivalentes?

Estratégia

Duas fórmulas A e B são logicamente equivalentes sse $A \leftrightarrow B$ é **válida**.
$A \leftrightarrow B$ é válida sse $\neg(A \leftrightarrow B)$ é **insatisfatível**.

Equivalência lógica via SMT

Pergunta: Essas duas codificações são equivalentes?

Estratégia

Duas fórmulas A e B são logicamente equivalentes sse $A \leftrightarrow B$ é **válida**.
$A \leftrightarrow B$ é válida sse $\neg(A \leftrightarrow B)$ é **insatisfatível**.

Ideia: Utilizar o cvc5 para decidir a satisfatbilidade da fórmula $\neg(A \leftrightarrow B)$.

Equivalência lógica via SMT

Pergunta: Essas duas codificações são equivalentes?

Estratégia

Duas fórmulas A e B são logicamente equivalentes sse $A \leftrightarrow B$ é **válida**.
$A \leftrightarrow B$ é válida sse $\neg(A \leftrightarrow B)$ é **insatisfatível**.

Ideia: Utilizar o cvc5 para decidir a satisfatbilidade da fórmula $\neg(A \leftrightarrow B)$.

O que acabamos de fazer

- Usamos o cvc5 como um **verificador de equivalência lógica** para fórmulas de lógica de predicados.

O que acabamos de fazer

- Usamos o cvc5 como um **verificador de equivalência lógica** para fórmulas de lógica de predicados.
- A ideia geral serve para **qualquer par de codificações**: se você e seu colega formalizaram um enunciado em português de maneiras diferentes, o cvc5 pode certificar que a resposta de vocês é equivalente!

O que acabamos de fazer

- Usamos o cvc5 como um **verificador de equivalência lógica** para fórmulas de lógica de predicados.
- A ideia geral serve para **qualquer par de codificações**: se você e seu colega formalizaram um enunciado em português de maneiras diferentes, o cvc5 pode certificar que a resposta de vocês é equivalente!
- Dualmente: se o resultado for `sat`, o cvc5 devolve uma **interpretação** $\mathcal{V}$ onde A e B diferem, isto é, um **contraexemplo** para a equivalência desejada.

N-rainhas

O problema das n rainhas

Dispor n rainhas em um tabuleiro $n \times n$ sem que uma ataque a outra:

- Nunca duas na mesma **linha**;
- Nunca duas na mesma **coluna**;
- Nunca duas na mesma **diagonal**.

O problema das n rainhas

Dispor n rainhas em um tabuleiro $n \times n$ sem que uma ataque a outra:

- Nunca duas na mesma **linha**;
- Nunca duas na mesma **coluna**;
- Nunca duas na mesma **diagonal**.

Vocês já viram este problema codificado como SAT (na aula sobre satisfatibilidade). Agora vamos:

1. Completar a codificação booleana (Q1-Q5) em `C`.

O problema das n rainhas

Dispor n rainhas em um tabuleiro $n \times n$ sem que uma ataque a outra:

- Nunca duas na mesma **linha**;
- Nunca duas na mesma **coluna**;
- Nunca duas na mesma **diagonal**.

Vocês já viram este problema codificado como SAT (na aula sobre satisfatibilidade). Agora vamos:

1. Completar a codificação booleana (Q1-Q5) em `C`.
2. Apresentar uma **codificação alternativa** usando variáveis inteiras.

Codificação Booleana: Variáveis e Restrições

Para cada posição (i, j) do tabuleiro, uma variável booleana $p_{i,j}$:

$$p_{i,j} = T \iff \text{há uma rainha em } (i, j).$$

Codificação Booleana: Variáveis e Restrições

Para cada posição (i, j) do tabuleiro, uma variável booleana $p_{i,j}$:

$$p_{i,j} = T \iff \text{há uma rainha em } (i, j).$$

Q1: Pelo menos uma rainha por linha

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} p_{i,j}$$

Codificação Booleana: Variáveis e Restrições

Para cada posição (i, j) do tabuleiro, uma variável booleana $p_{i,j}$:

$$p_{i,j} = T \iff \text{há uma rainha em } (i, j).$$

Q1: Pelo menos uma rainha por linha

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} p_{i,j}$$

Q2/Q3: No máximo uma por linha/coluna

Para cada linha i e colunas $j \neq k$: $\quad p_{i,j} \rightarrow \neg p_{i,k}$.

Q4/Q5: Codificação Booleana: Variáveis e Restrições

Diagonais

Para cada (i, j) e cada $k \geq 1$ válido:

$$p_{i,j} \rightarrow \neg p_{i+k,j+k} \quad \text{(diagonal } \searrow\text{)}$$
$$p_{i,j} \rightarrow \neg p_{i+k,j-k} \quad \text{(diagonal } \nearrow\text{)}$$

Codificação alternativa: inteiros + `distinct`

Observação: Em qualquer solução, cada linha contém **exatamente** uma rainha. Logo, podemos representar a configuração com uma única variável por linha:

$$\texttt{col}[i] \in \{0, \ldots, n-1\} \quad = \text{coluna da rainha na linha } i.$$

Codificação alternativa: inteiros + `distinct`

Observação: Em qualquer solução, cada linha contém **exatamente** uma rainha. Logo, podemos representar a configuração com uma única variável por linha:

$$\texttt{col}[i] \in \{0, \ldots, n-1\} \quad = \text{coluna da rainha na linha } i.$$

Restrições (Teoria de Inteiros)

- $0 \leq \texttt{col}[i] < n$ (domínio).
- $\texttt{distinct}(\texttt{col}[0], \ldots, \texttt{col}[n-1])$: nenhuma coluna repetida.
- $\texttt{distinct}(\texttt{col}[0] - 0, \texttt{col}[1] - 1, \ldots, \texttt{col}[n-1] - (n-1))$: Restrições na diagonal $\searrow$
- $\texttt{distinct}(\texttt{col}[0] + 0, \texttt{col}[1] + 1, \ldots, \texttt{col}[n-1] + (n-1))$: Restrições na diagonal $\nwarrow$.

Sudoku

Sudoku 9×9

Preencher a grade 9×9 com dígitos 1–9 tais que:

- Cada **linha** contenha todos os 9 dígitos;
- Cada **coluna** contenha todos os 9 dígitos;
- Cada **bloco** 3×3 contenha todos os 9 dígitos;
- As **pistas** iniciais sejam respeitadas.

Sudoku 9×9

Preencher a grade 9×9 com dígitos 1–9 tais que:

- Cada **linha** contenha todos os 9 dígitos;
- Cada **coluna** contenha todos os 9 dígitos;
- Cada **bloco** 3×3 contenha todos os 9 dígitos;
- As **pistas** iniciais sejam respeitadas.

Novamente, apresentaremos duas codificações seguindo a mesma lógica.

- **Booleana**: $x_{i,j,v}$ = "a célula (i, j) contém o valor $v + 1$".
  $9 \times 9 \times 9 = 729$ variáveis.

Sudoku 9×9

Preencher a grade 9×9 com dígitos 1–9 tais que:

- Cada **linha** contenha todos os 9 dígitos;
- Cada **coluna** contenha todos os 9 dígitos;
- Cada **bloco** 3×3 contenha todos os 9 dígitos;
- As **pistas** iniciais sejam respeitadas.

Novamente, apresentaremos duas codificações seguindo a mesma lógica.

- **Booleana**: $x_{i,j,v}$ = "a célula (i, j) contém o valor $v + 1$". $9 \times 9 \times 9 = 729$ variáveis.
- **Aritmética**: $c_{i,j}$ entre 1 e 9 inteiro: "a célula i, j contém o valor $c_{i,j}$".

Sudoku: Codificação Booleana

Variável: $x_{i,j,v}$ para $i, j, v \in \{0, \ldots, 8\}$.

Sudoku: Codificação Booleana

Variável: $x_{i,j,v}$ para $i, j, v \in \{0, \ldots, 8\}$.

Cada célula tem exatamente um valor

$$\bigvee_{v=0}^{8} x_{i,j,v} \ \wedge \ \bigwedge_{v<w} \neg(x_{i,j,v} \wedge x_{i,j,w})$$

Sudoku: Codificação Booleana

Variável: $x_{i,j,v}$ para $i, j, v \in \{0, \ldots, 8\}$.

Cada célula tem exatamente um valor

$$\bigvee_{v=0}^{8} x_{i,j,v} \ \wedge \ \bigwedge_{v<w} \neg(x_{i,j,v} \wedge x_{i,j,w})$$

Cada linha/coluna/bloco contém cada valor

$$\bigwedge_{v} \bigvee_{j} x_{i,j,v} \quad \text{(análogo p/ colunas e blocos)}$$

Sudoku: Codificação Booleana

Variável: $x_{i,j,v}$ para $i, j, v \in \{0, \ldots, 8\}$.

Cada célula tem exatamente um valor

$$\bigvee_{v=0}^{8} x_{i,j,v} \ \wedge \ \bigwedge_{v<w} \neg(x_{i,j,v} \wedge x_{i,j,w})$$

Cada linha/coluna/bloco contém cada valor

$$\bigwedge_{v} \bigvee_{j} x_{i,j,v} \quad \text{(análogo p/ colunas e blocos)}$$

Pistas

Adicione $x_{i,j,v}$ às hipóteses para cada pista dada.

Sudoku: Codificação aritmética

Variável: $c_{i,j}$ inteira com $1 \leq c_{i,j} \leq 9$.

Sudoku: Codificação aritmética

Variável: $c_{i,j}$ inteira com $1 \leq c_{i,j} \leq 9$.

Restrições

- `distinct` em cada linha: $\texttt{distinct}(c_{i,0}, \ldots, c_{i,8})$.
- `distinct` em cada coluna.
- `distinct` em cada bloco 3×3.
- Pistas: $c_{i,j} = v$ para cada pista.

Parte II

Lean 4 e Dedução Natural

O que é Lean 4?

Lean 4 é uma *linguagem de programação* que é, ao mesmo tempo, um *assistente de demonstração*.

- **Maior confiança em demonstrações:** Toda inferência realizada numa demonstração tem sua validade verificada pelo núcleo de lean.
- Criado por Leonardo de Moura, um brasileiro! Também conhecido pela criação do Z3, um solucionador SMT também bastante popular.
- **Notoriedade na comunidade matemática:** Sua principal biblioteca voltada para formalização de matemática tem mais de 200,000 teoremas e 2.2 milhões de linhas de código!
- **Automação de Matemática?** Mais recentemente, ganhou bastante atenção por algumas start-ups nos EUA ao combinar LLMs e assistentes de demonstração.

Instalação via VSCode

Passo 1: instale o VSCode. `https://code.visualstudio.com`

Instalação via VSCode

Passo 1: instale o VSCode. `https://code.visualstudio.com`

Passo 2: na aba *Extensions*, instale a extensão oficial:

- Nome: `lean4`
- Publisher: `leanprover`

Instalação via VSCode

Passo 1: instale o VSCode. `https://code.visualstudio.com`

Passo 2: na aba *Extensions*, instale a extensão oficial:

- Nome: `lean4`
- Publisher: `leanprover`

Passo 3: abra qualquer `.lean` vazio. A extensão detecta que o `elan` (gerenciador de versões Lean) não está instalado e oferece um botão **"Install Lean 4"**. Clique — isso instala `elan` e a *toolchain* estável automaticamente.

Instalação via VSCode

Passo 1: instale o VSCode. `https://code.visualstudio.com`

Passo 2: na aba *Extensions*, instale a extensão oficial:

- Nome: `lean4`
- Publisher: `leanprover`

Passo 3: abra qualquer `.lean` vazio. A extensão detecta que o `elan` (gerenciador de versões Lean) não está instalado e oferece um botão **"Install Lean 4"**. Clique — isso instala `elan` e a *toolchain* estável automaticamente.

Alternativa: instalar apenas o `elan` via terminal (equivalente ao que o botão do VSCode faria, mas sem depender dele):

```
$ curl https://raw.githubusercontent.com/leanprover/\
elan/master/elan-init.sh -sSf | sh
$ source $HOME/.elan/env      # ou reabra o shell
$ lean --version               # confirma instalação
```

Criando um projeto

```
$ lake new meu_projeto .lean
$ cd meu_projeto
$ lake build
```

Criando um projeto

```
$ lake new meu_projeto .lean
$ cd meu_projeto
$ lake build
```

Abrindo no VSCode: `File` → `Open Folder` e aponte para `meu_projeto/`.

Importante: abrir a **pasta**, não um arquivo isolado — a extensão precisa do `lean-toolchain` para escolher a versão certa.

Criando um projeto

```
$ lake new meu_projeto .lean
$ cd meu_projeto
$ lake build
```

Abrindo no VSCode: `File` → `Open Folder` e aponte para `meu_projeto/`.

Importante: abrir a **pasta**, não um arquivo isolado — a extensão precisa do `lean-toolchain` para escolher a versão certa.

Primeiro teste: crie `Test.lean` e digite:

```
#check (1 + 1)
```

O painel *InfoView* (Ctrl+Shift+Enter) deve mostrar:

```
1 + 1 : Nat
```

Atenção à primeira compilação

- O primeiro `lake build` pode levar **alguns minutos**: o `elan` baixa a toolchain ($\sim$ 200 MB).

Atenção à primeira compilação

- O primeiro `lake build` pode levar **alguns minutos**: o `elan` baixa a toolchain ($\sim$ 200 MB).
- Se você adicionar **Mathlib**, ela tem $\sim$500 MB de caches pré-compilados — rodar `lake exe cache get` **antes** de `lake build` evita recompilação de **horas**.

Atenção à primeira compilação

- O primeiro `lake build` pode levar **alguns minutos**: o `elan` baixa a toolchain ($\sim$ 200 MB).
- Se você adicionar **Mathlib**, ela tem $\sim$500 MB de caches pré-compilados — rodar `lake exe cache get` **antes** de `lake build` evita recompilação de **horas**.

Dedução Natural em Lean 4

O tipo `Form`

Primeiro, definimos o que é uma **fórmula proposicional**:

```
inductive Form : Type
  | var : Nat  → Form
  | and : Form → Form → Form
  | or  : Form → Form → Form
  | imp : Form → Form → Form
  | neg : Form → Form
  | bot : Form
  deriving Repr, DecidableEq
```

O tipo `Form`

Primeiro, definimos o que é uma **fórmula proposicional**:

```
inductive Form : Type
  | var : Nat  → Form
  | and : Form → Form → Form
  | or  : Form → Form → Form
  | imp : Form → Form → Form
  | neg : Form → Form
  | bot : Form
  deriving Repr, DecidableEq
```

- `var n` representa a proposição ψ_n.
- Os construtores correspondem um-a-um aos conectivos $\wedge, \vee, \to, \neg, \bot$.

O tipo `Proof` (1/2)

`Proof` Γ φ significa: *"a partir das hipóteses em* Γ*, deriva-se* φ*"*.

```
inductive Proof : List Form → Form → Prop
  | hyp    {Γ φ}      : φ ∈ Γ → Proof Γ φ

  -- Conjunção:
  | andI   {Γ φ ψ}    : Proof Γ φ → Proof Γ ψ
                      → Proof Γ (φ ∧ ψ)
  | andEd  {Γ φ ψ}    : Proof Γ (φ ∧ ψ) → Proof Γ φ
  | andEe  {Γ φ ψ}    : Proof Γ (φ ∧ ψ) → Proof Γ ψ

  -- Implicação:
  | impI   {Γ φ ψ}    : Proof (φ :: Γ) ψ → Proof Γ (φ ⇒ ψ)
  | impE   {Γ φ ψ}    : Proof Γ (φ ⇒ ψ) → Proof Γ φ
                      → Proof Γ ψ
```

O tipo `Proof` (2/2)

```
-- Disjunção:
| orId   {Γ φ ψ}    : Proof Γ φ → Proof Γ (φ ∨ ψ)
| orIe   {Γ φ ψ}    : Proof Γ ψ → Proof Γ (φ ∨ ψ)
| orE    {Γ φ ψ χ} : Proof Γ (φ ∨ ψ)
                   → Proof (φ :: Γ) χ
                   → Proof (ψ :: Γ) χ
                   → Proof Γ χ

-- Negação e absurdo:
| notI   {Γ φ}      : Proof (φ :: Γ) ⊥ₚ → Proof Γ (¬φ)
| notE   {Γ φ}      : Proof Γ (¬φ) → Proof Γ φ
                   → Proof Γ ⊥ₚ
| absurd {Γ φ}      : Proof ((¬φ) :: Γ) ⊥ₚ → Proof Γ φ
```

O tipo `Proof` (2/2)

```
-- Disjunção:
| orId   {Γ φ ψ}    : Proof Γ φ → Proof Γ (φ ∨ ψ)
| orIe   {Γ φ ψ}    : Proof Γ ψ → Proof Γ (φ ∨ ψ)
| orE    {Γ φ ψ χ} : Proof Γ (φ ∨ ψ)
                   → Proof (φ :: Γ) χ
                   → Proof (ψ :: Γ) χ
                   → Proof Γ χ

-- Negação e absurdo:
| notI   {Γ φ}      : Proof (φ :: Γ) ⊥ₚ → Proof Γ (¬φ)
| notE   {Γ φ}      : Proof Γ (¬φ) → Proof Γ φ
                   → Proof Γ ⊥ₚ
| absurd {Γ φ}      : Proof ((¬φ) :: Γ) ⊥ₚ → Proof Γ φ
```

- Cada construtor corresponde a uma regra de inferência dos slides.
- `impI`, `notI` e `absurd` *descartam* hipóteses: a premissa tem um Γ aumentado.
- Um termo `t : Proof Γ φ` é a árvore de dedução.

Cada regra é um *construtor*

Cada regra de dedução natural será aplicada por meio da tática `apply`:

$\wedge_I$	$\leftrightarrow$	`apply Proof.andI`
$\wedge_{E_d}$ / $\wedge_{E_e}$	$\leftrightarrow$	`apply Proof.andEd / .andEe`
$\vee_{I_d}$ / $\vee_{I_e}$	$\leftrightarrow$	`apply Proof.orId / .orIe`
$\vee_E$	$\leftrightarrow$	`apply Proof.orE`
$\rightarrow_I$ / $\rightarrow_E$	$\leftrightarrow$	`apply Proof.impI / .impE`
hipótese	$\leftrightarrow$	`apply Proof.hyp`

Exercício da Prova

Empregando dedução natural, construa uma árvore que conclua:

$$\Big((\psi_1 \vee \psi_2) \wedge \big((\psi_1 \wedge \psi_3) \vee (\psi_1 \wedge \psi_3)\big)\Big) \rightarrow \big((\psi_1 \vee \psi_2) \wedge \psi_3\big)$$

Exercício da Prova

Empregando dedução natural, construa uma árvore que conclua:

$$\Big((\psi_1 \vee \psi_2) \wedge \big((\psi_1 \wedge \psi_3) \vee (\psi_1 \wedge \psi_3)\big)\Big) \to \big((\psi_1 \vee \psi_2) \wedge \psi_3\big)$$

Observação: os dois disjuntos internos são idênticos — os casos de $\vee_E$ ficam simétricos, mas o exercício continua ilustrando a estrutura canônica.

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

1. Assumir H (hipótese 1).

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

1. Assumir H (hipótese 1).
2. De H, por $\wedge_{E_d}$: obter $A = \psi_1 \vee \psi_2$.

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

1. Assumir H (hipótese 1).
2. De H, por $\wedge_{E_d}$: obter $A = \psi_1 \vee \psi_2$.
3. De H, por $\wedge_{E_e}$: obter $D = (\psi_1 \wedge \psi_3) \vee (\psi_1 \wedge \psi_3)$.

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

1. Assumir H (hipótese 1).
2. De H, por $\wedge_{E_d}$: obter $A = \psi_1 \vee \psi_2$.
3. De H, por $\wedge_{E_e}$: obter $D = (\psi_1 \wedge \psi_3) \vee (\psi_1 \wedge \psi_3)$.
4. Análise de casos ($\vee_{E,2,3}$) sobre D:
   - Caso rotulado 2: assumir $\psi_1 \wedge \psi_3$, por $\wedge_{E_e}$ obter ψ_3.
   - Caso rotulado 3: idem.

   Ambos os ramos concluem ψ_3, então ψ_3 segue de D.

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

1. Assumir H (hipótese 1).
2. De H, por $\wedge_{E_d}$: obter $A = \psi_1 \vee \psi_2$.
3. De H, por $\wedge_{E_e}$: obter $D = (\psi_1 \wedge \psi_3) \vee (\psi_1 \wedge \psi_3)$.
4. Análise de casos ($\vee_{E,2,3}$) sobre D:
   - Caso rotulado 2: assumir $\psi_1 \wedge \psi_3$, por $\wedge_{E_e}$ obter ψ_3.
   - Caso rotulado 3: idem.

   Ambos os ramos concluem ψ_3, então ψ_3 segue de D.
5. Combinar A e ψ_3 por $\wedge_I$: obter $(\psi_1 \vee \psi_2) \wedge \psi_3$.

Estratégia de demonstração (passo a passo)

Seja H o antecedente.

1. Assumir H (hipótese 1).
2. De H, por $\wedge_{E_d}$: obter $A = \psi_1 \vee \psi_2$.
3. De H, por $\wedge_{E_e}$: obter $D = (\psi_1 \wedge \psi_3) \vee (\psi_1 \wedge \psi_3)$.
4. Análise de casos ($\vee_{E,2,3}$) sobre D:
   - Caso rotulado 2: assumir $\psi_1 \wedge \psi_3$, por $\wedge_{E_e}$ obter ψ_3.
   - Caso rotulado 3: idem.

   Ambos os ramos concluem ψ_3, então ψ_3 segue de D.
5. Combinar A e ψ_3 por $\wedge_I$: obter $(\psi_1 \vee \psi_2) \wedge \psi_3$.
6. Descartar H via $\rightarrow_{I,1}$.

Materiais Complementares

- **cvc5**: https://cvc5.github.io/docs/latest/api/c/c.html
- **Exemplos em C**: https://github.com/cvc5/cvc5/tree/main/examples/api/c
- **Lean 4**: https://lean-lang.org/lean4/doc/quickstart.html
- **Mathematics in Lean** (livro gratuito): https://leanprover-community.github.io/mathematics_in_lean/
- **Theorem Proving in Lean 4**: https://leanprover.github.io/theorem_proving_in_lean4/

Materiais Complementares

- **Aplicações de SAT/SMT:**
  `https://smt.st/SAT_SMT_by_example.pdf`
- **Tutorial detalhado sobre SMT:**
  `https://hanielbarbosa.com/papers/fm2024.pdf`
- **Aprenda Lean com jogos:** `https://adam.math.hhu.de/`

Apêndice A: Automação em Lean 4

Automação em Lean 4

Na demonstração que vimos, realizamos tudo passo a passo, construindo a demonstração diretamente. Mas... e se quissermos incluir alguma forma de automação nisso tudo?

Isso é possível. Lean possuí muitas opções diferentes para automação, por meio de *táticas*.

- `simp`: Simplifica o objetivo, usando regras conhecidas.
- `intro`, `rcases`, `exact`: táticas básicas para manipulação do objetivo a ser demonstrado.
- `tauto`: Demonstra tautologias.
- `smt`: Utiliza um solucionador smt para verificar/demonstrar a validade do objetivo.

Apêndice B: Exercício: SEND + MORE = MONEY

Exercício: quebra-cabeça criptoaritmético

Enunciado

Atribuir **dígitos distintos** de 0–9 a cada letra em

$$\texttt{S E N D}, \quad \texttt{M O R E}, \quad \texttt{M O N E Y}$$

de modo que a soma na base 10 seja consistente:

$$\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$$

com $S \neq 0$ e $M \neq 0$ (sem zeros à esquerda).

Faça um programa em C utilizando o cvc5 que encontra o valor de cada uma dessas variáveis.

Repositório com o código

Todo o código apresentado nesta aula está disponível no repositório:

`https://github.com/psaccomani15/ILC-lab`