Pós-Graduação em Ciência da Computação

Eunice Palmeira da Silva

Conversão de Provas em Lógica de Descrições $\mathcal{ALC}$ Geradas pelo Método de Conexões para Sequentes

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

RECIFE

2017

Eunice Palmeira da Silva

Conversão de Provas em Lógica de Descrições $\mathcal{ALC}$ Geradas pelo Método de Conexões para Sequentes

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutora em Ciência da Computação.

Universidade Federal de Pernambuco

Orientador: Frederico Luiz Gonçalves de Freitas
Coorientador: Jens Otten

RECIFE
2017

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

S586c Silva, Eunice Palmeira da.
Conversão de provas em lógica de descrições ALC geradas pelo método de conexões para sequentes / Eunice Palmeira da Silva. – 2017.
125f.: fig., tab.

Orientador: Frederico Luiz Gonçalves de Freitas.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, Recife, 2017.
Inclui referências e apêndices.

1. Ciência da computação 2. Teoria da computação. 3. Inteligência artificial. 4. Raciocínio automático I. Freitas, Frederico Luiz Gonçalves de. (Orientador). II. Titulo.

004 CDD (22. ed.) UFPE-MEI 2017-270

Eunice Palmeira da Silva

Conversão de Provas em Lógica de Descrições ALC Geradas pelo Método de Conexões para Sequentes

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutora em Ciência da Computação

Aprovado em: 15/09/2017.

_____

Orientador: Prof. Dr. Frederico Luiz Gonçalves de Freitas

BANCA EXAMINADORA

_____

Profa. Dra. Anjolina Grisi de Oliveira
Centro de Informática / UFPE

_____

Prof. Dr. Edward Hermann Haeusler
Departamento de Informática / PUC-RJ

_____

Profa. Dra. Ana Teresa de Castro Martins
Departamento de Computação / UFC

_____

Prof. Dr. Ivan José Varzinczak
Centre de Recherche en Informatique de Lens
Univiversité d'Artois

_____

Prof. Dr. Evandro de Barros Costa
Instituto de Computação / UFAL

Dedico este trabalho aos meus pais, ***Sebastião Palmeira*** *(in memoriam) e* ***Joselita Ferreira****, pelo imenso amor a mim dedicado, pelos valores transmitidos, e pela sabedoria e empenho em me oferecer acesso a educação formal, um bem básico que lhes foi negado.*

Agradecimentos

Realizar uma tarefa como o doutorado exige dedicação, esforço e renúncias... e nessa longa trajetória, a presença, compreensão e colaboração, direta ou indireta, de amigos e familiares, contribuiram para que eu pudesse concluir esse trabalho. E por isso sou muito grata, e aqui devo registrar, não necessariamente em ordem, minha gratidão. Agradeço:

- a Deus, essa força maior que tenho percebido cada vez mais forte e mais presente em minha vida;

- aos meus orientadores, Fred Freitas e Jens Otten, que foram fundamentais nesse caminhar, com suas orientações, apoio, confiança e pelo tratamento franco, respeitoso e humanizado;

- aos meus familiares pela torcida, compreensão e apoio, especialmente a minha mãe, fiel companheira, que, sobretudo nos momentos mais difíceis, esteve comigo, me resguardou para que o foco no doutorado não fosse perdido;

- ao IFAL, por me permitir dedicação exclusiva aos estudos, após toda uma vida onde trabalho e estudo precisaram ser conciliados; aos colegas de trabalho do IFAL, que torceram e compreenderam a minha ausência nesse período;

- a CAPES, pelo financiamento dessa pesquisa junto ao IFAL;

- aos amigos do grupo 'Papo de Doutorado', Alixandre, Gláucia e Chico, pelas conversas animadas, encontros divertidos e apoio nos momentos próprios do doutorado e da vida;

- a Pasqueline Scaico, pelo estímulo, motivação e compartilhamento de momentos alegres e difíceis... uma irmã que encontrei nesse caminho;

- a Ruan, que de forma tão afável e didática me ajudou a dirimir dúvidas técnicas acerca de Lógica;

- a Diandra, por docemente trocar informações comigo sobre Sequentes;

- a Cledja e Jéssyka Flavyanne, que me deram apoio nos primeiros momentos da minha estadia em Recife, tornando menos solitário o meu primeiro período longe de casa;

- a Roosevelt, por compartilhar informações sobre a área de Direito, as quais foram úteis nos projetos da disciplina de Raciocínio Automático e Representação do Conhecimento Simbólico;

- a Bruno Maciel, sempre muito prestativo me ajudou a resolver problemas de instalação de ferramentas;

- a Felipe Alencar, Fernando Kenji e Rodrigo Leon, com quem compartilhei, de forma harmoniosa e amigável, moradias na minha estada em Recife;

- aos membros da banca de qualificação e defesa, que lançaram luz sobre pontos do trabalho provocando mudanças no mesmo;

- aos colegas do grupo de pesquisa SWORD, pela torcida e colaboração mútua;

- e, antes de finalizar um agradecimento especial ao meu amigo Chico, que com seu companheirismo, paciência e amizade, me mostrou Recife e me proporcionou dias mais felizes, regados a longas, divertidas e filosóficas conversas.

"Nada lhe posso dar que já não exista em você mesmo. Não posso abrir-lhe outro mundo de imagens, além daquele que há em sua própria alma. Nada lhe posso dar a não ser a oportunidade, o impulso, a chave. Eu o ajudarei a tornar visível o seu próprio mundo, e isso é tudo." Hermann Hesse - escritor e pintor alemão.

Resumo

O método de conexões ganhou boa reputação na área de prova automática de teoremas por cerca de três décadas, devido à sua simplicidade, clareza, eficiência e uso racional de memória. Este método recentemente tem sido aplicado em provadores automáticos que raciocinam sobre ontologias escritas em lógica de descrições $\mathcal{ALC}$. No entanto, as provas geradas por esse método são de difícil compreensão, consistindo em um conjunto de pares de conexões que são formados por fórmulas atômicas complementares encontradas ao longo de cada caminho de uma matriz. A legibilidade das provas é em grande parte perdida pelo ganho de desempenho e transformações aplicadas à fórmula a ser provada. Esse trabalho apresenta um método de conversão das provas em $\mathcal{ALC}$ geradas pelo método de conexões para um sistema de sequentes $\mathcal{ALC}$. Com a transformação para sequentes, obtém-se uma representação mais legível e inteligível. O método de conversão proposto aqui recebe a fórmula $\mathcal{ALC}$ e sua correspondente prova de conexões em formato não-clausal. Uma representação em árvore da fórmula $\mathcal{ALC}$ é construída e serve como guia no processo de conversão. À medida que a prova em conexões é percorrida, busca-se na árvore da fórmula os pares de literais complementares que formam as conexões; paralelamente a este processo, uma prova em sequentes vai sendo construída. Por fim, é apresentado o algoritmo que implementa o método de conversão, cuja complexidade sugere a viabilidade do método.

Palavras-chaves: Lógica de Descrições. *Attributive Concept Language with Complements* ($\mathcal{ALC}$). Método de Conexões (MC). Cálculo de Sequentes.

Abstract

The connection method earned good reputation in the field of automated theorem proving for around three decades, thanks to its simplicity, clarity, efficiency and parsimonious use of memory. It has recently been applied in automatic provers that reason over ontologies written in the description logics $\mathcal{ALC}$. However, its proofs are not very readable, consisting of a set of pairs of connections that are formed by complementary atomic formulas found in each path through a matrix. The readability is largely lost by the gain of performance and transformations applied to the formula to be proved. This work presents a conversion method to translate $\mathcal{ALC}$ connection proofs into $\mathcal{ALC}$ sequent proofs. With the translation into sequent, a more readable and intelligible representation is obtained. The conversion method proposed here receives the $\mathcal{ALC}$ formula and its corresponding connection proof in non-clausal form. A tree representation of the $\mathcal{ALC}$ formula is built and serves as a guide in the conversion process. As the connection proof is traversed, the pairs of complementary literals that form the connections are searched in the formula tree; in parallel to this process, a sequent proof is being built. Finally, the algorithm that implements the process is presented, of which the complexity suggests the viability of the method.

Key-words: Description Logics. *Attributive Concept Language with Complements* ($\mathcal{ALC}$). Connection Method. Sequent Calculus.

Lista de ilustrações

Lista de tabelas

Lista de símbolos

α	alpha
β	beta
γ	gama
Γ	Gama
δ	delta
Δ	Delta
θ	theta
Θ	Theta
σ	Sigma
μ	Mi
τ	tau
φ	varphi
ψ	psi
$\in$	pertence
$\forall$	para todo
$\exists$	existe

Sumário

1 INTRODUÇÃO

Lógica de Descrições (BAADER et al., 2003) é uma família de formalismos para representar conhecimento. Amplamente utilizada como linguagem para modelar ontologias, é considerada uma ferramenta fundamental para a Web Semântica (BERNERS-LEE; HENDLER; LASSILA, 2001). Ela constitui o formalismo subjacente à linguagem *Web Ontology Language (OWL)*. Lógica de Descrições é entendida como um subconjunto decidível de Lógica de Primeira Ordem, e é bem sucedida em várias aplicações devido à sua expressividade. Ela dá um significado preciso e inequívoco às descrições de um domínio, e sua semântica formal também permite o desenvolvimento de algoritmos para raciocínio automatizado, que podem ser usados para responder corretamente consultas arbitrariamente complexas sobre o domínio (HORROCKS, 2008).

A perspectiva de fornecer serviços de raciocínio, que automaticamente podem deduzir conhecimento implícito do conhecimento explicitamente representado, tem provocado o desenvolvimento de provadores automáticos de teorema para estes formalismos. Assim, existe uma tendência em oferecer esses serviços no processo de apoio à decisão, sobretudo em problemas complexos, como checagem de consistência e classificação.

A junção do poder de expressividade de Lógica de Descrições com o bom desempenho de provadores automáticos de teorema aponta para uma nova forma de oferecer serviços de raciocínio automático. A utilização dessa união como apoio no processo decisório exige que os resultados inferidos sejam apresentados em formato amigável, capaz de descrever como as conclusões foram obtidas. Nos Sistemas Baseados em Conhecimento essas descrições são extremamente relevantes, já que elas deixam claras as regras básicas do domínio usado pelos sistemas, o raciocínio percorrido pelos motores de inferência e a estratégia utilizada para chegar a uma conclusão.

Recentemente tem surgido alguns sistemas baseados no Método de Conexões, motivados pela rapidez e eficiência que o método apresenta na sua busca de prova.

No entanto, para sistemas baseados no Método de Conexões (BIBEL, 1987), oferecer uma forma legível de comunicar esses resultados esbarra no fato de suas provas serem de difícil compreensão, consistindo em um conjunto de pares de conexões encontradas ao longo de cada caminho de uma matriz. Esse modelo de prova impede que tais sistemas contribuam efetivamente na interação com seus utilizadores e forneçam descrições dos passos usados para inferências para apoiar as tomadas de decisões.

Diante disso, a motivação para esse trabalho é tornar provas em Lógica de Descrições $\mathcal{ALC}$ geradas pelo método de conexões em um formato mais legível e inteligível. Dedução Natural, Cálculo de Sequentes e Tableau, são alguns métodos que possuem formatos de provas mais legíveis que o método de conexões. Embora haja uma relação entre Cálculo de Sequentes e Dedução Natural, o primeiro possui uma abordagem mais estrutural e simétrica, favorecendo

a implementação algorítmica, ao contrário de Dedução Natural. Tableau, por outro lado, é um método de raciocínio baseado em refutação, isto é, para mostrar que uma fórmula é válida, a abordagem demonstra o fato de que essa fórmula não é satisfatível. Esse modelo, não favorece uma descrição da prova e o entendimento por parte dos usuários (BORGIDA; FRANCONI; HORROCKS, 2000). O cálculo de Sequentes é, em essência, um estilo de argumentação de lógica formal onde cada linha de uma prova é uma tautologia condicional. Consequentemente, isso deve contribuir para uma melhor interação com os utilizadores dos sistemas baseados no Método de Conexões, bem como para a inclusão desses sistemas no processo de apoio à decisão.

1.1 Método de Conexões

O Método de Conexões ganhou uma boa reputação na área de prova automática de teoremas por cerca de três décadas, devido à sua simplicidade, clareza, eficiência e uso racional de memória. Ele tem sido utilizado para lógicas clássicas e não-clássicas. O método representa fórmulas como matrizes, e sua prova consiste em percorrer caminhos através da matriz com o objetivo de conectar um literal L com o seu complementar $\neg L$. O par $\{L, \neg L\}$, é chamado de conexão, que corresponde à validade de um caminho. Assim, uma fórmula é válida se todo caminho através da matriz tem uma conexão.

Uma variante desse método é o Cálculo Não-clausal de Conexões para Lógica de Primeira Ordem (OTTEN, 2011), que não requer a transformação da fórmula de entrada para quaisquer forma normal, ao contrário do método original.

No grupo de pesquisa ao qual esse trabalho de tese faz parte, vários trabalhos vêm sendo desenvolvidos na área de provadores de teoremas, como o Cálculo de θ-Conexões $\mathcal{ALC}$ (FREITAS; OTTEN, 2016) e o raciocinador RACCOON (*Reasoner based on the Connection Calculus Over ONtologies*) (FILHO; FREITAS; OTTEN, 2017). O Cálculo de θ-Conexões $\mathcal{ALC}$ é baseado no Método de Conexões, e foi desenvolvido especificamente para inferir sobre Lógica de Descrições $\mathcal{ALC}$. O cálculo inclui técnicas e características típicas de Lógica de Descrições, como notação sem variáveis, ausência de funções de Skolem e unificação e inclusão de uma regra de bloqueio para lidar com ciclos, que garante o término no caso de ontologias cíclicas.

1.2 Questão de Pesquisa e Objetivos da Tese

- **Questão de Pesquisa:**
  Dado o cenário descrito acima e a motivação para esse trabalho, a questão de pesquisa que orienta esta tese é: Como encontrar uma maneira de converter provas em Lógica de Descrições $\mathcal{ALC}$ fornecidas pelo Método de Conexões em uma estrutura que possibilite uma compreensão mais clara dos passos de prova. Portanto, a principal questão subjacente a esta pesquisa é:

É possível tornar as provas em Lógica de Descrições $\mathcal{ALC}$ fornecidas pelo Método de Conexões mais compreensíveis?

- **Objetivo Geral da Tese:**
  O objetivo desta tese é propor um método para converter provas em Lógica de Descrições $\mathcal{ALC}$ geradas pelo Método de Conexões em uma representação mais compreensível.

  Para isso, esse trabalho busca os seguintes objetivos específicos:

  - Propor um Cálculo Não-clausal de Conexões para $\mathcal{ALC}$, a fim de trabalhar diretamente na estrutura da fórmula original, sem qualquer necessidade de conversão para forma normal disjuntiva;
  - Propor um método para converter provas em Lógica de Descrições $\mathcal{ALC}$ geradas pelo Cálculo Não-clausal de Conexões para Sequentes;
  - Elaborar um algoritmo para o método de conversão supracitado;
  - Avaliar a complexidade computacional da conversão.

1.3 Organização do Texto

O presente trabalho é assim constituído:

- O Capítulo 2 discorre sobre Lógica de Primeira Ordem e Lógica de Descrições, descrevendo conceitos subjacentes ao tema desse trabalho.
- O Capítulo 3 apresenta conceitos centrais de sistemas de prova, uma visão geral sobre Cálculo de Sequentes para Lógica de Primeiro Ordem e, por fim, apresenta um Cálculo de Sequentes para Lógica de Descrições $\mathcal{ALC}$.
- No Capítulo 4, o Cálculo de Conexões para $\mathcal{ALC}$ é apresentado, seguido pelo Cálculo Não-clausal de Conexões para $\mathcal{ALC}$.
- No Capítulo 5, é apresentado o método de conversão de provas em Lógica de Descrições $\mathcal{ALC}$ geradas pelo Cálculo Não-clausal de Conexões para Sequentes. O capítulo discorre sobre as definições do método e faz uma descrição do processo de conversão.
- O Capítulo 6 apresenta o algoritmo para o método de conversão e sua complexidade.
- No Capítulo 7, são levantadas as conclusões e contribuições do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Antes de apresentar os principais assuntos abordados nesta tese, é necessário cobrir as teorias subjacentes que dizem respeito a descrição do conhecimento sobre o qual os Sistemas Baseados em Conhecimento (SBC) operam. Para este fim, este capítulo introduz brevemente Lógica de Primeira Ordem (sub-seção 2.1) e Lógica de Descrições (sub-seção 2.2), resumindo alguns dos conceitos básicos dessas linguagens para melhor entender como uma base de conhecimento pode ser representada e como a partir dessa base os SBC tiram conclusões.

2.1 Lógica de Primeira Ordem

2.1.1 Introdução

A LPO (também chamada de lógica de predicados) é um sistema de lógica simbólica que estende a Lógica Proposicional adicionando os conceitos de *predicados* e *quantificadores*.

2.1.2 Sintaxe

O alfabeto da Lógica de Primeira Ordem assim como as regras que descrevem quais expressões são sintaticamente válidas são definidos a seguir. As definições em geral se baseiam em (VAN DALEN, 1994).

O alfabeto consiste dos símbolos:

- Símbolos de predicado: $P_1, \ldots, P_n, =$
- Símbolos de função: $f_1, \ldots, f_m$
- Símbolos de constante: c_i para $i \in I$
- Variáveis: $x_0, x_1, x_2, \ldots$
- Conectivos: $\vee, \wedge, \rightarrow, \neg, \leftrightarrow, \bot, \top, \forall, \exists$
- Símbolos auxiliares: (,),

$\forall$ e $\exists$ são chamados de quantificador *universal* e *existencial*, respectivamente;
I é o conjunto dos números irracionais.

Definição 1. ***(Termos).*** *Um conjunto T de* ***termos*** *é o menor conjunto X com as propriedades:*

(i) $c_i \in X\,(i \in I)$ *e* $x_i \in X\,(i \in N)$,

(ii) $t_1, \ldots, t_n \in X \Rightarrow f_i(t_1, \ldots, t_n) \in X$, *para* $1 \leq i \leq m$.

N é o conjunto dos números naturais.

Exemplo 1. *(Termos)*
$c, \quad x, \quad f(x), \quad f(x,y), \quad f(g(x)), \ldots$

Definição 2. ***(Fórmulas).*** *Um conjunto de* ***fórmulas*** φ *é o menor conjunto X com as propriedades:*

(i) $\bot \in X$; $P_i \in X$ *se* $r_i = 0$; $t_1, \ldots, t_{r_i} \in T \Rightarrow P_i(t_1, \ldots, t_{r_i}) \in X$; $t_1, t_2 \in T \Rightarrow t_1 = t_2 \in X$,

(ii) $\varphi, \psi \in X \Rightarrow (\varphi \square \psi) \in X$, *onde* $\square \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$,

(iii) $\varphi \in X \Rightarrow (\neg\varphi) \in X$,

(iv) $\varphi \in X \Rightarrow ((\forall x_i)\varphi), ((\exists x_i)\varphi) \in X$.

As fórmulas em (i) são chamadas de fórmulas atômicas;
r_i é uma relação de aridade n, onde $r_i \geq 0$.

Exemplo 2. *(Fórmulas)*
$Q(x,y), \quad P(f(x)), \quad \forall x(P(f(x)) \wedge Q(f(y),z)), \ldots$

Definição 3. ***(Literal, Literal Complementar).*** *Um* ***literal*** *é uma fórmula atômica ou sua negação. Para todo literal L, o* ***literal complementar*** *para L, denotado por* $\overline{L}$ *é definido como (ROBINSON; VORONKOV, 2001):*

$$\overline{L} = \begin{cases} A, & \text{se } L = \neg A \\ \neg L, & \text{caso contrário} \end{cases}$$

Definição 4. ***(Escopo do Quantificador).*** *O* ***escopo de um quantificador*** *é o segmento de uma fórmula para o qual quantificador se aplica. Em* $(\forall x)\varphi$ *e* $(\exists x)\varphi$, φ *é o escopo do quantificador. Se uma variável, termo ou fórmula ocorre em* φ, *então ele está no escopo do quantificador em* $\forall x\varphi$ *ou* $\exists x\varphi$.

Exemplo 3. *(Escopo do Quantificador)*
Seja a fórmula: $\forall x(P(x) \rightarrow \exists y(Q(y),z))$. *Nesta fórmula o escopo de* $\forall$ *é* $(P(x) \rightarrow \exists y(Q(y),z))$ *e o escopo de* $\exists$ *é (Q(y),z).*

Definição 5. ***(Conjunto de Variáveis Livres em Termos).*** *O* ***conjunto*** $VL(t)$ ***de variáveis livres ocorrendo em um termo*** t *é definido por:*

(i) $VL(x_i) := \{x_i\}$,
$VL(c_i) := \emptyset$

(ii) $VL(f(t_1, \ldots, t_n)) := VL(t_1) \cup \cdots \cup VL(t_n)$.

Definição 6. ***(Conjunto de Variáveis Livres em Fórmulas).*** *Seja φ uma fórmula. O* ***conjunto*** $VL(\varphi)$ ***de variáveis livres em uma fórmula*** φ *é definida por:*

(i) $VL(P(t_1, \ldots, t_p)) := VL(t_1) \cup \ldots \cup VL(t_p)$

$VL(t_1 = t_2) := VL(t_1) \cup VL(t_2)$

$VL(\bot) = VL(P) := \emptyset$, *onde P é um símbolo proposicional,*

(ii) $VL(\varphi \square \psi), := VL(\varphi) \cup VL(\psi)$

$VL(\neg\varphi), := VL(\varphi)$

(iii) $VL(\forall x_i \varphi) := VL(\exists x_i \varphi) := VL(\varphi) - \{x_i\}.$

Definição 7. ***(Conjunto de Variáveis Ligadas em Fórmulas).*** *Seja φ uma fórmula. O* ***conjunto*** $VG(\varphi)$ ***de variáveis ligadas em uma fórmula*** φ *é definida por:*

(i) $VG(P(t_1, \ldots, t_p)) := \emptyset$

$VG(t_1 = t_2) := \emptyset$

$VG(\bot) = VG(P) := \emptyset$, *onde P é um símbolo proposicional,*

(ii) $VG(\varphi \square \psi), := VG(\varphi) \cup VG(\psi)$

$VG(\neg\varphi), := VG(\varphi)$

(iii) $VG(\forall x_i \varphi) := VG(\exists x_i \varphi) := VG(\varphi) \cup \{x_i\}.$

Exemplo 4. *(Conjunto de Variáveis Livres/Ligadas)*
Seja a fórmula: $\forall x(P(x, y) \rightarrow \exists y Q(x, y))$. Nesta fórmula a primeira ocorrência de y é livre, enquanto as ocorrências de y em Q, e no quantificador existencial são ligadas; as ocorrências de x em P, em Q e no quantificador universal são ligadas.

Se x é uma variável, t um termo e φ uma fórmula, a notação $\varphi[x/t]$ significa a fórmula obtida da substituição de todas as ocorrência livres de x em φ por t.

Definição 8. ***(Substituição).*** *Seja t um termo e φ uma fórmula. Uma* ***substituição*** $\varphi[x/t]$ *é definida por:*

(i) $\bot\,[x/t] := \bot,$

$P[x/t] := P,$

$P(t_1, \ldots, t_p)[x/t] := P(t_1[x/t], \ldots, t_p[x/t]),$

$(t_1 = t_2)[x/t] := t_1[x/t] = t_2[x/t],$

(ii) $(\varphi \square \psi)[x/t] := \varphi[x/t] \square \psi[x/t],$

$(\neg\varphi)[x/t] := \neg\varphi[x/t]$

(iii) $(\forall y\varphi)[x/t]$ $\begin{cases} \forall y\varphi[x/t] \text{ se } x \not\equiv y \\ \forall y\varphi \text{ se } x \equiv y \end{cases}$

$(\exists y\varphi)[x/t]$ $\begin{cases} \exists y\varphi[x/t] \text{ se } x \not\equiv y \\ \exists y\varphi \text{ se } x \equiv y \end{cases}$

$x \equiv y$ significa "x e y são as mesmas variáveis", e $x \not\equiv y$, caso contrário.

O item (iii) na definição 8 proíbe a substituição de variáveis ligadas. No entanto, isso não restringe que uma variável livre se torne ligada após a substituição. Para esse caso algumas restrições são estabelecidas:

- A condição de uma ocorrência de variável não pode ser alterada após uma substituição. Ou seja, se antes da aplicação da substituição ela era livre, ela deve permanecer livre após a substituição;
- A variável não pode ocorrer no termo que a substitui. Por exemplo, o termo $f(x)$ não pode substituir a variável x.

Exemplo 5. *(Substituição)*
Seja φ *a fórmula* $\forall x(P(x) \wedge Q(x)) \rightarrow (\neg P(x) \vee Q(y))$.
$\varphi[x/f(x,y)]$ *é a fórmula* $\forall x(P(x) \wedge Q(x)) \rightarrow (\neg P(f(x,y)) \vee Q(y))$.

Definição 9. ***(Composição de Substituições).*** *Sejam as substituições* $\theta = \{x_1/t_1, \ldots, x_m/t_m\}$ *e* $\sigma = \{y_1/s_1, \ldots, y_n/s_n\}$, *onde* $m \geqslant 1$ *e* $n \geqslant 1$. *A composição destas substituições, denotada por* $\theta\sigma$, *é obtida por remover do conjunto* $\{x_1/t_1\sigma, \ldots, x_m/t_m\sigma, y_1/s_1, \ldots, y_n/s_n\}$ *todos os elementos* $x_i/t_i\sigma$ *para os quais* $x_i = t_i\sigma$, *e, além disso, todos os elementos* y_i/s_j *para os quais* $y_j \in \{x_1, \ldots, x_m\}$.

Exemplo 6. *(Composição de Substituições)*
Considere a expressão $E = Q(x, f(y), g(z,x))$ *e as duas substituições* $\sigma = \{x/f(y), y/z\}$ *e* $\theta = \{x/a, y/b, z/y\}$. *A composição de* $\sigma\theta$ *de* σ *e* θ *é dada por:* $\sigma\theta = \{x/f(b), z/y\}$. *A aplicação da composição da substituição* $\sigma\theta$ *para E produz a instância* $E(\sigma\theta) = Q(f(b), f(y), g(y, f(b)))$.

A substituição de variáveis por termos em fórmulas torna essas fórmulas sintaticamente iguais e possui um papel central em um método conhecido como unificação. A unificação compara expressões, computando a substituição que é necessária para tornar as expressões sintaticamente iguais. Quando essa substituição não existe, o método de unificação conclui que as expressões não são unificáveis. Mais precisamente, o objetivo do método de unificação, é definir um grupo de substituição, para um dado problema, que são permitidos ocorrer em cada fórmula do problema e com nenhum membro redundante. Esse grupo de substituição é denominado unificador mais geral, conforme definido abaixo.

Definição 10. ***(Unificador, Unificador Mais Geral).*** *Sejam* $\varphi_1, \ldots, \varphi_k$ *fórmulas atômicas. Um* ***unificador*** *para o conjunto* $\{\varphi_1, \ldots, \varphi_k\}$ *é uma substituição* σ *tal que* $\varphi_1\sigma = \varphi_2\sigma = \ldots = \varphi_k\sigma$, *onde* $=$ *representa a identidade semântica. Um unificador* σ *é dito ser um* ***unificador mais geral*** *se, para todo unificador* ι *para o mesmo conjunto, existe um unificador* ρ *tal que* $\iota = \sigma\rho$ *(BUSS, 1998).*

Exemplo 7. *(Unificador Mais Geral)*
i) Seja a fórmula $\varphi = \forall x(P(x,y) \rightarrow (\neg Q(y) \vee \exists y P(x,y)))$*, todas as variáveis livres em* φ *são substituídas por t, assim temos* $\varphi[y/f(y,z)] = \forall x(P(x,f(y,z)) \rightarrow (\neg Q(f(y,z)) \vee \exists y P(x,y)))$*. A substituição* $[y/f(y,z)]$ *é um unificador mais geral.*
ii) O termo $f(x,y)$ *é livre para substituir y em* $P(y,z)$ *na fórmula* $\psi = \forall x(P(x,z) \wedge \exists y Q(y)) \rightarrow P(y,z)$*, resultando em* $\psi[y/f(x,y)] = \forall x(P(x,z) \wedge \exists y Q(y)) \rightarrow P(f(x,y),z)$*, mas ele não é livre para substitutir z em* ψ*. Então os termos não são unificáveis.*

Em Lógica de Primeira Ordem, uma fórmula sem variáveis livres é chamada de *sentença de primeira ordem*. Essas sentenças são usadas para representar conhecimento.

Definição 11. ***(Forma Normal Prenex).*** *Uma fórmula* φ *está em forma (normal) prenex se* φ *consiste de uma cadeia de quantificadores (possivelmente vazia) seguida por uma fórmula aberta (isto é, livre de quantificador).* φ *também é chamada de fórmula prenex (VAN DALEN, 1994).*

Exemplo 8. *(Forma Normal Prenex)*
$\forall x \forall y \exists z(\neg P(x,y) \vee \neg Q(y,x) \vee P(z,z))$

Definição 12. ***(Função de Skolem).*** *Seja* φ *uma fórmula na linguagem L e* $\{x_1, \ldots, x_n, y\}$ *o conjunto de variáveis livres de* φ*. Associado a* φ *uma função* f_φ *de aridade n, chamada função de Skolem de* φ*. A sentença* $\forall x_1 \ldots x_n(\exists y \varphi(x_1, \ldots, x_n, y) \rightarrow \varphi(x_1, \ldots, x_n, f_\varphi(x_1, \ldots, x_n)))$ *é chamada de axioma de Skolem para* φ*. Se n = 0, então* f_φ *é uma constante (VAN DALEN, 1994).*

O método de **Skolemização**, bastante utilizado na prova automatizada de teoremas, elimina sistematicamente os quantificadores existenciais (resp. universais) em uma fórmula de primeira ordem na forma normal prenex. O método substitui ocorrências $Qx\varphi(x,y)$ de quantificadores por $\varphi(f(y),y)$, onde Q é uma quantificador e f é uma nova função em que não há outra ocorrência dela na fórmula. Os termos e/ou funções introduzidos durante o processo são chamadas de termos e/ou funções de Skolem.

Exemplo 9. *(Skolemização, Função de Skolem)*
i) O resultado da skolemização da fórmula $\exists x \forall y \forall z(P(x,y) \rightarrow Q(x,z))$*, eliminando os quantificadores existenciais, é* $\forall y \forall z(P(c,y) \rightarrow Q(c,z))$*. c é um termo de Skolem.*
ii) A skolemização da fórmula $\forall x \exists y \forall z \exists u A(x,y,z,u)$*, eliminando os quantificadores universais, é* $\exists y \exists u A(c,y,f(y),u)$*. c e* $f(y)$ *são um termo e uma função de Skolem, respectivamente.*

2.1.3 Semântica

A semântica da Lógica de Primeira Ordem diz respeito ao significado dos símbolos lógicos e não lógicos. O significado é entendido pela interpretação dada que especifica um domínio ou universo de objetos e atribui um valor de verdade às sentenças.

Definição 13. ***(Estrutura e Interpretação).*** *De acordo com (CHANG; KEISLER, 1990), uma estrutura $\mathfrak{U}$ para a linguagem L é um par $\mathfrak{U} = \langle \Delta, \mathcal{I} \rangle$, onde:*

- Δ denota o universo,
- $\mathcal{I}$ denota uma função de interpretação mapeando os símbolos de L para apropriadas relações, funções e constantes em Δ,
- Cada predicado P de aridade n corresponde a uma relação de aridade n $P^{\mathcal{I}} \subseteq \Delta^n$ em Δ,
- Cada função f de aridade m corresponde a uma função de aridade m $f^{\mathcal{I}} : \Delta^m \to \Delta$ em Δ,
- Cada símbolo constante c corresponde a uma constante $c^{\mathcal{I}} \in \Delta$.

Exemplo 10. *(Interpretação) Seja φ a fórmula: $\exists x(P(x) \wedge Q(x))$ e Δ o conjunto de animais tais que*

- em uma interpretação $\mathcal{I}$, P é interpretado como "é animal"e Q como "está em risco de extinção". Então φ é verdade se e somente se x é animal e x está em risco de extinção. Existe ao menos um animal que está em risco de extinção.
- em uma interpretação $\mathcal{I}'$, P é interpretado como "é vertebrado"e Q como "é mamífero". Então ϕ é verdade se e somente se x é um vertebrado e x é um mamífero. Existe ao menos um vertebrado que é mamífero.

Dado o conceito de interpretação, o significado de fórmulas lógicas de primeira ordem pode ser definido em termos de satisfação como a seguir.

Definição 14. ***(Satisfação).*** *Seja a linguagem L, $\mathfrak{U}$ uma estrutura para L e α uma atribuição em $\mathfrak{U}$. Para cada fórmula φ, a declaração $\mathfrak{U} \models \varphi[\alpha]$, φ é satisfeita por α em $\mathfrak{U}$, é definida como segue (CHANG; KEISLER, 1990; DOETS, 1996).*

- $\mathfrak{U} \models (t_1 = t_2)[\alpha]$ se e somente se $t_1^{\mathfrak{U}}[\alpha] = t_2^{\mathfrak{U}}[\alpha]$,
- $\mathfrak{U} \models R(t_1, \ldots, t_n)[\alpha]$ se e somente se $R^{\mathfrak{U}}(t_1^{\mathfrak{U}}[\alpha], \ldots, t_n^{\mathfrak{U}}[\alpha])[\alpha]$,
- $\mathfrak{U} \models \neg\varphi[\alpha]$ se e somente se $\mathfrak{U} \not\models \varphi[\alpha]$,
- $\mathfrak{U} \models (\varphi \wedge \psi)[\alpha]$ se e somente se $\mathfrak{U} \models \varphi[\alpha]$ e $\mathfrak{U} \models \psi[\alpha]$,
- $\mathfrak{U} \models (\varphi \vee \psi)[\alpha]$ se e somente se ao menos $\mathfrak{U} \models \varphi[\alpha]$ ou $\mathfrak{U} \models \psi[\alpha]$,
- $\mathfrak{U} \models (\varphi \to \psi)[\alpha]$ se $\mathfrak{U} \not\models \varphi[\alpha]$ ou $\mathfrak{U} \models \psi[\alpha]$,
- $\mathfrak{U} \models (\varphi \leftrightarrow \psi)[\alpha]$ se $\mathfrak{U} \models \varphi[\alpha]$ e $\mathfrak{U} \models \psi[\alpha]$ ou $\mathfrak{U} \not\models \varphi[\alpha]$ e $\mathfrak{U} \not\models \psi[\alpha]$,
- $\mathfrak{U} \models \exists x(\varphi)$ se e somente se para ao menos um $a \in \Delta$, $\mathfrak{U} \models \varphi_a^x$,

- $\mathfrak{U} \models \forall x(\varphi)$ se e somente se para todo $a \in \Delta$, $\mathfrak{U} \models \varphi_a^x$.

Uma **atribuição** é uma função que leva um conjunto de variáveis V de uma linguagem L ao universo Δ de $\mathfrak{U}$, denotada por: $a : V \rightarrow \Delta$.

Exemplo 11. *(Satisfação) Seja φ a fórmula: $\exists x(P(x) \wedge Q(x))$, Δ o conjunto de animais e uma interpretação $\mathcal{I}'$ onde P é interpretado como "é vertebrado"e Q como "é mamífero". Assim, existe ao menos um x que é vertebrado e mamífero. Então φ é satisfatível em Δ pois existe ao menos uma interpretação que é verdadeira.*

Definição 15. ***(Validade).*** *Uma fórmula φ é chamada válida (uma tautologia), denotada por $\models \varphi$, se e somente se $\mathfrak{U} \models \alpha$ para toda interpretação em $\mathfrak{U}$ (VAN DALEN, 1994).*

Exemplo 12. *(Validade) Seja a fórmula $\forall x(P(x) \vee \neg P(x))$, ela é verdadeira em todas as interpretações em uma dada $\mathfrak{U}$.*

2.2 Lógica de Descrições

Esta seção começa com uma breve introdução sobre Lógica de Descrições que são fragmentos de Lógica de Primeira Ordem. Na sequência é apresentada uma definição formal da sintaxe e da semântica da lógica de descrições básica $\mathcal{ALC}$. A base de conhecimento em $\mathcal{ALC}$ é abordada com exemplos e conclusões que podem ser extraídas dela. Por fim, problemas básicos de raciocínio são analisados.

2.2.1 Introdução

Lógica de Descrições é uma família de formalismos lógicos com semântica bem definida, bastante usados para representar conhecimento em um domínio. Cada linguagem de descrição é diferenciada pelos construtores que podem ser usados. Esse trabalho lida especificamente com $\mathcal{ALC}$, que é um tipo de lógica de descrições a qual inclui um conjunto de construtores básicos, como visto mais adiante.

Lógicas de Descrições permitem representar conceitos, instâncias, relações entre conceitos e entre conceitos e instâncias. Essa capacidade de representação do conhecimento é amplamente explorada em vários domínios e sua popularidade tornou-se evidente no contexto da Web Semântica como ferramenta poderosa na modelagem ontológica.

Seu formalismo lógico possibilita aos sistemas raciocinar sobre bases de conhecimento e inferir informações adicionais a partir do conhecimento explícito. Esse formalismo dá aos sistemas o poder de expressão e raciocínio, oferecendo apoio para solucionar problemas complexos como por exemplo, verificação de consistência de leis (FREITAS; CANDEIAS JR; STUCKENSCHMIDT, 2011), reutilização forense de análise de padrões de comportamento na vigilância visual (HAN; HUTTER; STECHELE, 2011), representação de terminologia médica (NOY et al., 2008), importação e reutilização de conceitos de múltiplos domínios (BORGIDA; SERAFINI, 2003).

Este potencial para resolver problemas complexos é devido ao fato de que as Lógicas de Descrições são fragmentos de Lógica de Primeira Ordem com a vantagem de sua inferência ser decidível (RIBEIRO, 2012). De acordo com (SATTLER; CALVANESE; MOLITOR, 2003; MORTIMER, 1975), uma possibilidade de definir fragmentos decidíveis de lógica de primeira ordem é restringir o conjunto de variáveis que são permitidas dentro de fórmulas e a aridade de símbolos de relação. A tradução de conceitos de Lógica de Descrições em fórmulas da lógica de predicado envolve predicados com aridade de no máximo 2 e embora algumas Lógicas de Descrições possam ser menos expressivas que L^2 e C^2 [1], $\mathcal{ALC}$ é tão expressiva quanto L^2 e portanto decidível.

$\mathcal{ALC}$ constitui a base para Lógica de Descrições muito mais expressivas, e de acordo com (COTTA, 2008) ela é simples mas poderosa o suficiente para definir ontologias não triviais, como descrito nas próximas seções.

2.2.2 Sintaxe e Semântica de $\mathcal{ALC}$

Apresentada pela primeira vez por Manfred e Smolka (1991), $\mathcal{ALC}$ é formalmente definida como uma tupla ordenada (N_C, N_R, N_O), onde N_C **representa um conjunto de conceitos** (símbolos de predicado unário), N_R **um conjunto de relações** (símbolos de predicado binário) e N_O **um conjunto de indivíduos** (constantes), instâncias de N_C e N_R. Sua linguagem compreende apenas os construtores:

- $\sqcap$ (conjunção),
- $\sqcup$ (disjunção),
- $\neg$ (negação),
- $\exists$ (restrição existencial) e
- $\forall$ (restrição de valor).

As definições abaixo estão de acordo com (BAADER; HORROCKS; SATTLER, 2008).

Definição 16. ***(Sintaxe** $\mathcal{ALC}$**).*** *O conjunto de conceitos* $\mathcal{ALC}$ *sobre* N_C *e* N_R *é definido de tal modo que:*

- $\top$ (*top concept*), $\bot$ (*bottom concept*) e cada conceito $A \in N_C$ são conceitos $\mathcal{ALC}$;
- se C e D são conceitos $\mathcal{ALC}$ e $R \in N_R$, então $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall R.C$, e $\exists R.C$ são conceitos $\mathcal{ALC}$.

[1] L^k denota lógica de predicado de primeira ordem sobre predicados unários e binários com no máximo k variáveis. C^k denota lógica de predicado de primeira ordem sobre predicados unários e binários com no máximo k variáveis e contagem de quantificadores, que tem a forma $\exists^{\geq n}$, $\exists^{\leq n}$, onde n é um índice numérico. $\exists^{\geq n} v\varphi$ é lido como "existe ao menos n v tal que φ"e $\exists^{\leq n} v\varphi$ como "existe no máximo n v tal que φ".

Exemplo 13. *(Sintaxe de $\mathcal{ALC}$)*

- *Animal*, *AnimalRiscoExtincao* e *Mamifero* são exemplos de conceitos que definem classes ou subconjuntos de objetos do universo;
- $AnimalRiscoExtincao \sqcap Mamifero$ descreve o conjunto de indivíduos que são ao mesmo tempo membros de *AnimalRiscoExtincao* e $Mamifero$;
- $AnimalRiscoExtincao \sqcup Mamifero$ descreve o conjunto indivíduos que são ao menos membros de *AnimalRiscoExtincao* ou $Mamifero$;
- $\neg AnimalRiscoExtincao$ descreve o conjunto complementar de *AnimalRiscoExtincao*, ou seja, o conjunto de indivíduos que não são membros de $AnimalRiscoExtincao$;
- $\forall trafica.AnimalRiscoExtincao$ define o conjunto de todos os indivíduos x tal que todos os elementos relacionados a x através de $trafica$ são membros de *AnimalRiscoExtincao*;
- $\exists trafica.AnimalRiscoExtincao$ define todos os indivíduos x tal que existe ao menos um indivíduo y, relacionado a x através de $trafica$, que é membro de $AnimalRiscoExtincao$.

Definição 17. ***(Semântica de $\mathcal{ALC}$).*** *Uma interpretação $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ sobre a tupla (N_C, N_R, N_O) consiste de um conjunto não vazio $\Delta^{\mathcal{I}}$, chamado o domínio de $\mathcal{I}$, e uma função $.^{\mathcal{I}}$ que mapeia*

- todo conceito $\mathcal{ALC}$ a um subconjunto de $\Delta^{\mathcal{I}}$,
- e toda relaçção a subconjunto de $\Delta^{\mathcal{I}}$ x $\Delta^{\mathcal{I}}$

tal que, para todos os conceitos $\mathcal{ALC}$ C, D e todas as relações R, o mapeamento $.^{\mathcal{I}}$ pode ser estendido para conceitos como abaixo:

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$
- $\bot^{\mathcal{I}} = \emptyset$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
- $(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}}, \text{ se } (x, y) \in R^{\mathcal{I}}, \text{ então } y \in C^{\mathcal{I}}\}$
- $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} \text{ com } (x, y) \in R^{\mathcal{I}} \text{ e } y \in C^{\mathcal{I}}\}$

É dito que $C^{\mathcal{I}}(R^{\mathcal{I}})$ é a extensão do conceito C (da função R) na interpretação $\mathcal{I}$. Se $x \in C^{\mathcal{I}}$, então é dito que x é uma instância de C em $\mathcal{I}$.

Exemplo 14. *(Semântica de $\mathcal{ALC}$)*
Considere o universo Δ^I, onde o universo do discurso trata sobre as relações com o meio ambiente.

- *AnimalRiscoExtincao* é um conceito de Δ^I, sua interpretação é $AnimalRiscoExtincao^I \subseteq \Delta^I$;
- CHITA é uma instância do universo, tal que $CHITA^I \in \Delta^I$ e CHITA é um animal em risco de extinção é declarado como *AnimalRiscoExtincao(CHITA)*;
- A relação $trafica$ relaciona dois indivíduos do universo, tal que $trafica^I \in (\Delta^I \times \Delta^I)$. Se JOAO é uma instância da classe Pessoa e ele caça CHITA, a relação pode ser declarada como $trafica(JOAO, CHITA)$.

2.2.3 Ontologia em $\mathcal{ALC}$

Em geral, as linguagens de descrição são formalismos para representar conhecimento em uma ontologia e raciocinar sobre ela. Para melhor representar uma ontologia, $\mathcal{ALC}$ utiliza dois componentes: conhecimento *terminológico*, chamado TBox, e conhecimento *assertivo*, chamado ABox. O primeiro define as características gerais dos conceitos e funções e como eles estão relacionados, representando o conhecimento sobre os grupos que têm as mesmas características. O segundo representa o conhecimento de cada indivíduo que é parte de um conjunto. Assim, como tratado formalmente em (BAADER; HORROCKS; SATTLER, 2008), uma ontologia é definida abaixo:

Definição 18. ***(Ontologia).*** *Uma ontologia é um par $(\mathcal{T}; \mathcal{A})$, onde $\mathcal{T}$ é uma TBox, e $\mathcal{A}$ é uma ABox.*

Definição 19. ***(TBox).*** *Uma TBox é um conjunto finito de inclusões de conceito geral da forma $C \sqsubseteq D$, onde C, D são conceitos $\mathcal{ALC}$. $C \equiv D$ é uma abreviação para o par simétrico de $C \sqsubseteq D$ e $D \sqsubseteq C$. Uma interpretação $\mathcal{I}$ é um modelo de inclusão de conceito geral $C \sqsubseteq D$ se $C^I \subseteq D^I$; $\mathcal{I}$ é um modelo de uma TBox $\mathcal{T}$ se $\mathcal{I}$ é um modelo de cada inclusão de conceito geral em $\mathcal{T}$.*

Um axioma da forma $A \equiv C$, onde A é um conceito, é chamado de *definição*.

Definição 20. ***(ABox).*** *Uma ABox é um conjunto finito de axiomas da forma $C(a)$ ou $R(b, c)$, onde C é um conceito $\mathcal{ALC}$, R é uma função, e a, b e c são indivíduos. Uma interpretação $\mathcal{I}$ é um modelo de um axioma $C(a)$ se $a^I \in C^I$, e $\mathcal{I}$ é um modelo de um axioma $R(b, c)$ se $(b^I, c^I) \in R^I$; $\mathcal{I}$ é um modelo de uma ABox A se $\mathcal{I}$ é um modelo de cada axioma em A.*

Exemplo 15. *(TBox e ABox)*

$$Animal \equiv Mamifero \sqcup Passaro \tag{2.1}$$

$$AnimalRiscoExtincao \equiv Animal \sqcap EspecieRiscoExtincao \tag{2.2}$$

$$AnimalSemRiscoExtincao \equiv Animal \sqcap \neg AnimalRiscoExtincao \tag{2.3}$$

$$Traficante \equiv Pessoa \sqcap \exists trafica.Animal \tag{2.4}$$

$$CriminosoAmbiental \equiv Pessoa \sqcap \exists trafica.AnimalRiscoExtincao \tag{2.5}$$

$$Pessoa(JOAO) \tag{2.6}$$

$$AnimalRiscoExtincao(CHITA) \sqcap Mamifero(CHITA) \tag{2.7}$$

$$trafica(JOAO, CHITA) \tag{2.8}$$

Considere o Exemplo 15, *Animal*, *Mamifero*, *Passaro*, *AnimalRiscoExtincao*, *AnimalSemRiscoExtincao*, *Traficante*, *Pessoa* and *CriminosoAmbiental* são conceitos; *trafica* é uma relação, e todos eles são definidos em TBox. Já as declarações (2.6), (2.7) e (2.8) são definidas em ABox. (2.7) descreve que *CHITA* é uma instância de *AnimalRiscoExtincao* e de *Mamifero*, já em (2.8) o par *JOAO* e *CHITA* é uma instância da função trafica. Em (2.1) um *Animal* é definido como sendo *Mamifero* ou *Passaro*. (2.4) expressa que um traficante é uma pessoa que trafica ao menos um animal e (2.5) que um *criminoso ambiental* é uma pessoa que trafica um animal em risco de extinção.

Ao raciocinar sobre estes conceitos, é possível inferir que *JOAO* traficou um animal em risco de extinção, logo *JOAO* é um criminoso ambiental. Esta é uma informação implícita que pode parecer intuitiva para os seres humanos, mas para os sistemas a sintaxe $C \equiv D$ em lógicas de descrições descreve D especificando condições necessárias e suficientes para instâncias de C. Então para ser um criminoso ambiental é necessário ser uma instância de classe *Pessoa* e ter pelo menos uma relação *trafica* com outro indivíduo que é membro de *AnimalRiscoExtincao*. Assim, os sistemas de prova inferem relações implícitas, que são geralmente mais complexas do que este exemplo, nas quais problemas de inferência relevantes são enfrentados. Tais problemas são descritos na próxima seção.

2.2.4 Inferências em $\mathcal{ALC}$

Inferir conteúdo implícito a partir de declarações explícitas em TBox e ABox exige algoritmos capazes de lidar com serviços de inferências básicas em Lógica de Descrições. As inferências

sobre expressões conceituais são satisfação, subsunção, equivalência e disjunção. Já o raciocínio para ABox, é verificar a consistência. Todos esses tipos de inferências são formalmente definidos abaixo como em (BAADER et al., 2003).

Seja $\mathcal{T}$ uma TBox como no Exemplo 15.

Definição 21. ***(Satisfação).*** *Um conceito C é satisfatível com respeiro a $\mathcal{T}$ se existe um modelo $\mathcal{I}$ de $\mathcal{T}$, tal que $C^{\mathcal{I}}$ é não vazio. Neste caso também é dito que $\mathcal{I}$ é um modelo de C.*

Exemplo 16. *(Satisfação) O conceito $AnimalRiscoExtincao \sqcap Mamifero$ é satisfatível, enquanto o conceito $AnimalRiscoExtincao \sqcap AnimalSemRiscoExtincao$ não é.*

Definição 22. ***(Subsunção).*** *Um conceito C é subsumido por um conceito D com respeito a $\mathcal{T}$ se $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ para todo modelo $\mathcal{I}$ de $\mathcal{T}$. Este caso é denotado por $\mathcal{T} \models C \sqsubseteq D$.*

Exemplo 17. *(Subsunção) $ManterEmCativeiro \sqsubseteq Acao \sqcap \exists feitaPor.Pessoa \sqcap \exists contra.Animal$. Esta subsunção declara que ManterEmCativeiro é uma entre, outras possíveis ações, praticada por pessoas contra os animais.*

Definição 23. ***(Equivalência).*** *Dois conceitos C e D são equivalentes com respeito a $\mathcal{T}$ se $C^{\mathcal{I}} = D^{\mathcal{I}}$ para todo modelo $\mathcal{I}$ de $\mathcal{T}$. Este caso é denotado por $\mathcal{T} \models C \equiv D$.*

Exemplo 18. *(Equivalência) $Traficante \equiv Pessoa \sqcap \exists trafica.Animal$. O conceito Traficante define indivíduos que são obrigatoriamente membros da classe Pessoa e caçam pelo menos um Animal.*

Definição 24. ***(Disjunção).*** *Dois conceitos C e D são disjuntos com respeito a $\mathcal{T}$ se $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ para todo modelo $\mathcal{I}$ de $\mathcal{T}$.*

Exemplo 19. *(Disjunção) $AnimalRiscoExtincao \sqcup AnimalSemRiscoExtincao$ são disjuntos, já que não deve haver animal que esteja ao mesmo tempo em risco e sem risco de extinção.*

Definição 25. ***(Consistência).*** *Uma ABox A é consistente com relação a $\mathcal{T}$, se houver uma interpretação que é um modelo de A e $\mathcal{T}$.*

Exemplo 20. *(Consistência) $\{AnimalRiscoExtincao(CHITA), Mamifero(CHITA)\}$ é um conjunto de afirmações consistente com $\mathcal{T}$, enquanto $\{Traficante(CHITA), Mamifero(CHITA)\}$ não é.*

3 SISTEMAS DE PROVA E CÁLCULOS DE SEQUENTES

Este capítulo começa com uma breve descrição de alguns dos conceitos centrais de sistemas de prova e provas. Na sequência é fornecida uma visão geral sobre Cálculo de Sequentes para Lógica de Primeira Ordem e por fim é apresentado um Cálculo de Sequentes para $\mathcal{ALC}$.

3.1 Sistemas de Prova e Provas em Geral

Definição 26. ***(Sistema de Prova).*** *Seja L uma linguagem e Λ um conjunto de fórmulas de L. Um* ***sistema de prova*** $\mathcal{S}$ *é um par $\mathcal{S} = (\varphi, \mathcal{R})$, onde $\varphi \subseteq \Lambda$ e $\mathcal{R}$ é um conjunto finito de regras de inferência (KFOURY; MOLL; ARBIB, 2012).*

Definição 27. ***(Prova Formal).*** *Uma* ***prova formal*** *(ou dedução) de φ a partir de um conjunto de fórmula Γ é uma sequência finita $\langle \alpha_0, \ldots, \alpha_n \rangle$ de fórmulas tal que α_n é φ e para cada $k \leq n$, ou*
(a) α_k está em $\Gamma \cup \Lambda$, onde Λ é um conjunto de axiomas, ou
(b) α_k é obtida por regra de inferência a partir de duas fórmulas anteriores na sequência; isto é, para algum i e j menor que k, α_j é $\alpha_i \rightarrow \alpha_k$ (ENDERTON; ENDERTON, 2001).

Se tal prova existe, é dito que φ **é dedutível de** Γ, ou φ **é consequência lógica de** Γ, ou ainda que φ **é um teorema de** Γ, escrito como $\Gamma \vdash \varphi$. O símbolo $\vdash$ é chamado de *turnstile* ou catraca. Algumas abordagens usadas pelos sistemas de prova para provar teoremas são:

- **Prova Direta:** Uma prova direta de uma sentença da forma $P \rightarrow Q$ é uma prova que supõe que P é verdadeiro e então mostra, usando regras de inferência, axiomas previamente aceitos, definições, e equivalências lógicas, que Q é verdadeiro (ROSEN, 1999).
- **Prova por Contraposição:** Para provar $P \rightarrow Q$ considere sua equivalente contrapositiva $\neg Q \rightarrow \neg P$ verdadeira. Se $\neg Q \rightarrow \neg P$ é demonstrada verdadeira, então $P \rightarrow Q$ também o é. (HAMMACK, 2009).
- **Prova por Contradição:** Para provar $P \rightarrow Q$, $P \rightarrow Q$ é suposta como falsa, (isto é, suponhe que P é verdadeira e Q é falsa). Se uma contradição é derivada, $P \rightarrow Q$ não pode ser falsa, e portanto deve ser verdadeira (ROSEN, 1999).

A seguir são definidas duas importantes propriedades para um sistema de dedução, conforme (ENDERTON; ENDERTON, 2001):

Definição 28. ***(Corretude do Sistema de Prova).*** *Seja* Γ *um conjunto de fórmulas, e* φ *uma fórmula a ser provada,*

$$se\ \Gamma \vdash \varphi\ então\ \Gamma \models \phi.$$

O teorema da corretude certifica que qualquer fórmula derivada em um sistema de prova é também verdadeira em todas as interpretações.

Definição 29. ***(Completude do Sistema de Prova).*** *Seja* Γ *um conjunto de fórmulas, e* φ *uma fórmula a ser provada,*

$$se\ \Gamma \models \varphi\ then\ \Gamma \vdash \varphi.$$

O teorema da completude declara que se Γ implica logicamente em φ então existe uma prova de φ a partir de Γ no sistema dedutivo. Em outras palavras, tudo que é semanticamente obtido pode ser também obtido no sistema dedutivo.

3.2 Cálculos de Sequentes

Esta seção apresenta o cálculo de sequentes para Lógica de Primeira Ordem seguido pelo cálculo de sequentes para $\mathcal{ALC}$.

3.2.1 Cálculo de Sequentes para Lógica de Primeira Ordem

O cálculo de sequentes foi definido originalmente por Gentzen (1934), como um meio de investigar as propriedades do sistema de Dedução Natural. Chamado de *LK* para lógica clássica de primeira ordem, o cálculo de sequentes é considerado um sistema elegante e flexível para escrever provas. De modo geral, as definições abaixo são com base em (GENTZEN, 1934):

Definição 30. ***(Sequente).*** *Um* ***sequente*** *é uma expressão da forma* $\Gamma \vdash \Delta$*, onde* Γ *e* Δ *são sequências finitas de fórmulas, como* $\Gamma = \{A_1, \ldots, A_\mu\}$ *e* $\Delta = \{B_1, \ldots, B_\nu\}$*. Ambas as expressões podem ser vazias.* Γ *forma o antecedente e* Δ *o sucedente do sequente.*

O sequente $\{A_1, \ldots, A_\mu\} \vdash \{B_1, \ldots, B_\nu\}$ é verdadeiro (em uma particular interpretação) se $A_1 \wedge \ldots \wedge A_\mu$ implica em $B_1 \vee \ldots \vee B_\nu$. Se o antecedente é vazio, o sequente se reduz à fórmula $B_1 \vee \ldots \vee B_\nu$. Se o sucedente é vazio, o sequente significa o mesmo que a fórmula $\neg(A_1 \wedge \ldots \wedge A_\mu)$ ou que $(A_1 \wedge \ldots \wedge A_\mu)$ implica em falso. Se ambas as partes são vazias, a proposição é falsa.

Definição 31. ***(Cálculo de Sequentes, Prova).*** *As provas ou deduções são rotuladas como árvores finitas com uma única raiz, com axiomas nos nós superiores e cada rótulo do nó conectado com os rótulos dos nós sucessores (imediatos) (se houver) de acordo com uma das regras. As regras são divididas em regras esquerda (l-) e direita (r-). Para um operador lógico* $\Box$*, l*$\Box$ *e r*$\Box$ *indicam as regras onde uma fórmula com* $\Box$ *como operador principal é introduzida no lado*

esquerdo e no lado direito, respectivamente. Os axiomas e as regras para cálculo de sequentes, dado A, B e C arbitrárias fórmulas e Γ, Δ, Σ, *e* Π *sequências arbitrárias de fórmulas, são (GENTZEN, 1934; TROELSTRA; SCHWICHTENBERG, 2000):*

Axioma:

$$\frac{}{A \vdash A} \ (Ax)$$

Regras Lógicas:

$$\frac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \ (l\wedge_1) \qquad \frac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \ (l\wedge_2) \qquad \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B} \ (r\wedge)$$

$$\frac{A, \Gamma \vdash \Delta \qquad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta} \ (l\vee) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} \ (r\vee_1) \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B} \ (r\vee_2)$$

$$\frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \to B \vdash C} \ (l\to) \qquad \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \to B} \ (r\to)$$

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta} \ (l\neg) \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A} \ (r\neg)$$

$$\frac{A[x/t], \Gamma \vdash \Delta}{\forall x A, \Gamma \vdash \Delta} \ (l\forall) \qquad \frac{\Gamma \vdash \Delta, A[x/y]}{\Gamma \vdash \Delta, \forall x A} \ (r\forall) \ *$$

$$\frac{A[x/y], \Gamma \vdash \Delta}{\exists x A, \Gamma \vdash \Delta} \ (l\exists) \ * \qquad \frac{\Gamma \vdash \Delta, A[x/t]}{\Gamma \vdash \Delta, \exists x A} \ (r\exists)$$

Regras Estruturais:

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \ (l\text{-w}) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta} \ (r\text{-w})$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \ (l\text{-c}) \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} \ (r\text{-c})$$

$$\frac{\Gamma, A, B, \Sigma \vdash \Delta}{\Gamma, B, A, \Sigma \vdash \Delta} \ (l\text{-p}) \qquad \frac{\Gamma \vdash \Delta, A, B, \Pi}{\Gamma \vdash \Delta, B, A, \Pi} \ (r\text{-p})$$

$$\frac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi} \ (Corte)$$

Restrição de autovariável: A variável nas regras $r\forall^*$ e $l\exists^*$, denotada por y, designada na literatura por autovariável (*Eigenvariable*), não deve ocorrer em nenhum lugar nos sequentes mais abaixo (isto é, não deve ocorrer em Γ, Δ e A).

A aplicação das regras ocorre da conclusão para a premissa e é chamada de **redução**.

Teorema 1 (Gentzen)**.** *Uma fórmula F é válida em lógica clássica se e somente se existe uma dedução para F no correspondente cálculo de sequentes.*

Exemplo 21. *(Prova no Cálculo de Sequentes - fórmula válida). Seja* $G = \vdash \forall x\neg\neg A(x) \rightarrow \neg\neg\forall x A(x)$*. Uma dedução para G usando o cálculo de sequentes é:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A(a) \vdash A(a)}Ax}{\vdash \neg A(a), A(a)}r\neg}{\neg\neg A(a) \vdash A(a)}l\neg}{\forall x\neg\neg A(x) \vdash A(a)}l\forall}{\forall x\neg\neg A(x) \vdash \forall x A(x)}r\forall}{\forall x\neg\neg A(x), \neg\forall x A(x) \vdash}l\neg}{\forall x\neg\neg A(x) \vdash \neg\neg\forall x A(x)}r\neg}{\vdash \forall x\neg\neg A(x) \rightarrow \neg\neg\forall x A(x)}r\rightarrow$$

Existe uma dedução para G, logo G é válida.

Exemplo 22. *(Prova no Cálculo de Sequentes - fórmula não-válida). Seja a fórmula* $G_1 = \exists x A(x) \rightarrow \forall x A(x)$

Uma tentativa de provar a validade de G_1 *é fornecida abaixo. Contudo não é possível construir a dedução completa devido às restrições para as regras* $r\forall$ *e* $l\exists$*. Perceba que após a aplicação de* $r\rightarrow$ *só há duas regras a serem aplicadas,* $l\exists$ *e* $r\forall$*. Se aplicada a regra* $l\exists$*, conforme o exemplo, fica impossibilitada a aplicação de* $r\forall$ *na sequência. O mesmo ocorre com* $l\exists$ *se* $r\forall$ *for aplicada após* $r\rightarrow$*. Logo,* G_1 *não é uma fórmula válida.*

$$\dfrac{\dfrac{\dfrac{}{A(a) \vdash \forall x A(x)}\cancel{r\forall}}{\exists x A(x) \vdash \forall x A(x)}l\exists}{\vdash \exists x A(x) \rightarrow \forall x A(x)}r\rightarrow$$

Ao propor o Cálculo de Sequentes, Gentzen demonstrou o teorema de eliminação do corte (ou *Gentzen's Hauptsatz*) para lógica clássica:

Teorema 2 (Teorema da Eliminação do Corte)**.** *Se um sequente é provável em lógica clássica, então ele é provável em lógica clássica sem corte.*

Teorema 3. *(Propriedade da Subfórmula) Em uma derivação normal (derivação sem passos supérfluos) de* $\Gamma \vdash \varphi$*, cada fórmula é uma subfórmula de uma hipótese em* Γ *ou de* φ *(VAN DALEN, 1994).*

Conforme demonstrado, o Cálculo de Sequentes para Lógica de Primeira Ordem possui um conjunto de regras de inferência simples que facilitam verificar a validade de suas deduções. Ele apresenta uma maneira otimizada de dedução sem fazer muitos saltos intuitivos, tornando a compreensão da prova mais fácil e a regra de corte pode ser eliminada da prova sem afetar qualquer sequente provável que use essa regra.

3.2.2 Cálculo de Sequentes para Subsunção em $\mathcal{ALC}$

De acordo com (BORGIDA; FRANCONI; HORROCKS, 2000), o cálculo de sequentes axiomatiza a relação de consequência lógica (*entailment*), e isto tem um paralelo óbvio com a relação de subsunção. Assim, Borgida, Franconi e Horrocks (2000) propõem um cálculo de sequentes para inferências de subsunção em $\mathcal{ALC}$.

Nessa proposta não existem as regras da implicação, dada a relação com a subsunção mencionada acima; os termos não são movidos de um lado para o outro do *turnstile* durante a prova, ao contrário do Cálculo de Sequentes para Lógica de Primeira Ordem, preservando assim a estrutura da subsunção original. Para isso, regras adicionais foram criadas nas quais a negação é inserida na frente de cada construto e assim eliminam-se as regras de negação (l¬, r¬), as quais exigem mudança dos antecedentes do sequente para sucedentes e vice-versa.

A figura 1 mostra o cálculo organizado em três partes, onde as duas primeiras descrevem conjuntos de regras, enquanto a última descreve um conjunto de axiomas ($\wedge$ e $\vee$ são substituídos por seus correspondentes $\sqcap$ e $\sqcup$ em $\mathcal{ALC}$):

- **Regras para fórmulas proposicionais:** as regras $\sqcap$ e $\sqcup$ são duplicadas pela adição das regras de negação ($\neg\sqcap$, $\neg\sqcup$), enquanto as regras $\neg$ são modificadas para incluir mais uma negação ($\neg\neg$), se tornando duplamente negadas;

- **Regras para fórmulas quantificadas:** as regras com quantificadores também possuem a negação na frente do $\forall$ e $\exists$, gerando as regras $l\neg\forall$ e $r\neg\exists$. Além disso, uma condição é explicitamente considerada para a aplicação das regras $\forall$ e $\exists$. A condição declara que a regra é aplicável se todas as fórmulas universais e existenciais homólogas estão 'reunidas' juntas nos lados esquerdo e direito do sequente na pré-condição; a regra é então aplicada uma única vez;

- **Axiomas de terminação:** neste cálculo existem seis possíveis axiomas de terminação, ao contrário do cálculo de sequentes padrão no qual todos os axiomas de terminação podem ser reduzidos a $X, a \vdash a, Y$ pela aplicação das regras $\neg$. A aplicação das regras $\neg$ força a mudança das fórmulas do antecedente do sequente para o sucedente ou vice e versa até chegar a $X, a \vdash a, Y$, procedimento que é evitado nesse cálculo. Portanto, os axiomas adicionais de terminação são necessários para garantir que as fórmulas nunca sejam deslocadas de um lado para o outro do sequente.

Regras para fórmulas proposicionais

$$\frac{X, a, b \vdash Y}{X,\ a \sqcap b \vdash Y}\ (l\sqcap) \qquad \frac{X \vdash a, Y \quad X \vdash b, Y}{X, \vdash a \sqcap b, Y}\ (r\sqcap)$$

$$\frac{X, \neg a \vdash Y \quad X, \neg b \vdash Y}{X, \neg(a \sqcap b) \vdash Y}\ (l\neg\sqcap) \qquad \frac{X \vdash \neg a, \neg b, Y}{X \vdash \neg(a \sqcap b), Y}\ (r\neg\sqcap)$$

$$\frac{X,\ a \vdash Y \quad X,\ b \vdash Y}{X,\ a \sqcup b \vdash Y}\ (l\sqcup) \qquad \frac{X \vdash a, b, Y}{X \vdash a \sqcup b,\ Y}\ (r\sqcup)$$

$$\frac{X, \neg a, \neg b \vdash Y}{X, \neg(a \sqcup b) \vdash Y}\ (l\neg\sqcup) \qquad \frac{X \vdash \neg a, Y \quad X \vdash \neg b, Y}{X \vdash \neg(a \sqcup b), Y}\ (r\neg\sqcup)$$

$$\frac{X,\ a \vdash Y}{X, \neg\neg a \vdash Y}\ (l\neg\neg) \qquad \frac{X \vdash a, Y}{X \vdash \neg\neg a, Y}\ (r\neg\neg)$$

Regras para fórmulas quantificadas

$$\frac{X' \vdash b, Y'}{X \vdash \forall r.b,\ Y}\ (r\forall) \qquad \frac{X', b \vdash Y'}{X,\ \exists r.b \vdash Y}\ (l\exists)$$

$$\frac{X',\ \neg b \vdash Y'}{X, \neg\forall r.b \vdash Y}\ (l\neg\forall) \qquad \frac{X' \vdash \neg b, Y'}{X \vdash \neg\exists r.b, Y}\ (r\neg\exists)$$

onde $X' = \{a \mid \forall r.a \in X\} \cup \{\neg a \mid \neg\exists r.a \in X\}$, e
$Y' = \{a \mid \exists r.a \in Y\} \cup \{\neg a \mid \neg\forall r.a \in Y\}$

Axiomas de Terminação

$$X,\ a \vdash a,\ Y\ (=) \qquad X, \neg a \vdash \neg a,\ Y\ (=)$$
$$X,\ a,\ \neg a \vdash Y\ (\mathrm{l}\uparrow) \qquad X \vdash a,\ \neg a,\ Y\ (\mathrm{r}\uparrow)$$
$$X, \bot \vdash Y\ (\mathrm{l}\bot) \qquad X \vdash \top,\ Y\ (\mathrm{l}\top)$$

Figura 1 – Regras para $\mathcal{ALC}$ (BORGIDA; FRANCONI; HORROCKS, 2000).

No cálculo de primeira ordem, a propriedade da subfórmula depende do teorema da eliminação do corte, aqui é preciso lembrar o teorema abaixo.

Proposição 1. *Teorema da Eliminação do Corte (GIRARD; LAFONT; TAYLOR, 1989) conforme citado por Royer e Quantz (1992). Seja S um conjunto de sequentes (axiomas) e s um sequente individual. $S \vdash_{SC} s$, se e somente se, existe uma prova em SC de s cujas folhas ou são axiomas lógicos ou sequentes obtidos pela substituição de sequentes pertencentes a S, onde a regra do corte é somente aplicada com uma premissa sendo um axioma.*

Para ilustrar um procedimento de prova usando o cálculo acima, considere os exemplos abaixo com suas respectivas provas representadas em uma abordagem *bottom-up*:

Exemplo 23. *(Prova no Cálculo de Sequentes para $\mathcal{ALC}$). Considere a fórmula F_1:*
$\{\exists hasPet.Cat \sqsubseteq CatOwner, OldLady \sqsubseteq \exists hasPet.Animal \sqcap \forall hasPet.Cat\} \models OldLady \sqsubseteq CatOwner$

Observação 1. *Por uma questão de espaço os nomes dos literais foram abreviados na prova abaixo, assim como ocorre em outras partes do texto que se referem a F_1. Assim, hasPet é representado por h, OldLady por OL, Cat por C, CatOwner por CO e Animal por A.*

$$\dfrac{\dfrac{OL \vdash \exists h.A \sqcap \forall h.C \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{TRUE}}{A, C \vdash C}=}{\exists h.A, \forall h.C \vdash \exists h.C}\,l\exists}{\exists h.A \sqcap \forall h.C \vdash \exists h.C}\,l\sqcap}{OL \vdash \exists h.C}\,\text{corte} \qquad \exists h.C \vdash CO}{OL \vdash CO}\,\text{corte}$$

Figura 2 – Prova em sequentes $\mathcal{ALC}$ para F_1.

Exemplo 24. *(Prova no Cálculo de Sequentes para Subsunção em $\mathcal{ALC}$) Considere F_2 o axioma de subsunção $\mathcal{ALC}$:*

$$\exists child.\top \sqcap \forall child.\neg((\exists child.\neg Doctor) \sqcup (\exists child.Lawyer)) \sqsubseteq \exists child.\forall child.(Rich \sqcup Doctor)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{TRUE}}{Doctor, \neg Lawyer \vdash Rich, Doctor}=}{Doctor, \neg Lawyer \vdash Rich \sqcup Doctor}\,r\sqcup}{\neg\neg Doctor, \neg Lawyer \vdash Rich \sqcup Doctor}\,l\neg\neg}{\top, \neg(\exists child.\neg Doctor), \neg(\exists child.Lawyer)) \vdash \forall child.(Rich \sqcup Doctor)}\,r\forall}{\top, \neg((\exists child.\neg Doctor) \sqcup (\exists child.Lawyer)) \vdash \forall child.(Rich \sqcup Doctor)}\,l\neg\sqcup}{\exists child.\top, \forall child.\neg((\exists child.\neg Doctor) \sqcup (\exists child.Lawyer)) \vdash \exists child.\forall child.(Rich \sqcup Doctor)}\,l\exists}{\exists child.\top \sqcap \forall child.\neg((\exists child.\neg Doctor) \sqcup (\exists child.Lawyer)) \vdash \exists child.\forall child.(Rich \sqcup Doctor)}\,l\sqcap$$

Figura 3 – Prova em sequentes $\mathcal{ALC}$ para F_2.

Subsunção é considerada a inferência chave em Lógica de Descrições e essa proposta foca nas relações de subsunção de conceitos $\mathcal{ALC}$ usando um cálculo de sequentes modificado. Borgida, Franconi e Horrocks (2000) ressaltam que a principal vantagem desse cálculo é preservar a estrutura original dos dois conceitos e que os conceitos nunca são transferidos de antecedente para sucedente ou vice-versa.

4 CÁLCULOS DE CONEXÕES

Este capítulo descreve dois cálculos baseados no Método de Conexões introduzido por Bibel (1987). Ambos os cálculos são para Lógica de Descrições $\mathcal{ALC}$, sendo um clausal e o outro não-clausal. A primeira seção (4.1) descreve os principais conceitos, a formalização e o funcionamento do cálculo de conexões clausal, enquanto a segunda (4.2) trata do não-clausal.

4.1 Cálculo Clausal de Conexões para $\mathcal{ALC}$

Antes de apresentar propriamente o cálculo de conexões para $\mathcal{ALC}$, é preciso dar uma visão geral sobre o Cálculo de Conexões (ou Método de Conexões) introduzido por Bibel (1987).

O Cálculo de Conexões é considerado simples e eficiente (BIBEL, 1987; OTTEN; KREITZ, 1995). Ele trabalha com fórmulas em Forma Normal Disjuntiva (FND) (aqui chamada pelo termo que a generaliza, forma clausal). As fórmulas que não estão nesta forma devem ser convertidas em forma clausal para este cálculo. A forma clausal tem a forma $C_1 \sqcup \ldots \sqcup C_n$, onde cada C_i é uma cláusula. Cada cláusula é uma conjunção de literais na forma $L_1 \sqcap \ldots \sqcap L_m$ e cada L_j é um literal. Uma fórmula em forma clausal pode ser escrita como um conjunto de cláusulas $\{C_1, \ldots, C_n\}$, chamado de matriz. A matriz representa a disjunção de suas cláusulas. Na representação gráfica da matriz, suas cláusulas são dispostas horizontalmente, enquanto os literais de cada cláusula são dispostos verticalmente.

O processo de verificação de validade do Cálculo de Conexões consiste em percorrer caminhos através da matriz, com o objetivo de conectar um literal L ao seu complementar $\neg L$, que estão em diferentes cláusulas. Um caminho é uma disjunção de literais, na forma $L_1 \sqcup \ldots \sqcup L_n$. Se um caminho contém um literal L e o seu complementar, esse caminho terá $L \sqcup \neg L$. Seja qual for o valor verdade de L na fórmula, um dos dois literais será verdadeiro e, portanto, sua disjunção será verdadeira. Isso é equivalente a encontrar um par de literais complementares nas cláusulas de uma fórmula na Forma Normal Conjuntiva (FNC)[1] não negada. Os dois literais conectados formam uma conexão. Uma conexão define a validade de um ou mais caminhos. Logo, uma fórmula é válida se cada caminho através da sua matriz tem uma conexão.

Tendo posto isso, as próximas seções apresentam, de acordo com Freitas (2011) e Freitas e Otten (2016), o Cálculo Clausal de Conexões para $\mathcal{ALC}$ desenvolvido para realizar inferências especificamente sobre Lógica de Descrições $\mathcal{ALC}$. O cálculo inclui técnicas e características típicas de Lógica de Descrições, como notação sem variáveis, ausência de funções de Skolem e unificação e inclusão de uma regra de bloqueio para lidar com ciclos, que garante o término de ontologias cíclicas.

[1] Uma fórmula $\mathcal{ALC}$ na Forma Normal Conjuntiva (FNC) é uma conjunção de disjunções (como $C_1 \sqcap \ldots \sqcap C_n$), onde cada C_i é uma cláusula. Uma cláusula é uma disjunção de literais na forma $L_1 \sqcup \ldots \sqcup L_m$ e cada L_i é um literal.

4.1.1 Forma Normal e Matriz para $\mathcal{ALC}$

Definição 32. ***(Literal em $\mathcal{ALC}$). Literais*** *(em $\mathcal{ALC}$) são conceitos atômicos ou relações, possivelmente negados e/ou instanciados.*

Exemplo 25. *(Literal em $\mathcal{ALC}$)*
Com base no exemplo 15, Animal, $\neg AnimalRiscoExtincao$, $\neg trafica$, $Mamifero(CHITA)$ e $trafica(JOAO, CHITA)$ são alguns exemplos de literais em $\mathcal{ALC}$.

Definição 33. ***(Disjunção $\mathcal{ALC}$).*** *Uma **disjunção** $\mathcal{ALC}$ ou é um literal L, uma disjunção $(E_0 \sqcup E_1)$ ou uma restrição universal $\forall r.E_0$, onde E_0 e E_1 são expressões de conceitos arbitrárias.*

Exemplo 26. *(Disjunção $\mathcal{ALC}$)*
$Animal$, $\forall temPet.Gato$, $(\exists mata.AnimalRiscoExtincao) \sqcup (\exists pesca.Animal \sqcap \neg \exists tem.Autorizacao)$

Definição 34. ***(Conjunção $\mathcal{ALC}$).*** *Uma **conjunção** $\mathcal{ALC}$ ou é um literal L, uma conjunção $(E_0 \sqcap E_1)$ ou uma restrição existencial $\exists r.E_0$, onde E_0 e E_1 são expressões de conceitos arbitrárias.*

Exemplo 27. *(Conjunção $\mathcal{ALC}$)*
$\exists temPet.(Gato \sqcap Cachorro)$, $(Mamifero \sqcup Reptil) \sqcap Aquatico \sqcap AnimalSemRiscoExtincao$

Definição 35. ***(Impureza, Disjunção/Conjunção Pura). Impureza*** *em uma fórmula $\mathcal{ALC}$ é uma disjunção em uma conjunção, ou uma conjunção em uma disjunção. Uma **disjunção/conjunção pura** é uma disjunção/conjunção que não contém impurezas.*

Exemplo 28. *(Impureza, Disjunção Pura, Conjunção Pura)*

(i) $\exists r.A$ *e* $\bigwedge_{i=1}^{n} A_i$ *são conjunções puras se A e cada A_i são também uma Conjunções Puras.*

(ii) $(\forall r.(D_0 \sqcup \ldots \sqcup D_n \sqcup (C_0 \sqcap \ldots \sqcap C_m) \sqcup (A_0 \sqcap \ldots \sqcap A_p)))$ *não é uma Disjunção Pura pois ela contém duas impurezas:* $(C_0 \sqcap \ldots \sqcap C_m)$ *e* $(A_0 \sqcap \ldots \sqcap A_p)$.

Definição 36. ***(Forma Normal Disjuntiva (FND) em $\mathcal{ALC}$).*** *Uma fórmula $\mathcal{ALC}$ na **forma normal disjuntiva** ou forma clausal é uma disjunção de conjunções (como $C_1 \sqcup \ldots \sqcup C_n$), onde cada C_i é uma cláusula. Uma cláusula é uma conjunção pura de literais na forma $L_1 \sqcap \ldots \sqcap L_m$ e cada L_i é um literal.*

Exemplo 29. *(FND em $\mathcal{ALC}$)*
$(Pessoa \sqcap \neg Mamifero) \sqcup (\neg Pessoa) \sqcup (CriminosoAmbiental)$, $\forall temPet.(Gato \sqcap Cachorro \sqcap Peixe)$, $(\exists protege.Animal) \sqcup (\exists pesca.Animal \sqcap \exists tem.Autorizacao)$

Definição 37. ***(Forma Normal da Negação (FNN) em $\mathcal{ALC}$).*** *Uma fórmula $\mathcal{ALC}$ está na **forma normal da negação** se a negação $(\neg)$ é aplicada apenas na frente de conceitos (atômicos ou não). Um conceito $\mathcal{ALC}$ arbitrário pode ser transformado em uma FNN equivalente, usando uma combinação das leis de De Morgan (ver tabela 14) e a dualidade entre restrições existenciais e universais $\neg \exists r.C \equiv \forall r.\neg C$ e $\neg \forall r.C \equiv \exists r.\neg C$ (BAADER et al., 2003).*

Exemplo 30. *(FNN em $\mathcal{ALC}$)*
$(\neg\exists r.C)$ não está em FNN, mas sua equivalente $(\forall r.\neg C)$ está; $\neg(\exists r.A \sqcap \forall s.B)$, onde A e B são conceitos, pode ser transformada na equivalente FNN $\forall r.\neg A \sqcup \exists s.\neg B$.

Uma forma normal disjuntiva, chamada de Forma Normal Disjuntiva com Duas Linhas, foi criada especificamente para esse cálculo. A motivação, segundo seus autores, é que essa forma poupa memória por evitar redundâncias na matriz, e ajuda a provar a corretude, completude e terminação do sistema, restringindo os casos problemáticos a colunas com duas linhas. Nessa forma normal, as linhas representam as retrições existenciais e universais. Como o cálculo não usa variáveis nem funções de Skolem, as linhas são usadas para indicar quais literais possuem restrições. A Forma Normal Disjuntiva com Duas Linhas é formalmente definida abaixo e exemplos são dados na sequência para melhor compreensão dessa forma normal.

Definição 38. ***(Forma Normal Disjuntiva com Duas Linhas).*** *Um axioma $\mathcal{ALC}$ está na **forma normal disjuntiva com duas linhas** se e somente se está na FND e em uma das seguintes formas normais:*

(i) $\hat{E} \sqsubseteq \check{D}$;

(ii) $E \sqsubseteq \hat{E}$; *e*

(iii) $\check{D} \sqsubseteq E$, *onde E é um conceito*[2], $\hat{E}$ *é uma conjunção pura* $(\hat{E} = \bigsqcap_{i=1}^{n} C_i)$, *e* $\check{D}$ *é uma disjunção pura* $(\check{D} = \bigsqcup_{j=1}^{m} D_j)$.

Exemplo 31. *(Forma Normal Disjuntiva com Duas Linhas)*
A tabela 1 mostra três exemplos de fórmulas $\mathcal{ALC}$ contendo restrições de quantificação. Na representação gráfica da matriz (ver definições 39 e 40), linhas verticais representam restrições existenciais $(\exists r.C)$, linhas horizontais representam restrições universais $(\forall r.C)$ no lado esquerdo da sub-fórmula do axioma ou o oposto no lado direito. As linhas podem se sobrepor. Note também que, quando escritas em Lógica de Primeira Ordem (LPO) (ver a tradução para LPO na tabela 15), as funções de Skolem devem aparecer nas duas últimas formas normais na tabela 1 (por exemplo, $\neg r(x, f(x))$ substituiria $\exists y \ldots \neg r(x, y)$).

Observação 2. *Para alcançar essas formas, novos símbolos podem ser introduzidos. Esses símbolos podem não ocorrer na fórmula original. No entanto, os TBoxes normalizados são extensões conservativas*[3] *(GHILARDI; LUTZ; WOLTER, 2006) de seus originais, uma vez que a cada modelo do primeiro existe um modelo (às vezes distinto) do último, e a validade é preservada.*

[2] Os símbolos E e $\hat{E}$ foram escolhidos aqui para designar um nome de conceito e uma conjunção pura ao invés do habitual C e $\hat{C}$, para evitar confusão com cláusulas, que também são indicados por C.

[3] Formalmente, uma ontologia *O'* é uma extensão conservativa de uma ontologia O, se e somente se, toda subsunção de O envolvida por *O'* já está envolvida por O (de modo equivalente, se cada conceito de O que é insatisfatível com respeito a *O'* já é insatisfatível com relação a O)(GHILARDI; LUTZ; WOLTER, 2006).

Tabela 1 – Exemplos de restrições de quantificação

Axioma	**Matriz**	**Mapeamento para LPO Negada**
$\exists r.\hat{E} \sqsubseteq \forall s.\check{D}$ onde $\hat{E}$ é conjunção pura e $\check{D}$ disjunção pura	$\left[\begin{array}{c\|c} r & \\ E_1 & \\ \ldots & \\ E_n & \\ s & \\ \neg D_1 & \\ \ldots & \\ \neg D_m & \end{array}\right]$	$\exists x \exists y \exists z$ $(r(x,y) \wedge$ $E_1(y) \wedge \ldots \wedge E_n(y)$ $\wedge$ $(s(x,z) \wedge$ $\neg D_1(z) \wedge \ldots \wedge \neg D_m(z))$
$A \sqsubseteq \exists r.\hat{E}$ A é um nome de conceito e $\hat{E}$ uma conjunção pura	$\left[\begin{array}{cccc} A & \ldots & \ldots & A \\ \underline{\neg r} & \underline{\neg E_1} & \ldots & \underline{\neg E_n} \end{array}\right]$	$\exists x \forall y((A(x) \wedge \neg r(x,y))$ $\vee (A(x) \wedge \neg E_1(y))$ $\vee \ldots \vee (A(x) \wedge \neg E_n(y)))$
$\forall r.\check{D} \sqsubseteq A$ A é um nome de conceito e $\check{D}$ uma disjunção pura	$\left[\begin{array}{cccc} \underline{\neg r} & \underline{D_1} & \ldots & \underline{D_m} \\ \neg A & \ldots & \ldots & \neg A \end{array}\right]$	$\forall x \exists y$ $(\neg r(x,y) \wedge \neg A(x)) \vee$ $(D_1(y) \wedge \neg A(x)) \vee \ldots \vee$ $(D_m(y) \wedge \neg(A(x))$

Exemplo 32. *(Introdução de símbolos)*
Na representação gráfica da matriz dada a seguir, o literal A é um símbolo novo, introduzido para transformar a fórmula abaixo em forma normal com duas linhas.

$$\{W \sqcap \exists hC.P \sqsubseteq Mo, Mo \sqcap \forall hC.He \sqsubseteq Ha, W(a), hC(a,b), P(b), He(b)\} \models Ha(a)$$

$$\left[\begin{array}{c\|cccccccc} W & Mo & \neg A & \neg A & \neg W(a) & \neg hC(a,b) & \neg P(b) & \neg He(b) & Ha(a) \\ hC & \neg Ha & \underline{\neg hC_1} & \underline{He_1} & & & & & \\ P & A & & & & & & & \\ \neg Mo & & & & & & & & \end{array}\right]$$

Definição 39. ***(Matriz $\mathcal{ALC}$).*** *A **matriz** ($\mathcal{ALC}$) de uma fórmula $\mathcal{ALC}$ na FND é a sua representação como um conjunto $\{C_1, \ldots, C_n\}$, onde cada cláusula C_i tem a forma $\{L_1, \ldots, L_m\}$ com literais L_i. Literais envolvidos em uma restrição universal ($\forall r.C$) ou em uma restrição existencial ($\exists r.C$) são sublinhados na matriz da fórmula. Quando uma restrição envolve mais de uma cláusula, seus literais são indexados com o mesmo índice na coluna da matriz.*

Exemplo 33. *(Matriz $\mathcal{ALC}$)*

$$\{\{\underline{hasPet},\ \underline{Cat},\ \neg CatOwner\},\ \{OldLady, \underline{\neg hasPet_1}\},\ \{OldLady,\ \underline{\neg Animal_1}\},$$
$$\{OldLady,\ \underline{hasPet},\ \underline{\neg Cat}\},\ \{\neg OldLady(a)\},\ \{CatOwner(a)\}\}$$

Definição 40. ***(Representação Gráfica (Positiva) da Matriz $\mathcal{ALC}$).*** *Em uma **representação gráfica da matriz**, as cláusulas são colunas, as restrições com índices são representadas por*

linhas contínuas horizontais, enquanto as restrições sem índices por linhas contínuas verticais. A representação gráfica é dita positiva, quando as cláusulas são representadas como colunas.

Observação 3. ***(Representação Gráfica (Positiva) da Matriz $\mathcal{ALC}$).*** *Uma linha vertical representaria a ligação entre variáveis em Lógica de Primeira Ordem; uma linha horizontal representaria a ligação de uma variável em Lógica de Primeira Ordem a uma função de Skolem (ver também exemplo 36).*

Exemplo 34. *(Representação Gráfica da Matriz $\mathcal{ALC}$)*

$$\left[\begin{array}{l|lll|ll} hasPet & OldLady & OldLady & OldLady & & \\ Cat & \underline{\neg hasPet_1} & \underline{\neg Animal_1} & hasPet & \neg OldLady(a) & CatOwner(a) \\ \neg CatOwner & & & \neg Cat & & \end{array}\right]$$

Definição 41. ***(Ciclo, Ontologias e Matrizes cíclicas/acíclicas).*** *Se A e B são conceitos atômicos em uma ontologia O, A <u>usa diretamente</u> B, se B aparece no lado direito de um axioma de subsunção cujo o lado esquerdo é A. A relação <u>usa</u> é o fecho transitivo de <u>usa diretamente</u>. Uma **ontologia cíclica** ou **matriz** tem um **ciclo** quando um conceito atômico <u>usa</u> a si próprio; caso contrário, é **acíclica** (BAADER et al., 2003);*

Exemplo 35. *(Ciclo, Ontologias e Matrizes cíclicas/acíclicas)*
$O = \{A \sqsubseteq \exists r.B, B \sqsubseteq \exists s.A\}$ é uma ontologia cíclica.

4.1.2 Transformação para Forma Normal

Para provar a validade de uma fórmula, os axiomas em $\mathcal{ALC}$ devem estar na FND e em uma das três formas normais definidas pelo conceito forma normal disjuntiva com duas linhas, apresentado anteriormente, de modo que possam ser facilmente representados em matriz. Assim, esta seção descreve a transformação dos axiomas até obter a matriz de representação.

Definição 42. ***(Consulta).*** *Uma **consulta** α (um axioma TBox ou ABox) sobre uma ontologia O é uma fórmula $\mathcal{ALC}$ para a qual a conseqüência lógica $O \models \alpha$ deve ser provada.*

Observação 4. ***(Consulta).*** *Para deduzir $O \models \alpha$, com $O = C_1 \wedge \ldots \wedge C_n$, a validade da fórmula $C_1 \wedge \ldots \wedge C_n \rightarrow \alpha$ $(O \rightarrow \alpha)$, isto é, a validade de $\neg O \vee \alpha$, deve ser provada. Com isso os efeitos para o FND são:*

(i) axiomas da forma $E \sqsubseteq D$ são convertidos em $E \wedge \neg D$;

(ii) asserções em ABox são negadas;

(iii) variáveis livres são quantificadas existencialmente;

(iv) a skolemização para Lógica de Primeira Ordem é aplicada a variáveis universais; e

(v) a consulta α não é negada.

Exemplo 36. *(Consulta). Considere a consulta F_1:*

$$\{\exists hasPet.Cat \sqsubseteq CatOwner, OldLady \sqsubseteq \exists hasPet.Animal \sqcap \forall hasPet.Cat\} \models OldLady \sqsubseteq CatOwner$$

Tal consulta é lida em Lógica de Primeira Ordem como (de acordo com o mapeamento descrito na tabela 15):

$$\left.\begin{array}{c} \forall x((\exists y\ hasPet(x,y) \wedge Cat(y)) \rightarrow CatOwner(x)) \\ \forall z(OldLady(z) \rightarrow \exists v(hasPet(z,v) \wedge Animal(v))) \\ \forall z(OldLady(z) \rightarrow \forall k(hasPet(z,k) \rightarrow Cat(k))) \end{array}\right\} \models \forall u(OldLady(u) \rightarrow CatOwner(u))$$

Em seguida obtem-se a equivalente FND com duas linhas:

$$\begin{array}{r} \exists x \exists y((hasPet(x,y) \wedge Cat(y)) \wedge CatOwner(x)) \vee \\ \exists z \forall v(OldLady(z) \wedge (\neg hasPet(z,v) \wedge \neg Animal(v))) \vee \\ \exists z \exists k(OldLady(z) \wedge (hasPet(z,k) \wedge \neg Cat(k))) \vee \\ \forall u(\neg OldLady(u) \vee CatOwner(u)) \end{array}$$

No Cálculo de Conexões de Primeira Ordem, a skolemização é aplicada sobre quantificadores universais, os quais são eliminados e suas variáveis substituídas por termos ou funções de Skolem. O resultado consiste de uma sequência de quantificadores existenciais em cada subfórmula. A sequência de quantificadores existenciais é normalmente escrita como um único quantificador seguido pela sequência de variáveis quantificadas. Existindo um único tipo de quantificador, o quantificador por si só é supérfluo, significando que ele pode ser omitido. O que resta é uma matriz em Lógica de Primeira Ordem (onde a é um termo de Skolem, f uma função de Skolem):

$$\{\{hasPet(x,y), Cat(y), \neg CatOwner(x)\},\ \{OldLady(z), \neg hasPet(z, f(z))\}, \{OldLady(z),$$
$$\neg Animal(f(z))\},\ \{OldLady(z), hasPet(z,k), \neg Cat(k)\}, \{\neg OldLady(a)\},\ \{CatOwner(a)\}\}$$

A representação gráfica correspondente da matriz em Lógica de Primeira Ordem é:

$$\begin{bmatrix} hasPet(x,y) & OldLady(z) & OldLady(z) & OldLady(z) & \neg OldLady(a) & CatOwner(a) \\ Cat(y) & \neg hasPet(z,f(z)) & \neg Animal(f(z)) & hasPet(z,k) & & \\ \neg CatOwner(x) & & & \neg Cat(k) & & \end{bmatrix}$$

A matriz correspondente em $\mathcal{ALC}$ é dada abaixo (o índice da coluna marca as duas cláusulas envolvidas na mesma restrição; variáveis são omitidas dado que estão especificadas implicitamente):

$$\{\{\underline{hasPet,\ Cat},\ \neg CatOwner\},\ \{OldLady, \underline{\neg hasPet_1}\},\ \{OldLady,\ \underline{\neg Animal_1}\},$$
$$\{OldLady,\ \underline{hasPet,\ \neg Cat}\},\ \{\neg OldLady(a)\},\ \{CatOwner(a)\}\}$$

A representação gráfica da matriz em $\mathcal{ALC}$ é M_1:

$$\left[\begin{array}{c|ccc|cc} hasPet & OldLady & OldLady & OldLady & & \\ Cat & \underline{\neg hasPet_1} & \underline{\neg Animal_1} & hasPet & \neg OldLady(a) & CatOwner(a) \\ \neg CatOwner & & & \neg Cat & & \end{array}\right]$$

Vê-se claramente na matriz em $\mathcal{ALC}$ que as linhas verticais na primeira e quarta colunas representaria a ligação entre variáveis em Lógica de Primeira Ordem; as linhas horizontais nos literais $\neg hasPet$ e $\neg Animal$, na segunda e terceita colunas representaria a ligação de uma variável em Lógica de Primeira Ordem a uma função de Skolem. O índice 1 nesses dois literais, que estão em cláusulas diferentes, indica que eles possuem a mesma restrição.

4.1.3 O Cálculo de θ-Conexões $\mathcal{ALC}$

O Cálculo de θ-Conexões $\mathcal{ALC}$ substitui funções de Skolem e unificação por θ-substituições, e, assim como sistemas que trabalham com Lógica de Descriões, emprega bloqueio para garantir a terminação.

Definição 43. ***(Caminho).*** *Um* ***caminho*** *em uma matriz $M = \{C_1, \ldots, C_n\}$ é um conjunto de literais $\{L_1, \ldots, L_n\}$ que contém um literal L_i de cada cláusula $C_i \in M$.*

Exemplo 37. *(Caminho)*
Considere a matriz M_1 do Exemplo 36. $\{hasPet\ |, \underline{\neg hasPet_1}, \underline{\neg Animal_1}, hasPet\ |, \neg OldLady(a), CatOwner(a)\}$ e $\{Cat\ |, \underline{\neg hasPet_1}, \underline{\neg Animal_1}, \neg Cat\ |, \neg OldLady(a), CatOwner(a)\}$ são alguns caminhos através de M_1.

Definição 44. ***(Conexão).*** *Uma* ***conexão*** *é um conjunto de literais da forma $\{L, \neg L\}$.*

Definição 45. ***(θ-Substituição, Conexão θ-complementar).*** *Seja x uma variável (possivelmente omitida) em um literal E e y um indivíduo ou outra variável. Uma* ***θ-substituição*** *é uma atribuição de um indivíduo ou variável a cada variável x (possivelmente omitida) em um literal E, denotada por $\theta(E) = E(\theta(x))$, onde $\theta(x) = y$. Uma* ***conexão θ-complementar*** *é um par de literais $\mathcal{ALC}$ $\{E(x), \neg E(y)\}$ ou $\{p(x, v), \neg p(y, u)\}$, com $\theta(x) = \theta(y), \theta(v) = \theta(u)$. O complemento $\overline{L}$ de um literal L é E se $L = \neg E$, ou é $\neg E$ se $L = E$.*

Observação 5. ***(θ-substituição).*** *A unificação simples sem funções de Skolem é usada para calcular θ-substituições. A aplicação de uma θ-substituição a um literal é uma aplicação para suas variáveis, ou seja $\theta(E) = E(\theta(x))$ e $\theta(r) = r(\theta(x), \theta(y))$, onde E é um conceito atômico e r é uma relação. Além disso, $x^\theta = \theta(x)$.*

Exemplo 38. *(θ-substituição, Conexão θ-complementar)*
Ainda na matriz M_1 do Exemplo 36, considere os literais OldLady e hasPet, ambos na quarta coluna, e $\neg oldLady(a)$, na penúltima coluna. São exemplos de θ-substituições $\theta(OldLady) =$

$OldLady(\theta(y))$ e $\theta(hasPet) = hasPet(\theta(y), x)$, onde $\theta(y) = a$, e $\{OldLady, \neg OldLady(a)\}$ uma conexão θ-complementar,

Definição 46. ***(Conjunto de conceitos).*** *O **conjunto de conceitos** $\tau(x)$ de uma variável ou indivíduo x contém todos os conceitos atômicos que foram instanciados por x no caminho P até o momento, ou, mais formalmente, $\tau(x) \stackrel{def}{=} \{E \in N_C | E(x) \in P\}$.*

Definição 47. ***(Condição de Skolem).*** *A **condição de Skolem** define que no máximo um conceito é sublinhado na forma gráfica da matriz. Esta condição é formalmente definida como $(\forall a \mid \{\underline{E_i} \in N_C | \underline{E_i(a)} \in P\}| \leq 1)$, onde i é um índice da coluna e P o caminho.*

Observação 6. ***(Condição de Skolem).*** *A condição de Skolem evita a situação em que, na lógica clássica, a unificação falha para duas funções de Skolem distintas. Sua implementação é simples: apenas uma 'flag' indicando se cada variável/indivíduo em qualquer caminho contém um conceito sublinhado é suficiente.*

Esse cálculo também usa uma **Regra de Cópia** (descrita na definição 49) que cria uma cópia de uma cláusula existente e adiciona-a à matriz M. Tal procedimento de cópia existe embutido no Método de Conexões para Lógica de Primeira Ordem, no entanto nesse cálculo a regra foi criada com um esquema de bloqueio para lidar com ontologias cíclicas. A Regra de Cópia então cria uma cláusula C^μ, a μ-ésima cópia da cláusula C, onde μ é um número natural acrescido de uma unidade quando a regra de cópia é aplicada para aquela cláusula, e a variável x em C^μ é denotada por x_μ. A **Condição de Bloqueio** (definida a seguir), usual em sistemas que trabalham com Lógica de Descrições, evita a criação ciclos infinitos de cópias de cláusulas.

Definição 48. ***(Condição de Bloqueio).*** *A **condição de bloqueio** verifica se o novo indivíduo x_μ^θ (se ele é novo, então $x_\mu^\theta \notin N_O$, como na condição) gerado numa cópia de cláusulas tem seu conjunto de conceitos $\tau(x_\mu^\theta)$ comparado ao conjunto de conceitos de indivíduos copiados previamente, ou seja, $\tau(x_\mu^\theta) \not\subseteq \tau(x_{\mu-1}^\theta)$ (SCHMIDT; TISHKOVSKY, 2007). Isso testa se o primeiro conjunto de conceitos é um subconjunto do segundo.*

Exemplo 39. *(Condição de Skolem)*
Considere a consulta $F_3 = E \sqsubseteq \exists r.A$, $\forall r.A \sqsubseteq E \models E(a)$, representada em Lógica de Primeira Ordem na figura 4. Em Lógica de Primeira Ordem, a unificação impede a conexão (indicada por uma linha tracejada). Para o cálculo de θ-conexões $\mathcal{ALC}$ esta conexão também é proibida, devido a condição de Skolem. Na figura 5, para a nova variável y, $\tau(y) = \{\underline{A}_2\}$; logo não pode conter também $\neg\underline{A}_1$, caso contrário, isto violaria a condição de Skolem.

Lema 1. ***(Equivalência entre θ-substituição e unificação no método de conexões para fórmulas $\mathcal{ALC}$).*** *Seja F uma fórmula $\mathcal{ALC}$ e L um literal em F. θ-substituição é equivalente à unificação para fórmulas $\mathcal{ALC}$, denotado por $\theta(L) \equiv \sigma(L)$, isto é, $\theta(L) = E(a)$ e $\sigma(L) = E(a)$ ou $\theta(L) = \varnothing$ então $\sigma(L) = \varnothing$, quando θ-substituição e unificação recebem as mesmas entradas.*

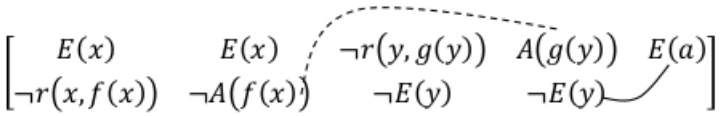

Figura 4 – Tentativa de prova em conexões para F_3 representada em Lógica de Primeira Ordem e com $\sigma = \{y/a\}$. A linha tracejada representa uma ligação proibida (FREITAS; OTTEN, 2016).

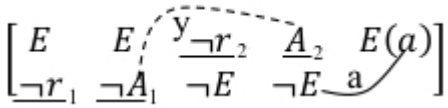

Figura 5 – Tentativa de prova em conexões para F_3 em Lógica de Descrições $\mathcal{ALC}$. A linha tracejada representa uma ligação proibida (FREITAS; OTTEN, 2016).

Os casos que ocorrem em $\mathcal{ALC}$, todos cobertos pela θ-substituição, estão na tabela 2.

Tabela 2 – Equivalência entre unificação e θ-substituição no Cálculo de θ-Conexões $\mathcal{ALC}$

Unificação		**θ-substituição**	
Entrada	**Saída**	**Entrada**	**Saída**
$L_1 = E(x)$ $L_2 = \neg E(a)$	$\sigma(L_1) = E(a)$	$L_1 = E$ ou $E(x)$ $L_2 = \neg E(a)$	$\theta(L_1) = E(a)$
$L_1 = E(x)$ $L_2 = \neg E(y)$	$\sigma(L_1) = E(y)$	$L_1 = E$ ou $E(x)$ $L_2 = \neg E$ ou $\neg E(y)$	$\theta(L_1) = E(y)$
$L_1 = E(b)$ $L_2 = \neg E(a)$	Não unificável	$L_1 = E(b)$ $L_2 = \neg E(a)$	Sem θ-substituição (Não unificável)
$L_1 = E(x)$ $L_2 = \neg E(f(y))$	$\sigma(L_1) = \{E(f(y))\}$	$L_1 = E$ $L_2 = \underline{\neg E_i}$	$\theta(L_1) = E(y)$ $\tau(y) = \tau(y) \cup \{\underline{\neg E_i}\}$
$L_1 = E(g(x))$ $L_2 = \neg E(f(y))$	Não unificável	$L_1 = \underline{E_k}$ $L_2 = \underline{\neg E_j}$	Sem θ-substituição: $\forall a \mid \{\underline{E_i} \mid \underline{E_i(a)} \in P\} \mid \leq 1$

Nota-se que, em todos os casos, a θ-substituição e a unificação produzem a mesma substituição ou nenhuma substituição; a única exceção reside no caso onde $L_1 = E(x)$ e $L_2 = \neg E(f(y))$ ($L_1 = E$, $L_2 = \neg E_i$ na notação sem variáveis). Contudo, o Lema 1 mostra que elas são equivalentes, no sentido em que a unificação no método de conexões para Lógica de Primeira Ordem impede as mesmas conexões que o Cálculo de θ-Conexões $\mathcal{ALC}$ e a θ-substituição também impedem.

Definição 49. ***(Cálculo de θ-Conexões $\mathcal{ALC}$).*** *A figura 6 mostra o* ***Cálculo de θ-Conexões $\mathcal{ALC}$*** *formal, adaptado do Cálculo de Conexões para Lógica de Primeira Ordem de (OTTEN, 2010). As regras do cálculo são aplicadas de baixo para cima. A estrutura básica é a tupla <C, M, P>, onde M é a matriz correspondente à consequência lógica $O \models \alpha$ e α é uma consulta*

na forma TBox ou ABox, e C e P são conjuntos de literais ou uma sentença vazia representada por ε.

- *C é chamada cláusula-alvo e possui o conjunto de literais que precisam ser conectados no momento;*
- *P é o caminho ativo, isto é, o (sub-)caminho que está sendo investigado no momento; P possui o conjunto de literais das cláusulas que foram conectadas para alcançar o caminho atual da prova;*
- *C_1 e C_2 são cláusulas;*
- *θ representa a θ-substituição e*
- *C^μ denota a cópia de índice μ da cláusula C.*

Axioma (Ax)	$\dfrac{}{\{\}, M, P}$
Início (In)	$\dfrac{C_1, M, \{\}}{\varepsilon, M, \varepsilon}$ com $C_1 \in \alpha$
Redução (Red)	$\dfrac{C, M, P \cup \{L_2\}}{C \cup \{L_1\}, M, P \cup \{L_2\}}$ com $\theta(L_1) = \theta(\overline{L_2})$ e a condição de Skolem é válida
Extensão (Ext)	$\dfrac{C_2 \setminus \{L_2\}, M, P \cup \{L_1\} \qquad C, M, P}{C \cup \{L_1\}, M, P}$ com $C_2 \in M$, $L_2 \in C_2, \theta(L_1) = \theta(\overline{L_2})$ e a condição de Skolem é válida
Cópia (Cop)	$\dfrac{C \cup \{L_1\}, M \cup \{C_2^\mu\}, P}{C \cup \{L_1\}, M, P}$ com C_2^μ é uma cópia de C_1, $L_2 \in C_2^\mu, \theta(L_1) = \theta(\overline{L_2})$ e a condição de bloqueio é válida

Figura 6 – Cálculo de θ-Conexões $\mathcal{ALC}$ (FREITAS; OTTEN, 2016).

- **Regra de Início:** A regra de início é a primeira regra a ser aplicada no cálculo. A regra começa com a cláusula-alvo e o caminho ativo vazios e seleciona uma cláusula inicial C_1 que pertence a α, ou seja, C_1 pertence ao consequente da consulta. C_1 é a cláusula com o conjunto de literais que precisam ser conectados.
- **Regra de Redução:** A regra de redução conecta L_1, na cláusula-alvo, ao seu complementar L_2 encontrado no caminho ativo P por meio de uma θ-substituição que torna possível a conexão $\{\theta(L_1), \theta(L_2)\}$, ou seja, $\theta(L_1) = \theta(\overline{L_2})$. A redução só pode ser aplicada quando a θ-substituição existe e é possível.

- **Regra de Extensão:** A regra de extensão conecta um literal L_1 da cláusula-alvo a um complementar L_2 de uma cláusula C_2 da matriz M. Esta regra gera dois ramos a serem provados:

  (a) o ramo direito com a cláusula-alvo sem o literal L_1 que foi conectado, e

  (b) o ramo esquerdo com C_2 como a nova cláusula-alvo. L_2 não faz parte da nova cláusula-alvo. L_1 é adicionado ao caminho ativo da nova cláusula-alvo.

  A regra de extensão só é aplicada se existe uma θ-substituição tal que $\theta(L_1) = \theta(\overline{L_2})$.

- **Regra de Cópia:** A regra de cópia cria uma cópia, denotada por C_2^{μ}, de uma cláusula copiada existente C_1 e a adiciona à matriz M. μ é um número natural atribuído a cada cláusula em M, especificando quantas cópias desta cláusula são consideradas em uma prova. Novas variáveis são criadas para substituir as variáveis implícitas da cláusula copiada, onde a variável x em C^{μ} é denotada por x_{μ}. Essa regra implementa o bloqueio (BAADER et al., 2003), técnica típica em raciocínio em Lógica de Descrições, quando nenhuma alternativa de conexão está disponível e ontologias cíclicas são tratadas. Esta regra regula a criação de novos indivíduos, impedindo, assim, a não terminação na ocorrência de um ciclo infinito. Quando a regra de cópia é usada, ela é seguida pela aplicação da regra de extensão ou da regra de redução, para evitar não-determinismo na aplicação das regras. Ao invés de trabalhar com a matriz original M, será usada uma matriz M' que corresponde a M com algumas cláusulas copiadas.

- **Regra Axioma:** A regra axioma é aplicada para fechar os ramos, com a cláusula-alvo vazia. A regra do axioma representa a validade do ramo.

4.1.4 Verificação de validade de uma Fórmula no Cálculo de θ-Conexões $\mathcal{ALC}$

Esta seção mostra o método de verificação de validade de uma fórmula $\mathcal{ALC}$ do Cálculo de θ-Conexões $\mathcal{ALC}$ utilizando a representação gráfica da matriz e o cálculo formal, respectivamente.

Dada uma fórmula em $\mathcal{ALC}$, o processo de verificação de validade desse cálculo checa se os caminhos possíveis através da matriz da fórmula possuem conexões que, sob θ-substituições, são complementares. Este processo é guiado por um caminho ativo, um (sub-)caminho através da matriz que está sendo investigado, consistindo de um conjunto de literais que foram conectados para alcançar o caminho atual da prova.

Considere a fórmula F_1 e sua representação gráfica M_1 dadas no exemplo 36. Uma demonstração dos passos da busca para F_1 é feita a seguir cuja representação gráfica é ilustrada na figura 7. A prova formal é ilustrada pela figura 8.

Na figura 7, os literais de cada conexão são ligados por um arco. Os literais do caminho ativo (inicialmente vazio) são envolvidos por um retângulo. Uma θ-substituição, instância ou variável atribuída a cada variável (omitida), aparece em cada arco - representando um indivíduo fictício sobre o qual os literais estão predicando.

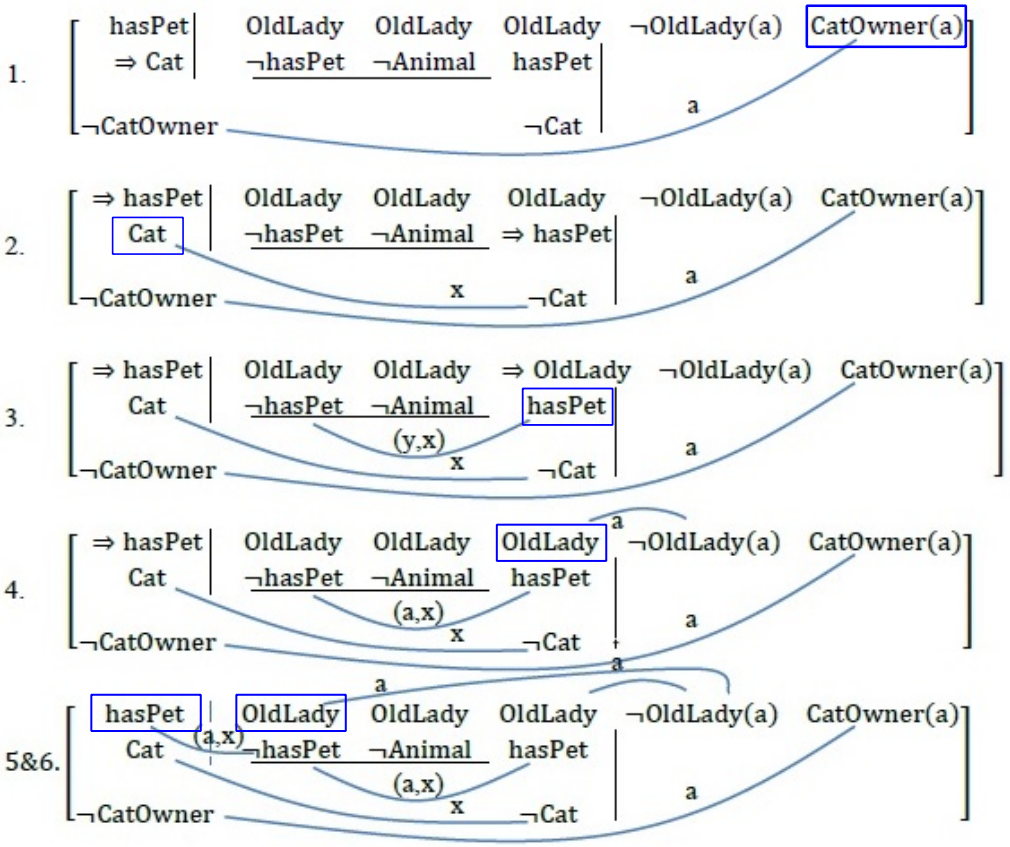

Figura 7 – Prova para F_1 usando a representação gráfica da matriz (FREITAS; OTTEN, 2016).

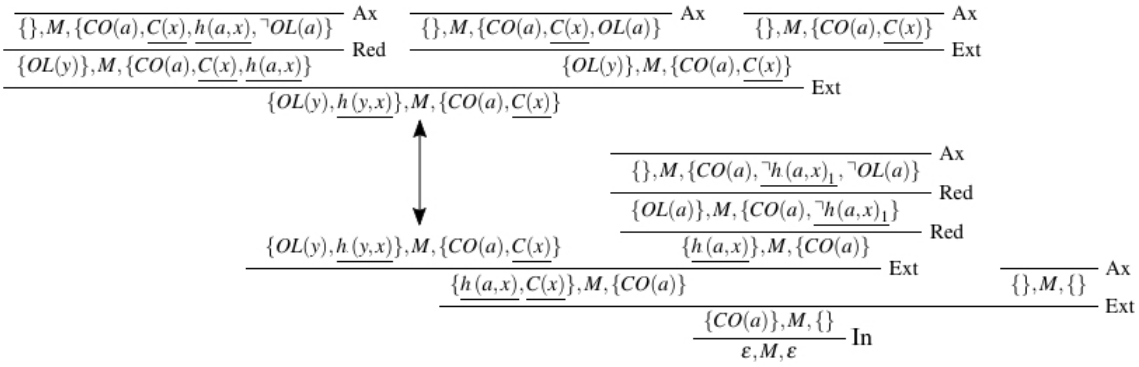

Figura 8 – Prova formal do MC-θ $\mathcal{ALC}$ para F_1 (FREITAS; OTTEN, 2016).

- No passo 1, uma cláusula do consequente é escolhida como *cláusula inicial*, nesse caso a cláusula $\{CatOwner(a)\}$, e um literal a partir dela é selecionado, $CatOwner(a)$. Esse

literal é conectado a $\neg CatOwner$ por um passo de extensão e a instância a está na θ-substituição de $\neg CatOwner$ e $CatOwner(a)$. A seta aponta para literais que ainda serão verificados na cláusula.

- No passo 2, um novo passo de extensão ocorre formando a conexão $\{Cat, \neg Cat\}$, e estabelecida sobre a variável x, produzida pela θ-substituição. Essa conexão ainda não é suficiente para provar todos os caminhos decorrentes da cláusula a qual *Cat* pertence, pois existem literais restantes ainda não verificados na cláusula, começando pelo literal apontado por uma seta.

- No passo 3, quando o predicado *hasPet* é conectado, uma variável (ou indivíduo fictício), por exemplo y, estabelece uma relação com a variável ou indivíduo que estava sendo tratado até o momento pela cláusula a qual pertence *hasPet*, ou seja x. Ocorre então uma θ-substituição. Nesse momento y representa um indivíduo que tem *pet* que não é *cat*. A relação é indicada por (y, x).

- No passo 4, *OldLady* é conectado a $\neg OldLady(a)$ predicando sobre a através de um passo de extensão. A variável y agora é conhecida e θ-substituída pelo indivíduo a (isto é, $\theta(y) = a$).

- Os passos 5 e 6, formam as conexões $\{OldLady, \neg OldLady(a)\}$ e $\{hasPet, \neg hasPet\}$. Cada conexão é formada utilizando um passo de redução. A redução é desencadeada por suas duas condições: (i) existe uma conexão para o literal atual na prova; e (ii) existe uma θ-substituição.

Cada caminho através da matriz possui uma conexão que é θ-complementar. Logo, a consulta em $\mathcal{ALC}$ em questão é válida.

Na prova formal, todos os ramos terminam em axioma Ax, provando a validade de F_1.

Esse cálculo é correto e completo para $\mathcal{ALC}$. Suas provas de Corretude, Completude e Terminação podem ser consultadas em (FREITAS; OTTEN, 2016).

4.2 Cálculo Não-clausal de Conexões para $\mathcal{ALC}$

O Cálculo Não-clausal de Conexões para $\mathcal{ALC}$ apresentado nesta seção é uma variante do Cálculo de θ-Conexões $\mathcal{ALC}$. Ele diverge do Cálculo de θ-Conexões $\mathcal{ALC}$ por trabalhar diretamente na estrutura da fórmula original, sem qualquer necessidade de conversão para FND, e por expandir a definição de cláusula para conter (sub-)matrizes. Essas características têm como base o Cálculo Não-clausal de Conexões para Lógica de Primeria Ordem proposto em (OTTEN, 2011), conforme explicado nas próximas seções.

4.2.1 Matriz Não-clausal para $\mathcal{ALC}$

O Cálculo Não-clausal de Conexões para $\mathcal{ALC}$ usa a definição de **Fórmula com Polaridade** e **Matriz Não-clausal** do Cálculo Não-clausal de Conexões para Lógica de Primeria Ordem proposto em (OTTEN, 2011). A definição de fórmula com polaridade é usada sem qualquer alteração, no entanto, a definição de matriz não-clausal foi adaptada para contemplar fórmulas $\mathcal{ALC}$ arbitrárias.

Definição 50. ***(Fórmula com Polaridade)*** *Uma **fórmula com polaridade**, denotada por F^p, consiste de uma fórmula F e uma polaridade p, onde $p \in \{0, 1\}$. Esse conceito é usado para representar negação em uma matriz.*

Observação 7. *A polaridade é útil em uma conversão para sequentes, onde a polaridade com valor 1 representa uma regra a esquerda, e com valor 0, representa uma regra a direita (ver tabela 7 no capítulo 5).*

Exemplo 40. *(Fórmula com Polaridade)*
Seja A e $\neg A$ duas fórmulas. A tem polaridade 0 e é representada por A^0 e $\neg A$ tem polaridade 1 e é representada por A^1.

Definição 51. ***(Matriz $\mathcal{ALC}$ Não-Clausal).*** *Uma **matriz $\mathcal{ALC}$ não-clausal** é um conjunto de cláusulas, na qual uma cláusula é um conjunto de literais e matrizes. Seja F uma fórmula $\mathcal{ALC}$ e p uma polaridade. A matriz de F^p, denotada por $M(F^p)$, é definida indutivamente de acordo com a tabela 3, a qual indica como a polaridade é herdada pelas matrizes de uma F^p. A matriz de F é a matriz $M(F^0)$.*

Tabela 3 – Matriz de uma fórmula $\mathcal{ALC}$ F^p.

Tipo	F^p	$M(F^p)$	Tipo	F^p	$M(F^p)$
Atômico	A^0	$\{\{A^0\}\}$	β	$(G \sqcap H)^0$	$\{\{M(G^0), M(H^0)\}\}$
	A^1	$\{\{A^1\}\}$		$(G \sqcup H)^1$	$\{\{M(G^1), M(H^1)\}\}$
α	$(\neg G)^0$	$M(G^1)$		$(G \sqsubseteq H)^1$	$\{\{M(G^0), M(H^1)\}\}$
	$(\neg G)^1$	$M(G^0)$	γ	$(\forall GH)^1$	$\{\{M(\underline{G^0}), M(\underline{H^1})\}\}$
	$(G \sqcap H)^1$	$\{\{M(G^1)\}, \{M(H^1)\}\}$		$(\exists GH)^0$	$\{\{M(\underline{G^0}), M(\underline{H^0})\}\}$
	$(G \sqcup H)^0$	$\{\{M(G^0)\}, \{M(H^0)\}\}$	δ	$(\forall GH)^0$	$\{\{M(\underline{G^1})\}, \{M(\underline{H^0})\}\}$
	$(G \sqsubseteq H)^0$	$\{\{M(G^1)\}, \{M(H^0)\}\}$		$(\exists GH)^1$	$\{\{M(\underline{G^1})\}, \{M(\underline{H^1})\}\}$
	$(G \models H)^0$	$\{\{M(G^1)\}, \{M(H^0)\}\}$			

A matriz de uma fórmula F^p em Lógica de Primeria Ordem é definida conforme mostra a tabela 4.

Analisando as duas tabelas (3 e 4), note que as adaptações na definição de matriz não-clausal para fórmulas $\mathcal{ALC}$ se deram nos seguintes pontos:

- os conectivos $\wedge$ e $\vee$ foram substituídos por $\sqcap$ e $\sqcup$, sem perda na definição das matrizes;

Tabela 4 – Matriz de uma fórmula F^p em LPO (OTTEN, 2011).

Tipo	F^p	$M(F^0)$	Tipo	F^p	$M(F^0)$
Atômico	A^0	$\{\{A^0\}\}$	β	$(G \wedge H)^0$	$\{\{M(G^0), M(H^0)\}\}$
	A^1	$\{\{A^1\}\}$		$(G \vee H)^1$	$\{\{M(G^1), M(H^1)\}\}$
α	$(\neg G)^0$	$M(G^1)$		$(G \rightarrow H)^1$	$\{\{M(G^0), M(H^1)\}\}$
	$(\neg G)^1$	$M(G^0)$	γ	$(\forall x G)^1$	$M(G[x\backslash x^*]^1)$
	$(G \wedge H)^1$	$\{\{M(G^1)\}, \{M(H^1)\}\}$		$(\exists x G)^0$	$M(G[x\backslash x^*]^0)$
	$(G \vee H)^0$	$\{\{M(G^0)\}, \{M(H^0)\}\}$	δ	$(\forall x G)^0$	$M(G[x\backslash t^*]^0)$
	$(G \rightarrow H)^0$	$\{\{M(G^1)\}, \{M(H^0)\}\}$		$(\exists x G)^1$	$M(G[x\backslash t^*]^1)$

- o conectivo $\rightarrow$ foi substituído por $\sqsubseteq$, sem alteração na definição das matrizes;
- foi adicionada uma definição de matriz para a fórmula com construtor $\models$ e polaridade zero, com isso é possível contemplar fórmulas na forma $O \models C \sqsubseteq D$;
- as fórmulas do tipo γ e δ (fórmulas com restrições universais e existenciais) e suas respectivas matrizes, foram redefinidas para refletir as relações e ausência de variáveis em Lógica de Descrições, bem como os sublinhados dos literais do Cálculo de Conexões para $\mathcal{ALC}$ (ver definição 39).

Definição 52. ***(Representação Gráfica (Positiva) da Matriz).*** *Na* ***representação gráfica (positiva) de uma matriz****, suas cláusulas são dispostas horizontalmente, enquanto que os literais e (sub-)matrizes de cada cláusula estão dispostos verticalmente. As restrições são representadas por linhas contínuas; restrições com índices, isto é $L_{i,j}$, são representadas com linhas horizontais; restrições sem índices, com linhas verticais.*

Observação 8. *Matrizes da forma $M = \{\ldots, \{C_1, \ldots, C_n\}, \ldots\}$ podem ser simplificadas para $M' = \{\ldots, C_1, \ldots, C_n, \ldots\}$ onde $C_1, \ldots, C_n$ são cláusulas. Cláusulas da forma $C = \{\ldots, \{ M_1, \ldots, M_m\}, \ldots\}$ podem ser simplificadas para $C' = \{\ldots, M_1, \ldots, M_m, \ldots\}$ onde $M_1, \ldots, M_m$ são matrizes.*

Observação 9. *Para definir a matriz não-clausal de uma fórmula $\mathcal{ALC}$ o processo inicia pelo seu principal construtor com polaridade zero, que deverá ser $\models$ ou $\sqsubseteq$.*

Exemplo 41. *(Matriz $\mathcal{ALC}$ Não-Clausal, Representação Gráfica (Positiva) da Matriz) Considere novamente a consulta F_1 do exemplo 36:*

$$\left.\begin{array}{c}(\exists(hasPet.Cat) \sqsubseteq CatOwner) \sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))\end{array}\right\} \models (OldLady(a) \sqsubseteq CatOwner(a))$$

Os conceitos contidos na tabela 3 são aplicados à fórmula acima, conforme demonstra em detalhes a tabela 5.

Tabela 5 – Passos para obter a simplificada matriz não-clausal para a consulta F_1

F^p	Fórmula/$M(F^p)$
$(G \models H)$	$\left.\begin{array}{c}(\exists(hasPet.Cat) \sqsubseteq CatOwner)\sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))\end{array}\right\} \models (OldLady(a) \sqsubseteq CatOwner(a))$
$(G \models H)^0$	$\left\{\begin{array}{c}\{(((\exists(hasPet.Cat) \sqsubseteq CatOwner)\sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat)))^1\}, \\ \{(OldLady(a) \sqsubseteq CatOwner(a))^0\}\end{array}\right\}$
$(G \sqsubseteq H)^0$	$\left\{\begin{array}{c}\{(((\exists(hasPet.Cat) \sqsubseteq CatOwner)\sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat)))^1\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(G \sqcap H)^1$	$\left\{\begin{array}{c}\{\{\{(\exists(hasPet.Cat) \sqsubseteq CatOwner)^1\}, \\ \{(OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))^1\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(G \sqsubseteq H)^1$	$\left\{\begin{array}{c}\{\{\{\{(\exists(hasPet.Cat))^0,\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ (\exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))^1\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(G \sqcap H)^1$	$\left\{\begin{array}{c}\{\{\{\{(\exists(hasPet.Cat))^0,\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{(\exists(hasPet.Animal))^1\}, \{(\forall(hasPet.Cat))^1\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(\exists GH)^0$	$\left\{\begin{array}{c}\{\{\{\{\{\underline{hasPet}^0,\ \underline{Cat}^0\}\},\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{(\exists(hasPet.Animal))^1\}, \{(\forall(hasPet.Cat))^1\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(\forall GH)^1$	$\left\{\begin{array}{c}\{\{\{\{\{\underline{hasPet}^0,\ \underline{Cat}^0\}\},\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{(\exists(hasPet.Animal))^1\},\ \{\{\{\underline{hasPet}^0, \underline{Cat}^1\}\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(\exists GH)^1$	$\left\{\begin{array}{c}\{\{\{\{\{\underline{hasPet}^0,\ \underline{Cat}^0\}\},\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{\{\{\underline{hasPet}_1^1\},\ \{\underline{Animal}_1^1\}\}\}, \{\{\{\underline{hasPet}^0, \underline{Cat}^1\}\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$

Assim a matriz não-clausal simplificada para F_1 é:

$$\begin{array}{r}\{\ \{\underline{hasPet^0,\ Cat^0},\ CatOwner^1\}, \\ \{\ OldLady^0, \{\{\underline{hasPet}_1^1\}, \{\underline{Animal}_1^1\},\ \{\underline{hasPet^0,\ Cat^1}\}\}\ \}, \\ \{OldLady(a)^1\},\ \{CatOwner(a)^0\}\ \}\end{array}$$

Essa mesma matriz não-clausal simplificada sem a notação com polaridade é dada por:

$$\begin{array}{r}\{\ \{\underline{hasPet,\ Cat},\ \neg CatOwner\}, \\ \{\ OldLady, \{\{\underline{\neg hasPet}_1\}, \{\underline{\neg Animal}_1\},\ \{\underline{hasPet,\ \neg Cat}\}\}\ \}, \\ \{\neg OldLady(a)\},\ \{CatOwner(a)\}\ \}\end{array}$$

Sua representação gráfica é M_2:

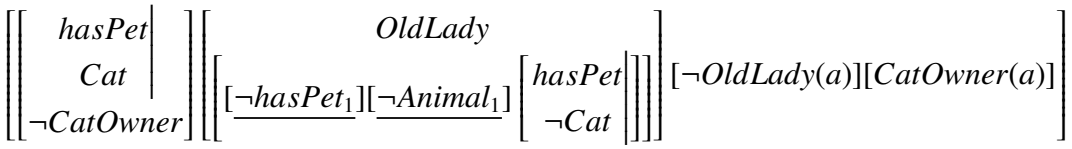

Figura 9 – Representação gráfica da matriz não-clausal para F_1.

4.2.2 Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$

O Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$ é uma fusão do Cálculo de θ-Conexões $\mathcal{ALC}$ com o Cálculo Não-clausal de Conexões para Lógica de Primeira Ordem (OTTEN, 2011). Nessa fusão, alguns conceitos são aproveitados, sem qualquer alteração, do Cálculo de θ-Conexões $\mathcal{ALC}$. São eles:

- conexão,
- θ-substituição,
- conexão θ-complementar,
- conjunto de conceitos e
- condição de Skolem.

Da mesma forma, os conceitos abaixo são trazidos do Cálculo Não-clausal de Conexões para Lógica de Primeira Ordem de (OTTEN, 2011):

- cláusula α-relacionada,
- cláusula-mãe,
- cláusula de extensão,
- β-cláusula e
- a definição de caminho, a qual é generalizada para matrizes não-clausais.

Definição 53. ***(Caminho).*** *Um* ***caminho*** *através de uma matriz $M = \{C_1, \ldots, C_n\}$ é um conjunto de literais que contém um literal L_i de cada cláusula $C_i \in M$, isto é $\bigcup_{i=1}^{n}\{L_i\}$ com $L_i \in C_i$. Um caminho através de uma matriz M (ou uma cláusula C) é indutivamente definido como se segue. O (único) caminho através de um literal L é $\{L\}$. Se $p_1, \ldots, p_n$ são caminhos através das cláusulas $C_1, \ldots, C_n$, respectivamente, então $p_1 \cup \ldots \cup p_n$ é um caminho através da matriz $M = \{C_1, \ldots, C_n\}$. Se $p_1, \ldots, p_n$ são caminhos através das matrizes/literais $M_1, \ldots, M_n$, respectivamente, então $p_1, \ldots, p_n$ são também caminhos através da cláusula $C = \{M_1, \ldots, M_n\}$.*

Exemplo 42. *(Caminho)*
São exemplos de caminhos através da matriz M_2, dada no exemplo 41, $\{hasPet|, OldLady, \neg OldLady(a), CatOwner(a)\}$ e $\{Cat|, \underline{\neg hasPet_1}, \underline{\neg Animal_1}, \neg Cat, \neg OldLady(a), CatOwner(a)\}$.

Definição 54. ***(Cláusula α-relacionada).*** *Seja C uma cláusula em uma matriz M e L um literal em M. C é uma* ***cláusula α-relacionada*** *com L se e somente se M contém (ou é igual a) uma matriz $\{C_1, \ldots, C_n\}$ tal que $C = C_i$ ou C_i contém C, e C_j contém L para algum $1 \leq i, j \leq n$ com $i \neq j$. C é α-relacionada com um conjunto de literais $\mathcal{L}$ se e somente se C é α-relacionada com todos os literais $L \in \mathcal{L}$.*

Exemplo 43. *(Cláusula α-relacionada)*
Ainda na matriz M_2 do exemplo 41, $\{\underline{\neg Animal_1}\}$ é α-relacionada com $\{hasPet, \neg Cat\}$.

Definição 55. ***(Cláusula-mãe).*** *Seja M uma matriz e C uma cláusula em M. A cláusula $C' = \{M_1, \ldots, M_n\}$ em M é chamada* ***cláusula-mãe*** *de C se e somente se $C \in M_i$ para algum $1 \leq i \leq n$.*

Exemplo 44. *(Cláusula-mãe)*
$\{OldLady, \{\{\underline{\neg hasPet_1}\}, \{\underline{\neg Animal_1}\}, \{hasPet, \neg Cat\}\}\}$ é a cláusula-mãe de $\{\underline{\neg hasPet_1}\}$.

Definição 56. ***(Cláusula de Extensão).*** *Seja M uma matriz e P um caminho (um conjunto de literais). Então a cláusula C em M é uma* ***cláusula de extensão*** *de M com respeito a P, se e somente se ou C contém um literal de P, ou C é α-relacionada a todos os literais de P ocorrendo em M e se C tem uma cláusula mãe, ela contém um literal de P.*

Observação 10. *Na regra de extensão do Cálculo de θ-Conexões $\mathcal{ALC}$ a nova cláusula-alvo (conjunto de literais que precisam ser conectados) é $C_2 \setminus \{L_2\}$ (no ramo esquerdo). No Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$, a cláusula de extensão C_2 pode conter cláusulas que são α-relacionadas a L_2 e não precisam ser consideradas para a nova cláusula-alvo. Logo, estas cláusulas podem ser excluídas da cláusula-alvo. A cláusula resultante é chamada a β-Cláusula de C_2 com respeito a L_2.*

Definição 57. ***(β-Cláusula).*** *Seja $C = \{M_1, \ldots, M_n\}$ uma cláusula e L um literal em C. A* ***β-Cláusula*** *de C com respeito a L, denotada por β-Cláusula$_L(C)$, é indutivamente definida:*

$$\beta\text{-Cláusula}_L(C) := \begin{cases} C \setminus \{L\} & \text{se } L \in C, \\ M_1, \ldots, M_{i-1}, \{C^\beta\}, M_{i+1}, \ldots, M_n & \text{caso contrário,} \end{cases}$$

onde $C' \in M_i$ contém L e $C^\beta := \beta$-Cláusula$_L(C')$.

Exemplo 45. *(Cláusula de Extensão, β-Cláusula)*
Na matriz M_2 do exemplo 41, $C = \{OldLady, \{\{\underline{\neg hasPet_1}\}, \{\underline{\neg Animal_1}\}, \{\underline{hasPet}, \underline{\neg Cat}\}\}\}$ é uma cláusula de extensão em relação ao caminho $p = \{CatOwner(a), \underline{Cat}\}$, enquanto a cláusula $\{OldLady, \{\{\underline{\neg hasPet_1}\}, \{\underline{\neg Animal_1}\}, \{hasPet\}\}\}$ é uma β-Cláusula de C em relação a $L = \underline{\neg Cat}$.

Definição 58. ***(Cálculo Não-clausal de θ-conexões $\mathcal{ALC}$). O Cálculo Não-clausal de θ-conexões*** *$\mathcal{ALC}$ consiste das regras Axioma, Início, Redução, Extensão, Decomposição e Cópia conforme ilustra a figura 10.*

Axioma (A) $\dfrac{}{\{\}, M, P}$

Início (I) $\dfrac{C_1, M, \{\}}{\varepsilon, M, \varepsilon}$ com $C_1 \in \alpha$

Redução (R) $\dfrac{C, M, P \cup \{L_2\}}{C \cup \{L_1\}, M, P \cup \{L_2\}}$
com $\theta(L_1) = \theta(\overline{L_2})$ e a condição de Skolem é válida

Extensão (E) $\dfrac{C_3, M, P \cup \{L_1\} \qquad C, M, P}{C \cup \{L_1\}, M, Path}$ com $C_3 := \beta\text{-}cláusula_{L_2}(C_2)$,
C_2 é uma cláusula de extensão de M com respeito a $P \cup \{L_1\}$,
$L_2 \in C_2$, $\theta(L_1) = \theta(\overline{L_2})$ e a condição de Skolem é válida

Decomposição(D) $\dfrac{C \cup C_1, M, P}{C \cup \{M_1\}, M, P}$ com $C_1 \in M_1$

Cópia (C) $\dfrac{C \cup \{L_1\}, M \cup \{C_2^{\mu}\}, P}{C \cup \{L_1\}, M, P}$ onde C_2^{μ} é uma cópia de C_1,
$L_2 \in C_2^{\mu}$, $\theta(L_1) = \theta(\overline{L_2})$ e a condição de bloqueio é válida

Figura 10 – Cálculo Não-clausal de θ-conexões $\mathcal{ALC}$.

- **Regras de Início, Redução, Cópia e Axioma:** Essas regras são as mesmas do Cálculo de θ-Conexões $\mathcal{ALC}$, e funcionam da mesma forma como foi descrito anteriormente.
- **Regra de Extensão:** Essa regra é semelhante à regra de extensão do Cálculo de θ-Conexões $\mathcal{ALC}$. A diferança é que ela foi modificada para conter β-Cláusulas. Ela conecta um literal L_1 da cláusula-alvo a um complementar L_2 de uma cláusula C_3 da matriz M, onde C_3 é uma β-Cláusula de C_2 com respeito a L_2. Com isso o ramo esquerdo pode conter β-Cláusula (C_3) como a nova cláusula-alvo.
- **Regra de Decomposição:** A regra de decomposição consiste em decompor uma cláusula $\{M_1\}$ (composta de subcláusulas) em suas subcláusulas, extraindo a subcláusula C_1 de $\{M_1\}$.

Exemplo 46. *(Cálculo Não-clausal de θ-conexões $\mathcal{ALC}$).*
Considere novamente a fórmula F_1 e sua matriz não-clausal M_2. As figuras 11 e 12 mostram a prova para F_1 usando a representação gráfica da matriz e o cálculo formal, respectivamente. O processo de verificação de validade da fórmula é similar ao mostrado no Cálculo de θ-conexões $\mathcal{ALC}$. Como todo caminho através de M_2 contém uma conexão θ-complementar, F_1 é válida. Na prova formal, a regra de Decomposição divide a cláusula-alvo obtendo uma nova (sub)cláusula-alvo, enquanto que a θ-substituição é possível com $\theta(y) = a$. Uma vez que todas as folhas são axiomas A, a prova formal é concluída e F_1 é provada válida.

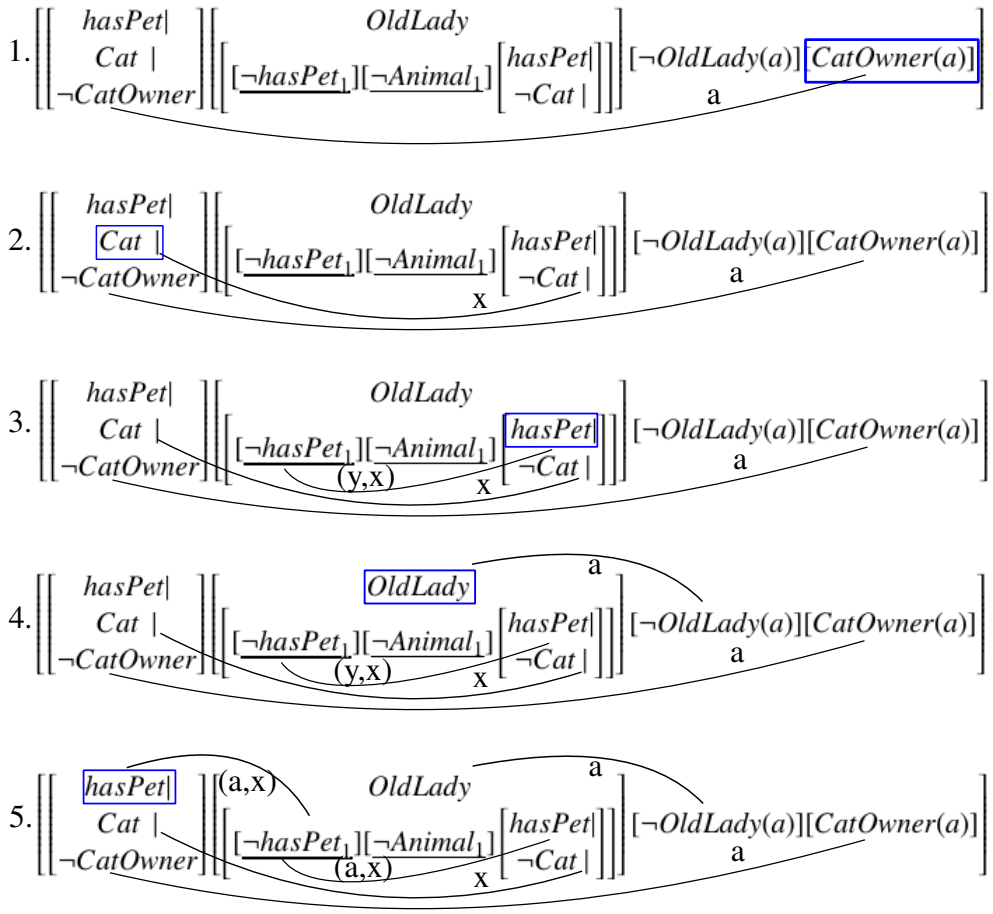

Figura 11 – Prova no Cálculo Não-clausal de θ-conexões $\mathcal{ALC}$ usando a representação gráfica da matriz.

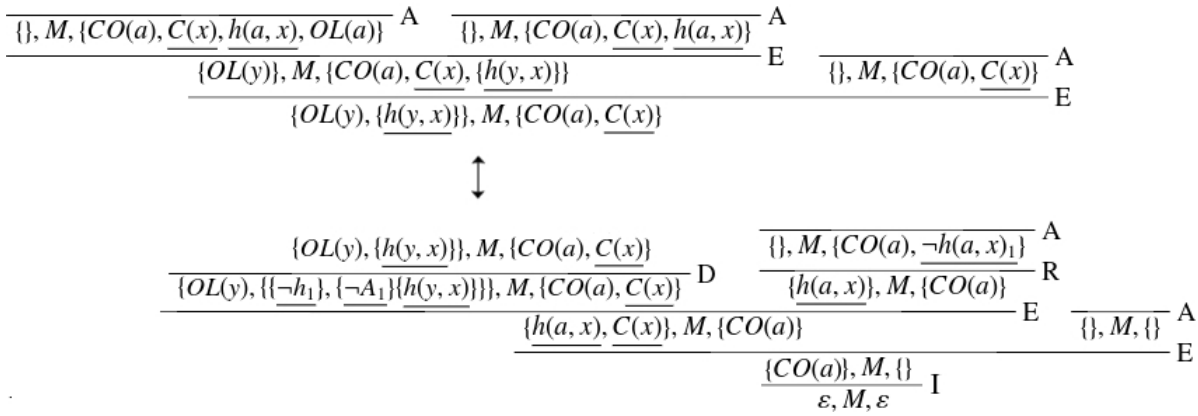

Figura 12 – Prova formal do Cálculo Não-clausal de θ-conexões $\mathcal{ALC}$ para F_1

As provas de corretude, completude e terminação desse cálculo se encontram no Apêndice B.

5 CONVERTENDO PROVAS $\mathcal{ALC}$ EM CONEXÕES PARA SEQUENTES

Este capítulo apresenta um método para converter provas $\mathcal{ALC}$ construídas com o método de conexões não-clausal, visto na seção 4.2, em provas no cálculo de sequentes para $\mathcal{ALC}$ apresentado na seção 7.2. A conversão aqui apresentada baseia-se no método de prova para Lógica de Primeira Ordem apresentado em Otten e Kreitz (1995). Tal método constrói uma matriz de prova e uma prova em sequentes simultaneamente, cuja técnica utiliza noções de caminhos e conexões. O método descrito neste trabalho difere do original por utilizar uma notação sem variável e construtores específicos da Lógica de Descrições, como por exemplo o construtor para subsunção; além da adição de uma substituição para lidar com instâncias e trabalhar com regras próprias do cálculo de sequentes para $\mathcal{ALC}$.

Desse modo, as próximas seções se encarregam em descrever o método de conversão iniciando pela seção 5.1, a qual aborda as principais ideias, conceitos e formalização do método. Já a seção 5.2 descreve o processo de conversão e detalha suas etapas utilizando um exemplo.

5.1 Definições Preliminares

Definição 59. ***(Árvore de Fórmula, Posição, Rótulo, Polaridade, Tipo).*** *Uma* ***árvore de fórmula*** *é uma representação sintática de uma fórmula F como árvore, onde cada nó poderá ter até dois nós filhos. Cada nó possui os seguintes dados:*

- ***Posição:*** *uma posição é um índice que identifica cada elemento (predicado e conectivo) na fórmula. Sua forma é* $a_0, a_1, a_2, a_3 \ldots$*;*
- ***Rótulo:*** *um rótulo ou é um conectivo* ($\sqcap, \sqcup, \neg, \sqsubseteq, \models$) *ou um quantificador ou um predicado, se é uma (sub-)fórmula atômica. Nós rotulados com predicados são folhas da árvore, enquanto os demais nós são chamados de nós internos;*
- ***Polaridade:*** *a polaridade (0 ou 1) de um nó é determinada pelo rótulo e polaridade de seu nó pai. O nó raiz, primeiro nó da árvore, tem polaridade 0;*
- ***Tipo:*** *o tipo de um nó é representado por uma das letras gregas:* α, β, α', β', γ *e* δ*. Ele é determinado por seu rótulo e sua polaridade. Nós folha não têm tipo. A polaridade e o tipo de um nó são definidos na tabela 6. Por exemplo, na primeira linha dessa tabela,* $(A \sqcap B)^1$ *significa que o nó rotulado com* $\sqcap$ *e polaridade 1 tem tipo* α *e seus nós sucessores têm polaridade 1.*

Tabela 6 – Polaridade e tipos dos nós para $\mathcal{ALC}$

Tipo α			Tipo β			Tipo δ		
$(A \sqcap B)^1$	A^1	B^1	$(A \sqcap B)^0$	A^0	B^0	$(\forall rA)^0$	r^1	A^0
$(A \sqcup B)^0$	A^0	B^0	$(A \sqcup B)^1$	A^1	B^1	$(\exists rA)^1$	r^1	A^1
$(\neg A)^1$	A^0							
$(\neg A)^0$	A^1							
Tipo α'			Tipo β'			Tipo γ		
$(A \sqsubseteq B)^0$	A^1	B^0	$(A \sqsubseteq B)^1$	A^0	B^1	$(\forall rA)^1$	r^0	A^1
$(A \models B)^0$	A^1	B^0				$(\exists rA)^0$	r^0	A^0

Observação 11. *Nós do tipo α e α' correspondem às regras em sequentes que não causam ramificação na prova. Nós do tipo γ e δ correspondem às regras em sequentes que possuem quantificadores. As regras associadas ao tipo δ possuem a restrição de autovariável. Nós do tipo β e β' (isto é, $\sqcap^0$, $\sqcup^1$, e $\sqsubseteq^1$) são particularmente importantes, uma vez que as regras em sequentes correspondentes a esses tipos (descritas nas tabelas 7 e 8) causam uma ramificação da prova em duas sub-provas independentes. Estes nós têm seus tipos indexados na árvore de fórmula para facilitar sua identificação, por exemplo β_1, β_2, β'_1, β'_2. Além disso, cada ramo cuja raiz é do tipo β ou β' é marcado com uma letra (por exemplo: a,b,c,...).*

Os nós folhas que possuem instâncias são filhos de nós do tipo α, α' e β. Os nós folha que não possuem instâncias têm associado ao seu rótulo as posições dos seus nós antecessores mais próximos, de acordo com os critérios listados abaixo (isso ajuda a checar complementaridade em uma conexão entre dois nós, como será visto mais adiante):

- se o rótulo do nó folha representa um conceito, ele terá associado ao seu rótulo uma única posição;
- se o rótulo do nó folha representa uma relação, ele terá associado ao seu rótulo duas posições na forma (a_1, a_2), onde a_2 é a posição do nó antecessor mais próximo;
- apenas as posições dos nós do tipo γ, δ e β' são associadas aos rótulos.

Um nó interno é representado como na figura 13, enquanto um nó folha é representado como na figura 14.

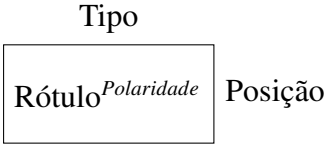

Figura 13 – Representação de nó interno.

Figura 14 – Representação de nó folha.

Observação 12. *A construção da árvore se dá pela identificação do construtor (conectivo ou quantificador) principal da (sub-)fórmula, o qual será um rótulo no nó da árvore. Esse nó possui no máximo dois ramos que os liga a seus nós filhos, ou seja, novas (sub-)fórmulas. O tipo do nó e as polaridades de seus filhos são atribuídos conforme a tabela 6. Se os nós filhos não são (sub-)fórmulas atômicas, o processo se repete identificando o construtor principal dessas (sub-)fórmulas e gerando outros nós na árvore até chegar aos nós que são folhas.*

Exemplo 47. *(Processo de Construção da Árvore de Fórmula, com Posição, Rótulo, Polaridade, Tipo). Considere novamente a fórmula para F_1:*

$$((\exists h.C \sqsubseteq CO) \sqcap (OL \sqsubseteq \exists h.A \sqcap \forall h.C)) \models (OL(a) \sqsubseteq CO(a))$$

- **Passo 1:** O principal construtor de F_1 é $\models$. Esse construtor será o rótulo do nó raiz, que por definição tem polaridade zero. Sua posição é a_0, um índice que o identifica. De acordo com a tabela 6, seu tipo é α', seus nós filhos, à direita e à esquerda, têm polaridade zero e 1, respectivamente. O nó filho à direita, é a sub-fórmula à direita de $\models$ em F_1, e o nó filho à esquerda, a sub-fórmula à sua esquerda. Esse passo é demonstrado na figura 15.

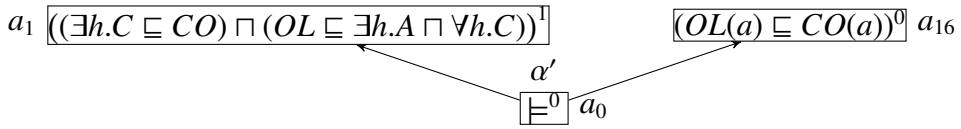

Figura 15 – Passo 01 - Processo de Construção da Árvore de Fórmula para F_1.

- **Passo 2:** Na sub-fórmula do lado esquerdo da árvore, o principal conector é $\sqcap$. Sua polaridade é 1, seu tipo é α e ambos os filhos têm polaridade 1. Na sub-fórmula do lado direito, o conectivo $\sqsubseteq$ tem polaridade zero, tipo α' e dois nós filhos que são folhas. O filho do lado direito tem polaridade zero e o filho do lado esquerdo, polaridade 1.

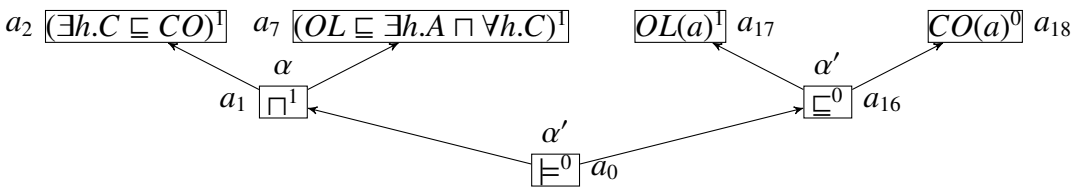

Figura 16 – Passo 02 - Processo de Construção da Árvore de Fórmula para F_1.

- **Passo 3:** Nas duas sub-fórmulas que precisam ser desdobradas, o conectivo principal é $\sqsubseteq$, com polaridade 1. Assim, os nós para esses conectivos são do tipo β'. Para diferenciá-los, um índice é inserido em cada um deles, obtendo β'_1 e β'_2. Seus ramos são também identificados com as letras 'a' e 'b', para os ramos de β'_1, e 'c' e 'd', para os ramos de β'_2. Os filhos do lado direito desses nós, têm polaridade 1, e os filhos do lado esquerdo, zero. O nó do tipo β'_1, tem um filho que é nó folha. Esse nó folha tem a posição do seu pai (a_2) associada ao seu rótulo. O nó β'_2 também possui um nó folha, que tem associada ao seu rótulo a posição a_7.

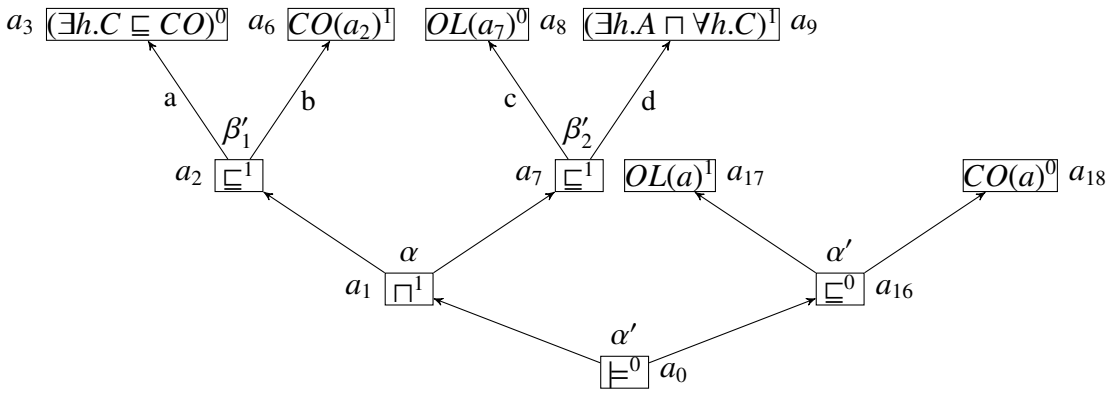

Figura 17 – Passo 03 - Processo de Construção da Árvore de Fórmula para F_1.

Esse processo continua até chegar aos nós folha, como mostra a figura 18.

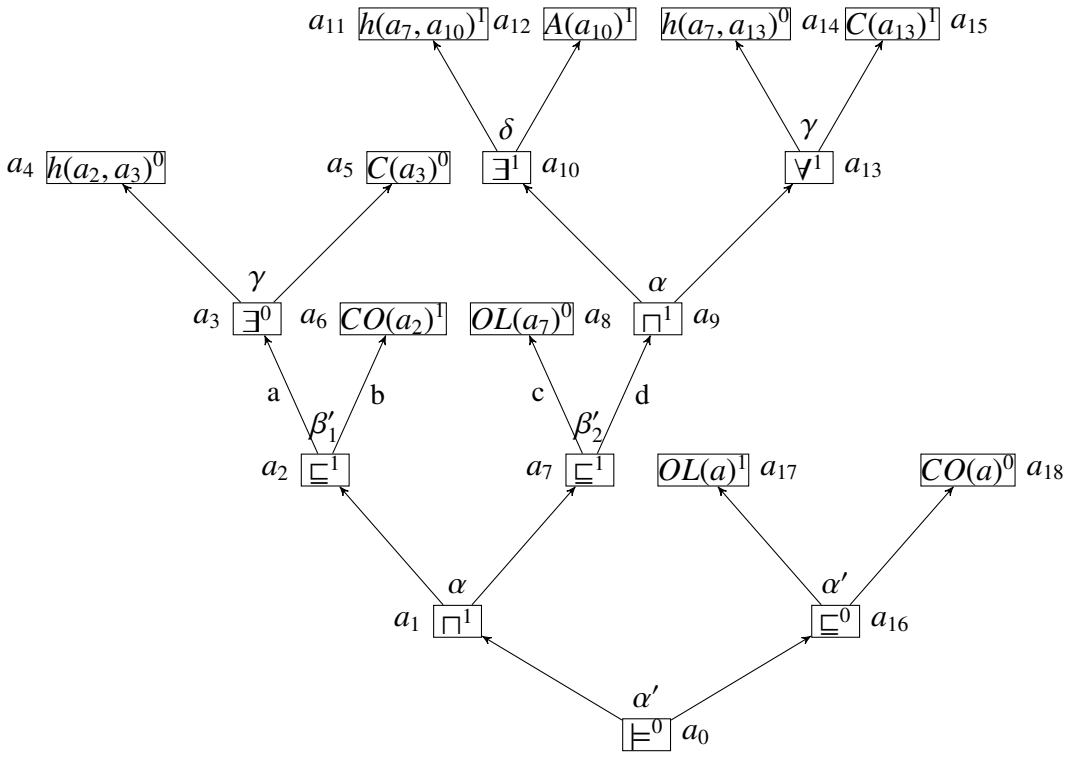

Figura 18 – Árvore de fórmula para F_1 com rótulos, polaridades e tipos.

Perceba que os nós a_4, a_{11} e a_{14}, possuem duas posições associadas aos seus rótulos. Isso ocorre dado que esses rótulos representam relações, que são binárias. Logo, o nó a_4 tem associado ao seu rótulo h as posições a_2 e a_3, que são as posições dos seus antecessores do tipo β' e γ, respectivamente. Perceba que o nó a_{11} é precedido por nós do tipo β', α e δ, assim o nó a_{11} tem associado ao seu rótulo h as posições dos nós do tipo β' e δ, que são a_7 e a_{10}, obedecendo aos critérios mencionados anteriormente. De forma semelhante, o nó a_{14} tem associado ao seu rótulo as posições a_7 e a_{13}.

Definição 60. ***(Caminho entre dois nós).*** *Sejam dois nós, um com posição a_i e o outro com posição a_j. Um* ***caminho entre*** *esses* ***dois nós*** *através de uma árvore de fórmula F, é um subconjunto de posições dos nós de sua árvore que ligam esses dois nós na forma $\{a_i, a_2, \ldots, a_{j-1}, a_j\}$.*

Exemplo 48. *(Caminho entre dois nós)*
Na figura 18, o caminho entre o nó de posição a_{11}, com rótulo h^1, e o nó de posição a_{14}, rotulado com h^0, através da árvore é $\{a_{11}, a_{10}, a_9, a_{13}, a_{14}\}$. A figura 19 ilustra esse caminho.

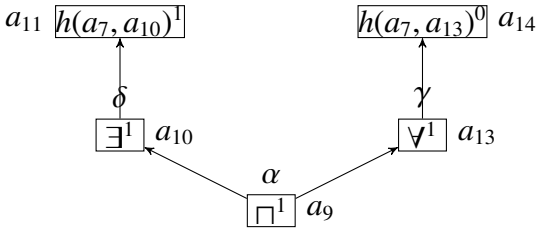

Figura 19 – Recorte da árvore de fórmula para F_1 com o caminho entre os nós de posição a_{11} e a_{14}.

São definidas a seguir três substituições de posições, para substituir posições associadas aos rótulos dos nós folha:

- substituição σ_δ: substitui posições do tipo γ por posições do tipo δ;
- substituição $\sigma_{\beta'}$: substitui posições do tipo β', γ, δ por instâncias ou posições do tipo β';
- substituição combinada σ_{Final}: consiste de uma combinação das duas primeiras.

Observação 13. *Para uma dada fórmula é usado A, A', B, B', Γ e Δ para denotar os conjuntos de posições de nós do tipo α, α', β, β', γ, e δ, respectivamente.*

Definição 61. ***(Substituição de posições σ_δ, relação de ordenamento $\sqsubset_\delta$)*** *Uma* ***substituição de posições*** *σ_δ é um mapeamento do conjunto Γ de posições dos nós do tipo γ para o conjunto Δ de posições de nós do tipo δ. A substituição σ_δ induz uma* ***relação de ordenamento parcial*** *$\sqsubset_\delta$ em $\Delta \times \Gamma$ da seguinte forma: seja $u \in \Gamma$ e $v \in \Delta$; se $\sigma_\delta(u) = p$ então $v \sqsubset_\delta u$ para todo $v \in \Delta$ ocorrendo na posição p.*

Observação 14. *Dado que as regras do sequentes $r\forall$ e $l\exists$ e suas homólogas $l\neg\forall$ e $r\neg\exists$ são restritas a condição de autovariável, a relação $v \sqsubset_\delta u$ expressa que o nó rotulado por v deve ser reduzido antes de reduzir um rotulado por u.*

Definição 62. ***(Substituição de posições $\sigma_{\beta'}$).*** *As posições dos nós do tipo β', γ e δ, assim como instâncias aparecem em fórmulas atômicas. Consequentemente, uma **substituição de posições** $\sigma_{\beta'}$ é um mapeamento do conjunto $B'/\Gamma/\Delta$ de posições dos nós do tipo $\beta'/\gamma/\delta$ para instâncias ou posições dos nós do tipo β'. Seja u um nó folha com posições dos nós do tipo $\beta'/\gamma/\delta$ associadas a seu rótulo e $v \in B'$; se $\sigma_{\beta'}(u) = p$, onde $p \in B'$ ou p é uma instância.*

Observação 15. *Reduzir um nó significa aplicar a regra do sequentes que corresponde àquele nó sobre uma dada (sub-)fórmula. Nós folhas não são reduzidos.*

Exemplo 49. *(Substituição de posições σ_δ, relação de ordenamento $\sqsubset_\delta$)*
Considere a árvore de fórmula na figura 18. Seja u o nó rotulado por $\forall^1$, com posição a_{13} e tipo γ, e seja v o nó rotulado por $\exists^1$, com posição a_{10} e tipo δ. Para substituir a posição de um nó do tipo γ pela posição de um nó do tipo δ, é preciso reduzir primeiro o nó do tipo δ, então o nó com posição a_{10} deve ser reduzido antes do nó com posição a_{13}. Assim, para esse exemplo, a relação de ordenamento $\sqsubset_\delta$ é dada por $\exists^1 a_{10} \sqsubset_\delta \forall^1 a_{13}$, e a substituição $\sigma_\delta(\forall^1 a_{13}) = a_{10}$. Com isso, têm-se $\sigma_\delta = \{a_{13}/a_{10}\}$.

Exemplo 50. *(Substituição de posições $\sigma_{\beta'}$)*
Considere a árvore de fórmula na figura 18. Seja u o nó rotulado por OL^0, com posição a_8 e posição a_7 do tipo β' associada ao seu rótulo, e seja v o nó rotulado por OL^1, com posição a_{17} e instância a. A substituição para esse nó folha u nesse caso é $\sigma_{\beta'}(OL(a_7)^0) = a$. Logo, $\sigma_{\beta'} = \{a_7/a\}$.

Definição 63. ***(Substituição σ_{Final}).*** *Uma **substituição** σ_{Final} consiste de uma substituição σ_δ e uma substituição $\sigma_{\beta'}$, onde $\sigma_{Final} := \sigma_\delta \cup \sigma_{\beta'}$.*

Exemplo 51. *(Substituição σ_{Final})*
Considere os dois exemplos dados anteriormente. Assim, $\sigma_{Final} = \{a_{13}/a_{10}, a_7/a\}$.

Definição 64. ***(Conexão, conexão σ_{Final}-complementar).*** *Uma **conexão** é um par de nós folha rotulados com o mesmo símbolo de predicado e a mesma posição associada ao rótulo ou mesma instância, mas com diferentes polaridades. Se eles são idênticos sob σ_{Final}, a conexão é chamada de **conexão σ_{Final}-complementar**.*

Exemplo 52. *(Conexão, conexão σ_{Final}-complementar)*
Seja a árvore de fórmula da figura 18. Os nós folha $h(a_2, a_3)^0$ e $h(a_7, a_{10})^1$ com posições a_4 e a_{11}, respectivamente, formam uma conexão que é complementar sob $\sigma_{Final} = \{a_2/a_7, a_3/a_{10}\}$.

Definição 65. ***(Ordenação da árvore $\prec$).*** *A **ordenação da árvore** $\prec$ de uma fórmula F é a ordenação parcial das posições dos nós na árvore de fórmula. $\prec$ é definida como se segue:*

(i) a raiz ocupa a menor posição com respeito a esta ordenação,

(ii) $a_i \prec a_j$ se e somente se a posição a_i está abaixo de a_j na árvore de fórmula.

Exemplo 53. *(Ordenação da árvore $\prec$)*
Na árvore representada na figura 18, são exemplos de ordenação de árvore $a_7 \prec a_9 \prec a_{13} \prec a_{15}$ e $a_0 \prec a_1 \prec a_2 \prec a_3$.

Definição 66. ***(Ordem de redução $\lhd$).*** *O fecho transitivo da união de $\sqsubset_\delta$, $\sqsubset_{\beta'}$ e $\prec$ é chamado de* ***ordem de redução*** *$\lhd$, isto é, $\lhd := (\prec \cup \sqsubset_\delta \cup \sqsubset_{\beta'})^+$.*

Observação 16. *$v \lhd u$ significa que o nó rotulado por v deve ser reduzido antes do nó rotulado por u na prova em sequentes. $\lhd$ determina a ordem de redução dos nós da árvore, bem como ajuda a determinar quais e em que ordem as regras do cálculo de sequentes devem ser aplicadas.*

Exemplo 54. *(Ordem de redução $\lhd$)*
Na árvore representada na figura 18, os nós com posições a_7, a_{10}, a_{16} e a_{13}, possuem as seguintes relações:

- $a_7 \prec a_{10}$;
- $a_7 \prec a_{13}$;
- $a_{10} \sqsubset_\delta a_{13}$.

Assim, a união do ordenamento e a ordenação de árvore determinam a ordem de redução para esses nós, que é: $a_7 \lhd a_{10} \lhd a_{13}$.

Definição 67. ***(Substituição σ_{Final} admissível).*** *Uma* ***substituição σ_{Final} é admissível*** *se a ordem de redução $\lhd$ não é reflexiva. Neste caso é possível construir uma prova no cálculo de sequentes.*

Exemplo 55. *(Substituição σ_{Final} não-admissível) Aqui, será dado um contra-exemplo de uma substituição σ_{Final} admissível. Considere a fórmula F_3 :*

$$\forall child.\exists child.Doctor \sqsubseteq \exists child \forall child(Doctor(a) \sqcup Lawyer(a))$$

Sua árvore de fórmula é mostrada na figura 20. $\sigma_\delta = \{a_1/a_8, a_6/a_3\}$ é uma substituição que induz às ordens de redução $a_8 \lhd a_1$ e $a_3 \lhd a_6$. A união dessas ordens de redução e a ordenação da árvore formam loops, como por exemplo $a_8 \lhd a_1 \lhd a_3 \lhd a_6 \lhd a_8$. Esse loop tem $a_8 \lhd a_8$, isto é, a ordem de redução é reflexiva, então a substituição não é admissível. Portanto, não é possível conectar $child^1$, na posição a_9, a $child^0$, na posição a_2 sob a substituição σ_δ no método de conexões, bem como não é possível construir uma prova em sequentes para essa fórmula. Consequentemente, F_3 não é válida.

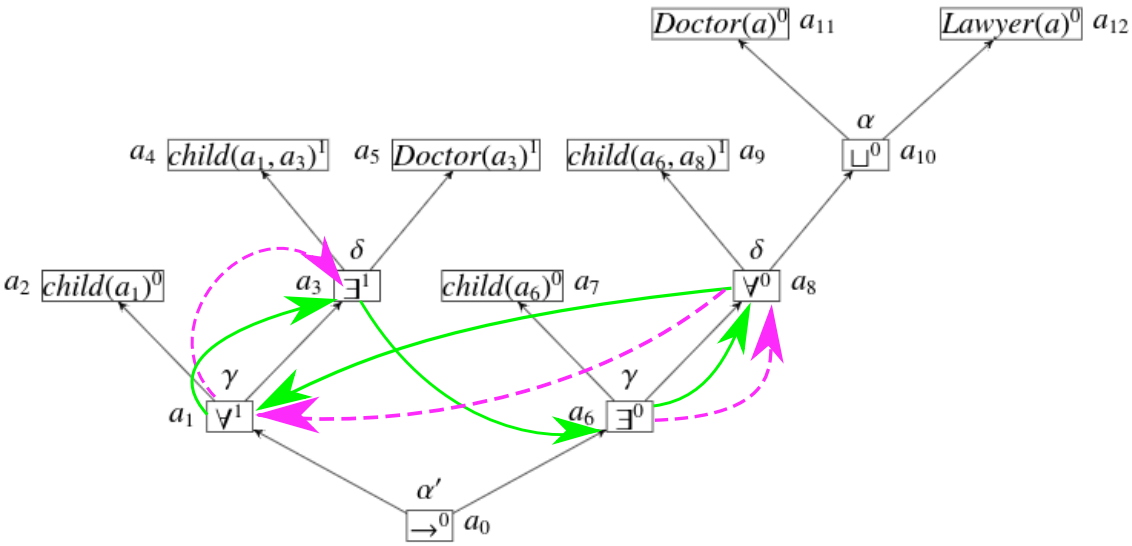

Figura 20 – Árvore de fórmula para F_3.

Observação 17. *Os arcos tracejados e os arcos com linhas contínuas representam, respectivamente, os loops formados por* $a_8 \lhd a_1 \lhd a_3 \lhd a_6 \lhd a_8$ *e* $a_6 \lhd a_3 \lhd a_8 \lhd a_1 \lhd a_6$ *que impedem a validação da fórmula.*

Uma correspondência entre rótulo, polaridade e tipo de um nó com as regras do sequentes, apresentadas na seção 7.2, é estabelecida nas tabelas 7 e 8. Tal correspondência é útil para a construção da prova em sequentes, onde a polaridade auxilia na identificação da regra. A polaridade 1 representa uma regra à esquerda (left ou l), enquanto a polaridade 0, à direita (right ou r), para os casos em que há uma regra associada. Por exemplo, na primeira linha da tabela 7, para o nó $\sqcup^1$ a regra é l$\sqcup$, e para o nó $\sqcap^0$ a regra é r$\sqcap$.

Para os casos em que nós internos são precedidos por um nó rotulado por uma negação, a correspondência é dada pela tabela 8.

Tabela 7 – Correspondência entre rótulo, polaridade e tipo de um nó, não precedido por um nó rotulado por uma negação, com as regras do sequentes

Tipo α	Regra
$\sqcap^1$	l$\sqcap$
$\sqcup^0$	r$\sqcup$
$\neg^1$	$\varnothing$
$\neg^0$	$\varnothing$

Tipo β	Regra
$\sqcap^0$	r$\sqcap$
$\sqcup^1$	l$\sqcup$

Tipo δ	Regra
$\forall^0$	r$\forall$
$\exists^1$	l$\exists$

Tipo α'	Regra
$\sqsubseteq^0$	$\varnothing$
$\models^0$	$\varnothing$

Tipo β'	Regra
$\sqsubseteq^1$	$\dfrac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$

Tipo γ	Regra
$\forall^1$	$\varnothing$
$\exists^0$	$\varnothing$

Tabela 8 – Correspondência entre rótulo, polaridade e tipo de um nó, precedido por um nó rotulado por uma negação, com as regras do sequentes

Tipo α	Regra
$\neg^1$	r$\neg\neg$
$\neg^0$	l$\neg\neg$
$\sqcap^1$	r$\neg\sqcap$
$\sqcup^0$	l$\neg\sqcup$

Tipo β	Regra
$\sqcap^0$	l$\neg\sqcap$
$\sqcup^1$	r$\neg\sqcup$

Tipo δ	Regra
$\forall^0$	l$\neg\forall$
$\exists^1$	r$\neg\exists$

5.2 Processo de Conversão

Dada uma fórmula $\mathcal{ALC}$ e sua prova em conexões na matriz não-clausal gerada pelo método de conexões não-clausal apresentado na seção 4.2, o procedimento de conversão converte a prova em conexões para uma prova em sequentes no cálculo de sequentes para $\mathcal{ALC}$ descrito na seção 7.2. Esse processo de conversão realiza quatro etapas:

- **Etapa 1- Construção da árvore de fórmula:** Nessa etapa, uma representação sintática em forma de árvore é construída para a fórmula de entrada contendo nós, conforme descrito na definição 59;
- **Etapa 2- Atribuição de posições aos elementos da matriz:** Como os elementos da matriz de prova correspondem aos predicados existentes na fórmula, que por sua vez também correspondem aos nós folha na árvore de fórmula, essa etapa atribui a cada elemento da matriz a posição do predicado correspondente;
- **Etapa 3- Construção da estrutura da prova (parcial) em sequentes:** Para cada conexão da matriz, a árvore de fórmula é examinada em busca dos nós folha que correspondem à conexão. Os caminhos entre o nó raiz e esses nós na árvore são analisados, a fim de determinar a ordem dos nós a ser trabalhada e com isso construir uma estrutura da prova (parcial) em sequentes. Essa estrutura fornece informações sobre a ordem de redução $\lhd$, que ajuda a determinar as regras que devem ser aplicadas, e sobre a existência de ramificação da prova, dada pela identificação de nós do tipo β e β'.
- **Etapa 4- Construção da prova completa em sequentes:** Esta é a quarta e última etapa do processo onde uma prova completa em sequentes é construída a partir da estrutura da prova (parcial) em sequentes e da correspondência entre o nó e as regras do sequentes, descritas nas tabelas 7 e 8.

Exemplo 56. *(Processo de Conversão) Para a geração da prova em sequentes, considere novamente a fórmula para a consulta F_1 e sua prova na matriz não-clausal:*

$$\{\exists hasPet.Cat \sqsubseteq CatOwner, OldLady \sqsubseteq \exists hasPet.Animal \sqcap \forall hasPet.Cat\} \models OldLady(a) \sqsubseteq CatOwner(a)$$

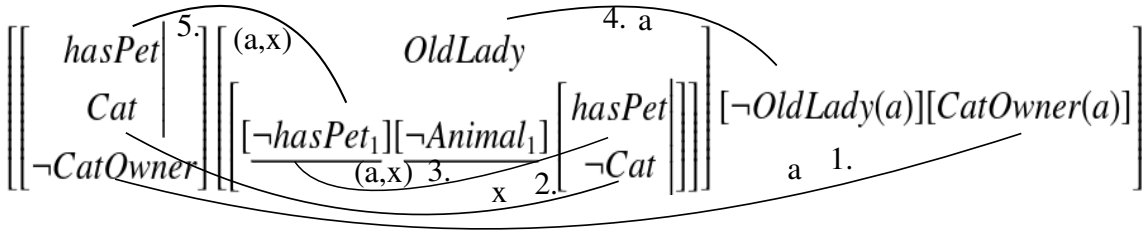

Figura 21 – Prova no Cálculo Não-clausal de θ-conexões $\mathcal{ALC}$ para F_1 usando a representação gráfica da matriz.

Etapa 1: Foi construída a árvore de fórmula de F_1 contendo todas as informações necessárias, conforme mostra a figura 22.

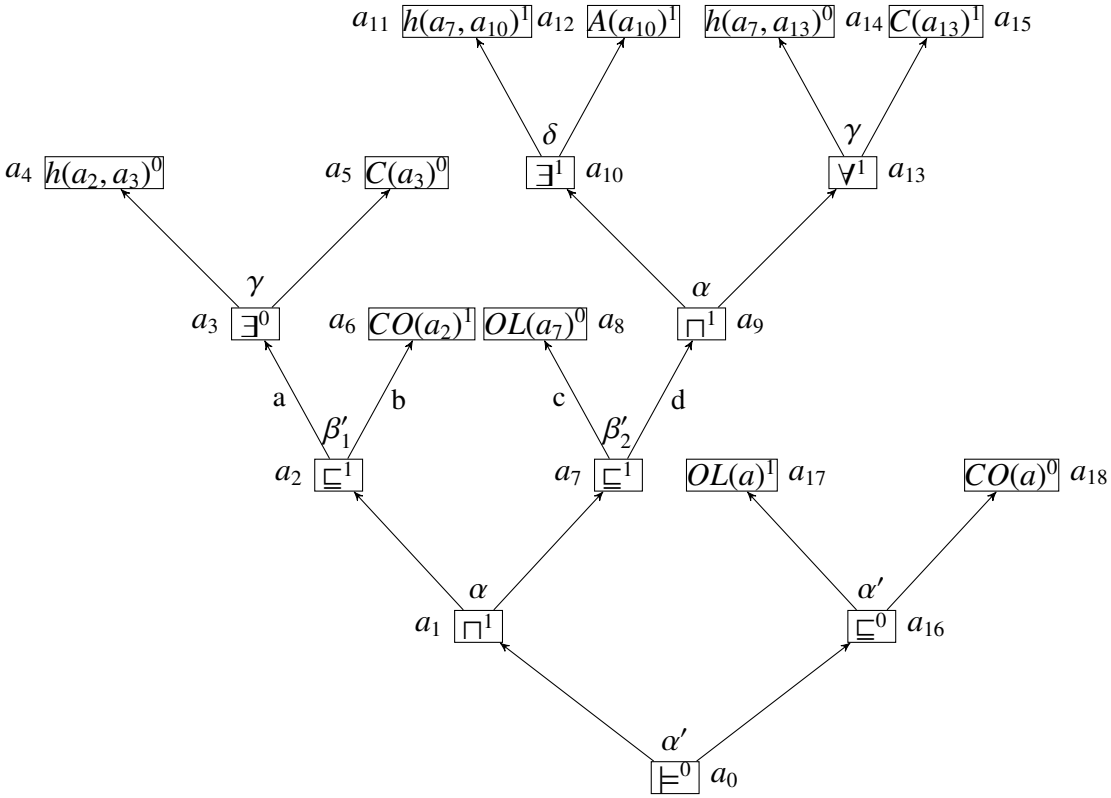

Figura 22 – Árvore de fórmula para F_1.

Etapa 2: A matriz com os elementos contendo a posição do nó correspondente na árvore de fórmula é representada na figura 23 (por uma questão de espaço os nomes dos literais foram abreviados em todas as matrizes).

$$\left[\left[\begin{array}{|c} h^0a_4 \\ C^0a_5 \\ CO^1a_6 \end{array}\right]\left[\left[\begin{array}{c} OL^0a_8 \\ [\underline{h^1_1a_{11}}][\underline{A^1_1a_{12}}]\left[\begin{array}{|c} h^0a_{14} \\ C^1a_{15} \end{array}\right] \end{array}\right]\right][OL(a)^1a_{17}][CO(a)^0a_{18}]\right]$$

Figura 23 – Representação gráfica da matriz não-clausal para F_1 com posições.

Etapa 3: As conexões começam a ser analisadas. A primeira conexão liga o elemento $CO(a)^0$, de posição a_{18}, ao elemento CO^1, de posição a_6. Essa conexão é complementar sob a substituição $\sigma_{\beta'} = \{a_2/a\}$, ver tabela 9. Analisando o caminho entre os nós folha que correspondem a essa conexão, temos o caminho $\{a_{18}, a_{16}, a_0, a_1, a_2, a_6\}$. Como não há uma relação de ordenamento $\sqsubset_\sigma$ entre os nós desse caminho e há duas ordenações de árvore dada por $a_0 \prec a_{16}$ e $a_0 \prec a_1 \prec a_2$, é possível começar por qualquer uma dessas ordenações de árvore. Escolhendo a primeira, tem-se a ordem de redução nesse momento igual a: $a_0 \lhd a_{16} \lhd a_1 \lhd a_2$. Dado que o nó com posição a_2 é do tipo β', o sequente é dividido em dois ramos, chamados 'a' e 'b', como na árvore de fórmula. Assim, essa conexão fecha o ramo 'b', ramo onde se encontra o nó CO^1, e leva ao axioma $h^0, C^0 \vdash CO^1$, porque nós do tipo β' são associados a regra do corte (ver tabela 7). A figura 24, mostra esse estágio com a prova em conexões e uma estrutura parcial da prova em sequentes, e a tabela 9, mostra uma relação entre a conexão, as substituições e os ordenamentos.

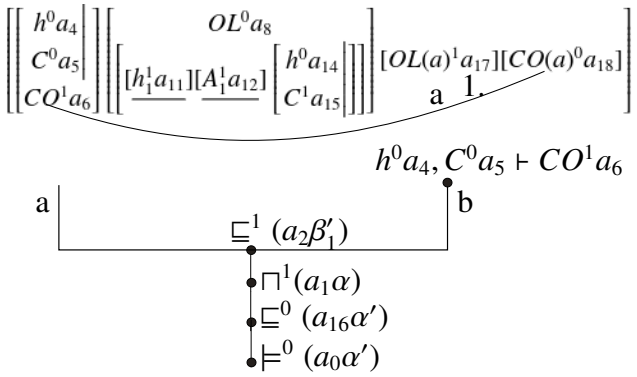

Figura 24 – Primeiro passo na prova de conexão/sequentes $\mathcal{ALC}$ de F_1.

Tabela 9 – Relação entre conexões, substituições e ordenamentos

Nº	Nós	σ_δ	$\sigma_{\beta'}$	$\sqsubset_\delta$	$\lhd$
1	$CO(a_2)^1a_6, CO(a)^0a_{18}$		a_2/a		$a_0 \lhd a_{16} \lhd a_1 \lhd a_2$
$\sigma_{Final} = a_2/a$					
$a_0 \lhd a_{16} \lhd a_1 \lhd a_2$					

Na segunda conexão, $\underline{C^0}$, com posição a_5 no ramo 'a', é conectado a $\underline{C^1}$, com posição a_{15} no ramo 'd'. O caminho entre os nós com essas posições é $\{a_5, a_3, a_2, a_1, a_7, a_9, a_{13}, a_{15}\}$. Como os nós com posições a_1 e a_2 já foram reduzidos, é preciso reduzir os nós das posições a_3, a_7, a_9 e a_{13}. Analisando essas posições, temos a ordenação de árvore $a_7 \prec a_9 \prec a_{13}$ e as relações $a_{10} \sqsubset_\delta a_3$ e $a_{10} \sqsubset_\delta a_{13}$. Logo, no momento só é possível reduzir o nó com posição a_7 e em seguida o nó com posição a_9, ou seja, $a_7 \lhd a_9$. Como o nó de posição a_7 possui o tipo β', sua redução divide o ramo 'a' nos ramos 'c' e 'd'. Em seguida, o nó de posição a_9, no ramo 'd', é reduzido. Como nesse caminho existem nós pendentes, ainda não é possível formar um axioma e fechar o ramo 'd'. Esse cenário é ilustrado através da figura 25 e da tabela 10.

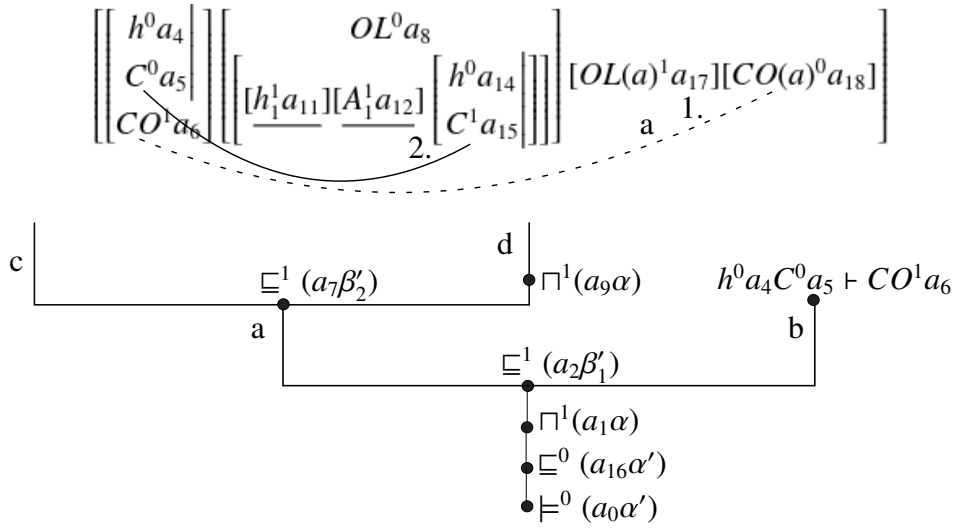

Figura 25 – Segundo passo na prova de conexão/sequentes $\mathcal{ALC}$ de F_1.

Tabela 10 – Relação entre conexões, substituições e ordenamentos

Nº	Nós	σ_δ	$\sigma_{\beta'}$	$\sqsubset_\delta$	$\lhd$
1	$CO(a_2)^1a_6, CO(a)^0a_{18}$		a_2/a		$a_0 \lhd a_{16} \lhd a_1 \lhd a_2$
2	$C(a_3)^0a_5, C(a_{13})^1a_{15}$			$a_{10} \sqsubset_\delta a_3$, $a_{10} \sqsubset_\delta a_{13}$	$a_7 \lhd a_9$
$\sigma_{Final} = a_2/a$					
$a_0 \lhd a_{16} \lhd a_1 \lhd a_2 \lhd a_7 \lhd a_9$					

A terceira conexão é analisada, onde $\underline{h^0}$, de posição a_{14}, é conectado a $\underline{h^1}$, de posição a_{11}, ambos no ramo 'd'. O caminho entre os nós é $\{a_{14}, a_{13}, a_9, a_{10}, a_{11}\}$. Como a_9 já foi reduzido, e existem as relações $a_{10} \sqsubset_\delta a_{13}$ e $a_{10} \sqsubset_\delta a_3$, o nó de posição a_{10} é reduzido, e 'junto' com ele os nós de posição a_{13} e a_3. A redução do nó de posição a_{10} torna a terceira e a segunda conexão complementares sob as substituições $\sigma_\delta = \{a_{13}/a_{10},\ a_3/a_{10}\}$, ver tabela 11. Com isso as duas últimas conexões são refletidas na prova de sequentes levando ao fechamento do ramo 'd'. Perceba que a segunda conexão só foi alcançada na árvore após a terceira conexão, isso leva ao axioma na forma $\Gamma, C^1 \vdash C^0, \Delta$, conforme ilustra a figura 26.

Tabela 11 – Relação entre conexões, substituições e ordenamentos

Nº	Nós	σ_δ	$\sigma_{\beta'}$	$\sqsubset_\delta$	$\lhd$
1	$CO(a_2)^1a_6, CO(a)^0a_{18}$		a_2/a		$a_0 \lhd a_{16} \lhd a_1 \lhd a_2$
2	$C(a_3)^0a_5, C(a_{13})^1a_{15}$	a_{13}/a_{10}, a_3/a_{10}		$a_{10} \sqsubset_\delta a_3$, $a_{10} \sqsubset_\delta a_{13}$	$a_7 \lhd a_9$
3	$h(a_7, a_{13})^0a_{14}, h(a_7, a_{10})^1a_{11}$	a_{13}/a_{10}			a_{10}
$\sigma_{Final} = a_2/a,\ a_{13}/a_{10},\ a_3/a_{10}$					
$a_0 \lhd a_{16} \lhd a_1 \lhd a_2 \lhd a_7 \lhd a_9 \lhd a_{10}$					

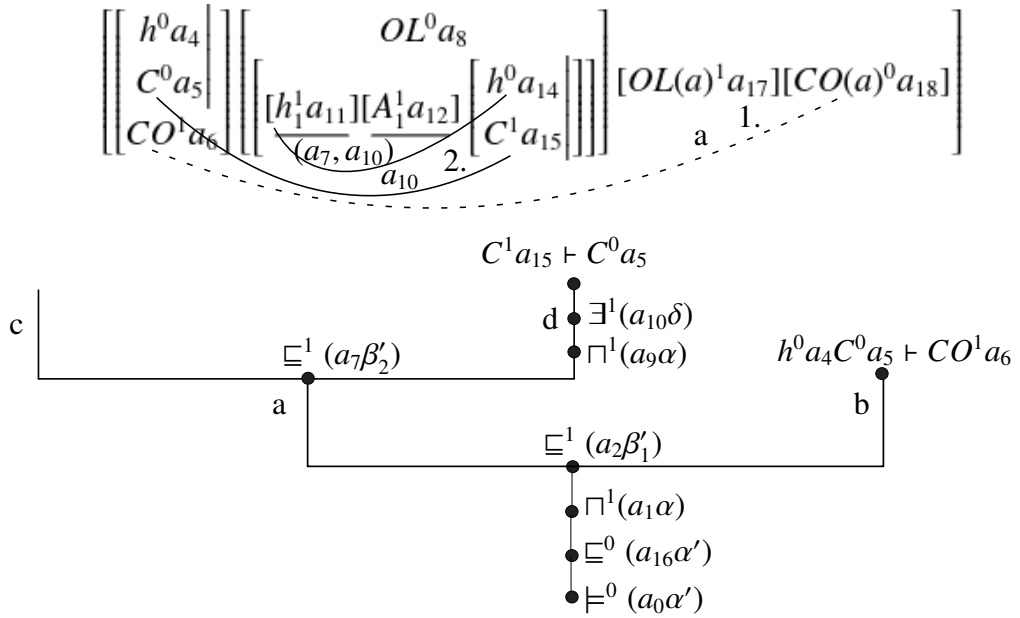

Figura 26 – Segundo e terceiro passos na prova de conexão/sequentes $\mathcal{ALC}$ de F_1.

A quarta conexão conecta OL^0, com posição a_8 no ramo 'c', a OL^1, com posição a_{17}. O caminho entre os nós com essas posições é $\{a_8, a_7, a_1, a_0, a_{16}, a_{17}\}$. Como todos nós do caminho já foram reduzidos, nenhuma redução será necessária nesse passo. Assim, o ramo 'c' é fechado com um axioma na forma $OL^0 \vdash h^1A^1, h^0C^1$, dada a regra do corte. A figura 27 ilustra esse cenário. Essa conexão é complementar sob $\sigma_{\beta'} = \{a_7/a\}$.

Na quinta e última conexão, que liga $\underline{h^0}$, com posição a_4, a $\underline{h^1}$, com posição a_{11}, também não necessita de redução de nós, uma vez que todos os nós no caminho já foram reduzidos. A conexão é complementar sob $\sigma_\delta = \{a_3/a_{10}\}$. Note que a_2/a e a_7/a foram substituições $\sigma_{\beta'}$ realizadas anteriormente.

Perceba que todas as conexões são complementares sob uma substituição σ_{Final}, todos os ramos da estrutura da prova em sequentes foram fechados, e a ordem de redução não é reflexiva, como mostram a figura 27 e a tabela 12.

Tabela 12 – Relação entre conexões, substituições e ordenamentos

Nº	Nós	σ_δ	$\sigma_{\beta'}$	$\sqsubset_\delta$	$\lhd$
1	$CO(a_2)^1a_6, CO(a)^0a_{18}$		a_2/a		$a_0 \lhd a_{16} \lhd a_1 \lhd a_2$
2	$C(a_3)^0a_5, C(a_{13})^1a_{15}$	a_{13}/a_{10}, a_3/a_{10}		$a_{10} \sqsubset_\delta a_3$, $a_{10} \sqsubset_\delta a_{13}$	$a_7 \lhd a_9$
3	$h(a_7, a_{13})^0a_{14}, h(a_7, a_{10})^1a_{11}$	a_{13}/a_{10}			a_{10}
4	$OL(a_7)^0a_8, OL(a)^1a_{17}$		a_7/a		
5	$h(a_2, a_3)^0a_4, h(a_7, a_{10})^1a_{11}$	a_3/a_{10}	a_2/a		
$\sigma_{Final} = a_2/a,\ a_{13}/a_{10},\ a_3/a_{10},\ a_7/a$					
$a_0 \lhd a_{16} \lhd a_1 \lhd a_2 \lhd a_7 \lhd a_9 \lhd a_{10}$					

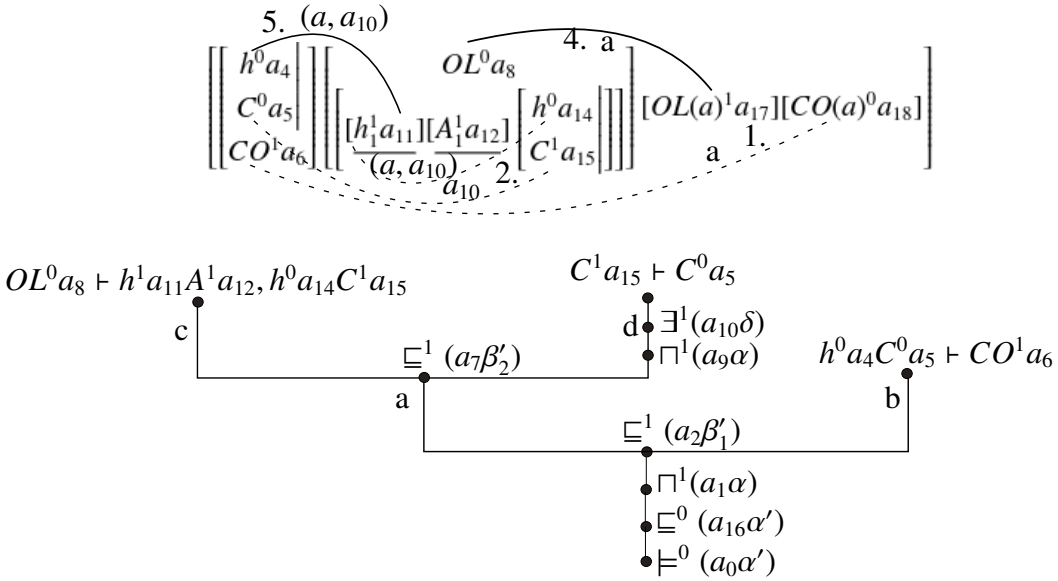

Figura 27 – Quarto e quinto passos na prova de conexão/sequentes $\mathcal{ALC}$ de F_1.

Etapa 4: Nessa etapa as instâncias da fórmula $\mathcal{ALC}$, utilizadas e fornecidas pelo método não-clausal, são omitidas. A estrutura obtida na etapa anterior é percorrida, e a cada nó encontrado é feita uma busca nas tabelas 7 e 8 para encontrar a regra correspondente. Nesse exemplo, os dois primeiros nós são do tipo α'. Como não existe regra associada a esse tipo, a prova começa propriamente com a redução do terceiro nó da estrutura, o nó de posição a_1. A regra l$\sqcap$ é aplicada, como ilustrado abaixo.

$$\frac{(\exists h.C \ \vdash \ CO,\ OL \ \vdash \ \exists h.A \sqcap \forall h.C) \ \vdash \ (OL \ \vdash \ CO)}{\Big(((\exists h.C \ \vdash \ CO) \sqcap (OL \ \vdash \ \exists h.A \sqcap \forall h.C)) \ \vdash \ (OL \ \vdash \ CO)\Big)}\ \mathrm{l}\sqcap$$

Em seguida, o nó de posição a_2, que possui o tipo β', é reduzido com a aplicação da regra do corte sob a consulta α, ou seja, sobre $(OL \vdash CO)$. A prova é divida nos ramos 'a' e 'b'. O ramo 'b' é fechado com o axioma inicial $\exists h.C \vdash CO$, enquanto o ramo 'a' fica aberto, no qual $OL \vdash \exists h.C$ deve ser provado.

$$\frac{\dfrac{OL \ \vdash \ \exists h.C \qquad \exists h.C \ \vdash \ CO}{(\exists h.C \ \vdash \ CO,\ OL \ \vdash \ \exists h.A \sqcap \forall h.C) \ \vdash \ (OL \ \vdash \ CO)}\ \mathrm{cut}}{\Big(((\exists h.C \ \vdash \ CO) \sqcap (OL \ \vdash \ \exists h.A \sqcap \forall h.C)) \ \vdash \ (OL \ \vdash \ CO)\Big)}\ \mathrm{l}\sqcap$$

O próximo nó é o de posição a_7 e do tipo β'. A redução desse nó, divide o ramo 'a' nos ramos 'c' e 'd', por meio da aplicação de uma nova regra do corte sobre $OL \vdash \exists h.C$. O ramo 'c' é fechado com o axioma inicial $OL \vdash \exists h.A \sqcap \forall h.C$, enquanto o ramo 'd' fica aberto.

$$\dfrac{\dfrac{OL \vdash \exists h.A \sqcap \forall h.C \qquad \exists h.A \sqcap \forall h.C \vdash \exists h.C}{OL \vdash \exists h.C}\,\text{cut} \qquad \exists h.C \vdash CO}{\dfrac{(\exists h.C \vdash CO,\ OL \vdash \exists h.A \sqcap \forall h.C) \vdash (OL \vdash CO)}{\Big(((\exists h.C \vdash CO) \sqcap (OL \vdash \exists h.A \sqcap \forall h.C)) \vdash (OL \vdash CO)\Big)}\,\text{l}\sqcap}\,\text{cut}$$

Para fechar o ramo 'd', o nó de posição a_9 é reduzido com a regra $l\sqcap$, seguido da redução do nó com posição a_{10}, através da regra $l\exists$. Isso conclui a prova em sequentes para F_1, como mostra a figura abaixo:

$$\dfrac{\dfrac{OL \vdash \exists h.A \sqcap \forall h.C \qquad \dfrac{\exists h.A \sqcap \forall h.C \vdash \exists h.C}{\dfrac{\exists h.A, \forall h.C \vdash \exists h.C}{\dfrac{}{A, C \vdash C}=}\,\text{l}\exists}\,\text{l}\sqcap}{OL \vdash \exists h.C}\,\text{cut} \qquad \exists h.C \vdash CO}{\dfrac{(\exists h.C \vdash CO,\ OL \vdash \exists h.A \sqcap \forall h.C) \vdash (OL \vdash CO)}{\Big(((\exists h.C \vdash CO) \sqcap (OL \vdash \exists h.A \sqcap \forall h.C)) \vdash (OL \vdash CO)\Big)}\,\text{l}\sqcap}\,\text{cut}$$

Figura 28 – Prova completa em sequentes $\mathcal{ALC}$ para F_1.

6 ALGORITMO E COMPLEXIDADE

Este capítulo apresenta os principais algoritmos para o método de conversão com suas complexidades, de acordo com as 4 etapas vistas na seção 5.2. Inicialmente é feita uma descrição sobre o funcionamento dos algoritmos, relacionando as variáveis e funções que eles utilizam. Logo depois seus pseudo-códigos são apresentados seguidos da complexidade e de um exemplo ilustrando possíveis entradas e saídas.

A complexidade de tempo é analisada de acordo com o tamanho da entrada de cada algoritmo. Por exemplo, alguns algoritmos aqui apresentados têm como entrada uma fórmula $\mathcal{ALC}$ F, assim o tamanho da entrada n representa o número de símbolos que F possui. Outros algoritmos terão como entrada a matriz de prova de F, nesse caso a entrada será o número de símbolos da matriz, incluindo as conexões formadas entre os literais. Essa entrada será representada por m. Entradas diferentes dessas são especificadas explicitamente durante a análise da complexidade.

A figura 29, apresenta uma visão geral da ordem de execução dos (oito) principais algoritmos por etapa. As linhas com setas indicam que a saída de um algoritmo é entrada para outro, por exemplo, a saída fornecida pelo algoritmo 02 (chamado de *converteEmPosFixa*), localizado na etapa 1, é utilizada como entrada para o algoritmo 04 (denominado *atribuiPosicao*), que se encontra na etapa 2.

Alguns dos algoritmos apresentados neste capítulo possuem chamadas para funções. Os algoritmos para essas funções são descritos no apêndice C junto com a análise de suas complexidades.

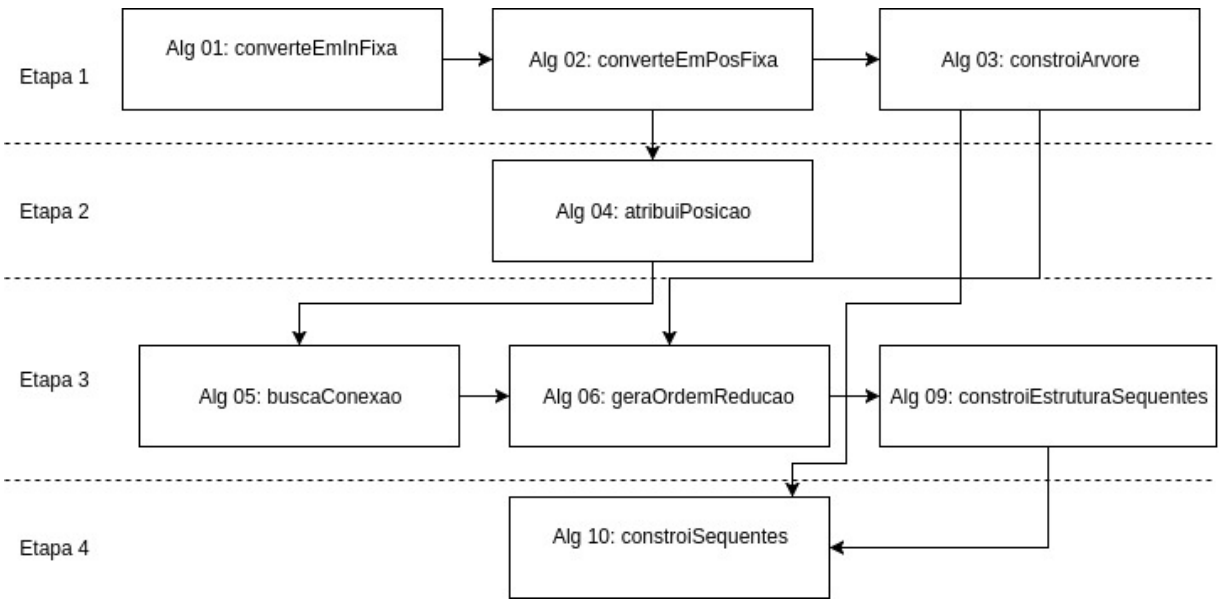

Figura 29 – Visão geral da ordem dos principais algoritmos.

Etapa 1- Construção da árvore de fórmula:

Algoritmo 1: Esse algoritmo converte uma fórmula $\mathcal{ALC}$ em sua forma infixa.

Ele recebe como entrada:

- $F[]$: um array com os elementos de uma fórmula $\mathcal{ALC}$. Os elementos são do tipo *Literal*.
  - *Literal*: é um modelo que representa um símbolo de uma fórmula. Possui o atributo *rotulo*, para representar um símbolo, e *posicao*, para representar a posição do símbolo na fórmula, conforme especificado na figura 30.

E como saída ele fornece:

- $Fin[]$: um array com os elementos de $F[]$, dispostos na sua forma infixa.

Literal
String rotulo;
inteiro posicao;

Figura 30 – O modelo LITERAL e seus atributos.

Algoritmo 1: CONVERTE FÓRMULA $\mathcal{ALC}$ PARA FORMA INFIXA

```
1  Função converteEmInFixa(F[])
2      Fin[];
3      quant;
4      i := 0;
5      para cada elemento m de F[] faça
6          se m.rotulo ≠ ∃ e m.rotulo ≠ ∀ e m.rotulo ≠ '.' então
7              Fin[i] := m;
8              i:=i+1;
9          senão se m.rotulo = ∃ ou m.rotulo = ∀ então
10             quant := m;
11         senão se m.rotulo = '.' então
12             Fin[i] := quant ;
13             i:=i+1;
14     retorna Fin;
```

Complexidade do algoritmo 1: O(n), pois na linha 5 o algoritmo processa *n* elementos de entrada. As operações dentro do loop aumentam a complexidade em um fator constante, não modificando a complexidade assintótica.

Exemplo 57. *(Algoritmo 1)*

Entrada:*F[] =*

(	(	(	∃	h	.	C	)	⊑	CO	)	⊓	(	OL	⊑	(	∃	h	.	A	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

⊓	(	∀	h	.	C	)	)	⊨	(	OL(a)	⊑	CO(a)	)	)
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Saída:*Fin[] =*

(	(	(	h	∃	C	)	⊑	CO	)	⊓	(	OL	⊑	(	h	∃	A	)
0	1	2	4	3	6	7	8	9	10	11	12	13	14	15	17	16	19	20

⊓	(	h	∀	C	)	)	⊨	(	OL(a)	⊑	CO(a)	)	)
21	22	24	23	26	27	28	29	30	31	32	33	34	35

Algoritmo 2: Converte uma fórmula infixa para sua forma pós-fixa. O algoritmo recebe:

- Fin[]: é um array com os elementos de uma dada fórmula F na sua forma infixa.

E utiliza:

- *pilha*: é uma variável do tipo Pilha usada para empilhar os elementos da fórmula;
- Fp[]: um array usado para armazenar os elementos (construtores e literais) da fórmula.

Algoritmo 2: CONVERTE FÓRMULA NA FORMA INFIXA PARA A FORMA PÓS-FIXA

```
1  Função converteEmPosFixa(Fin[])
2      Pilha pilha; construtor := {⊨, ⊑, ⊓, ⊔, ¬, ∃, ∀};
3      para cada elemento m de Fin[] faça
4          se m.rotulo = '(' então
5              empilhe m no topo da pilha;
6          senão se m.rotulo ∉ construtor e m.rotulo ≠ ')' então
7              empilhe m em Fp[];
8          senão se m.rotulo ∈ construtor então
9              empilhe m no topo da pilha;
10         senão se m.rotulo = ')' então
11             enquanto rotulo do elemento do topo da pilha ≠ '(' faça
12                 se o rotulo do elemento topo da pilha ∈ construtor então
13                     empilhe o elemento do topo da pilha em Fp[];
14             desempilhe o elemento do topo da pilha;
15     retorna Fp;
```

Complexidade do algoritmo 2: O(n^2), dado que existem dois loops aninhados, linha 4 e linha 12, e o tamanho máximo da pilha será o número de símbolos da fórmula.

Exemplo 58. *(Algoritmo 2)*
Considere Fin[] um array contendo os elementos de uma fórmula infixa, como representado abaixo, onde a linha numerada simboliza as posições de cada elemento nessa fórmula:

Entrada:*Fin[ ] =*

(	(	(	h	∃	C	)	⊑	CO	)	⊓	(	OL	⊑	(	h	∃	A	)
0	1	2	4	3	6	7	8	9	10	11	12	13	14	15	17	16	19	20

⊓	(	h	∀	C	)	)	⊨	(	OL(a)	⊑	CO(a)	)	)
21	22	24	23	26	27	28	29	30	31	32	33	34	35

Saída: *Fp[] =*

h	C	∃	CO	⊑	OL	h	A	∃	h	C	∀	⊓	⊑	⊓	OL(a)	CO(a)	⊑	⊨
4	6	3	9	8	13	17	19	16	24	26	23	21	14	11	31	33	32	29

Algoritmo 3: O algoritmo 3 constrói uma árvore sintática para representar uma fórmula.

Ele recebe como entrada:

- Fp[]: é um array que armazena os elementos de uma fórmula dispostos na notação pós-fixa,
- *pol*: um valor inteiro (0 ou 1), que representa a polaridade do literal. O valor inicial de *pol* é zero,
- *arcos*[]: um array de tamanho 4, com os índices para dois ramos do tipo β e β', respectivamente. *arcos*[] é inicialmente vazia;
- *posBetal*: variável que armazena a posição de um nó do tipo β', a fim de associá-la a nós folha;
- *posGDelta*: variável que armazena as posições de nós do tipo γ ou δ, a fim de associá-la a nós folha;
- *inStr* e *inEnd*: são duas variáveis inteiras que auxiliam na identificação dos nós que são folha. Seus valores iniciais são zero e n-1, respectivamente. Onde n é igual ao tamanho de Fp[];

O algoritmo utiliza ainda:

- *index*: uma variável global do tipo inteira que ajuda a controlar os índices de F[] e de *pos*[]. Seu valor inicial é n-1 (n = tamanho de Fp[]);
- *pos*[]: é uma variável global do tipo array gerada pela função 11, a qual simula a retirada dos parênteses da fórmula gerando novas posições para os literais na fórmula sem parênteses;
- *trl*[]: um array de tamanho 3. Onde trl[0] armazena o tipo do nó, trl[1] a polaridade do filho da direita do nó, e trl[2] a polaridade do filho da esquerda do nó. Esses valores são retornados por uma função chamada *buscaTipoPol*;
- $buscaTipoPol$: é uma função que recebe o rótulo e polaridade de um nó, e busca em uma tabela com 12 entradas (ver tabela 13, que representa as definições na tabela 6), o tipo do nó e as polaridades de seus filhos.
- *no*: é um ponteiro do tipo NoArvore. Ele é usado para representar nós na árvore de fórmula, e possui atributos conforme especificado na figura 31. Esse algoritmo retorna *no*, que deverá representar o nó raiz da árvore de fórmula de Fp[].

Tabela 13 – Tipos e Polaridades

Tipo	**Con**	**Pol**	**PolNoE**	**PolNoD**
α	$\sqcap$	1	1	1
α	$\sqcup$	0	0	0
α	$\neg$	1		0
α	$\neg$	0		1
α'	$\sqsubseteq$	0	1	0
α'	$\models$	0	1	0
β'	$\sqsubseteq$	1	0	1
β	$\sqcap$	0	0	0
β	$\sqcup$	1	1	1
δ	$\forall$	0	1	0
δ	$\exists$	1	1	1
γ	$\forall$	1	0	1
γ	$\exists$	0	0	0

Algoritmo 3: CONSTRÓI ÁRVORE DE FÓRMULA

```
1  Função constroiArvore(inStr, inEnd, Fp[], pol, arcos[], posBetal, posGDelta)
2      NoArvore no;
3      trl[];
4      int posBl := posBetal;
5      int posGD := posGDelta;
6      construtor := {⊨, ⊑, ⊓, ⊔, ¬, ∃, ∀};
7      se inStrt > inEnd então
8          retorna null;
9      se Fp[index].rotulo ∈ construtor então
10         trl := buscaTipoPol(Fp[index].rotulo,pol);
11         se trl[0] = β então
12             arcos[0]:= arcos[0] +1;
13             arcos[1]:= arcos[0] +1;
14             no := (Fp[index].rotulo, Fp[index].posicao, pol,trl[0], arcos[0],arcos[1]);
15         senão se trl[0] = β′ então
16             arcos[2]:= arcos[2] +1;
17             arcos[3]:= arcos[2] +1;
18             posBl:= Fp[index].posicao;
19             no := (Fp[index].rotulo, Fp[index].posicao, pol,trl[0], arcos[2],arcos[3]);
20         senão se trl[0] = γ ou δ então
21             posGD:= Fp[index].posicao;
22             no := (Fp[index].rotulo, Fp[index].posicao, pol,trl[0]);
23         senão
24             no := (Fp[index].rotulo, pos[index].posicao, pol,trl[0]);
25     senão
26         no := (Fp[index].rotulo, Fp[index].posicao, pol, posBl, posGD);
27     int m := pos[index];
28     index := index - 1;
29     se inStrt = inEnd então
30         retorna no;
31     no.direita := constroiArvore(m+1, inEnd, Fp[], trl[1], arcos, posBl, posGD);
32     no.esquerda := constroiArvore(inStr, m-1, Fp[], trl[2], arcos, posBl, posGD);
33     retorna no;
```

NoArvore
String rotulo;
inteiro polaridade;
inteiro posicao;
String tipo;
array posSubst[2]
String instancia
apontador direita
apontador esquerda

Figura 31 – O modelo para um NoArvore e seus atributos.

Complexidade: $O(n^2)$. A cada iteração, no pior caso, reduz-se a fórmula à metade. Logo, haverá $log_2 n$ iterações de complexidade $O(n)$.

Exemplo 59. *(Algoritmo 3)*
A saída do algoritmo 2 é entrada para o algoritmo 3.
Entrada: $Fp[] =$

h	C	∃	CO	⊑	OL	h	A	∃	h	C	∀	⊓	⊑	⊓	OL(a)	CO(a)	⊑	⊨
4	6	3	9	8	13	17	19	16	24	26	23	21	14	11	31	33	32	29

Saída:

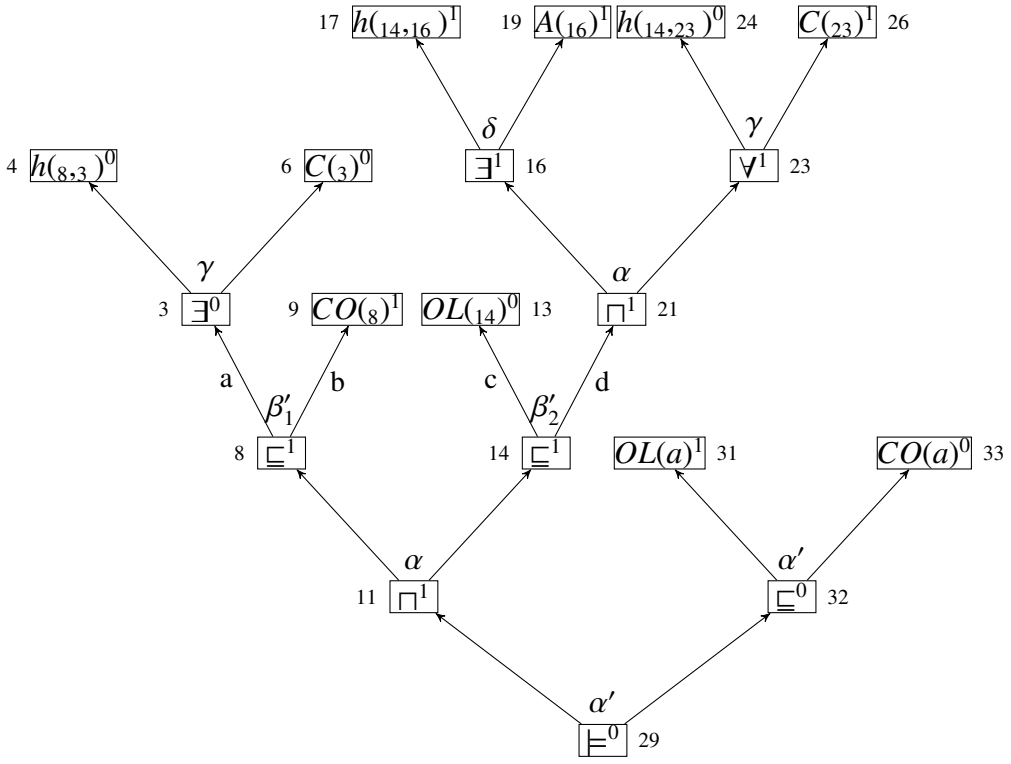

Figura 32 – Árvore de fórmula para $Fp[]$.

Etapa 2- Atribuição de posições aos elementos da matriz:

Algoritmo 4: Tanto os elementos da matriz, como os nós na árvore de fórmula, são símbolos da fórmula de entrada. Esses símbolos possuem uma posição na fórmula. Um valor para essas posições nos nós da árvore foi atribuído na etapa anterior. Na etapa 2, esse algoritmo tem a função de atribuir os valores das posições aos elementos da matriz. Note que, todo elemento da matriz de prova é um nó folha na árvore de fórmula, e a construção da matriz segue a ordem dos símbolos da fórmula. Logo, assim como ocorreu na construção da árvore de fórmula, esse algoritmo utiliza a fórmula pós-fixa e as posições dos seus símbolos para atribuir as posições aos elementos da matriz.

Esse algoritmo recebe como entrada:

- *matriz*[]: é um array que representa a matriz de prova gerada pelo método de conexões não-clausal para $\mathcal{ALC}$. Seus elementos podem ser arrays ou do tipo *Elemento*:
  - *Elemento*: é um modelo que representa um literal. Ele possui os seguintes atributos: id, literal, polaridade, posicao e conexoes, conforme especificado na figura 33;
  - *Conexao*: é um modelo que representa uma conexão entre dois literais, e contém os seguintes atributos: idelemento1, idelemento2 e ordem. Os dois primeiros são usados para identificar os elementos da matriz, e ordem, para representar a ordem e também identificar a conexão, conforme especificado na figura 34.
- Fp[]: é um array com os elementos de uma fórmula dispostos na notação pós-fixa;
- *indice*: é uma variável inteira, usada para controlar os índices de Fp[]. Ela é inicialmente zero.

Como saída:

- *matriz*[]: esse algoritmo retorna a *matriz*[] atualizada, contendo os valores para posições dos seus literais.

Elemento
inteiro id;
String literal;
inteiro polaridade;
inteiro posicao;
Conexao conexoes[]

Figura 33 – O modelo ELEMENTO e seus atributos.

Conexao
inteiro idelemento1;
inteiro idelemento2;
inteiro ordem

Figura 34 – O modelo CONEXAO e seus atributos.

Algoritmo 4: ATRIBUI POSIÇÕES AOS ELEMENTOS DA MATRIZ

```
1  Função atribuiPosicao(matriz[], Fp[],indice)
2      achou := false;
3      construtor := {⊨, ⊑, ⊓, ⊔, ¬, ∃, ∀};
4      para cada elemento i da matriz[] faça
5          se matriz[i] não é do tipo Elemento então
6              atribuiPosicao(matriz[i],Fp[],indice);
7          senão
               /* é um elemento                                    */
8              achou := false;
9              repita
10                 se Fp[indice].rotulo ∉ construtor então
11                     se Fp[indice].rotulo = matriz[i].literal então
12                         matriz[i].posicao := Fp[indice].posicao;
13                         achou := true;
14                         indice := indice + 1;
15                     senão
16                         indice := indice + 1;
17                 senão
18                     indice := indice + 1;
19             até (indice > tamanho de Fp[]) ou (achou = true);
20     retorna matriz;
```

Complexidade do algoritmo 4: $m^2 \times n$, logo O($m^2 \cdot n$), dada as iterações da linha 4 e n iterações da linha 9.

Exemplo 60. *(Algoritmo 4)*
Esse algoritmo recebe como entrada uma matriz de prova e uma fórmula pós-fixa. A fórmula pós-fixa é a saída do algoritmo 2. Na matriz, cada elemento possui seus atributos dispostos na seguinte ordem: { *id, literal, polaridade, posição, conexões* }.
Entrada: *matriz =*

$$\{\{1, h, 0, null, 5;\ 2, C, 0, null, 2;\ 3, CO, 1, null, 1\},\ \{4, OL, 0, null, 4; \{5, h, 1, null, \{3, 5\}\},$$
$$\{6, A, 1, null, null\}\ \{7, h, 0, null, 3; 8, C, 1, null, 2\}\}, \{9, OL(a), 1, null, 4\}, \{10, CO(a), 0, null, 1\}\ \}$$

$Fp[] =$

h	C	∃	CO	⊑	OL	h	A	∃	h	C	∀	⊓	⊑	⊓	OL(a)	CO(a)	⊑	⊨
4	6	3	9	8	13	17	19	16	24	26	23	21	14	11	31	33	32	29

Saída:

$$\{\{1,h,0,4,5;\ 2,C,0,6,2;\ 3,CO,1,9,1\},\ \{4,OL,0,13,4;\{5,h,1,17,\{3,5\}\},$$
$$\{6,A,1,19,null\}\{7,h,0,24,3;8,C,1,26,2\}\},\{9,OL(a),1,31,4\},\{10,CO(a),0,33,1\}\ \}$$

Etapa 3- Construção da estrutura da prova (parcial) em sequentes:

Algoritmo 5: O algoritmo 5, percorre uma matriz de prova em busca das conexões entre seus elementos.

Esse algoritmo recebe como entrada:

- *matriz*[]: é um array que representa a matriz de prova gerada pelo método de conexões não-clausal para $\mathcal{ALC}$, contendo as posições de seus elementos;
- *C*[]: é um array usado para armazenar um conjunto de conexões entre nós. Ele é inicialmente vazio. Seus elementos são do tipo ConexaoNo:
  - *ConexaoNo*: é um modelo usado para representar conexões entre dois nós. Seus atributos são: posicao1, posicao2 e ordem. posicao1 e posicao2, armazenam as posições dos nós folhas conectados, enquanto o último identifica e representa a ordem da conexão entre os nós. Ver figura 35.
- *indice*: variável inteira, usada para controlar os índices de *C*[]. Seu valor inicial é zero;

O algoritmo utiliza ainda:

- *constaConexao*: é uma função que checa se uma conexão consta no array de conexões, dados o array *C*[] e a ordem *ordem*;
- *buscaPosicao*: é uma função que localiza a posição de um elemento na matriz, dado o identificador do elemento (ver algoritmo 13). Ele recebe uma matriz de prova, *matriz*[], e um identificador do elemento da matriz, *idelemento*, e retorna a posição do elemento correspondente ao identificador através da variável *poselemento*;
- *ordenaConexao*: é uma função que ordena as conexões em ordem crescente (ver algoritmo 14). Ele recebe um conjunto de conexões *C*[] e retorna esse conjunto ordenado crescentemente pelo campo ordem.

Como saída, o algoritmo fornece:

- *C*[]: é um array contendo um conjunto de conexões encontradas pelo algoritmo.

ConexaoNo
inteiro posicao1;
inteiro posicao2;
inteiro ordem

Figura 35 – O modelo ConexaoNo e seus atributos.

Algoritmo 5: Busca Conexões

```
1  Função buscaConexao(matriz[], C[], indice)
2      para cada elemento i da matriz[] faça
3          se matriz[i] não é do tipo Elemento então
4              buscaConexao(matriz[i], C[], indice);
5          senão
               /* se é diferente de null, então possui pelo menos uma conexão
                  */
6              se matriz[i].conexao[] ≠ null então
7                  para cada conexao j de matriz[i].conexao[] faça
8                      se constaConexao(C[],matriz[i].conexao[j].ordem) = false então
                           /* conexão ainda não consta no conjunto de conexões.
                              */
9                          se matriz[i].id = matriz[i].conexao[j].idelemento1 então
10                             C[indice].posicao1 := matriz[i].posicao;
11                             C[indice].posicao2 =
                                buscaPosicao(matriz[i].conexao[j].idelemento2);
12                         senão
13                             C[indice].posicao2 := matriz[i].posicao;
14                             C[indice].posicao1 :=
                                buscaPosicao(matriz[i].conexao[j].idelemento1);
15                         C[indice].ordem := matriz[i].conexao[j].ordem;
16                         indice := indice + 1;
       /* ordenar as conexões pelo campo ordem                                */
17     C[] := ordenaConexao(C[]);
18     retorna C;
```

Complexidade do algoritmo 5: O(m^4), dada as m iterações da linha 2 e m iterações da linha 7, e as chamadas, dentro do escopo dessas iterações, das funções:

- buscaPosicao (ver algoritmo 13), linhas 11 e 14, com complexidade O(m^2),

- constaConexao (ver algoritmo 12), linha 8, com complexidade O(c),

temos a complexidade: $m \times m \times (m^2 + c) = m^2 \times (m^2 + c) = m^4 + m^2 \times c$.

- ordenaConexao (ver algoritmo 14), na linha 17, tem complexidade: O(c^2). Note que, a chamada para essa função está fora do escopo das iterações citadas acima. Assim: $m^4 + m^2 \times c + c^2 = O(m^4)$.

Exemplo 61. *(Algoritmo 5)*
Esse algoritmo recebe como entrada a saída do algoritmo 4, ou seja, uma matriz com as posições atribuídas aos elementos. Busca as conexões formadas entre os elementos, gerando um conjunto de conexões, que são ordenadas pelo campo ordem.
Entrada: *matriz =*

$$\{\{1,h,0,4,5;\ 2,C,0,6,2;\ 3,CO,1,9,1\},\ \{4,OL,0,13,4;\{5,h,1,17,\{3,5\}\},$$
$$\{6,A,1,19,null\}\{7,h,0,24,3;8,C,1,26,2\}\},\{9,OL(a),1,31,4\},\{10,CO(a),0,33,1\}\ \}$$

Saída: *C[]=*

ordem	posicao1	posicao2
1	33	9
2	6	26
3	24	17
4	13	31
5	4	17

Algoritmo 6: Esse algoritmo gera o ordenamento de redução dos nós de uma fórmula F. À medida que ele vai gerando esse ordenamento, ele verifica se esse ordenamento é reflexivo. Se for reflexivo, não é possível construir uma prova em sequentes, assim, ele interrompe sua execução. Ele também verifica a relação de ordem entre as posições, para evitar que uma posição γ, por exemplo, seja reduzida antes de uma posição δ. Ele ainda realiza as substituições de posições, verificando se elas são permitidas. Seu pseudo-código foi divido em partes devido a sua extensão, além disso ele utiliza várias funções para auxiliar no seu processamento.

O algoritmo recebe como entrada:

- $C[]$: é um array que possui um conjunto de conexões formadas entre os nós folha,
- *no*: é um nó raiz da árvore de uma fórmula F.

E utiliza:

- cn_1: é um vetor usado para armazenar os nós que são caminho entre o nó raiz e um nó folha na árvore que forma uma conexão;

- cn_2: é um vetor usado para armazenar os nós que são caminho entre o nó raiz e um nó folha na árvore que forma uma conexão;
- R: é um vetor usado para armazenar os nós que foram reduzidos durante o processamento do algoritmo;
- P: é um vetor que armazena um conjunto de nós que ainda precisam ser reduzidos pelo algoritmo. Esses nós podem não ter sido visitados ainda na árvore ou estão pendentes por alguma restrição de ordenamento;
- BUSCACAMINHO: uma função que busca o caminho entre o nó raiz e um nó folha, dado o nó raiz, a posição do nó folha que formou uma conexão;
- SUBSTITUIPOSICAO: é uma função que realiza a substituição de posições σ_δ ou $\sigma_{\beta'}$. O que vai determinar se a substituição é σ_δ ou $\sigma_{\beta'}$, são as entradas para essa função. Assim, a função recebe dois nós, $no1$, $no2$ e um array de substituições σ, que pode ser σ_δ ou $\sigma_{\beta'}$;
- SUBSTITUIPOSICAOFINAL: é uma função que realiza a substituição final com a união de σ_δ e $\sigma_{\beta'}$
- CHECAREFLEXIVIDADE: essa função verifica se o ordenamento de redução de nós é reflexivo. Ela recebe $\lhd$ e retorna 0, se $\lhd$ NÃO é reflexiva, e 1, caso contrário;
- REMOVENO: essa função remove nós do conjunto de nós que precisavam ser reduzidos, aguardando alguma condição ser satisfeita, como por exemplo, nós do tipo γ que só podem ser reduzidos após nós do tipo δ. Para isso, ela recebe como entrada um nó e um array;
- CONSTANO: essa função checa se um determinado nó já consta no conjunto de nós que esperam para ser reduzidos. Seu objeetivo é evitar que esse conjunto possua elementos repetidos;
- BUSCANOTIPO: essa função busca em um conjunto de nós, nós de um dado tipo. Para isso ela recebe um *tipo* e um array, e retorna os nós com esse tipo, caso existam.
- O Algoritmo 7 e Algoritmo 8, são partes do código desse algoritmo, que estão separados devido a extensão do algoritmo como um todo.

Como saída, o algoritmo fornece:

- $\lhd$: é um array com nós dispostos de acordo com a ordem de redução obtida, incluindo um conjunto de nós (cujos os rótulos são predicados) que formaram conexão.

Algoritmo 6: GERA A ORDEM DE REDUÇÃO DAS POSIÇÕES

```
1  Função geraOrdemReducao(C, no)
2      para cada elemento i de C[] faça
           /* Busca o caminho para cada nó folha que forma a conexão      */
3          cn1[] = buscaCaminho(no, C[i].posicao1, ∅);
4          cn2[] = buscaCaminho(no, C[i].posicao2, ∅);
5          para cada caminho cnz faça
6              continua := true;
7              conectar := false;
8              j := 0;
9              repita
                   /* se o nó é do tipo folha                              */
10                 se cnz[j].tipo = null então
11                     empilhe cnz[j] no topo de P;
12                     continua := false;
13                     se z = 2 então
                           /* no 2º caminho e nó folha, então tentar conectar */
14                         conectar := true;
15                 senão
                       /* Se o nó cnz[j] não está no conj. R de nós que foram
                          reduzidos. Se ele está, não precisa mais inseri-lo. */
16                     se (cnz[j] ∉ R) então
17                         Executar Instruções do Algoritmo 7;
18                         j := j + 1;
19                     senão
20                         j := j + 1;
21             até continua = false;
22         se conectar = true então
23             Executar Instruções do Algoritmo 8;
24     retorna ◁;
```

Complexidade do algoritmo 6: O(n^2), dado que $c/2 \times 2n + 2 \times (n \times n) + n^2 = O(n^2)$, onde, na sequência temos:

- $c/2$ é número de iterações da linha 2;
- $2n$ se refere as duas chamadas para buscaCaminho (ver algoritmo 15), nas linhas 3 e 4;
- 2 é o número máximo de iterações da linha 5;
- n é o número de iterações da linha 9;
- n, número de operações da linha 17, que está dentro do escopo da iteração da linha 9;
- n^2, se refere as operações na linha 23, instruções do algoritmo 8.

Algoritmo 7: VERIFICA TIPO DA POSIÇÃO

```
1  se (cn_z[j].tipo = α ou β ou α') então
2      empilhe cn_z[j] em R;
3      empilhe cn_z[j] em em ◁;
4      se (cn_z[j].tipo = α') então
5          empilhe cn_z[j] em AlphaL;
6  senão se (cn_z[j].tipo = δ) então
7      empilhe cn_z[j] em Delta;
8      empilhe cn_z[j] em R;
9      empilhe cn_z[j] em ◁;
10     nosGamma[] := buscaNosTipo(γ, P);
11     se nosGamma[] ≠ ∅ então
12         para cada nó gamma em nosGamma[] faça
13             σ_δ:= SubstituiPosicao(gamma, Delta, σ_Final);
14             empilhe nó gamma em R;
15             removeNo(gamma, P);
16             σ_Final := σ_δ;
17 senão se (cn_z[j].tipo = β') então
18     se AlphaL ≠ ∅ então
19         σ_β':= SubstituiPosicao(cn_z[j], AlphaL, σ_Final);
20         empilhe cn_z[j] em R;
21         empilhe cn_z[j] em ◁;
22         σ_Final := σ_β';
23     senão
24         para cada nó no caminho cn_z a partir de j faça
25             se (ConstaNo(cn_z[j],P) = false) então
26                 empilhe cn_z[j] no topo de P;
27             j := j + 1;
28         continua := false;
29 senão se (cn_z[j].tipo = γ) então
30     se Delta ≠ ∅ então
31         σ_δ:= SubstituiPosicao(cn_z[j], Delta, σ_Final);
32         empilhe cn_z[j] em R;
33         σ_Final := σ_δ;
34     senão
35         para cada nó no caminho cn_z a partir de j faça
36             se (ConstaNo(cn_z[j],P) = false) então
37                 empilhe cn_z[j] no topo de P;
38             j := j + 1;
39         continua := false;
```

Algoritmo 8: CONECTAR

```
1  temp := SUBSTITUIPOSICAOFINAL(cn1[], cn2[], σFinal);
2  se temp ≠ null então
3      σFinal := temp;
4      resp := CHECAREFLEXIVIDADE(◁);
       /* Se resp = 0, então ◁ NÃO é reflexiva, caso contrário, é.      */
5      se (resp = 0) então
6          parLit[0]:= elemento do topo de cn1;
7          parLit[1]:= elemento do topo de cn2;
8          predicado := false;
           /* Relações são binárias, logo posSubst é igual a 2. Predicados,
              são unários, então posSubst = 1.                          */
9          se tamanho de parLit[0].posSubst = 1 então
10             predicado := true;
11         achou := false;
12         para (k := 0; k < 2, k++) faça
13             se (parLit[k] ∈ R) então
14                 achou := true;
15                 desempilhe elemento do topo de P;
16             senão
17                 empilhe o elemento do topo de P em R;
18                 desempilhe elemento do topo de P;
19         se (achou = false) e (predicado = true) então
               /* O par de literais que formam a conexão, parLit, são
                  armazenados em ◁. Isso fechará um ramo na estrutura da
                  prova parcial.                                        */
20             empilhe parLit em ◁;
21     senão
           /* ◁ é reflexiva, não é possível gerar prova em sequentes    */
22         retorna null;
23 senão
24     retorna null;
```

Complexidade do algoritmo 7: O(n), dado que todas as funções chamadas são de complexidade O(n), e estão dentro de estruturas de condição, sendo executadas apenas uma vez.

Complexidade do algoritmo 8: O(n^2), dado que a única iteração é a da linha 12, que pode ser executada no máximo 2 vezes, e as chamadas para as funções abaixo, não estão dentro de estrutura de repetição:
- SubstituiPosicaoFinal (ver algoritmo 21): complexidade $O(n^2)$
- checaReflexividade (ver algoritmo 20): complexidade $O(n^2)$
Assim, $O(n^2) + O(n^2) = O(n^2)$

Exemplo 62. *(Algoritmo 6)*
Esse algoritmo recebe como entrada a saída do algoritmo 5, ou seja, conjunto de conexões, que são ordenadas pelo campo ordem. Além disso, ele recebe o nó raiz de árvore de fórmula de F (ver figura 36).
Entrada: $C[]$ =

ordem	**posicao1**	**posicao2**
1	33	9
2	6	26
3	24	17
4	13	31
5	4	17

Saída:
$\lhd = \{17, 15, 11, 3, \{18, 4\}, 7, 5, 9, \{2, 14\}, \{6, 16\}\}$

Algoritmo 9: Esse algoritmo constrói uma estrutura de uma prova (parcial) em sequentes para F.

Ele recebe como entrada:

- $\lhd$: um array, onde os elementos são nós ou conjunto de nós, ordenados por ordem de redução. O conjunto de nós, representam nós folhas que formam uma conexão,
- idx: um variável inteira usada como índice de $\lhd$. O valor inicial de idx é zero.

Como saída ele retorna:

- $noEst$: um nó raíz da estrutura da prova (parcial) gerada.

Algoritmo 9: CONSTRÓI A ESTRUTURA DA PROVA (PARCIAL) EM SEQUENTES

```
1  Função constroiEstruturaSequentes(◁, idx)
2      i := idx;
3      se ◁[i] é um conjunto de nós então
4          retorna ◁[i];
5      senão
6          noEst := ◁[i];
7          noEst.direita := constroiEstruturaSequentes(◁, i + 1);
8          se noEst.tipo = β ou β′ então
9              noEst.esquerda := constroiEstruturaSequentes(◁, i + 2);
10     retorna noEst;
```

Complexidade ao algoritmo 9: O(n^2). A cada iteração, no pior caso, reduz-se a fórmula à metade. Portanto, haverá $\log_2 n$ iterações de complexidade $O(n)$.

Exemplo 63. *(Algoritmo 9)*

A saída do algoritmo 6, é entrada para esse algoritmo.

Entrada:

$\lhd = \{29, 32, 11, 8, \{33, 9\}, 14, 21, 16, \{6, 26\}, \{13, 31\}\}$

Saída:

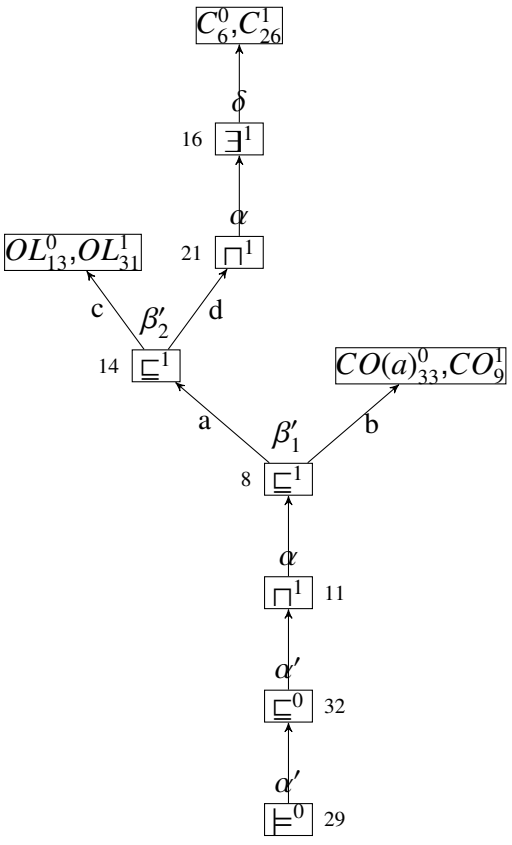

Figura 36 – Árvore de fórmula para $Fp[]$.

Etapa 4- Construção da prova completa em sequentes:

Algoritmo 10: Esse algortimo contrói a prova completa em sequentes.

O algortimo recebe como entrada:

- F: é um nó que possui três campos: *Lit*, *direita* e *esquerda*. O campo *Lit* um array do tipo *Literal*, como especificado na figura 30, e *direita* e *esquerda* são dois ponteiros;
- *noArv*: o nó raiz da árvore de fórmula;
- *noEst*: um nó raíz da estrutura da prova (parcial) de sequentes;
- *lado*: uma variável do tipo char, que recebe um caracter 'D' ou 'L'. Esses caracteres indicam se a aplicação das regras que causam ramificação na prova deve ser aplicada para ramificar o lado direito ou esquerdo. Inicialmente, o valor dessa variável é vazio.
- *regra*: indica uma das regras do cálculo de sequentes, por exemplo l$\sqcap$;
- *i*: um índice para diferenciar entre a primeira e demais execuções do algoritmo que é recursivo. Seu valor inicial é zero;
- S : é uma variável global, inicialmente vazia, que guarda o sequente a ser provado em um dos ramos da prova. Seu valor é gerado pela função aplicaRegraCorte.

E utiliza:

- aplicaRegraCorte: uma função que implementa a regra do corte;
- aplicaRegraL$\sqcap$R$\sqcup$: função que implementa as regras l$\sqcap$ e r$\sqcup$;
- aplicaRegraR$\neg\sqcap$L$\neg\sqcup$: função que implementa as regras r$\neg\sqcap$ e l$\neg\sqcup$;
- aplicaRegraL$\neg\neg$R$\neg\neg$: função que implementa as regras l$\neg\neg$ e r$\neg\neg$;
- aplicaRegraL$\sqcup$R$\sqcap$: função que implementa as regras l$\sqcup$, r$\sqcap$, l$\neg\sqcap$ e r$\neg\sqcup$;
- aplicaRegraR$\forall$L$\exists$: função que implementa as regras r$\forall$, l$\exists$, l$\neg\forall$ e r$\neg\exists$.

Algoritmo 10: CONSTRÓI PROVA EM SEQUENTES

```
1  Função constroiSequentes(F, noArv, noEst, lado, regra, i, S)
2      se i>0 então
3          se regra = l⊓ ou r⊔ então
4              F := aplicaRegraL⊓R⊔(F, noEst);
5          senão se regra = r¬⊓ ou l¬⊔ então
6              F := aplicaRegraR¬⊓L¬⊔(F, noEst, noArv);
7          senão se regra = l¬¬ ou r¬¬ então
8              F := aplicaRegraL¬¬R¬¬(F, noEst, noArv);
9          senão se regra = l⊔ ou r⊓ ou l¬⊓ ou r¬⊔ então
10             F := aplicaRegraL⊔R⊓(F, noEst, lado);
11         senão se regra = r∀ ou l∃ ou l¬∀ ou r¬∃ então
12             F := aplicaRegraR∀L∃(F, noEst)
13         senão se regra = cut então
14             se noEst é um conjunto de nós então
15                 F := aplicaRegraCorte(F, noEst, lado, S);
16                 retorna F;
17             senão
18                 j := 0;
19                 enquanto S ≠ ∅ faça
20                     F[j]:= o elemento do topo de S;
21                     desempilhe o elemento do topo de S;
22                     j:= j +1;
23     se noEst é um conjunto de nós então
24         retorna null;
25     senão
26         regra := buscaRegraSequentes(noArv, noEst);
27         enquanto regra = 0 faça
28             regra := buscaRegraSequentes(noArv, noEst.direita);
29     se noEst.tipo = β′ então
30         se noEst.direita é um conjunto de nós então
31             F.direita := constroiSequentes(F, noArv, noEst.direita, 'D', regra, 1, S);
32             F.esquerda := constroiSequentes(F,noArv,noEst.esquerda,'E', regra, 1, S);
33         senão
34             F.esquerda := constroiSequentes(F,noArv,noEst.esquerda,'E', regra, 1, S);
35             F.direita := constroiSequentes(F, noArv, noEst.direita, 'D', regra, 1, S);
36     senão
37         F.direita := constroiSequentes(F, noArv, noEst.direita,'D', regra, 1, S);
38     se noEst.tipo = β então
39         F.esquerda := constroiSequentes(F, noArv, noEst.esquerda, 'E', regra, 1, S);
40     retorna F;
```

Complexidade do algoritmo 10: O(n^3), pois analisando a chamada de:
- aplicaRegraL⊓R⊔, na linha 4, tem $O(n)$;
- aplicaRegraR¬⊓L¬⊔, na linha 6, $O(n)$;
- aplicaRegraL¬¬R¬¬, na linha 8, tem $O(n)$
- aplicaRegraL⊔R⊓, na linha 10, tem $O(n)$
- aplicaRegraR∀L∃, na linha 12, tem $O(n^2)$, e
- aplicaRegraCorte, na linha 15, tem $O(n)$.

Até aqui a complexidade é $O(n^2)$.
- buscaRegraSequentes, na linha 26, tem $O(n)$
- na linha 27, tem uma iteração para $n/2$ entradas, e
- buscaRegraSequentes, na linha 28, tem $O(n)$, dentro do escopo da iteração da linha 27.

Logo, temos nesse outro ponto, $O(n^2)$.
Como há uma recursividade, que reduz no pior caso a árvore pela metade, temos:
$\log n$ recursões $\times(O(n^2) + O(n^2)) = O(n^3)$,

7 CONCLUSÕES

7.1 Contribuições

Lógica de Descrições é um formalismo lógico que oferece meios poderosos de representar conhecimento e realizar inferências. Já existem boas soluções para inferir sobre bases de conhecimento descritas em Lógia de Descrições. O formato das provas, porém, ainda é alvo de pesquisas, quando se trata de descrever os passos usados para se chegar às conclusões. O Método de Conexões é considerado como um método eficiente para Lógica de Primeira Ordem e já possui uma variante para inferir sobre Lógica de Descrições $\mathcal{ALC}$. Porém, apresenta provas de difícil compreensão, que dificultam a interação com usuários em geral.

O presente trabalho partiu da premissa de que a junção do poder de expressividade de Lógica de Descrições com o bom desempenho de provadores automáticos de teorema aponta para uma forma promissora de oferecer serviços de raciocínio automático. E dado que os Sistemas Baseados em Conhecimento visam apoiar o processo decisório, a necessidade de descrições legíveis, de como os resultados foram inferidos, se torna imprescindível.

Isto posto, as contribuições desse trabalho são:

- Formulação de um Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$

  Conforme visto na seção 4.2, esse cálculo não requer a transformação da fórmula de entrada para quaisquer forma normal, e evita a introdução de símbolos novos, práticas que geralmente obscurecem a estrutura da fórmula original. Com isso, é possível trabalhar diretamente com a estrutura orginal da fórmula e facilitar o processo de conversão das provas. Além disso, ele mantém técnicas e características típicas de Lógica de Descrições, tais como notação sem variáveis, ausência de funções de Skolem e unificação e inclusão de uma regra de bloqueio para lidar com ciclos, que garante o término no caso de ontologias cíclicas, assim como no Cálculo Clausal de θ-Conexões $\mathcal{ALC}$.

- Elaboração de um Método de Conversão de Provas em Lógica de Descrições $\mathcal{ALC}$ geradas pelo Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$ para Sequentes.

  Conforme visto no Capítulo 5, o método desenvolvido utiliza noções de caminhos e conexões. Ele converte as provas geradas pelo Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$ para provas no Cálculo de Sequentes para $\mathcal{ALC}$ para Sequentes, obtendo assim, um formato de prova mais legível e inteligível. Isso certamente contribuirá para uma melhor descrição dos passos usados nas inferências, explicitando as razões para os resultados inferidos e fornecendo informações para tomadas de decisões.

- Desenvolvimento dos algoritmos para o método de conversão

Os algoritmos desenvolvidos e apresentados no Capítulo 6, possibilitam uma visão mais concreta do processo de conversão das provas aqui tratadas, mostrando a sequência de procedimentos necessários para a sua implementação. Ressalte-se ainda que a avaliação da complexidade computacional do algoritmo demonstra sua viabilidade prática, pois tem complexidade polinomial.

7.2 Trabalhos Futuros

O presente trabalho pode ser estendido em algumas direções, como:

- Uma implementação do Cálculo Não-clausal de θ-Conexões $\mathcal{ALC}$, produzindo com isso um raciocinador capaz de gerar provas não-clausais;
- A implementação do método de conversão, que automatize seu processo gerando provas em sequentes que, posteriormente, podem servir como entrada para uma descrição textual mais clara e legível dos passos de prova;
- Comparação do cálculo de sequentes para $\mathcal{ALC}$ apresentado na seção com outras abordagens em sequentes;
- Adaptação do método de conversão para outros cálculos de sequentes para $\mathcal{ALC}$;
- Comparação o método de conversão com outras abordagens;
- Uso prático com aplicações em áreas que empregam raciocínio em Lógica de Descrições e geram descrições sobre as inferências em linguagem natural para usuários leigos. Por exemplo, no grupo de pesquisa em que se insere este trabalho, o sistema LEGIS (RODRIGUES et al., 2015) permite consultas de situações para verificar se tais situações derivam crimes tipificados por leis, codificados em Lógica de Descrições. A conversão de provas pode ajudar aos usuários a entender porque uma determinada situação configura um crime, tornando a sua utilização viável na prática.

REFERÊNCIAS

BAADER, F.; CALVANESE, D.; MCGUINNESS, D. L.; NARDI, D.; PATEL-SCHNEIDER, P. F. *The Description Logic Handbook: Theory, Implementation, and Applications*. [S.l.]: Cambridge University Press, 2003. v. 32. 622 p. ISBN 0521781760.

BAADER, F.; HORROCKS, I.; SATTLER, U. *Description Logics*. [S.l.]: Elsevier, 2008. v. 3. 135–179 p.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The Semantic Web. *Scientific American*, Citeseer, v. 284, n. 5, p. 34–43, 2001. ISSN 00368733. Disponível em: <http://www.nature.com/doifinder/10.1038/scientificamerican0501-34>.

BIBEL, W. *Automated theorem proving*. [S.l.]: Vieweg Verlag, 1987.

BORGIDA, A.; FRANCONI, E.; HORROCKS, I. Explaining ALC Subsumption. *ECAI - European Conference on Artificial Intelligence*, 2000.

BORGIDA, A.; SERAFINI, L. Distributed description logics: Assimilating information from peer sources. *Journal on data semantics I*, p. 153—-184, 2003.

BUSS, S. R. *Handbook of proof theory*. [S.l.]: Elsevier, 1998.

CHANG, C. C.; KEISLER, H. J. *Model theory*. [S.l.]: Elsevier, 1990.

COTTA, C. *Knowledge-driven Computing: Knowledge Engineering and Intelligent Computations*. [S.l.]: Springer Science & Business Media, 2008.

DOETS, K. *Basic model theory*. [S.l.]: CLSI Publications Stanford, 1996.

ENDERTON, H.; ENDERTON, H. B. *A mathematical introduction to logic*. 2. ed. [S.l.]: Academic press, 2001.

FILHO, D. M.; FREITAS, F.; OTTEN, J. RACCOON: A Connection Reasoner for the Description Logic ALC. In: *LPAR-21: 21ST International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. [S.l.: s.n.], 2017.

FREITAS, F. A Connection Method for the Description Logic ALC. In: *24th International Workshop on Description Logics*. Barcelona, Espanha: [s.n.], 2011. p. 499.

FREITAS, F.; CANDEIAS JR, Z.; STUCKENSCHMIDT, H. Towards Checking Laws' Consistency through Ontology Design: The Case of Brazilian Vehicles' Laws . *Journal of Theoretical and Applied Electronic Commerce Research*, v. 6, n. 1, p. 112–126, 2011.

FREITAS, F.; OTTEN, J. A Connection Calculus for the Description Logic ALC. In: . [S.l.]: Canadian Artificial Intelligence Conference, 2016.

GENTZEN, G. Untersuchungen über das logische Schliessen. *Mathematische zeitschrift*, v. 39, n. 1, p. 176–210, 1934.

GHILARDI, S.; LUTZ, C.; WOLTER, F. Did I Damage my Ontology ? A Case for Conservative Extensions in Description Logic. In: *Kr*. [S.l.]: AAAI Press, 2006. p. 187–197. ISBN 9781577352716.

GIRARD, J.-Y.; LAFONT, Y.; TAYLOR, P. *Proofs and types*. 7. ed. [S.l.]: Cambridge University Press Cambridge, 1989.

HAMMACK, R. *Book of Proof*. [S.l.]: Virginia Commonwealth University, 2009.

HAN, S.; HUTTER, A.; STECHELE, W. A reasoning approach to enable abductive semantic explanation upon collected observations for forensic visual surveillance. *Multimedia and Expo (ICME), IEEE International Conference*, 2011.

HORROCKS, I. Ontologies and the Web Semantic. *Communications of the ACM*, v. 51, n. 12, p. 58–67, 2008.

KFOURY, A. J.; MOLL, R. N.; ARBIB, M. A. *A programming approach to computability*. [S.l.]: Springer Science \& Business Media, 2012.

MANFRED, S.-S.; SMOLKA, G. Attributive concept descriptions with complements. *Artificial intelligence*, v. 48, n. 1, p. 1–26, 1991.

MORTIMER, M. On languages with two variables. *Mathematical Logic Quarterly*, v. 21, n. 1, p. 135–140, 1975.

NOY, N. F.; CORONADO, S. de; SOLBRIG, H.; FRAGOSO, G.; HARTEL, F. W.; MUSEN, M. A. Representing the NCI Thesaurus in OWL DL: Modeling tools help modeling languages. *Applied ontology*, v. 3, n. 3, p. 173—-190, 2008.

OTTEN, J. Restricting backtracking in connection calculi. *Ai Communications*, v. 23, n. 2, p. 159–182, 2010.

OTTEN, J. A non-clausal connection calculus. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. [S.l.]: Springer, 2011. p. 226–241.

OTTEN, J.; KREITZ, C. A connection based proof method for intuitionistic logic. In: *Theorem Proving with Analytic Tableaux and Related Methods*. [S.l.]: Springer, 1995. p. 122–137.

RIBEIRO, M. M. *Belief revision in non-classical logics*. Campinas, SP: Springer Science & Business Media, 2012.

ROBINSON, A. J.; VORONKOV, A. *Handbook of automated reasoning*. 2. ed. [S.l.]: Elsevier, 2001.

RODRIGUES, C. M. d. O.; AZEVEDO, R. R. de; FREITAS, F. L. G. de; SILVA, E. P. da; da Silva Barros, P. V. An Ontological Approach for Simulating Legal Action in the Brazilian Penal Code. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. Salamanca, Spain: ACM, 2015. p. 376—-381.

ROSEN, K. H. *Handbook of Discrete and Combinatorial Mathematics*. [S.l.]: CRC press, 1999.

ROYER, V.; QUANTZ, J. Deriving inference rules for terminological logics. *Logics in AI*, p. 84—-105, 1992.

SATTLER, U.; CALVANESE, D.; MOLITOR, R. *"Relationships with other formalisms."The description logic handbook.* [S.l.]: Cambridge University Press, 2003. 142–183 p.

SCHMIDT, R.; TISHKOVSKY, D. Analysis of Blocking Mechanisms for Description Logics. In: . [S.l.]: Proceedings of the Workshop on Automated Reasoning, 2007.

TROELSTRA, A. S.; SCHWICHTENBERG, H. *Basic Proof Theory*. [S.l.]: Cambridge University Press, 2000.

VAN DALEN, D. *Logic and structure*. 3. ed. Berlin: Springer, 1994.

APÊNDICE A – EQUIVALÊNCIAS E MAPEAMENTOS LÓGICOS

- **Equivalências Lógicas**

Tabela 14 – Equivalências Lógicas.

Nome	**Equivalência**
Idempotência	$A \wedge A \Leftrightarrow A$
	$A \vee A \Leftrightarrow A$
Comutativa	$A \wedge B \Leftrightarrow B \wedge A$
	$A \vee B \Leftrightarrow B \vee A$
Associativa	$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$
	$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$
Distributiva	$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$
	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$
Complemento	$A \wedge (B \vee \neg B) \Leftrightarrow A$
	$A \vee (B \wedge \neg B) \Leftrightarrow A$
Absorção	$A \wedge (A \vee B) \Leftrightarrow A$
	$A \vee (A \wedge B) \Leftrightarrow A$
Leis de De Morgan	$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
	$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$
Lei da dupla negação	$\neg\neg A \Leftrightarrow A$
Implicação	$A \rightarrow B \Leftrightarrow \neg A \vee B$
	$\neg(A \rightarrow B) \Leftrightarrow A \wedge \neg B$
Equivalência	$A \leftrightarrow B \Leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$
	$A \leftrightarrow B \Leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B)$
Contraposição	$A \rightarrow B \Leftrightarrow \neg B \rightarrow \neg A$
Contradição	$A \wedge \neg A \Leftrightarrow \bot$
Tautologia	$A \vee \neg A \Leftrightarrow \top$

- **Mapeamento de Lógica de Descrições para Lógica de Primeira Ordem**

Tabela 15 – Lógica de Descrições e seu Mapeamento para Lógica de Primeira Ordem (BAADER; HORROCKS; SATTLER, 2008).

Mapeamento de Conceitos para Lógica de Primeira Ordem[1]					
$\pi_x(A)$	=	$A(x)$	$\pi_y(A)$	=	$A(y)$
$\pi_x(C \sqcap D)$	=	$\pi_x(C) \wedge \pi_x(D)$	$\pi_y(C \sqcap D)$	=	$\pi_y(C) \wedge \pi_y(D)$
$\pi_x(C \sqcup D)$	=	$\pi_x(C) \vee \pi_x(D)$	$\pi_y(C \sqcup D)$	=	$\pi_y(C) \vee \pi_y(D)$
$\pi_x(\exists r.C)$	=	$\exists y.r(x,y) \wedge \pi_y(C)$	$\pi_y(\exists r.C)$	=	$\exists x.r(y,x) \wedge \pi_x(C)$
$\pi_x(\forall r.C)$	=	$\forall y.r(x,y) \rightarrow \pi_y(C)$	$\pi_y(\forall r.C)$	=	$\forall x.r(y,x) \rightarrow \pi_x(C)$
Mapeamento de um TBox $\mathcal{T}$ e um ABox A[2]					
$\pi(\mathcal{T}) =$		$\bigwedge_{C \sqsubseteq D \in \mathcal{T}} \forall x.(\pi_x(C) \rightarrow \pi_x(D))$			
$\pi(A) =$		$\bigwedge_{a:C \in A} \pi_x(C)[x/a] \wedge \bigwedge_{(a,b):r \in A} r(a,b)$			

[1] π_x e π_y são funções de tradução, que indutivamente mapeiam conceitos $\mathcal{ALC}$ em fórmulas em lógica de primeira ordem com uma variável livre, x ou y.

[2] $\psi[x/a]$ denota a fórmula obtida a partir de ψ, substituindo todas as ocorrências livres de x por a.

APÊNDICE B – PROVAS DE CORRETUDE, COMPLETUDE E TERMINAÇÃO

- **Corretude, Completude e Terminação**

As seções seguintes mostram que o Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$ termina, é correto e completo. Doravante, algumas abreviações são usadas para os nomes dos cálculos da seguinte forma:

- Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$ é representado por θ-NC$\mathcal{ALC}$,
- Cálculo de θ-Conexões $\mathcal{ALC}$ por θ-$\mathcal{ALC}$ e
- Cálculo Não-Clausal de Conexões para Lógica de Primeira ordem por CC.

- **Terminação**

Definição 68. ***(Cláusula expandida).*** *Seja E uma cláusula com um símbolo introduzido S.$Xc(E)$, a cláusula expandida de E é a cláusula resultante da união de E', que é E sem S, mais L, onde L é o literal da cláusula que contém $\overline{S}$, ou seja, $Xc = E \setminus S \cup \{L\}, \{L, \overline{S}\} \in M$.*

Definição 69. ***(Ontologia ou consulta expandida).*** *Seja O uma ontologia ou consulta com alguma cláusula E com um símbolo introduzido. $Xo(O)$, a ontologia expandida de O, é O na qual cada E é substituída por $Xc(E)$ e sem $\{L, \overline{S}\}$, ou seja, $Xo(O) = \forall E \mid O \setminus \{E, \{L, \overline{S}\}\} \cup Xc(E)$.*

Lema 2. ***(Cláusulas correspondentes).*** *Qualquer cláusula no Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$ resultante de uma consulta Q existe em $Xo(Q)$.*

Demonstração. Para o Cálculo de θ-Conexões $\mathcal{ALC}$, fórmulas são convertidas para forma normal com duas linhas (FREITAS; OTTEN, 2016), possivelmente com novos símbolos introduzidos. Tem-se os seguintes casos:

1. Sem símbolos introduzidos no θ-$\mathcal{ALC}$: para cada cláusula em θ-$\mathcal{ALC}$ existe uma cláusula em θ-NC$\mathcal{ALC}$. Logo, para cada conexão em θ-$\mathcal{ALC}$ existe uma conexão em θ-NC$\mathcal{ALC}$,
2. Com símbolos introduzidos no θ-$\mathcal{ALC}$: toda cláusula ou cláusula expandida em θ-$\mathcal{ALC}$, existe também no θ-NC$\mathcal{ALC}$. Assim, toda conexão sem símbolos introduzidos no primeiro sistema de inferência ocorre também no segundo. A consulta expandida de θ-$\mathcal{ALC}$ é composta por fórmulas na FND existentes na consulta θ-NC$\mathcal{ALC}$ correspondente.

□

Lema 3. ***(Provas Correspondentes).*** *Uma prova em Cálculo de θ-Conexões $\mathcal{ALC}$, sem as conexões sobre os símbolos introduzidos, é uma prova em Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$.*

Demonstração. Uma prova é um conjunto de conexões com θ-substituições adequadas em ambos os sistemas. A existência das mesmas conexões, exceto aquelas com símbolos introduzidos, segue diretamente do lema anterior. □

Definição 70. ***(Caminho vertical através da cláusula).*** *Seja X uma matriz, cláusula, ou literal. Um caminho vertical p através de X, denotado por $p \parallel X$, é um conjunto de literais de X e indutivamente definido como: $\{L\} \parallel L$ para o literal L; $p \parallel M$ para a matriz M onde $p \parallel C$ para alguma cláusula $C \in M$; $p \parallel C$ para uma cláusula $C \neq \{\}$ onde $p = \bigcup_{M_i \in C} p_i$ e $p_i \parallel M_i$; $\{\} \parallel C$ para a cláusula $C = \{\}$.*

Lema 4. ***(Cláusulas e Caminhos Verticais).*** *Seja M a matriz não-clausal de uma fórmula $\mathcal{ALC}$ F e M' a matriz de F na forma clausal (ver Exemplo 36). Então para cada cláusula $C' \in M'$, sem ou com símbolos introduzidos, existe uma cláusula 'original' $C \in M$ com $C' \parallel C$. Se existe uma prova em conexões para C', M, P, então existe também uma prova em conexões para C, M, P. Isto também é verdade se as cláusulas $D' := C' \setminus \{L\}$ e $D := \beta\text{-cláusula}_L(C)$ são usados, para algum literal L, ao invés de C' e C, respectivamente.*

Demonstração. A existência de uma cláusula $C \in M$ para cada $C' \in M'$ com $C' \parallel C$ segue da definição 51 e definição 70. C, M, P pode ser derivado de C', M, P por vários passos de decomposição. Em D o literal L e todas as cláusulas que são α-relacionadas a L, isto é, que não contém literais de D', são deletados de C. Portanto, $D' \parallel D$ e D, M, P podem ser derivado de D', M, P também. □

Teorema 4. ***(Terminação).*** *Dada a matriz não-clausal M que representa a consulta arbitrária $O \vdash_{\theta-NC\mathcal{ALC}} \alpha$, e uma cláusula inicial C, qualquer sequência de regras no θ-NC$\mathcal{ALC}$ aplicada sobre a tupla "$\varepsilon, M, \varepsilon$" termina.*

Demonstração. Todas as conexões de uma prova θ-$\mathcal{ALC}$ da consulta Q existem em θ-NC$\mathcal{ALC}$, exceto aquelas com símbolos introduzidos. Consultas na forma normal com duas linhas (para θ-$\mathcal{ALC}$) são extensões conservativas (GHILARDI; LUTZ; WOLTER, 2006) de suas consultas originais (FREITAS; OTTEN, 2016). Portanto, cada passo de prova de θ-NC$\mathcal{ALC}$ corresponde a um passo em θ-$\mathcal{ALC}$ em consultas acíclicas ou cíclicas, exceto quando a regra Decomposição é aplicada. A regra Decomposição é aplicada uma vez em cada cláusula (possivelmente contendo matrizes). Segue-se que se θ-$\mathcal{ALC}$ termina, então θ-NC$\mathcal{ALC}$ também termina. □

- **Corretude**

Definição 71. ***(Superconjunto através da cláusula).*** *Seja p um conjunto de literais. p é um superconjunto através de C, denotado por $C \sqsubseteq p$, se e somente se existe um caminho p' através de $\{C\}$ com $p' \subseteq p$.*

Lema 5. ***(Corretude das regras que não são a regra Início).*** *Se existe uma prova de conexões para $\{C, M, P\}$ com o termo substituição θ, então existe uma multiplicidade μ tal que cada caminho p através de M^μ com $P \subseteq p$ e $C \sqsubseteq p$ contém uma conexão θ-complementar.*

1. A prova é bastante semelhante ao análogo Lema 1 do Cálculo Não-Clausal de Conexões Clássico (OTTEN, 2011), pois as regras são quase iguais, com as seguintes diferenças:

   a) No Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$, a θ-substituição e a aplicação nas regras da condição de Skolem substituem a unificação. A validade é preservada, uma vez que θ-substituição e unificação são equivalentes (ver Lema 1, Table 2);

   b) A introdução da regra Cópia, com a condição de bloqueio.

A prova também é por indução estrutural para as provas em conexões (durante a sua construção). *Hipótese de Indução (HI)*: Se *Prova* é uma prova em conexões para $\{C, M, P\}$, então existe um μ tal que cada caminho p através de M^μ com $P \subseteq p$ e $C \sqsubseteq p$ contém uma conexão θ-complementar.

As provas para as regras Redução, Extensão, Decomposição e Axioma são completamente análogas ao Lema 1 em (OTTEN, 2011), desde que as referências aos unificadores arbitrários σ e σ' sejam substituídas por referências a θ-substituições θ e θ' análogas, e referências a conexões σ-complementar sejam lidas como conexões θ-complementar. Além disso, na regra Extensão $M[C_1 \setminus C_2]$ deve ser substituída por M. O último caso diz respeito a regra Cópia, descrita no final.

1. *Axioma*: Seja $\frac{}{\{\},M,P}$ uma prova em θ-conexões $\mathcal{ALC}$. Seja $\mu \equiv 1$ e $\theta(x) = x$ para todo x. Então $\{\} \sqsubseteq p$ é válido para nenhum caminho p através de M^μ. Assim, a *HI* é verdadeira.

2. *Redução*: Seja $\frac{Prova}{C,M,P\cup\{L_2\}}$ uma prova em θ-conexões $\mathcal{ALC}$ para $C, M, P \cup \{L_2\}$ para algum θ. De acordo com a *HI* existe um μ tal que todo p através de M^μ com $P \cup \{L_2\} \subseteq p$ e $C \sqsubseteq p$ contém uma conexão θ-complementar. Então a derivação $\frac{\frac{Prova}{C,M,P\cup\{L_2\}}}{C\cup\{L_1\},\ M,\ P\cup\{L_2\}}$ R com $\tau(\theta(L_1)) = \tau(\theta(\overline{L_2}))$ para algum termo de substituição τ é uma prova em conexões para $C \cup \{L_1\},\ M,\ P \cup \{L_2\}$. Seja $\mu' := \mu$ e $\theta' := \tau \circ \theta$. Todo caminho p' através de $M^{\mu'}$ com $P \cup \{L_2\} \subseteq p'$ e $C \cup \{L_1\} \sqsubseteq p'$ contém uma conexão θ'-complementar também, desde que $\theta'(L_1) = \theta'(\overline{L_2})$.

3. *Extensão*: Seja $\frac{Prova_1}{C_3,\ M,\ P\cup\{L_1\}}$ e $\frac{Prova_2}{C,\ M,\ P}$ provas em θ-conexões $\mathcal{ALC}$ para $C_3,\ M[C_1 \setminus C_2], P \cup \{L_1\}$ e $C,\ M,\ P$, respectivamente, para algum θ, com $C_3 := \beta\text{-cláusula}_{L_2}(C_2)$, C_2 é uma cópia de C_1, C_1 é uma cláusula de extensão de M com respeito a. $P \cup \{L_1\}$, e C_2 contém o literal L_2 com $\tau(\theta(L_1)) = \tau(\theta(\overline{L_2}))$ para alguma substituição τ. De acordo com a *HI*

existe um μ_1 tal que todo caminho p através $(M[C_1 \setminus C_2])^{\mu_1}$ com $P \cup \{L_1\} \subseteq p$ e $C_3 \sqsubseteq p$ contém uma conexão θ-complementar, e existe um μ_2 tal que todo p através de M^{μ_2} com $P \subseteq p$ e $C \sqsubseteq p$ contém uma conexão θ-complementar. Então $\frac{\frac{Prova_1}{C_3,\, M,\, P\cup\{L_1\}} \quad \frac{Prova_2}{C,\, M,\, P}}{C\cup\{L_1\},M,P}$ E é uma prova em conexões $C \cup \{L_1\}, M, P$. Este último passo de extensão é ilustrado abaixo. Deve ser demonstrado que existe uma multiplicidade μ' e uma substituição θ' tal que todo caminho p' através de $M^{\mu'}$ com $P \subseteq p'$ e $C \cup \{L_1\} \sqsubseteq p'$ contém uma conexão θ'-complementar. Seja M' a matriz M na qual a (sub-)matriz $\{\ldots, C_1, \ldots\}$ que contém C_1 é substituída pela matriz $\{\ldots, C_1, C_2, \ldots\}$, isto é, a cláusula C_2 é adicionada a M como mostrado abaixo.

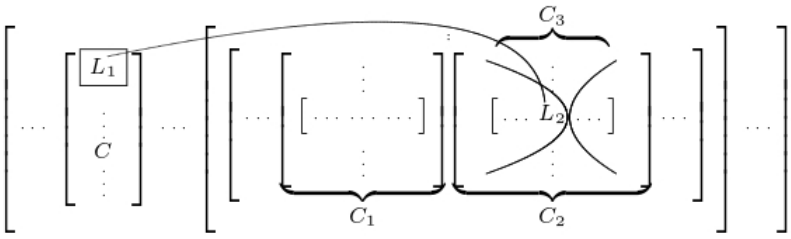

Figura 37 – Passo de Extensão (OTTEN, 2011).

De acordo com a definição 56, os seguintes casos para a cláusula de extensão C_1 precisam ser considerados:

a) Se a cláusula de extensão C_1 contém um literal do P, então todo caminho p' através de M' com $P \cup \{L_1\} \subseteq p'$ é também um superconjunto através da cláusula C_2, isto é. $C_2 \sqsubseteq p'$.

b) Por outro lado, C_1 é α-relacionado a todos os literais em $P \cup \{L_1\}$ ocorrendo em M'.

   - Se C_1 não tem cláusula mãe, então para todo p' através de M' $C_1 \sqsubseteq p'$ é válida. Então para todo p' através de M' com $P \cup \{L_1\} \subseteq p'$ é $C_1 \sqsubseteq p'$ e, portanto $C_2 \sqsubseteq p'$.
   - Por outro lado, a cláusula mãe de C_1 contém um elemento de $P \cup \{L_1\}$. Como C_1 é α-relacionado a $P \cup \{L_1\}$, existe um $\widehat{C}$ contendo um literal de $P \cup \{L_1\}$ com $C_1, \widehat{C} \in \widehat{M}$. Portanto, para todo p' através de M' com $P \cup \{L_1\} \subseteq p'$ é $C_1 \sqsubseteq p'$ e, portanto $C_2 \sqsubseteq p'$.

C_3 é a β-cláusula de C_2 com respeito a L_2, isto é, L_2 e todas as cláusulas que são α-relacionadas a L_2 são excluídas de C_2. De acordo com a definição 57 o único elemento excluído de uma *cláusula* é o literal L_2. Portanto, para todo p' através de M' com $P \cup \{L_1\} \subseteq p'$ é $C_3 \sqsubseteq p'$ ou $\{L_2\} \sqsubseteq p'$. O mesmo é válido se cópias de cláusulas são adicionadas a M'. Seja μ' a multiplicidade onde todas as cópias, de acordo com μ_1 e μ_2, bem como a cópia de C_2 são consideradas. Seja $\theta' := \tau \circ \theta$. Então $C_3 \sqsubseteq p'$ ou $L_2 \in p'$ vale para todo p' através de $M^{\mu'}$ com $P \cup \{L_1\} \subseteq p'$. Como existe uma prova para C_3, $M[C_1 \setminus C_2]$, $P \cup \{L_1\}$ com θ, todo p' através de $(M[C_1 \setminus C_2])^{\mu_1}$, e portanto através de $M^{\mu'}$, com

$P \cup \{L_1\} \subseteq p$ e $C_3 \sqsubseteq p$ contém uma conexão θ-complementar, e portanto uma conexão θ'-complementar. Mais adiante, todo p' com $P \cup \{L_1\} \subseteq p'$ que inclui L_2 contém uma conexão θ'-complementar como $\theta'(L_1) = \theta'(\overline{L_2})$. Portanto, todo p' através de $M^{\mu'}$ com $P \cup \{L_1\} \subseteq p'$ contém uma conexão θ'-complementar. Então, todo p' através de $M^{\mu'}$, com $P \subseteq p'$ e $\{L_1\} \sqsubseteq p'$ contém uma conexão θ'-complementar. Como existe uma prova para $C,\ M,\ P$ com θ, todo p através de M^{μ_2}, e portanto através de $M^{\mu'}$, com $P \subseteq p$ e $C \sqsubseteq p$ contém uma conexão θ-complementar, e portanto uma conexão θ'-complementar. Assim, todo p' através de $M^{\mu'}$, com $P \subseteq p'$ e $C \cup \{L_1\} \sqsubseteq p'$ contém uma conexão θ'-complementar.

4. *Decomposição*: Seja $\frac{Prova}{C \cup C_1, M, P}$ uma prova em θ-conexões $\mathcal{ALC}$ para $C \cup C_1, M, P$ para algum θ. De acordo com a *HI* existe um μ talque todo p através de M^{μ} com $P \subseteq p$ e $C \cup C_1 \sqsubseteq p$ contém uma conexão θ-complementar. Então $\frac{\frac{Prova}{C \cup C_1, M, P}}{C \cup \{M_1\}, M, P}$ D com $C_1 \in M_1$ é uma prova de conexões para $C \cup \{M_1\}, M, P$. Seja $\mu' := \mu$ e $\theta' := \theta$. Todo p' através de $M^{\mu'}$ com $P \subseteq p'$ e $C \cup \{M_1\} \sqsubseteq p'$ contém também uma conexão θ'-complementar, logo $C_1 \in M_1$ e, portanto, para todo p' o seguinte é verdade: se $C \cup \{M_1\} \sqsubseteq p'$ então $C \cup C_1 \sqsubseteq p'$.

1. *Cópia:* Seja $\frac{Prova}{C \cup \{L_1\}, M \cup C_2^{\mu}, P}$ uma prova em θ-conexões $\mathcal{ALC}$ para $C \cup \{L_1\}, M \cup C_2^{\mu}, P$. De acordo com *(HI)*, existe uma multiplicidade μ tal que cada caminho p através de M^{μ} com $P \subseteq p$ e $C \cup \{L_1\} \sqsubseteq p$ contém uma conexão θ-complementar. Então $\frac{\frac{Prova}{C \cup \{L_1\}, M \cup C_2^{\mu}, P}}{C \cup \{L_1\},\ M,\ P}$ Cop onde C_2^{μ} é uma cópia de C_1, é uma prova em conexões para $C \cup \{L_1\}, M, P$. Seja $\mu' = \mu, \theta' = \theta$. Todo p' através $M^{\mu'}$ com $P \subseteq p', C \cup \{L_1\} \sqsubseteq p'$ contém também uma conexão θ-complementar, desde que $L_2 \in C_2^{\mu}$ e $\theta'(L_1) = \theta'(\overline{L_2})$.

Teorema 5. ***(Corretude do Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$).*** *Se existe uma prova no Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$ para uma consulta $O \vdash_{\theta - NC\mathcal{ALC}} \alpha$ em $\mathcal{ALC}$, então a consulta é válida em $\mathcal{ALC}$ ($O \models \alpha$).*

Demonstração. Seja M a matriz de uma fórmula F. Se existe uma prova no Cálculo Não-Clausal de θ-Conexões para $\varepsilon, M, \varepsilon$ ela tem a forma $\frac{\vdots}{\frac{C_1, M, \{\}}{\varepsilon, M, \varepsilon}}$ In com $C_1 \in M$. Deve haver uma prova para $C_1, M, \{\}$ para algum θ. De acordo com o Lema 5 existe uma μ, tal que todo caminho p através de M^{μ} com $\{\} \subseteq p$ e $C_1 \sqsubseteq p$ contém uma conexão θ-complementar. Logo, se existe um termo de substituição θ e um conjunto de conexões S, tal que todo caminho através de M contém uma conexão que é θ-complementar, a fórmula F é válida. □

- **Completude**

Teorema 6. ***(Completude do Cálculo Não-Clausal de θ-Conexões $\mathcal{ALC}$).*** *Seja M a matriz de uma fórmula $\mathcal{ALC}$ F e M' a matriz de F na forma clausal. Se existe uma prova não-clausal de conexões para a consulta F, existe uma derivação para F no cálculo não-clausal de θ-conexões $\mathcal{ALC}$.*

Demonstração. Se CC é completo, então $O \models \alpha$ implica $O \vdash_{CC} \alpha$. E se θ-NC$\mathcal{ALC}$ é completo, então $O \models \alpha$ implica em $O \vdash_{\theta-NC\mathcal{ALC}} \alpha$. Como $\mathcal{ALC}$ é um fragmento de Lógica de Primeira Ordem decidível (BAADER et al., 2003), e θ-NC$\mathcal{ALC}$ e CC são precedimentos de decisão para $\mathcal{ALC}$, é suficiente provar que $O \vdash_{CC} \alpha$ implica $O \vdash_{\theta-NC\mathcal{ALC}} \alpha$. Para casos cíclicos, o bloqueio deve ser aplicado para garantir a terminação, então a prova é contrapositiva: $O \nvdash_{\theta-NC\mathcal{ALC}} \alpha$ deve implicar $O \nvdash_{CC} \alpha$.

A prova contrapositiva é por indução estrutural na estrutura da sequência finita dos nomes dos indivíduos $x_1, \ldots, x_{i-1}$ que gera o próximo indivíduo do ciclo x_i.

Hipótese de Indução: $O \nvdash_{\theta-NC\mathcal{ALC}} \alpha$

Caso base: O caso base aqui é o ciclo mínimo, onde $O = \{E \sqsubseteq \exists r.E\}$ ou $O = \{\forall r.E \sqsubseteq E\}$. Então considere $O = \{E \sqsubseteq \exists r.E\}$, $O \nvdash_{\theta-NC\mathcal{ALC}} \alpha$ e α uma fórmula arbitrária, ou seja, $\alpha = \{\neg E(a)\}$. As figuras 38 e 39 mostram as tentativas para concluir a prova em conexões em θ-NC$\mathcal{ALC}$ e CC, respectivamente. A figura 38 retrata que após a conexão θ-complementar $\{E, \neg E(a)\}$, a regra Cópia é aplicada para pela primeira vez e o par $\{\underline{\neg E}^1, E\}$ é conectado, onde $\underline{\neg E(b)}^1$ é construído com uma θ-substituição. Em seguida, Cópia é aplicada mais uma vez e a cláusula $\{E, \{\{\underline{\neg r}\}\{\underline{\neg E}\}\}\}$ aparece pela terceira vez em M. Então, a conexão $\{\underline{\neg E(b)}^2, E(b)\}$ é resolvida, e b é reusada ao invés de criar um novo indivíduo. Assim, a conexão $\{\underline{\neg E(b)}, E(b)\}$ reaparece, o processo é bloqueado e $E \sqsubseteq \exists r.E \nvdash_{\theta-NC\mathcal{ALC}} \neg E(a)$.

Em relação ao CC, o caso é mostrado na figura 39. Após a conexão $\{E(y), \neg E(a)\}$, com $\theta = \{y/a\}$, devido a falta de um literal complementar para $\neg E(f(y))$, CC copia a primeira cláusula aumentando o μ da cláusula. Isso ocorre indefinidamente. Então CC entra em *loop* e falha, e, portanto, $E \sqsubseteq \exists r.E \nvdash_{CC} \neg E(a)$. Então, para o caso base com $\alpha \in O$, $O \nvdash_{\theta-NC\mathcal{ALC}} \alpha$ implica $O \nvdash_{CC} \alpha$.

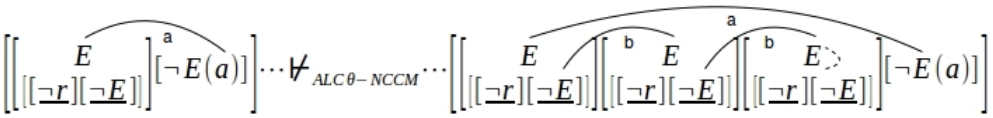

Figura 38 – Tentativa de concluir as provas em conexões no θ-NC$\mathcal{ALC}$.

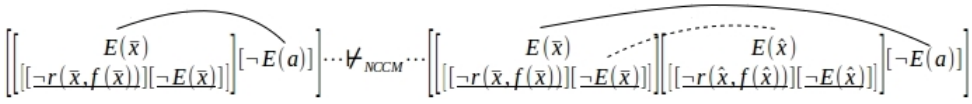

Figura 39 – Tentativa de concluir as provas em conexões no CC.

Caso Indutivo: Considere novamente $O = \{E \sqsubseteq \exists r.E\}$, $O \nvdash_{\theta-NC\mathcal{ALC}} \alpha$ e α uma fórmula arbitrária, isto é, $\alpha = \{\neg E(a)\}$. Seja x^{μ}_{n-1} um indivíduo, com $n \geq 1$. Em cada ciclo μ existe uma sequência: $\neg r_0, \ldots, \neg r_n (n \geq 1)$, onde uma sequência de indivíduos é criada com relações em cada ciclo. Portanto, para o primeiro ciclo (ciclo 0) em M a sequência $\neg r_1(x^0_0, x^0_1), \ldots, \neg r_n(x^0_{n-1}, x^0_n)$ é gerada. No segundo ciclo (ciclo 1) existe uma nova sequência: $\neg r_1(x^1_0, x^1_1), \ldots, \neg r_n(x^1_{n-1},$

x_n^1), onde $(x_0^0 = x_0^1, \ldots, x_n^0 = x_n^1)$. A condição de bloqueio identifica tais repetições e o bloqueio ocorre, dado que $\tau(x_n^1) \subseteq \tau(x_n^0)$ e θ-NC$\mathcal{ALC}$ pára. CC, por sua vez, contém a cláusula $[E(y), [[\neg r(y, f(y))][\neg E(f(y))]]]$, e mesmo depois de buscar longas seqüências finitas de indivíduos para cada cópia desta cláusula, ou seja, $x_1, \ldots, x_{i-1}$, CC entra em *loop* e falha, como no caso base. Portanto, para o caso indutivo onde $\alpha \in O$, $O \nvdash_{\theta-NC\mathcal{ALC}} \alpha$ implica $O \nvdash_{CC} \alpha$. □

APÊNDICE C – ALGORITMOS - FUNÇÕES

- **Funções dos Algoritmos do Método de Conversão**

Algoritmo 11: SIMULA A EXCLUSÃO DOS PARÊNTESES DA FÓRMULA INFIXA E ATUALIZA POSIÇÕES DOS ELEMENTOS NA VARIÁVEL GLOBAL POS

```
1  Função atualizaPosicao(F[])
2      int idx = 0;
3      int j;
4      para cada elemento m de F[] faça
5          se m ≠ '(' e m ≠ ')' então
6              idx:= idx + 1;
7              j = 0;
8              enquanto j < tamanho de pos[] faça
9                  j := j + 1;
10                 se pos[j] = posição atual de F[] então
11                     pos[j] := idx - 1;
12                     j := tamanho de pos[];
```

Complexidade do Algortimo 11: O(n), pois o algoritmo processa *n* entradas. As operações dentro do loop aumentam a complexidade em um fator constante, não alterando a complexidade assintótica.

Algoritmo 12: CHECA SE UMA CONEXÃO CONSTA NO CONJUNTO DE CONEXÕES.

```
1  Função constaConexao(C[], ordem)
2      se C[] = ∅ então
3          retorna false;
4      senão
5          para cada elemento i em C[] faça
6              se C[i].ordem = ordem então
7                  retorna true;
8              senão
9                  retorna false;
```

Complexidade do Algortimo 12: O(c), onde *c* representa o tamanho de *C*[], ou seja, o número de conexões contidas no array. O algoritmo processa *c* entradas. As operações dentro do

loop aumentam a complexidade em um fator constante, não alterando a complexidade assintótica.

Algoritmo 13: BUSCA A POSIÇÃO DO ELEMENTO NA MATRIZ DADO SEU IDENTIFICADOR

```
1  Função buscaPosicao(matriz[],idelemento)
2      achou := false;
3      int poselemento; i := 0;
4      enquanto (achou = false) ou (i <= tamanho de matriz[]) faça
5          se matriz[i] não é do tipo Elemento então
6              buscaPosicao(matriz[i], idelemento);
7          senão
8              se matriz[i].id = idelemento então
9                  poselemento := matriz[i].posicao;
10                 achou := true;
11             i := i + 1;
12     retorna poselemento;
```

Complexidade do Algortimo 13: O(m^2). A cada iteração, no pior caso, reduz-se a fórmula à metade.

Algoritmo 14: ORDENA CONEXÕES EM ORDEM CRESCENTE PELO CAMPO ORDEM

```
1  Função ordenaConexao(C[])
2      int n := tamanho de C[];
3      int min; ConexaoNo temp[];
4      para i = 0; i < n - 1; i++) faça
5          min := i;
6          para (j = i + 1; j < n; j++) faça
7              se C[j].ordem < C[min].ordem então
8                  min := j;
9          temp := C[i];
10         C[i] := C[min];
11         C[min] := temp;
12     retorna C;
```

Complexidade do Algortimo 14: O(c^2), pois possui duas iterações aninhadas, com c entradas. c representa o tamanho de C[], ou seja, o número de conexões contidas no array.

Algoritmo 15: BUSCA O CAMINHO ENTRE O NÓ RAIZ E UM DADO NÓ

```
1  Função buscaCaminho(no, pos, caminho)
       /* Busca o caminho entre o nó raiz e um nó na árvore sintática    */
       /* O noNulo com posição igual a -1, indica a árvore vazia         */
2      No noNulo;
3      noNulo.pos = -1;
4      se no = null então
5          empilhe noNulo no topo do caminho;
6          retorna caminho;
7      senão
8          empilhe no.pos no caminho;
9          se no.pos = pos então
10             retorna caminho;
11         senão
12             caminho := buscaCaminho(no.direita, pos, caminho);
13             se elemento c do topo do caminho tem c.pos = −1 então
14                 desempilhe c do caminho;
15                 caminho := buscaCaminho(no.esquerda, pos, caminho);
16                 se elemento c do topo do caminho tem c.pos = −1 então
17                     desempilhe os dois primeiros elementos do topo do caminho;
18                     empilhe noNulo no topo do caminho;
19                     retorna caminho;
20                 senão
21                     retorna caminho;
22             senão
23                 retorna caminho;
```

Complexidade do algoritmo 15: O(n). Dado que, no pior caso, o algoritmo deve percorrer toda a árvore duas vezes, indo e voltando com o backtracking.

Algoritmo 16: SUBSTITUI POSIÇÕES σ_δ / $\sigma_{\beta'}$

```
1 Função substituiPosicao(no1, no2, σ)
2     par[0] := no1.posicao;
3     par[1] := no2.posicao;
4     se σ ≠ ∅ então
5         para cada elemento σ' em σ faça
6             se σ'[0] = no2.posicao então
7                 par[1] := σ'[1];
8     σ := σ ∪ par;
9     retorna σ;
```

Complexidade do algoritmo 16: O(n), pois a entrada para esse algoritmo é $n/2$ e só tem um loop de tamanho proporcional a entrada.

Algoritmo 17: CHECA SE UM NÓ CONSTA EM UMA LISTA DE NÓS.

```
1 Função constaNo(no, lista)
      /* Checa se um nó consta em uma lista de nós, como por exemplo na
         lista de nós que precisam ser reduzidos P                    */
2     para cada elemento i em lista faça
3         se lista[i] = no então
4             retorna true;
5         senão
6             retorna false;
```

Complexidade do algoritmo 17: O(n), pois tem no máximo tamanho proporcional a n iterações.

Complexidade ao algoritmo 18: O(n), pois tem no máximo tamanho proporcional a n iterações.

Complexidade ao algoritmo 19: O(n), pois tem no máximo tamanho proporcional a n iterações.

Complexidade do algoritmo 20: O(n^2), pois tem duas iterações aninhadas com entrada de tamanho proporcional a n.

Algoritmo 18: BUSCA NÓS DE UM DADO TIPO EM UMA LISTA DE NÓS.

```
1 Função buscaNosTipo(tipo, lista)
      /* Procura nós em um array de acordo com um tipo, se encontrar
         informa quais são.                                          */
2     j:= 0;
3     v[];
4     para cada elemento i em lista faça
5         se lista[i].tipo = tipo então
6             v[j]:= lista[i];
7             j:= j + 1;
8     retorna v;
```

Algoritmo 19: REMOVE NÓ DE UMA LISTA.

```
1 Função removeNo(no, lista)
      /* Remove nó de um array.                                      */
2     j:= 0;
3     novaLista[];
4     para cada elemento i em lista faça
5         se lista[i] ≠ no então
6             novaLista[j]:= lista[i];
7             j:= j + 1;
8     retorna novaLista;
```

Algoritmo 20: CHECA SE $\triangleleft$ É REFLEXIVA

```
1 Função checaReflexividade(◁)
2     resp := 0;
3     tamanho := tamanho de ◁;
4     para (i := 0; i < tamanho; i++) faça
5         para (j: = i + 1; j < tamanho; j++) faça
6             se (◁[j] = ◁[i]) então
7                 resp := 1; /* há um valor repetido, ◁ é reflexiva  */
8     retorna resp;
```

Complexidade do algoritmo 23: O(n), pois tem no máximo n iterações. As funções chamadas nas linhas 5, 6, 7 e 8, estão fora do escopo da iteração, e todas possuem complexidade O(n). Assim: $O(n) + O(n) + O(n) + O(n) + O(n) = O(n)$.

Algoritmo 21: SUBSTITUI POSIÇÕES σ_{Final}

```
1  Função substituiPosicaoFinal(cn1, cn2, σFinal)
2      no1:= elemento do topo de cn1;
3      no2:= elemento do topo de cn2;
4      desempilhe o elemento do topo de cn1;
5      desempilhe o elemento do topo de cn2;
6      paiNo1:= elemento do topo de cn1;
7      paiNo2:= elemento do topo de cn2;
       /* tamanho de no1.posSubst[] é no máximo 2, para nós que são relações
          */
8      tam := tamanho de no1.posSubst[];
9      para (i = 0, i < tam, i++) faça
10         par[0] := no1.posSubst[i];
11         par[1] := no2.posSubst[i];
           /* se posições ou instâncias associadas ao rótulo Não são
              idênticas, busca se essas posições já foram substituídas; se
              são idênticas, não faz nada, nem mesmo insere em σFinal      */
12         se (par[0] ≠ par[1]) então
13             se σFinal ≠ ∅ então
14                 para cada elemento σ'Final em σFinal faça
15                     se σ'Final[0] = no1.posSubst[i] então
16                         par[0] := σ'Final[1];
17                     senão se σ'Final[0] = no2.posSubst[i] então
18                         par[1] := σ'Final[1];
19         se (par[0] ≠ par[1]) então
               /* faz a substituição da posição por uma instância se um dos
                  nós é filho de nó do tipo α ou α', caso contrário retorna
                  null, indicando conexão não complementar                  */
20             se paiNo1.tipo = α ou α' então
21                 par[0] := par[1];
22                 par[1] := no1.posSubst[i];
23                 σFinal := σFinal ∪ par;
24             senão se paiNo2.tipo = α ou α' então
25                 σFinal := σFinal ∪ par;
26             senão
27                 retorna null;
28     retorna σFinal;
```

Complexidade ao algoritmo 21: O(n^2), pois tem duas iterações aninhadas com entrada de tamanho prop

Algoritmo 22: APLICA REGRA L⊓ OU R⊔ SOBRE F.

```
1  Função aplicaRegraL⊓R⊔(noArv, noEst)
2      achou := false;
3      i := 0;
4      j := 0
5      idx := buscaIndice(F,noEst.posicao);
6      par := posicaoParenteses(F,idx);
7      enquanto i < tam faça
8          se (i = par[0]) ou (i = par[1]) então
9              i := i + 1;
10         senão se F.No[i].posicao = noEst.posicao então
11             auxF[j].rotulo := ',';
12             auxF[j].posicao := noEst.posicao;
13             j := j + 1;
14             i := i + 1;
15         senão
16             auxF[j] := F.No[i];
17             j := j + 1;
18             i := i + 1;
19     retorna F;
```

Complexidade do algoritmo 22: O(n), pois tem no máximo n iterações na linha 7.
As funções chamadas nas linhas 5 e 6, estão fora do escopo da iteração, e ambas possuem complexidade O(n).
Assim: $O(n) + O(n) + O(n) = O(n)$.

Complexidade do algoritmo 24: O(n), pois tem no máximo n iterações.
As funções chamadas nas linhas 4, e 5, estão fora do escopo da iteração, e todas possuem complexidade O(n). Assim: $O(n) + O(n) + O(n) = O(n)$.

Algoritmo 23: APLICA REGRA R¬⊓ OU L¬⊔ SOBRE F.

```
1  Função aplicaRegraR¬⊓L¬⊔(F, noEst, noArv)
2      achou := false;
3      i := 0; j := 0;
4      idx := buscaIndice(F,noEst.posicao);
5      cn1[] = buscaCaminho(noArv, noEst.posicao, ∅);
6      noNeg := buscaNoRotulo(¬, cn1[]);
7      par := posicaoParenteses(F,idx);
8      enquanto i < tam faça
9          se (i = par[0]) ou (i = par[1]) então
10             i := i + 1;
11         senão se F.No[i].posicao = noEst.posicao então
12             auxF[j].rotulo := ',';
13             auxF[j].posicao := noEst.posicao;
14             j := j + 1;
15             auxF[j].rotulo := noNeg.rotulo;
16             auxF[j].posicao := noNeg.posicao;
17             j := j + 1;
18             i := i + 1;
19         senão
20             auxF[j] := F.No[i];
21             j := j + 1;
22             i := i + 1;
23     F.No[]:= auxF;
24     retorna F;
```

Algoritmo 24: APLICA REGRA L¬¬ OU R¬¬ SOBRE F.

```
1  Função aplicaRegraL¬¬R¬¬(F, noEst, noArv)
2      achou := false;
3      j := 0;
4      cn1[] = buscaCaminho(noArv, noEst.posicao, ∅);
5      noNeg := buscaNoRotulo(¬, cn1[]);
6      para (i:=0; i<tam; i++) faça
7          se (F.No[i].posicao = noNeg.posicao) e (F.No[i].posicao = noEst.posicao)
            então
8              auxF[j] := F.No[i];
9              j := j + 1;
10     F.No[]:= auxF;
11     retorna F;
```

Algoritmo 25: APLICA REGRA (L⊔ OU R⊓) OU (L¬⊓ OU R¬⊔) SOBRE F.

```
1  Função aplicaRegraL⊔R⊓(F, noEst, lado)
2      par[];
3      achou := false;
4      i := 0;
5      j := 0;
6      idx := buscaIndice(F,noEst.posicao);
7      par := posicaoParenteses(F,idx);
8      se lado = 'd' então
9          enquanto i < tam faça
10             se (i >= par[0] e i <= idx) ou (i = par[1] ) então
11                 i := i + 1;
12             senão
13                 auxF[j] := F.No[i];
14                 j := j + 1;
15                 i := i + 1;
16     se lado = 'e' então
17         enquanto i < tam faça
18             se (i >= idx e i <= par[1]) ou (i = par[0] ) então
19                 i := i + 1;
20             senão
21                 auxF[j] := F.No[i];
22                 j := j + 1;
23                 i := i + 1;
24     F.No[]:= auxF;
25     retorna F;
```

Complexidade do algoritmo 25: O(n), pois as iterações das linhas 9 e 17 são exclusivas, e tem no máximo n iterações. As funções chamadas nas linhas 6, e 7, estão fora do escopo das

iterações, e todas possuem complexidade O(n).
Assim: $O(n) + O(n) + O(n) = O(n)$.

Algoritmo 26: APLICA REGRA (R$\forall$ OU L$\exists$) OU (L$\neg\forall$ OU R$\neg\exists$) SOBRE F.

```
1  Função aplicaRegraR∀L∃(F, noEst)
       /* armazena as posições de todos os quantificadores que devem ser
          reduzidos juntos.                                              */
2      posQuants[] := noEst.posNRJ[];
3      tam := tamanho de posQuants[];
4      posQuants[tam] := noEst.posicao;
5      tam := tamanho de posQuants[];
6      j :=0;
7      tam := tamanho de F.No[];
8      t := tamanho de idx[];
9      i := 0;
10     enquanto i<tam faça
11         para k := 0; k<t; k++ faça
12         se F.No[i].posicao = posQuants[k] então
13             i := i + 2;
               /* para 'eliminar' a relação associada ao quantificador.  */
14         senão
15             auxF[j] := F.No[i];
16             j := j + 1;
17     F.No[]:= auxF;
18     retorna F;
```

Complexidade do algoritmo 26: O(n^2), pois existem duas iterações aninhadas e executadas em um número proporcional a n. A primeria, na linha 10 tem entrada n, a segunda, na linhas 11, entrada $n/2$. Assim, a complexidade é $n \times n/2 = O(n^2)$.

Complexidade do algoritmo 27: O(n), pois todas as chamadas das funções listadas abaixo tem complexidade $O(n)$:
- buscaSequenteAxiomaD, na linha 10,
- buscaAntecedenteSequente, na linha 11,
- buscaSequenteAxiomaD, na linha 12,
- buscaAntecedenteSequente, na linha 13,
- buscaSequenteAxiomaE, na linha 15,
- buscaConsequenteSequente, linha 16,
- buscaSequenteAxiomaE, linha 17,
- buscaConsequenteSequente, linha 18,
- e as iterações nas linhas 19 e 23 são independentes, e cada uma tem complexidade O(n).

Assim: $O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) = O(n)$.

Algoritmo 27: APLICA REGRA DO CORTE SOBRE F.

```
1  Função aplicaRegraCorte(F, noEst, lado, S)
2      continua := true;
3      cont := 0;
       /* O nó noEst é um array com os 2 nós que formaram a conexão. O nó
          com menor posição, faz parte do axioma inicial.                  */
4      se noEst[0].posicao < noEst[1].posicao então
5          idNoAxioma := noEst[0].posicao;
6          idNo := noEst[1].posicao;
7      senão
8          idNoAxioma := noEst[1].posicao;
9          idNo := noEst[0].posicao;
10     se lado = 'D' então
           /* temos o axioma inicial para o ramo direito do sequentes      */
11         axioma := buscaSequenteAxiomaD(F, idNoAxioma);
           /* temos o consequente do sequente a ser provado                */
12         sucedente := buscaAntecedenteSequente(axioma, idNoAxioma)
           /* temos um sequente do consequente da fórmula. Falta extrair seu
              antecedente.                                                 */
13         seqCons := buscaSequenteAxiomaD(F, idNo);
           /* temos o antecedente do sequente a ser provado                */
14         antecedente := buscaAntecedenteSequente(seqCons, idNoAxioma)
15     senão
           /* temos o axioma inicial para o ramo esquerdo do sequentes     */
16         axioma := buscaSequenteAxiomaE(F, idNoAxioma);
           /* temos o antecedente do sequente a ser provado                */
17         antecedente := buscaConsequenteSequente(sequente)
           /* temos um sequente do consequente da fórmula. Falta extrair seu
              sucedente.                                                   */
18         seqCons := buscaSequenteAxiomaE(F, idNo);
           /* temos o sucedente do sequente a ser provado                  */
19         sucedente := buscaConsequenteSequente(sequente)
       /* O sequente a ser provado é a junção: antecedente + → + sucedente */
       /* S recebe seus elementos na ordem da direita para esquerda.       */
20     enquanto sucedente ≠ ∅ faça
21         empilhe o elemento do topo de sucedente em S;
22         desempilhe o elemento do topo de sucedente;
23     empilhe '→' em S;
24     enquanto antecedente ≠ ∅ faça
25         empilhe o elemento do topo de antecedente em S;
26         desempilhe o elemento do topo de antecedente;
27     F.No[]:= axioma;
28     retorna F;
```

Complexidade do algoritmo 28: O(n), pois:
- buscaIndice, na linha 2, tem complexidade $O(n)$,
- e as iterações nas linhas 3, 11 e 20 não são aninhadas, e cada uma tem complexidade O(n).
Assim: $O(n) + O(n) + O(n) = O(n)$.

Complexidade do algoritmo 29: O(n), pois as iterações nas linhas 3 e 11, não são aninhadas e cada uma pode realizar n iterações.
Assim: $O(n) + O(n) = O(n)$.

Algoritmo 28: BUSCA UM SEQUENTE OU AXIOMA INICIAL, DA DIREITA PARA ESQUERDA DA DADA FÓRMULA

```
1  Função buscaSequenteAxiomaD(F, idNo)
       /* pega o índice de ')' na fórmula que delimita o axioma inicial ou
          sequente que estamos procurando. Essa busca é feita da direita
          para a esquerda.                                                */
2      idx := buscaIndice(F,idNo);
3      enquanto continua = true faça
4          idx := idx + 1 ;
5          se F.No[idx].rotulo = ')' então
6              cont := cont + 1;
7          senão
8              continua := false;
9      idx := idx - 1;
10     continua := true; cont := 0;
       /* empilha em auxF o axioma inicial                                */
11     enquanto continua = true faça
12         se F.No[idx].rotulo = ')' então
13             cont := cont + 1;
14         senão se F.No[idx].rotulo = '(' então
15             cont := cont - 1;
16             se cont = 0 então
17                 continua = false;
18         empilhe em F.No[idx] em auxF;
19         idx := idx - 1 ;
       /* o axioma agora é emplilhado em axioma, sentido esquerda - direita
          */
20     enquanto auxF ≠ ∅ faça
21         empilhe o elemento do topo de auxF em axioma;
22         desempilhe o elemento do topo de auxF;
23 retorna axioma
```

Algoritmo 29: BUSCA UM SEQUENTE OU AXIOMA INICIAL, DA ESQUERDA PARA DIREITA DA DADA FÓRMULA

```
1  Função buscaSequenteAxiomaE(F, idNo)
       /* pega o índice de '(' na fórmula que delimita o axioma inicial ou
          sequente que estamos procurando. Essa busca é feita da esquerda
          para direita.                                                    */
2      idx := buscaIndice(F,idNo);
3      enquanto continua = true faça
4          idx := idx - 1 ;
5          se F.No[idx].rotulo = '(' então
6              cont := cont + 1;
7          senão
8              continua := false;
9      idx := idx + 1;
10     continua := true; cont := 0;
       /* empilha em auxF o axioma inicial                                 */
11     enquanto continua = true faça
12         se F.No[idx].rotulo = '(' então
13             cont := cont + 1;
14         senão se F.No[idx].rotulo = ')' então
15             cont := cont - 1;
16             se cont = 0 então
17                 continua = false;
18         empilhe em F.No[idx] em auxF;
19         idx := idx + 1 ;
       /* o axioma recebe auxF, que já está no sentido esquerda - direita
          (último elemento da direita = elemento do topo da pilha)         */
20     axioma := auxF;
21 retorna axioma
```

Complexidade do algoritmo 31: O(n), pois as iterações nas linhas 5 e 9, não são aninhadas e cada uma pode realizar n iterações.
Assim: $O(n) + O(n) = O(n)$.

Complexidade do algoritmo 32: O(n), pois realiza um tamanho proporcional a n iterações.

Algoritmo 30: BUSCA O ANTECEDENTE E UM SEQUENTE/AXIOMA INICIAL

```
1  Função buscaAntecedenteSequente(sequente, idNo)
       /* Armazena o antecedente de um Sequente/axioma. Para isso, é
          preciso desempilhar os elementos até o → e contar os ')'      */
2      continua := true;
3      cont := 0;
4      enquanto continua = true faça
5          se o atributo posicao do elemento do topo de sequente = idNo então
6              empilhe o elemento do topo de sequente em temp;
7              desempilhe o elemento do topo de sequente;
8              continua := false;
9          empilhe o elemento do topo de sequente em temp;
10         desempilhe o elemento do topo de sequente;
       /* Retira os parênteses excedentes                               */
11     cont := 0;
12     enquanto continua = true faça
13         se o atributo rotulo do elemento do topo de sequente = ')' então
14             cont := cont + 1;
15             empilhe o elemento do topo de sequente em auxSeq;
16             desempilhe o elemento do topo de sequente;
17         senão se o atributo rotulo do elemento do topo de sequente = '(' então
18             se cont > 1 então
19                 cont := cont - 1;
20                 empilhe o elemento do topo de sequente em auxSeq;
21                 desempilhe o elemento do topo de sequente;
22             senão se cont = 0 então
23                 continua = false;
24     enquanto auxSeq ≠ ∅ faça
25         empilhe o elemento do topo de auxSeq em antecedente;
26         desempilhe o elemento do topo de auxSeq;
27 retorna antecedente;
```

Algoritmo 31: BUSCA O CONSEQUENTE DE UM SEQUENTE/AXIOMA INICIAL

```
1  Função buscaConsequenteSequente(sequente)
       /* Armazena o consequente de um Sequente/axioma. Para isso, é
          preciso desempilhar os elementos após o → e contar os '('     */
2      continua := true;
3      cont := 0;
4      consequent;
       /* desempilha todos os elementos que antecedem →                 */
5      enquanto o rotulo do elemento do topo de sequente ≠ '→' faça
6          desempilhe o elemento do topo de sequente;
       /* desempilha o →                                                */
7      desempilhe o elemento do topo de sequente;
       /* Descobre o consequente, 'eliminando' os parênteses excedentes */
8      cont := 0;
9      enquanto continua = true faça
10         se o atributo rotulo do elemento do topo de sequente = '(' então
11             cont := cont + 1;
12             empilhe o elemento do topo de sequente em consequente ;
13             desempilhe o elemento do topo de sequente;
14         senão se o atributo rotulo do elemento do topo de sequente = ')' então
15             se cont > 1 então
16                 cont := cont - 1;
17                 empilhe o elemento do topo de sequente em consequente ;
18                 desempilhe o elemento do topo de sequente;
19             senão se cont = 0 então
20                 continua = false;
21         senão
               /* o atributo rotulo do elemento do topo de sequente é um
                  literal                                               */
22             empilhe o elemento do topo de sequente em consequente ;
23             desempilhe o elemento do topo de sequente;
24 retorna consequente
```

Algoritmo 32: BUSCA O ÍNDICE DE UM NÓ EM UM ARRAY.

```
1 Função buscaIndice(F,posicao)
      /* Busca índice de nó em um array                    */
2     tam := tamanho de F;
3     i := 0;
4     enquanto (i < tam) e (achou = false) faça
5         se F.No[i].posicao = posicao então
6             achou := true;
7             idx := i;
8         i := i + 1;
9     retorna idx
```

Algoritmo 33: ENCONTRA POSIÇÕES DOS PARÊNTESES QUE DELIMITAM A ABRANGÊNCIA DE UM CONECTIVO.

```
1 Função posicaoParenteses(F,idx)
2     achou := false;
3     tam := tamanho de F.No[];
4     j := 0;
5     i := idx - 1;
      /* Procura o '(' mais próximo do nó de índice idx.   */
6     enquanto (i >= 0) e (achou = false) faça
7         se F.No[i].rotulo = '(' então
8             achou := true;
9             par[j] := i;
10            j:= j + 1;
11        i := i + 1;
12    achou := false;
13    i := idx + 1;
14    enquanto (i < tam) e (achou = false) faça
15        se F.No[i].rotulo = ')' então
16            achou := true;
17            par[j] := i;
18            j:= j + 1;
19        i := i + 1;
20    retorna par;
```

Complexidade do algoritmo 30: O(n), pois as iterações nas linhas 4, 12 e 24, não são aninhadas e cada uma pode realizar n iterações.
Assim: $O(n) + O(n) + O(n) = O(n)$.

Complexidade do algoritmo 33: O(n), pois as iterações nas linhas 6 e 14, não são aninhadas e cada uma realiza um tamanho proporcional a n iterações. Assim: $O(n) + O(n) = O(n)$.

Algoritmo 34: BUSCA A REGRA NO CÁLCULO DE SEQUENTES CORRESPONDENTE AO NÓ.

```
1 Função buscaRegraSequentes(noArv, noEst)
2     cn1[] = buscaCaminho(noArv, noEst.posicao, ∅);
3     resp := constaNoRotulo(¬, cn1[]);
      /* Se a negação precede o nó, busca a regra em uma tabela X, que deve
         representar a tabela 8, senão, na tabela Y, representando a
         tabela 7.                                                        */
4     se resp = true então
5         regra := buscaRegra(noEst, tabela X);
6     senão
7         regra := buscaRegra(noEst, tabela Y);
8     retorna regra;
```

Complexidade do algoritmo 34: O(n), pois:
- buscaCaminho, na linha 2, tem complexidade $O(n)$;
- constaNoRotulo, na linha 3, tem complexidade $O(n)$;
- as linhas 5 e 7, são buscas em tabelas de tamanho fixo.

Assim: $O(n) + O(n) + O(1) + O(1) = O(n)$.

Algoritmo 35: CHECA UM NÓ COM UM DADO RÓTULO ESTÁ NO CAMINHO ENTRE O NÓ RAIZ E UM DADO NÓ.

```
1 Função constaNoRotulo(rotulo, caminho)
      /* Checa um nó com um dado rótulo está no caminho entre o nó raiz e
         um dado nó. Por exemplo, um nó com rótulo igual a ¬ precede um
         certo nó.                                                        */
2     achou := false;
3     i := 0;
4     tam := tamanho do caminho;
5     enquanto (i <= tam - 2) e (achou = false) faça
6         se caminho[i].rotulo = rotulo então
7             achou := true;
8     retorna achou;
```

Complexidade do algoritmo 35: O(n), pois ele realiza um tamanho proporcional a n iterações.

Algoritmo 36: BUSCA UM NÓ COM UM DADO RÓTULO NO CAMINHO ENTRE O NÓ RAIZ E UM DADO NÓ.

```
1 Função buscaNoRotulo(rotulo, caminho)
      /* Busca um nó com um dado rótulo no caminho entre o nó raiz e um
         dado nó. Por exemplo, um nó com rótulo igual a ¬ precede um certo
         nó.                                                              */
2     achou := false;
3     i := 0;
4     tam := tamanho do caminho;
5     enquanto (i <= tam - 2) e (achou = false) faça
6         se caminho[i].rotulo = rotulo então
7             achou := true;
8     retorna caminho[i];
```

Complexidade do algoritmo 36: O(n), pois ele realiza um tamanho proporcional a n iterações.