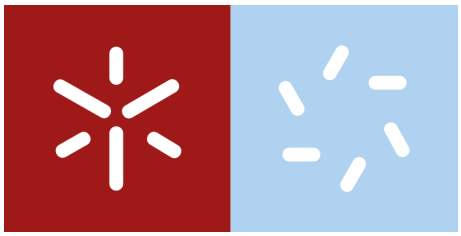

Ferramenta para Construção Assistida de Provas em Dedução Natural

Ano letivo 2024/2025

Daniel Andrade
A100057

Pedro Malainho
A100050

Simão Ribeiro
A102877

Universidade do Minho
Escola de Ciências
Departamento de Matemática

Maio 2025

Ferramenta para Construção Assistida de Provas em Dedução Natural

Ano letivo 2024/2025

Relatório do Projeto
Licenciatura em Ciências da Computação

Orientador
Luís Filipe Ribeiro Pinto

Maio 2025

Resumo

O presente relatório descreve o desenvolvimento de uma ferramente computacional destinada à construção assistida de provas em Dedução Natural. Embora não tenha sido concebida diretamente no âmbito da unidade curricular de Lógica, a ferramenta foi projetada com o objetivo de apoiar o ensino e aprendizagem de sistemas formais de prova, sendo particularmente relevante para UCs que exploram os fundamentos da lógica clássica e intuicionista - como é o caso da própria unidade curricular de Lógica, onde se privilegia o raciocínio formal e a construção rigorosa de demonstrações.

A aplicação foi desenvolvida com base numa arquitetura web modular. O backend, implementado em Python 3.12 com os packages Flask e PLY, é responsável pela análise e validação lógica das provas. O frontend, desenvolvido com Vite, ReactJS, CSS e Axios, oferece uma interface interativa que permite ao utilizador construir e manipular provas de forma assistida. Um dos componentes estruturais centrais do sistema é a modelação do estado da prova, concebida para representar explicitamente os subproblemas em aberto e permitir o controlo incremental da construção dedutiva.

Ao longo do relatório, são apresentadas as principais decisões de modelação, a organização da arquitetura, e os detalhes da implementação das regras formais que sustentam a lógica subjacente ao funcionamento da ferramenta.

Indice

1. Introdução

A construção assistida de provas em lógica formal exige um conhecimento profundo dos sistemas dedutivos, em particular dos fundamentos da Dedução Natural. Este formalismo baseia-se na aplicação sistemática de regras de introdução e eliminação dos conectivos lógicos, permitindo construir demonstrações rigorosas a partir de hipóteses bem definidas. Uma das características centrais da Dedução Natural é a possibilidade de raciocinar no contexto de hipóteses, com regras específicas associadas à estrutura sintática dos conectivos envolvidos.

Este tipo de abordagem é estudado, por exemplo, na unidade curricular de Lógica da Licenciatura em Ciências da Computação (LCC), onde os estudantes são confrontados com a tarefa de construir, validar e compreender provas formais em lógicas clássica e intuicionista. No contexto do desenvolvimento computacional, a representação e manipulação de provas em Dedução Natural exige uma modelação precisa do estado da prova, capaz de refletir dinamicamente o progresso dedutivo e os subproblemas que permanecem por resolver. Cada passo deve ser verificado de acordo com as regras do sistema lógico em uso, assegurando a validade global da demonstração.

Este projeto tem como objetivo o desenvolvimento de uma ferramenta computacional para a construção assistida de provas em Dedução Natural, com suporte às lógicas clássica e intuicionista. A ferramenta visa apoiar o processo de ensino-aprendizagem, oferecendo um ambiente interativo que auxilia na aplicação correta das regras e no raciocínio formal estruturado. Um dos elementos centrais da sua conceção é a modelação do estado da prova, que, a cada momento, deve permitir identificar de forma clara os subproblemas ou subprovas ainda em aberto, orientando o utilizador na sua resolução.

As decisões relativas às linguagens e tecnologias utilizadas foram tomadas numa fase inicial do projeto, privilegiando ferramentas adequadas ao desenvolvimento web e à manipulação simbólica. Este relatório descreve a arquitetura geral da aplicação, a modelação e implementação das regras formais, e as decisões técnicas que orientaram a construção da ferramenta.

2. Modelação e Implementação

Esta secção descreve a criação e a realização técnica da ferramenta de construção assistida de provas em dedução natural. São abordadas a modelação do estado da prova, a arquitetura do sistema cliente-servidor e a implementação das regras de inferência nos contextos da lógica clássica e intuicionista.

2. 1. Modelação estado prova

A representação interna do estado da prova foi estruturada de forma a refletir com precisão a lógica da dedução natural, permitindo tanto a verificação formal da prova quanto a sua manipulação interativa pelo utilizador.

O estado é modelado como uma árvore de dedução, em que cada nó representa uma etapa da prova e contém os seguintes elementos:

- A fórmula lógica atual;
- A regra de inferência aplicada;
- As premissas envolvidas na aplicação da regra;
- As sub-provas associadas, caso a regra exija a demostração de hipóteses adicionais;
- A validação formal, isto é, a verificação de que a regra foi corretamente aplicada de acordo com os princípios da lógica em uso (clássica ou intuicionista);
- Uma identificação única para cada nó, permitindo a sua referência e manipulação.

Além da árvore principal de dedução, é mantido um histórico de ações efetuadas pelo utilizador. Este histórico permite implementar funcionalidades de refazer, facilitando a exploração e revisão de diferentes caminhos de prova.

Este modelo de estado fornece a base para a interface interativa da ferramenta, suportando:

- A construção passo-a-passo de provas;
- A visualização em tempo real da árvore de dedução e dos objetos pendentes.

2. 2. Arquitetura projeto

A ferramenta segue uma arquitetura cliente-servidor modular, que promove a separação de responsabilidades entre a interface de utilizador e o motor lógico de validação e resolução de provas.

2. 2. 1. Cliente

O cliente foi desenvolvido utilizando ReactJS em conjunto com Vite, com o objetivo de proporcionar uma experiência fluida e responsiva. As funcionalidades principais do sistema incluem:

- Edição interativa de provas, com visualização em tempo real da árvore de dedução, permitindo ao utilizador construir a prova de forma estruturada e compreensível;
- Destacamento de sub-provas em aberto, auxiliando o utilizador na identificação de objetivos pendentes e no planeamento da construção da prova;
- Interface intuitiva e acessível, com componentes reutilizáveis e um design centrado na usabilidade, o que favorece a aplicação tanto em contextos educativos como exploratórios;
- Reinício do trabalho, através de um botão que permite ao utilizador reiniciar a prova atual, eliminando todos os passos e estruturas previamente inseridas;
- Exportação da prova, possibilitando o download da árvore de dedução atual em formato JSON, para armazenamento local do progresso;
- Importação da prova, permitindo o upload de uma árvore de dedução previamente guardada, o que facilita a retomada dos trabalhos anteriores ou a partilha de provas entre utilizadores.

Esta abordagem centrada no utilizador, aliada à utilização de tecnologias modernas, assegura uma aplicação eficiente, com tempos de resposta rápidos e elevada extensibilidade, Assim, o sistema torna-se ideal para suporte ao ensino e prática da dedução natural em ambientes acadêmicos.

2. 2. 2. Servidor

O servidor foi implementado em Python, utilizando o micro framework Flask para a criação de uma API RESTful clara, bem estruturada e orientada à manipulação de dados em formato JSON. A principal responsabilidade do servidor é a avaliação e resolução lógica das provas, assegurando a verificação rigorosa dos passos de dedução fornecidos.

A arquitetura do servidor foi organizada em módulos distintos, cada um com responsabilidades específicas, conforme descrito a seguir:

- Compilador: Responsável pela análise sintática das expressões lógicas recebidas, convertendo-as em estruturas internas adequadas para posterior avaliação, garantindo a conformidade com a gramática definida.
- DTO Manager: Atende às requisições HTTP e gerencia os Data Transfer Objects (DTOs), facilitando a comunicação entre cliente e servidor ao encapsular e padronizar os dados trocados entre os módulos;
- Utils: Conjunto de utilitários e funções auxiliares utilizadas em diversos componentes do sistema. Este módulo visa a reutilização de lógica comum e a redução de duplicação de código, contribuindo para uma base de código mais coesa.
- Rules: Implementa o núcleo lógico da aplicação, definindo e aplicando as regras de inferência e validando os passos de dedução.

Para suportar as funcionalidades avançadas oferecidas no cliente, o servidor também expõe endpoints adicionais que permitem:

- Inicializar uma nova prova, eliminando o estado anterior e preparando a estrutura para um novo conjunto de deduções;
- Exportar o estado atual da árvore de dedução, retornando ao cliente uma representação JSON da prova em curso;
- Importar uma árvore previamente guardada, possibilitando a reconstrução do estado da prova a partir de um ficheiro JSON fornecido pelo utilizador.

Esta estrutura modular e extensível garante uma integração eficaz entre cliente e servidor, assegurando robustez, adaptabilidade e adequação ao contexto acadêmico.

2. 3. Implementação regras

As regras de dedução foram implementadas com base nos formalismos da dedução natural, assegurando rigor e correção tanto para a lógica clássica quanto para a lógica intuicionista.

Cada regra foi codificada como uma função pura, parametrizada por premissas e, quando aplicável, por metas ou contextos locais. Essas funções retornam o passo inferido quando a aplicação é válida, ou uma descrição detalhada do erro caso a aplicação seja inválida.

O sistema contempla um conjunto abrangente de regras, incluindo:

- Introdução e eliminação de conectivos lógicos: conjunção ($\wedge$), disjunção ($\vee$), implicação ($\rightarrow$) e negação ( );
- Introdução e eliminação do absurdo ($\perp$);

A organização das regras em módulos facilita a manutenção do sistema e a adaptação para diferentes formalismos lógicos, como a lógica clássica e a lógica intuicionista. Dessa forma, o sistema oferece um método confiável e claro para a construção e verificação automática das provas.

3. Regras suportadas

3. 1. Hipótese

A **regra da Hipótese** permite introduzir diretamente uma proposição que tenha sido previamente assumida como hipótese no contexto atual da prova. Esta regra valida a utilização de uma fórmula que está disponível no conjunto de premissas ou hipóteses ativas. Trata-se da base da construção dedutiva, uma vez que todas as provas se iniciam ou recorrem a hipóteses assumidas temporariamente ou globalmente.

Implementação ver: **Código 1**.

3. 2. Implicação

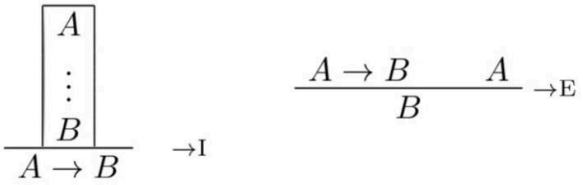

Figura 1: Regras da Implicação.

A **Introdução da Implicação** permite inferir uma proposição condicional $p \to q$ a partir de uma sub-prova em que p é assumido como hipótese e q é demonstrado sob essa suposição. Conclui-se que $p \to q$ é válida, encerrando o bloco de suposição correspondente à hipótese inicial. Esta regra formaliza a ideia de que, sempre que p for verdadeiro, q também o será.

Implementação ver: **Código 2**.

A **Eliminação da Implicação** (também conhecida como modus ponens) permite deduzir q a partir de duas premissas: uma implicação $p \to q$ e a afirmação de p. Sendo p verdadeiro, e sabendo que p implica q, então q segue-se logicamente.

Implementação ver: **Código 3**.

3. 3. Conjunção

$$\frac{A \qquad B}{A \wedge B}\ \wedge\mathrm{I} \qquad \frac{A \wedge B}{A}\ \wedge\mathrm{E} \qquad \frac{A \wedge B}{B}\ \wedge\mathrm{E}$$

Figura 2: Regras da Conjunção.

A **Introdução da Conjunção** permite inferir $p \wedge q$ a partir das proposições p e q individualmente demonstradas. Esta regra formaliza a junção de duas verdades em um única conjunta.

Implementação ver: **Código 4**.

A **Eliminação da Conjunção** permite obter qualquer uma das partes de uma conjunção. A partir de $p \wedge q$, pode-se inferir p, e igualmente q. Esta regra reconhece que ambas as componentes de um conjunção são, individualmente, verdadeiras.

Implementação ver: **Código 5**.

3. 4. Disjunção

$$\frac{A}{A \vee B}\ \vee\mathrm{I} \qquad \frac{B}{A \vee B}\ \vee\mathrm{I} \qquad \frac{A \vee B \qquad \begin{array}{|c|}\hline A \\ \vdots \\ C \\ \hline\end{array} \qquad \begin{array}{|c|}\hline B \\ \vdots \\ C \\ \hline\end{array}}{C}\ 1\ \vee\mathrm{E}$$

Figura 3: Regras da Disjunção.

A **Introdução da Disjunção** permite inferir $p \vee q$ a partir de p, ou alternativamente $p \vee q$ a partir de q. Esta regra expressa que, ao ter uma das disjunções como verdadeira, pode-se concluir que a disjunção completa é verdadeira, independentemente do valor de verdade da outra parte.

Implementação ver: **Código 6**.

A **Eliminação da Disjunção** permite concluir uma proposição r a partir de uma disjunção $p \vee q$, desde que r possa ser derivada tanto assumido p como assumindo q. Requer, portanto, duas sub-provas: uma em que, assumindo p, se demonstra r, e outra em que, assumindo q, se demonstra r. A conclusão é que, independentemente de qual disjunção seja verdadeira, a proposição r pode ser inferida em ambos os casos.

Implementação ver: **Código 7**.

3. 5. Equivalencia

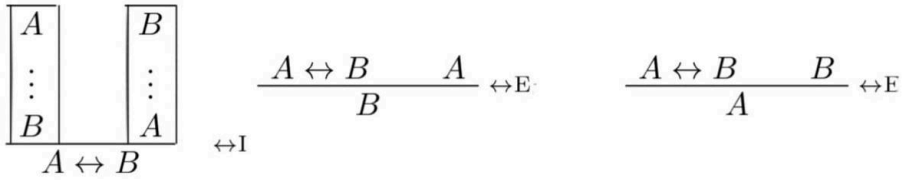

Figura 4: Regras da Equivalência.

A **Introdução da Equivalência** permite concluir $p \leftrightarrow q$ se for possível demonstrar tanto $p \rightarrow q$ como $q \rightarrow p$. A equivalência afirma uma relação lógica de dupla implicação entre p e q, indicando que cada uma das proposições implica a outra.

Implementação ver: **Código 8**.

A **Eliminação da Equivalência** permite deduzir $p \rightarrow q$ ou $q \rightarrow p$ a partir de $p \leftrightarrow q$. Como a equivalência expressa uma implicação em ambas as direções, qualquer uma das duas pode ser inferida isoladamente conforme necessário no raciocínio dedutivo.

Implementação ver: **Código 9**.

3. 6. Negação

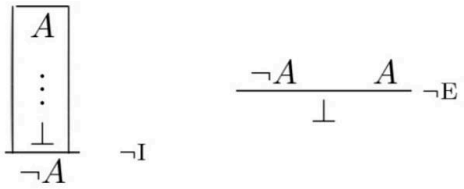

Figura 5: Regras da Negação.

A **Introdução da Negação** baseia-se na demonstração de uma contradição a partir da suposição de uma proposição p. Se ao assumir p se chega a um absurdo (uma contradição lógica), conclui--se que p não pode ser verdadeiro, e portanto $\neg p$ é válido. Esta regra expressa o princípio da redução ao absurdo para introdução da negação.

Implementação ver: **Código 10**.

A **Eliminação da Negação** permite inferir uma contradição a partir da proposição p e de sua negação $\neg p$. Quando se tem simultaneamente p e $\neg p$, chega-se logicamente a um absurdo, simbolizando a falência da coerência entre essas proposições.

Implementação ver: **Código 11**.

3. 7. Absurdo

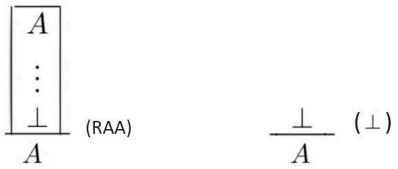

Figura 6: Regras do Absurdo.

A **Introdução do Absurdo** acontece quando, ao assumir uma hipótese, conseguimos provar uma contradição — ou seja, uma proposição e a sua negação. Isso mostra que a hipótese leva a um absurdo, representado por $\perp$.

Implementação ver: **Código 12**.

A **Eliminação do Absurdo** diz que, se temos um absurdo ($\perp$), podemos concluir qualquer proposição p. Isso mostra que contradições tornam qualquer coisa logicamente possível.

Implementação ver: **Código 13**.

4. Conclusão

Este trabalho incidiu no desenvolvimento de uma ferramenta computacional destinada à construção assistida de provas em Dedução Natural, com suporte às lógicas clássica e intuicionista. A aplicação foi concebida com o objetivo de apoiar o processo de ensino-aprendizagem de raciocínio formal, proporcionando ao utilizador um ambiente interativo que reflete, de forma estruturada, a dinâmica de uma prova lógica.

A implementação centrou-se na modelação do estado da prova, permitindo representar subprovas ativas, aplicar regras de inferência e acompanhar a progressão da demonstração. Ao longo do relatório, foram apresentadas as principais decisões técnicas que orientaram o desenvolvimento da aplicação, desde a definição da arquitetura tecnológica até à formalização das regras do sistema lógico.

O resultado obtido é uma aplicação funcional que promove a compreensão da estrutura das provas formais, servindo como recurso complementar em contextos educativos.

Como trabalho futuro, prevê-se a introdução de mecanismos de validação automática das provas construídas, bem como o aperfeiçoamento da interface gráfica, com vista a melhorar a usabilidade e a acessibilidade da ferramenta. Adicionalmente, considera-se a possibilidade de integrar a representação das provas em termos lambda, explorando a correspondência de Curry-Howard, o que permitirá estabelecer uma ponte entre lógica e programação funcional, alargando o potencial pedagógico da ferramenta.

5. Anexos

Código 1: Python

```python
def apply_axiom(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:

    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(),
        auxiliar_formula)

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("Auxiliary formula cannot be empty.")

        if logical_expr == new_problem_parsed:
            print("Inisde")
            result = [
                {
                    "name": None,
                    "parentId": "",
                    "child": [],
                    "knowledge_base": [],
                    "rule" : f"{auxiliar_formula.upper()}"
                }
            ]

            return result

    except Exception as e:
        print(f"Exception in axiom: {e}")
        raise
```

Código 2: Python

```python
def apply_implication_introduction(
        logical_expr: str,
```

```
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    arguments = split_expression(logical_expr)

    if len(arguments) != 3 or arguments[0] != '→':
        raise ValueError('Implication introduction requires 3 arguments or symbol not →')

    antecedent, consequent = arguments[1], arguments[2]

    local_knowledge_base = {}
    tmp = f'X{problem_id}'
    local_knowledge_base[tmp] = antecedent

    result = [
        {
            "name": consequent,
            "parentId": "",
            "child": [],
            "knowledge_base": local_knowledge_base,
            "rule": "→I"
        },
    ]

    return result
```

Código 3: Python

```python
def apply_implication_elimination(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:

    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(), auxiliar_formula)

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
```

```
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        result = [
            {
                "name": f"EBinOp(→, {new_problem_parsed}, {logical_expr})",
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "→E"
            },
            {
                "name": new_problem_parsed,
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "→E"
            },
        ]

        return result

    except Exception as e:
        print(f"Exception in implication elimination: {e}")
        raise
```

Código 4: Python

```
def apply_conjunction_introduction(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    arguments = split_expression(logical_expr)

    if len(arguments) != 3 or arguments[0] != '∧':
        raise ValueError('conjunction introduction requires 3 arguments or symbol not ∧')

    antecedent, consequent = arguments[1], arguments[2]

```

```
    result = [
        {
            "name": antecedent,
            "parentId": "",
            "child": [],
            "knowledge_base": [],
            "rule": "∧I",
        },
        {
            "name": consequent,
            "parentId": "",
            "child": [],
            "knowledge_base": [],
            "rule": "∧I",
        },
    ]

    return result
```

Código 5: Python

```python
def apply_conjunction_elimination_1(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:
    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(),
        auxiliar_formula)

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        result = [
            {
                "name": f"EBinOp(∧, {logical_expr}, {new_problem_parsed})",
```

```
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "∧E1",
            }
        ]

        return result

    except Exception as e:
        print(f"Exception in conjunction elimination: {e}")
        raise


def apply_conjunction_elimination_2(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:
    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(),
        auxiliar_formula)

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        result = [
            {
                "name": f"EBinOp(∧, {new_problem_parsed}, {logical_expr})",
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "∧E2",
            },
```

```
        ]

        return result

    except Exception as e:
        print(f"Exception in conjunction elimination: {e}")
        raise
```

Código 6: Python

```python
def apply_disjunction_introduction_1(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    arguments = split_expression(logical_expr)

    if len(arguments) != 3 or arguments[0] != 'v':
        raise ValueError('disjunction introduction requires 3 arguments or symbol not v')

    antecedent, consequent = arguments[1], arguments[2]

    result = [
        {
            "name": antecedent,
            "parentId": "",
            "child": [],
            "knowledge_base": [],
            "rule": "vI1"
        }
    ]

    return result


def apply_disjunction_introduction_2(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    arguments = split_expression(logical_expr)

    if len(arguments) != 3 or arguments[0] != 'v':
```

```
        raise ValueError('disjunction introduction requires 3 arguments or symbol not v')

    antecedent, consequent = arguments[1], arguments[2]

    result = [
        {
            "name": consequent,
            "parentId": "",
            "child": [],
            "knowledge_base": [],
            "rule": "vI2"
        },
    ]

    return result
```

Código 7: Python

```python
def apply_disjunction_elimination(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:

    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(), auxiliar_formula)

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        arguments = split_expression(new_problem_parsed)

        if len(arguments) != 3 or arguments[0] != 'v':
```

```
            raise ValueError('Disjunction elimination requires 3 arguments or symbol not v')

        antecedent, consequent = arguments[1], arguments[2]
        local_knowledge_base1 = {}
        tmp = f'X{problem_id}'
        local_knowledge_base1[tmp] = antecedent

        local_knowledge_base2 = {}
        tmp = f'Z{problem_id}'
        local_knowledge_base2[tmp] = consequent

        print(f"{tmp}:{antecedent}")
        print(f"{tmp}:{consequent}")

        result = [
            {
                "name": new_problem_parsed,
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "vE",
            },
            {
                "name": logical_expr,
                "parentId": "",
                "child": [],
                "knowledge_base": local_knowledge_base1,
                "rule": "vE",
            },
            {
                "name": logical_expr,
                "parentId": "",
                "child": [],
                "knowledge_base": local_knowledge_base2,
                "rule": "vE",
            },
        ]

        return result

    except Exception as e:
        print(f"Exception in disjunction elimination: {e}")
        raise
```

Código 8: Python

```
def apply_equivalence_introduction(
```

```python
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:
    arguments = split_expression(logical_expr)

    if len(arguments) != 3 or arguments[0] != '⇔':
        raise ValueError('equivalence introduction requires 3 arguments or symbol not ⇔')

    antecedent, consequent = arguments[1], arguments[2]
    local_knowledge_base1 = {}
    tmp = f'X{problem_id}'
    local_knowledge_base1[tmp] = antecedent

    local_knowledge_base2 = {}
    tmp = f'Z{problem_id}'
    local_knowledge_base2[tmp] = consequent

    result = [
        {
            "name": consequent,
            "parentId": "",
            "child": [],
            "knowledge_base": local_knowledge_base1,
            "rule": "⇔I",
        },
        {
            "name": antecedent,
            "parentId": "",
            "child": [],
            "knowledge_base": local_knowledge_base2,
            "rule": "⇔I",
        },
    ]

    return result
```

Código 9: Python

```python
def apply_equivalence_elimination_1(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:

```

```
    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(),
        auxiliar_formula)

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        result = [
            {
                "name": f"EBinOp(⇔, {new_problem_parsed}, {logical_expr})",
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "⇔E1",
            },
            {
                "name": new_problem_parsed,
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "⇔E1"
            },
        ]

        return result


    except Exception as e:
        print(f"Exception in equivalence elimination: {e}")
        raise


def apply_equivalence_elimination_2(
        logical_expr: str,
        available_hypothesis: set[tuple[str, Any]],
```

```python
        problem_id: str,
        auxiliar_formula: str
) -> list[dict[str, str | list[Any] | Any]]:

    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(),
        auxiliar_formula)

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        result = [
            {
                "name": f"EBinOp(⇔, {logical_expr}, {new_problem_parsed})",
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "⇔E2",
            },
            {
                "name": new_problem_parsed,
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "⇔E2"
            },
        ]

        return result


    except Exception as e:
        print(f"Exception in equivalence elimination: {e}")
        raise
```

Código 10: Python

```python
def apply_negation_introduction(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    content = None
    match = re.search(r"EUnOp\(~, (.+)\)", logical_expr)

    if match:
        content = match.group(1)
    else:
        raise ValueError("Logical expression must start with ~.")

    local_knowledge_base = {}
    tmp = f'X{problem_id}'
    local_knowledge_base[tmp] = content

    result = [
        {
            "name": "ABSURD_LITERAL",
            "parentId": "",
            "child": [],
            "knowledge_base": local_knowledge_base,
            "rule": "~I",
        },
    ]

    return result
```

Código 11: Python

```python
def apply_negation_elimination(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    if logical_expr != "ABSURD_LITERAL":
        raise ValueError("Logical expression must be ⊥")

    available_hypothesis_dict = dict(available_hypothesis)

    try:
        new_problem = available_hypothesis_dict.get(auxiliar_formula.upper(),
        auxiliar_formula)
```

```

        if not new_problem:
            raise ValueError("Auxiliary formula cannot be empty.")

        new_problem_parsed = CodeGenerator().generate_code(
            SemanticAnalyzer().analyze(
                Parser.parse(new_problem, debug=False)
            )
        )

        if not new_problem_parsed:
            raise ValueError("No new problem parsed.")

        result = [
            {
                "name": f"EUnOp(~,{new_problem_parsed})",
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "~E",
            },
            {
                "name": new_problem_parsed,
                "parentId": "",
                "child": [],
                "knowledge_base": [],
                "rule": "~E",
            },
        ]

        return result

    except Exception as e:
        print(f"Exception in negation elimination: {e}")
        raise
```

Código 12: Python

```python
def apply_RAA(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    local_knowledge_base = {}
    tmp = f'X{problem_id}'
    local_knowledge_base[tmp] = f'EUnOp(~, {logical_expr})'
```

```

    result = [
        {
            "name": "ABSURD_LITERAL",
            "parentId": "",
            "child": [],
            "knowledge_base": local_knowledge_base,
            "rule": "RAA",
        },
    ]

    return result
```

Código 13: Python

```python
def apply_absurd_elimination(
        logical_expr: str,
        available_hypothesis: set[str],
        problem_id: str,
        auxiliar_formula : str
) -> list[dict[str, str | list[Any] | Any]]:

    result = [
        {
            "name": "ABSURD_LITERAL",
            "parentId": "",
            "child": [],
            "knowledge_base": [],
            "rule": "AE",
        },
    ]

    return result
```