LETÍCIA CARVALHO PIVETTA FENDT

REFINAMENTOS
PARA
O MÉTODO DOS TABLEAUX

FLORIANÓPOLIS
2000

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO

REFINAMENTOS PARA O MÉTODO DOS TABLEAUX

Dissertação submetida à Universidade Federal de Santa Catarina
como parte dos requisitos para a obtenção
do grau de Mestre em Ciência da Computação.

LETÍCIA CARVALHO PIVETTA FENDT

Florianópolis, Fevereiro de 2000.

REFINAMENTOS PARA O MÉTODO DOS TABLEAUX

Letícia Carvalho Pivetta Fendt

'Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Ciências da Computação, Área de Concentração *Sistemas de Conhecimento*, e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina'.

Prof. Arthur Ronald de Vallauris Buchsbaum, Dr. - Orientador

Prof. Fernando Ostuni Gauthier, Dr.
Coordenador do Curso de Pós-Graduação em Ciência da Computação

Banca Examinadora:

Prof. Jorge Muniz Barreto, Dr. (CPGCC – UFSC)

Prof. Rosvelter João Coelho da Costa, Dr. (CPGCC – UFSC)

Prof. Tarcísio Haroldo Cavalcante Pequeno, Dr. (LIA - UFC)

Florianópolis, Fevereiro de 2000.

Para minha filha Thayna

AGRADECIMENTOS

Ao Curso de Pós-graduação em Ciência da Computação e à Universidade Federal de Santa Catarina pela infra-estrutura e organização que viabilizaram o desenvolvimento deste trabalho.

Ao meu orientador, Prof. Arthur Ronald de Vallauris Buchsbaum, pelas valiosas idéias, dedicação e ensinamentos.

Aos professores Jorge Muniz Barreto, Rosvelter João Coelho da Costa e Tarcísio Haroldo Cavalcante Pequeno, por terem julgado este trabalho.

À direção geral e acadêmica e à coordenação do Curso de Informática do Instituto Luterano de Ensino Superior de Ji-Paraná, pela sensibilidade e apoio à finalização deste trabalho.

Aos amigos Ana, Adja, Alessandra, Fabiane, Laura, e, principalmente, Cristiane e Nederson pelo apoio e companheirismo.

Aos meus pais José Luiz e Maria Conceição, e minhas irmãs Patrícia e Bruna, pelo incentivo e carinho.

Agradeço especialmente a minha filha Thayna, e ao meu esposo Flavio, pela compreensão, paciência e amor que me ajudaram nos momentos mais difíceis, que juntos possamos colher os melhores frutos deste trabalho.

RESUMO

No presente trabalho são propostos refinamentos sobre o método dos tableaux, tal como tradicionalmente apresentado, visando diminuir o número de nós das árvores de prova, bem como aumentar a possibilidade de obtenção de respostas. Para isto especificamos e implementamos três diferentes algoritmos baseados no método tradicional dos tableaux para a lógica clássica. O mais sofisticado e avançado dos três métodos recorre ao procedimento de unificação, comumente utilizado no método da resolução.

Finalizamos este trabalho apresentando uma série de testes, para verificar experimentalmente a consecução dos objetivos propostos.

ABSTRACT

In this work some refinements are proposed on the tableaux method, aiming to decrease the number of nodes of the proof trees, as well as to increase the possibility of obtaining answers. For this three different algorithms are specified and implemented, based on the traditional tableau method for classical logic. The most sophisticated and advanced among them appeals to the unification procedure, commonly used by the resolution method.

Finally, this work presents a series of tests, for verifying the attainment of the proposed objectives.

SUMÁRIO

LISTA DE FIGURAS

1 - INTRODUÇÃO

Ao mencionarmos a palavra inteligência o primeiro pensamento que nos vem à mente é o raciocínio. A maioria das pessoas crê na agilidade de raciocínio como um dos indicadores de inteligência. A possibilidade de ter acesso, analisar e guardar informações passa quase desapercebida pelo ser humano. Já dizia Descartes: "Penso, logo existo." Porém, uma série de tarefas realizadas pelo homem exige um esforço rotineiro, mecânico. Tais tarefas, se executadas por uma máquina, liberariam o homem deste esforço, possibilitando que o mesmo se dedique a tarefas que necessitam de criatividade.

A lógica, observada em [Barreto 1997] como estudo dos mecanismos do pensamento, possui um papel importante na Inteligência Artificial, sendo discutida desde os primórdios desta como um método de representação de conhecimento e modelagem de raciocínio.

Podemos realizar a simulação de uma dada forma de raciocínio através dos seguintes passos:

1º) Escolha e/ou modelagem de uma lógica que melhor reflita esta forma de raciocínio;

2º) Especificação de um método de prova automático, parcial ou total para tal lógica, que melhor reflita esta forma de raciocínio.

Um método automático de prova consiste na obtenção de um algoritmo sistemático que decida, da forma mais ampla possível, quando uma dada fórmula é conseqüência lógica de uma dada base de conhecimento. Como métodos de prova, criados visando a utilização humana, podemos citar sistemas de Hilbert ou cálculos axiomáticos, dedução natural e cálculo de seqüentes. Como métodos criados para prova

automática por computador temos, por exemplo, o método da resolução e o método dos tableaux.

O presente trabalho trata de algoritmos de automatização do processo de raciocínio para a lógica quantificacional clássica, e utiliza, como algoritmo de prova, o método dos tableaux.

Optamos pela lógica quantificacional clássica pelos exaustivos estudos sobre a mesma, pela simplicidade de compreensão e manuseio de suas estruturas e pelo fato desta lógica ser adequada e suficiente para a representação de uma infinidade de fatos. Além disso, diversas lógicas não clássicas possuem estruturas semelhantes à da lógica quantificacional clássica, consequentemente as diversas técnicas que mostramos no nosso trabalho são de fácil adaptação para algumas outras lógicas, tais como algumas das lógicas paraconsistentes, paracompletas e não aléticas, como por exemplo com respeito àquelas expostas em [Buchsbaum & Pequeno 1991].

Quanto à utilização do método dos tableaux, podemos citar pontos muito específicos para sua escolha; por exemplo, o algoritmo dos tableaux não necessita de transformações para qualquer forma normal. Segundo [Oppacher & Suen 1988], a principal vantagem do método dos tableaux, em relação ao método tradicional e conhecido de resolução é fato deste evitar utilização de formas clausais. Além disso, segundo [Buchsbaum 1988], o método dos tableaux é flexível e possui uma descrição bem simples.

Este trabalho estuda o método dos tableaux para a lógica quantificacional clássica e está disposto da seguinte maneira:

- Capítulo 2 - apresenta a lógica clássica;
- Capítulo 3 - apresenta o método dos tableaux;
- Capítulo 4 - apresenta o sistema de tableaux tradicional para a lógica quantificacional clássica;

- Capítulo 5 – apresenta a descrição dos refinamentos para o sistema de trableaux tradicional;
- Capítulo 6 - contém a descrição da implementação do sistema computacional desenvolvido, o qual integra o método tradicional, descrito no capítulo 4, e seus refinamentos, descritos no capítulo 5;
- Capítulo 7 – contém uma série de resultados obtidos através de testes realizados sobre o sistema implementado;
- Capítulo 8 – apresenta as conclusões e perspectivas futuras relacionadas a este trabalho.

2 - LÓGICA CLÁSSICA

1 – Linguagens de Primeira Ordem

1.1 Definição: Por um *alfabeto* **A** entende-se uma coleção não vazia de sinais gráficos. Uma *samblagem* de um dado alfabeto é uma seqüência finita de símbolos do alfabeto. $\mathbf{A}^*$ denota o conjunto de todas as samblagens de **A**.

1.2 Definição: Um *alfabeto de primeira ordem* é a união das seguintes coleções de sinais, disjuntas duas a duas:

- uma coleção infinita enumerável de sinais ditos *variáveis*, as quais são as mesmas em todo alfabeto de primeira ordem;
- a coleção $\{\neg, \wedge, \vee, \rightarrow\}$ de sinais ditos *conetivos lógicos*;
- a coleção $\{\forall, \exists\}$ de sinais ditos *quantificadores*;
- a coleção {")", "(", ","} de sinais ditos *de pontuação*;
- uma coleção (eventualmente vazia) de sinais ditos *constantes*;
- um alfabeto de primeira ordem *com igualdade* contém também a coleção {=};
- uma coleção (possivelmente vazia) de sinais ditos *funcionais*; a cada sinal funcional associamos uma coleção de números positivos, ditos as suas aridades;
- uma coleção (não vazia) de sinais ditos *predicativos*; a cada sinal predicativo associamos uma coleção de números naturais, ditos as suas aridades.

1.3 Convenção: Consideramos que a função sintática de cada sinal utilizada em dois alfabetos distintos é invariante. Em particular, temos que:

- todo sinal considerado constante em um dado alfabeto deve continuar a sê-lo em qualquer outro alfabeto considerado no mesmo contexto;
- todo sinal considerado funcional (predicativo) em um dado alfabeto deve continuar a sê-lo em qualquer outro alfabeto considerado no mesmo contexto, de modo que a sua coleção de aridades é invariante para todos os alfabetos considerados.

1.4 Convenção: De agora em diante, a não ser que seja dito expressamente o contrário, adotaremos as seguintes convenções notacionais:

- as letras x, y, z, w, seguidas ou não de plicas ou índices, denotam variáveis de um alfabeto de primeira ordem;
- as letras **a**, **b**, **c**, seguidas ou não de plicas ou índices, denotam constantes de um alfabeto de primeira ordem;
- as letras **f**, **g**, **h**, seguidas ou não de plicas ou índices, denotam sinais funcionais de um alfabeto de primeira ordem;
- as letras **p**, **q**, **r**, seguidas ou não de plicas ou índices, denotam sinais predicativos de um alfabeto de primeira ordem;
- a letra Σ, seguida ou não de plicas ou índices, denota um alfabeto de primeira ordem.

1.5 Definição: As seguintes cláusulas especificam o que entendemos por *Σ-termos*:

- toda variável é um Σ-termo;
- toda constante de Σ é um Σ-termo;
- se $\mathbf{t_1},\ldots, \mathbf{t_n}$ são Σ-termos e se **f** é um sinal funcional de Σ com aridade **n**, então $\mathbf{f}(\mathbf{t_1},\ldots, \mathbf{t_n})$ é um Σ-termo.

1.6 Definição: As seguintes cláusulas especificam o que entendemos por *Σ-fórmulas*:

- se $\mathbf{t_1},..., \mathbf{t_n}$ são Σ-termos e **r** é um sinal predicativo de Σ com aridade **n**, então $\mathbf{r(t_1,..., t_n)}$ é uma Σ-fórmula;
- se **P** é uma Σ-fórmula, então $\neg\mathbf{P}$ é uma Σ-fórmula;
- se P e Q são Σ-fórmulas, então $(P \wedge Q)$, $(P \vee Q)$ e $(P \rightarrow Q)$ são Σ-fórmulas;
- se **P** é uma Σ-fórmula e x é uma variável, então $\forall x\,\mathbf{P}$ e $\exists x\,\mathbf{P}$ são Σ-fórmulas.

1.7 Definição: Uma *linguagem de primeira ordem* é a coleção de todas as Σ-fórmulas de um dado alfabeto Σ; dizemos também neste caso que tal linguagem é a linguagem de primeira ordem gerada por Σ. Se **t** é um Σ-termo e **L** é a linguagem gerada por Σ, dizemos que **t** é um termo em **L**.

1.8 Convenção: Sempre que for irrelevante a indicação do alfabeto Σ específico, usaremos respectivamente as palavras "termos" e "fórmulas" no lugar de "Σ-termos" e "Σ-fórmulas."

1.9 Convenção: Adotamos as seguintes convenções sintáticas para todos os sinais abaixo, seguidos ou não de plicas ou índices:

- **t**, **u** denotam termos;
- **P**, **Q**, **R**, **S** denotam fórmulas;
- Γ, Θ denotam coleções de fórmulas;
- **L** denota uma linguagem de primeira ordem.

1.10: Definição: Um *designador* é um termo ou uma fórmula.

1.11 Definição: Pelas cláusulas abaixo especificamos quando um termo é *subtermo* de outro termo:

- **t** é subtermo de **t**;
- $\mathbf{t_i}$ é subtermo de $\mathbf{f(t_1,..., t_n)}$, onde $1 \leq \mathbf{i} \leq \mathbf{n}$;
- se $\mathbf{t_1}$ é subtermo de $\mathbf{t_2}$ e $\mathbf{t_2}$ é subtermo de $\mathbf{t_3}$, então $\mathbf{t_1}$ é subtermo de $\mathbf{t_3}$.

1.12 Definição: "$\mathbf{P} \leftrightarrow \mathbf{Q}$" é uma abreviatura para "$(\mathbf{P} \rightarrow \mathbf{Q}) \wedge (\mathbf{Q} \rightarrow \mathbf{P})$".

1.13 Definição: Pelas cláusulas abaixo especificamos quando uma dada fórmula é *subfórmula* de outra dada fórmula:

- **P** é subfórmula de **P**;
- **P** é subfórmula de $\neg\mathbf{P}$;
- **P** e **Q** são subfórmulas de $(\mathbf{P} \rightarrow \mathbf{Q})$, $(\mathbf{P} \vee \mathbf{Q})$ e $(\mathbf{P} \wedge \mathbf{Q})$;
- **P** é subfórmula de $\forall x\, \mathbf{P}$ e $\exists x\, \mathbf{P}$;
- se **P** é subfórmula de **Q** e **Q** é subfórmula de **R**, então **P** é subfórmula de **R**.

1.14 Convenção: Adotaremos as seguintes convenções para escrita informal de termos e fórmulas:

- notamos $\mathbf{f}(\mathbf{t_1}, \mathbf{t_2})$ por $(\mathbf{t_1}\ \mathbf{f}\ \mathbf{t_2})$;
- notamos $\mathbf{p}(\mathbf{t_1}, \mathbf{t_2})$ por $(\mathbf{t_1}\ \mathbf{p}\ \mathbf{t_2})$;
- quando $(\mathbf{t_1}\ \mathbf{f}\ \mathbf{t_2})$ não está escrito como subtermo de outro termo podemos prescindir do par exterior de parênteses;
- quando $(\mathbf{t_1}\ \mathbf{p}\ \mathbf{t_2})$ não está escrito como subfórmula de outra fórmula podemos prescindir do par exterior de parênteses;
- mesmo quando $(\mathbf{t_1}\ \mathbf{p}\ \mathbf{t_2})$ está escrito como subfórmula de outra fórmula, podemos prescindir do par exterior de parênteses, se isto não prejudicar a clareza da escrita da fórmula envolvida;
- quando $(\mathbf{P} \rightarrow \mathbf{Q})$, $(\mathbf{P} \wedge \mathbf{Q})$ e $(\mathbf{P} \vee \mathbf{Q})$ não estão escritos como subfórmulas de outra fórmula podemos prescindir do par exterior de parênteses;
- a seguinte lista fornece a ordem de prioridade para separação em subfórmulas: $\leftrightarrow$, $\rightarrow$, $\{\wedge, \vee\}$; por exemplo, "$\mathbf{P} \leftrightarrow \mathbf{Q} \vee \mathbf{R} \rightarrow \mathbf{S}$" representa "$\mathbf{P} \leftrightarrow ((\mathbf{Q} \vee \mathbf{R}) \rightarrow \mathbf{S})$";
- quando o mesmo conetivo de aridade 2 se suceder em uma fórmula, a parentização implícita se dá da direita para a esquerda; por exemplo, "$\mathbf{P} \rightarrow \mathbf{Q} \rightarrow \mathbf{P}$" representa "$\mathbf{P} \rightarrow (\mathbf{Q} \rightarrow \mathbf{P})$".

1.15 Definição: Especificamos abaixo quando uma determinada *variável* é *livre em uma fórmula*:

- x é livre em $\mathbf{r}(t_1,..., t_n)$ se x ocorre em um dos termos $t_1,..., t_n$;
- x é livre em $\neg\mathbf{P}$ se x é livre em $\mathbf{P}$;
- x é livre em $(\mathbf{P} \rightarrow \mathbf{Q})$, $(\mathbf{P} \wedge \mathbf{Q})$ e $(\mathbf{P} \vee \mathbf{Q})$ se x é livre em $\mathbf{P}$ ou x é livre em $\mathbf{Q}$;
- x não é livre em $\forall x\ \mathbf{P}$ e $\exists x\ \mathbf{P}$;
- x é livre em $\forall y\ \mathbf{P}$ e $\exists y\ \mathbf{P}$ se x é distinto de y e x é livre em $\mathbf{P}$.

1.16 Definição: Se $\mathbf{D}$ é um designador, dizemos que x é livre em $\mathbf{D}$ se uma das seguintes condições for satisfeita:

- $\mathbf{D}$ é um termo e x ocorre em $\mathbf{D}$;
- $\mathbf{D}$ é uma fórmula e x ocorre livre em $\mathbf{D}$.

Se $\mathbf{S}$ é uma coleção de designadores, dizemos que x é livre em $\mathbf{S}$ se x é livre em pelo menos um dos elementos de $\mathbf{S}$.

1.17 Definição: *Termos fechados* são termos nos quais não ocorrem variáveis.

1.18 Definição: *Sentenças* são fórmulas sem variáveis livres.

1.19 Definição: Especificamos abaixo o resultado da *substituição de uma variável por um dado termo em um termo ou uma fórmula* (consideramos aqui x distinto de y):

- $\mathbf{c}(x/\mathbf{t}) = \mathbf{c}$;
- $x(x/\mathbf{t}) = \mathbf{t}$;
- $y(x/\mathbf{t}) = y$;
- $\mathbf{f}(t_1,..., t_n)(x/\mathbf{t}) = \mathbf{f}(t_1(x/\mathbf{t}),..., t_n(x/\mathbf{t}))$;
- $\mathbf{p}(t_1,..., t_n)(x/\mathbf{t}) = \mathbf{p}(t_1(x/\mathbf{t}),..., t_n(x/\mathbf{t}))$;
- $(\neg\mathbf{P})(x/\mathbf{t}) = \neg\mathbf{P}(x/\mathbf{t})$;
- $(\mathbf{P} \rightarrow \mathbf{Q})(x/\mathbf{t}) = \mathbf{P}(x/\mathbf{t}) \rightarrow \mathbf{Q}(x/\mathbf{t})$;
- $(\mathbf{P} \wedge \mathbf{Q})(x/\mathbf{t}) = \mathbf{P}(x/\mathbf{t}) \wedge \mathbf{Q}(x/\mathbf{t})$;
- $(\mathbf{P} \vee \mathbf{Q})(x/\mathbf{t}) = \mathbf{P}(x/\mathbf{t}) \vee \mathbf{Q}(x/\mathbf{t})$;
- $(\forall x\ \mathbf{P})(x/\mathbf{t}) = \forall x\ \mathbf{P}$;

- $(\exists x\ \mathbf{P})(x/t) = \exists x\ \mathbf{P}$;

- $(\forall y\ \mathbf{P})(x/t) =$
  - * $\forall y\ \mathbf{P}$, se x não é livre em **P**;
  - * $\forall y\ \mathbf{P}(x/t)$, se x é livre em **P** e y não ocorre em **t**;
  - * $\forall z\ \mathbf{P}(y/z)(x/t)$, se x é livre em **P** e y ocorre em **t**, onde z é a primeira variável, distinta de x e y, que não é livre em **P** e não ocorre em **t**;

- $(\exists y\ \mathbf{P})(x/t) =$
  - * $\exists y\ \mathbf{P}$, se x não é livre em **P**;
  - * $\exists y\ \mathbf{P}(x/t)$, se x é livre em **P** e y não ocorre em **t**;
  - * $\exists z\ \mathbf{P}(y/z)(x/t)$, se x é livre em **P** e y ocorre em **t**, onde z é a primeira variável, distinta de x e y, que não é livre em **P** e não ocorre em **t**.

1.20 Definição: Dizemos que um termo **t** (uma fórmula **P**) *está no escopo de* uma variável x *em* uma fórmula **Q** se existe uma subfórmula de **Q** de uma das formas $\forall x\ \mathbf{R}$ ou $\exists x\ \mathbf{R}$, tal que **t** (**P**) ocorre em **R**.

1.21 Definição: Especificamos abaixo quando duas fórmulas são congruentes:

- **P** é congruente a **P**;
- se **P** é congruente a **P'**, então $\neg\mathbf{P}$ é congruente a $\neg\mathbf{P}'$;
- se **P** é congruente a **P'** e **Q** é congruente a **Q'**, então $\mathbf{P} \rightarrow \mathbf{Q}$ ($\mathbf{P} \wedge \mathbf{Q}$, $\mathbf{P} \vee \mathbf{Q}$) é congruente a $\mathbf{P}' \rightarrow \mathbf{Q}'$ ($\mathbf{P}' \wedge \mathbf{Q}'$, $\mathbf{P}' \vee \mathbf{Q}'$);
- se **P** é congruente a **P'**, então $\forall x\ \mathbf{P}$ ($\exists x\ \mathbf{P}$) é congruente a $\forall x\ \mathbf{P}'$ ($\exists x\ \mathbf{P}'$);
- se y não é livre em **P** e **P(x/y)** é congruente a **P'**, então $\forall x\ \mathbf{P}$ ($\exists x\ \mathbf{P}$) é congruente a $\forall y\ \mathbf{P}'(x/y)$ ($\exists y\ \mathbf{P}'(x/y)$).

2 - Semântica Clássica

Atribuímos significados a fórmulas de uma dada linguagem escolhendo uma coleção de *valores veritativos*, os quais são distintos juízos que podem ser feitos acerca do grau de verdade e falsidade de uma dada fórmula. Os valores veritativos que atribuem graus de veracidade a uma dada fórmula são também chamados de *valores distinguidos*.

Em qualquer lógica o conceito semântico fundamental é o de *valoração*, a qual é uma função que associa fórmulas a valores veritativos, possuindo determinadas propriedades, variando de lógica para lógica. Definimos assim uma semântica para uma dada lógica, especificando para cada linguagem **L** desta lógica qual é o conjunto associado de valorações para **L**.

2.1 Predefinição: Dada uma lógica consideramos conhecida a coleção correspondente de valores veritativos e, para cada linguagem desta lógica, a coleção correspondente de valorações.

2.2 Definição: Dizemos que **V** é uma valoração em uma dada lógica se existe uma linguagem **L**, desta lógica, tal que **V** é uma valoração para **L**.

Considere nos parágrafos seguintes uma lógica fixa, e seja **L** uma linguagem para esta lógica.

2.3 Definição: Dizemos que uma valoração **V** para **L** *satisfaz* uma fórmula **P** de **L** se **V(P)** é um valor distinguido. **V** *satisfaz* uma coleção de fórmulas de **L** se esta satisfizer todas as fórmulas desta coleção.

2.4 Definição: Uma valoração **V** satisfaz **P** (Γ) se existir uma linguagem **L** tal que **V** é uma valoração para **L**, **P** (Γ) é uma fórmula de **L** (uma coleção de fórmulas de **L**) e **V** satisfaz **P** (Γ).

2.5 Definição: Se $\mathbf{P} \in \mathbf{L}$ ($\Gamma \subseteq \mathbf{L}$) e **V** é uma valoração para **L**, dizemos que **V** é uma valoração para **P** (Γ).

2.6 Definição: Considere $\Gamma \cup \{\mathbf{P}\} \subseteq \mathbf{L}$. Nas cláusulas abaixo especificamos conceitos semânticos básicos para fórmulas e para coleções de fórmulas em uma dada lógica fixa:

- **P** (Γ) é *satisfatível em* **L** se existe uma valoração para **L** que satisfaz **P** (Γ);
- **P** (Γ) é *satisfatível* se existe uma valoração para **P** (Γ) que satisfaz **P** (Γ);
- **P** (Γ) é *insatisfatível em* **L** se toda valoração para **L** não satisfaz **P** (Γ);
- **P** (Γ) é *insatisfatível* se toda valoração para **P** (Γ) não satisfaz **P** (Γ);
- **P** (Γ) é *válido em* **L** se toda valoração para **L** satisfaz **P** (Γ);
- **P** (Γ) é *válido* se toda valoração para **P** (Γ) satisfaz **P** (Γ);
- **P** (Γ) é *inválido em* **L** se existe uma valoração para **L** que não satisfaz **P** (Γ);
- **P** (Γ) é *inválido* se existe valoração para **P** (Γ) não satisfaz **P** (Γ);
- **P** é *conseqüência semântica* de Γ *em* **L**, e notamos isto por $\Gamma \models \mathbf{P}$ (**L**), se toda valoração para **L** que satisfaz Γ satisfaz **P**;
- **P** é *conseqüência semântica* de Γ se toda valoração para $\Gamma \cup \{\mathbf{P}\}$ que satisfaz Γ satisfaz **P**.

2.7 Definição: Em qualquer um dos níveis proposicional, quantificacional ou equacional da lógica clássica, a coleção de valores veritativos correspondente é a mais simples possível, contendo apenas dois valores veritativos: v, representando "*verdadeiro*", e **f**, representando "*falso*". Dotamos o conjunto {v, **f**} de uma relação de ordem, considerando **f** estritamente menor que v.

2.8 Escólio: O leitor pode se convencer facilmente que este conjunto, munido da relação da ordem que acabamos de definir, é uma álgebra booleana, na qual qualquer subconjunto da mesma possui um elemento mínimo e possui um elemento máximo.

Temos também, nesta álgebra que o complemento de **v**, notado por **v'**, é **f**, e o complemento de **f**, notado por **f'**, é **v**.

2.9 Definição: Uma valoração proposicional para **L** é uma função **V** atendendo às seguintes condições:

- **V** é uma função de **L** em {v, f};
- $\mathbf{V}(\neg\mathbf{P}) = \mathbf{V}(\mathbf{P})'$;
- $\mathbf{V}(\mathbf{P} \rightarrow \mathbf{Q}) = \max\{\mathbf{V}(\mathbf{P})', \mathbf{V}(\mathbf{Q})\}$;
- $\mathbf{V}(\mathbf{P} \wedge \mathbf{Q}) = \min\{\mathbf{V}(\mathbf{P}), \mathbf{V}(\mathbf{Q})\}$;
- $\mathbf{V}(\mathbf{P} \vee \mathbf{Q}) = \max\{\mathbf{V}(\mathbf{P}), \mathbf{V}(\mathbf{Q})\}$.

2.10 Teorema: Na lógica proposicional clássica os conceitos de satisfatibilidade, insatisfatibilidade, validade, invalidade e conseqüência semântica são invariantes com respeito às mudanças de linguagem.

Para definirmos uma semântica da lógica clássica a nível quantificacional devemos atribuir significados a constantes, variáveis, sinais funcionais e sinais predicativos (através do conceito de *interpretação*), obtendo duas funções semânticas básicas: a primeira, dita *denotação*, associa termos a indivíduos do universo de discurso da interpretação, e a segunda, dita *valoração quantificacional*, associa fórmulas a valores veritativos.

2.11 Definição: Uma **L**-*interpretação* (para a lógica quantificacional clássica) é um terno $\mathbf{I} = \langle\Delta, \sigma, \mathbf{s}\rangle$ atendendo às seguintes condições:

- Δ é um conjunto não vazio, dito o *universo de discurso*;
- σ é uma função, dita a **L**-*atribuição de sinais* de **I**, satisfazendo as seguintes cláusulas:
  - para cada constante **c** em **L**, $\sigma(\mathbf{c}) \in \Delta$;
  - para cada sinal funcional **f** em **L** e cada aridade **n** de **f**, $\sigma(\mathbf{f},\mathbf{n})$ é uma função de $\Delta^{\mathbf{n}}$ em Δ;

- para cada sinal predicativo **p** em **L** e cada aridade **n** de **p**, $\sigma(\mathbf{p},\mathbf{n})$ está contido em $\Delta^{\mathbf{n}}$;

- **s** é uma função, dita a Δ-*atribuição de variáveis* de **I**, da coleção de variáveis para Δ.

Se **I** é uma **L**-interpretação para a lógica equacional clássica, então $\sigma(=, 2)$ é a coleção $\{(x, x)/\ x \in \Delta\}$. Uma *interpretação* é uma **L**-interpretação para alguma linguagem de primeira ordem **L**; dizemos neste caso que **L** é a linguagem da interpretação. Se $\mathbf{P} \in \mathbf{L}$ ($\Gamma \subseteq \mathbf{L}$) e **L** é a linguagem de uma interpretação **I**, dizemos que **I** é uma interpretação para **P** (Γ).

2.12 Definição: Dada uma Δ-atribuição **s** de variáveis e um elemento **d** de Δ, definimos uma Δ-atribuição de variáveis (diferindo possivelmente de **s** em x), notada por $\mathbf{s}(x/\mathbf{d})$, por:

$$\mathbf{s}(x/\mathbf{d})(y) = \begin{cases} \mathbf{d}, \text{ se } x = y, \\ \mathbf{s}(y), \text{ se } x \neq y \end{cases}$$

Se $\mathbf{I} = \langle\Delta, \sigma, \mathbf{s}\rangle$ é uma **L**-interpretação, então $\mathbf{I}(x/\mathbf{d})$ é a **L**-interpretação especificada pelo terno $\mathbf{I} = \langle\Delta, \sigma, \mathbf{s}(x/\mathbf{d})\rangle$.

2.13 Definição: Dada uma **L**-interpretação $\mathbf{I} = \langle\Delta, \sigma, \mathbf{s}\rangle$, definimos duas funções $\mathbf{I_t}$ e $\mathbf{I_f}$, ditas respectivamente a **L**-*denotação* e a **L**-*valoração quantificacional* geradas por **I**, pelas seguintes cláusulas:

- $\mathbf{I_t}$ é uma função da coleção de termos em **L** para Δ;
- $\mathbf{I_t}(\mathbf{c}) = \sigma(\mathbf{c})$;
- $\mathbf{I_t}(x) = \mathbf{s}(x)$;
- $\mathbf{I_t}(\mathbf{f}(\mathbf{t_1},..., \mathbf{t_n})) = \sigma(\mathbf{f},\mathbf{n})(\mathbf{I_t}(\mathbf{t_1}),..., \mathbf{I_t}(\mathbf{t_n}))$;
- $\mathbf{I_f}$ é uma **L**-valoração proposicional de **L** em $\{\mathbf{v}, \mathbf{f}\}$;
- $\mathbf{I_f}(\mathbf{p}(\mathbf{t_1},..., \mathbf{t_n})) = \mathbf{v}$ sss $\langle\mathbf{I_t}(\mathbf{t_1}),..., \mathbf{I_t}(\mathbf{t_n})\rangle \in \sigma(\mathbf{p},\mathbf{n})$;
- $\mathbf{I_f}(\forall x\ \mathbf{P}) = \min\{\mathbf{I}(x/\mathbf{d})_\mathbf{f}(\mathbf{P})/\ \mathbf{d} \in \Delta\}$;
- $\mathbf{I_f}(\exists x\ \mathbf{P}) = \max\{\mathbf{I}(x/\mathbf{d})_\mathbf{f}(\mathbf{P})/\ \mathbf{d} \in \Delta\}$.

2.14 Escólio: Se **I** é uma **L**-*interpretação para a lógica equacional clássica*, então $\mathbf{I}_f$ é dita a **L**-*valoração equacional* gerada por I, e obedece à seguinte cláusula adicional:

- $\mathbf{I}_f(t = t') = v$ sss $\mathbf{I}_t(t) = \mathbf{I}_t(t')$.

2.15 Definição: Dizemos que uma **L**-interpretação *satisfaz* uma fórmula **P** de **L** se $\mathbf{I}_f(\mathbf{P}) = v$. Uma **L**-interpretação *satisfaz* uma coleção Γ de fórmulas de **L** se esta satisfizer todas as fórmulas desta coleção. Uma interpretação **I** satisfaz **P** (Γ) se esta é uma **L**-interpretação para **P** (Γ) que satisfaz **P** (Γ).

2.16 Notação: Usamos as seguintes siglas:

- **LPC** = lógica proposicional clássica;
- **LQC** = lógica quantificacional clássica;
- **LEC** = lógica equacional clássica.

2.17 Teorema: Se $\Gamma \cup \{\mathbf{P}\}$ é uma coleção de fórmulas não contendo quantificadores, então:

- $\Gamma \models_{\mathbf{LPC}} \mathbf{P}$ sss $\Gamma \models_{\mathbf{LQC}} \mathbf{P}$;
- $\Gamma \models_{\mathbf{LPC}} \mathbf{P}$ sss $\Gamma \models_{\mathbf{LEC}} \mathbf{P}$.

2.18 Teorema: Se $\Gamma \cup \{\mathbf{P}\}$ é uma coleção de fórmulas não contendo o sinal de "=", então:

- $\Gamma \models_{\mathbf{LQC}} \mathbf{P}$ sss $\Gamma \models_{\mathbf{LEC}} \mathbf{P}$.

Os dois teoremas anteriores dizem que a lógica quantificacional clássica e equacional são extensões conservativas da lógica proposicional clássica, e que a lógica equacional clássica é uma extensão conservativa da lógica quantificacional clássica. Devido a este fato estabelecemos a convenção seguinte:

2.19 Convenção: De agora em diante, quando expusermos fatos semânticos, ficará implícito que estamos nos referindo à lógica clássica, sem nos preocuparmos se estamos

em um dos níveis proposicional, quantificacional ou equacional, a não ser que seja necessário.

2.20 Definição: Dizemos que duas interpretações $\mathbf{I} = \langle \Delta, \sigma, \mathbf{s} \rangle$ e $\mathbf{I'} = \langle \Delta, \sigma', \mathbf{s'} \rangle$ *concordam em* uma fórmula $\mathbf{P}(\Gamma)$ se as seguintes condições forem satisfeitas:

- **I** e **I'** são interpretações para $\mathbf{P}(\Gamma)$;
- para toda constante **c** ocorrendo em $\mathbf{P}(\Gamma)$, $\sigma(\mathbf{c}) = \sigma'(\mathbf{c})$;
- para todo sinal funcional **f** ocorrendo em $\mathbf{P}(\Gamma)$ e para toda aridade **n** de **f**, $\sigma(\mathbf{f},\mathbf{n}) = \sigma'(\mathbf{f},\mathbf{n})$;
- para todo sinal predicativo **p** ocorrendo em $\mathbf{P}(\Gamma)$ e para toda aridade **n** de **p**, $\sigma(\mathbf{p},\mathbf{n}) = \sigma'(\mathbf{p},\mathbf{n})$;
- para toda variável x livre em $\mathbf{P}(\Gamma)$, $\mathbf{s}(x) = \mathbf{s'}(x)$.

2.21 Lema da Concordância de Interpretações: Se duas interpretações **I** e **I'** concordam em $\mathbf{P}(\Gamma)$, então **I** satisfaz $\mathbf{P}(\Gamma)$ se, e somente se, **I'** satisfaz $\mathbf{P}(\Gamma)$.

Uma conseqüência imediata do Teorema 2.10 e do Lema da Concordância de Interpretações é o teorema a seguir, o qual cita uma série de propriedades semânticas invariantes com respeito às mudanças de linguagem nos níveis proposicional, quantificacional e equacional da lógica clássica.

2.22 Teorema: Os conceitos de satisfatibilidade, insatisfatibilidade, validade, invalidade e de conseqüência semântica são invariantes com respeito às mudanças de linguagem, em qualquer um dos níveis proposicional, quantificacional e equacional da lógica clássica. Para exemplificar o que entendemos por isto, detalhamos abaixo essa idéia para o conceito de satisfatibilidade. As seguintes proposições são equivalentes:

- $\mathbf{P}(\Gamma)$ é satisfatível em alguma linguagem **L**;
- $\mathbf{P}(\Gamma)$ é satisfatível em toda linguagem **L**, tal que $\mathbf{P} \in \mathbf{L}$ ($\Gamma \subseteq \mathbf{L}$);
- $\mathbf{P}(\Gamma)$ é satisfatível.

3 – Regras e Teoremas da Lógica Clássica

Nesta seção definiremos uma relação alternativa, a relação de conseqüência sintática para a lógica clássica, baseada em processos mecânicos de manipulação simbólica, a qual se revela equivalente à relação de conseqüência semântica da mesma lógica. Além disso, enumeramos as regras e teoremas da lógica clássica que consideramos de maior importância.

Existem algumas alternativas que poderiam servir para definirmos esta relação de conseqüência: cálculos axiomáticos ou sistemas de Hilbert, como por exemplo em [Enderton 1972]; sistemas de dedução natural, como por exemplo em [Prawitz 1965]; e cálculos de seqüentes, como por exemplo em [Ebbinghaus & outros 1989]. Escolhemos aqui a terceira opção, devido à sua simplicidade.

3.1 Definição: Um *seqüente* é uma lista de fórmulas (de alguma linguagem de primeira ordem). Um *esquema de seqüentes* é uma coleção de seqüentes. Uma *regra de seqüentes* é uma coleção de **n**-tuplas de seqüentes, onde **n** é um número natural maior ou igual a dois. Um *postulado de seqüentes* é um esquema de seqüentes ou uma regra de seqüentes. Um *cálculo de seqüentes* é uma coleção de postulados de seqüentes possuindo pelo menos um esquema de seqüentes.

3.2 Definição: Seja **C** um cálculo de seqüentes. Um *axioma seqüencial em* **C** é um elemento de um esquema de **C**. Uma *aplicação de regra em* **C** é um elemento de uma regra de seqüentes de **C**. Se $\langle S_1,\ldots, S_n, S\rangle$ é uma aplicação de uma regra de seqüentes de **C**, notamos usualmente $\langle S_1,\ldots, S_n, S\rangle$ por $\frac{S_1,\ldots, S_n}{S}$; dizemos neste caso que $S_1,\ldots, S_n$ são as *hipóteses* da aplicação e que **S** é a *conclusão* da aplicação. Uma *demonstração em* **C** é uma lista de seqüentes, tal que cada elemento desta lista é um axioma seqüencial em **C** ou é uma conclusão de uma aplicação de uma regra de seqüentes de **C**, tal que todas as hipóteses dessa aplicação precedem esta conclusão na lista.

3.3 Definição: Dizemos que **P** é *conseqüência sintática* de Γ em **C**, e notamos isto por $\Gamma \vdash_{C} \mathbf{P}$, se ($\Gamma$ **P**) é o último elemento de uma demonstração em **C**.

3.4 Definição: A seguir damos os postulados que formam os cálculos de seqüentes para a lógica clássica nos níveis proposicional, quantificacional e equacional.

Os axiomas e regras de seqüentes primitivos para o cálculo de seqüentes proposicional clássico são os seguintes:

- Esquema da Reflexividade: Se $\mathbf{P} \in \Gamma$, então "Γ **P**" é um axioma seqüencial da reflexividade;

- Regra da Transitividade: "$\dfrac{\Gamma\ \mathbf{P_1}, \ldots, \Gamma\ \mathbf{P_n},\ \{\mathbf{P_1},\ldots,\mathbf{P_n}\}\ \mathbf{Q}}{\Gamma\ \mathbf{Q}}$" é uma aplicação da regra da transitividade;

- Regra da Monotonicidade: Se $\Gamma \subseteq \Gamma'$, então "$\dfrac{\Gamma\ \ \mathbf{P}}{\Gamma'\ \mathbf{Q}}$" é uma aplicação da regra da monotonicidade;

- Regra da Dedução: "$\dfrac{\Gamma\ \ \mathbf{P}\ \ \mathbf{Q}}{\Gamma\ \ \mathbf{P} \rightarrow \mathbf{Q}}$" é uma aplicação da regra da dedução;

- Esquema Modus Ponens: "**P** **P** $\rightarrow$**Q** **Q**" é um axioma seqüencial do esquema modus ponens;

- Esquema do $\wedge$-introdução: "**P** **Q** **P** $\wedge$ **Q**" é um axioma seqüencial do esquema do $\wedge$-introdução;

- Esquemas do $\wedge$-eliminação:
  - "**P** $\wedge$ **Q** **P**" é um axioma seqüencial do primeiro esquema do $\wedge$-eliminação;
  - "**P** $\wedge$ **Q** **Q**" é um axioma seqüencial do segundo esquema do $\wedge$-eliminação;

- Esquemas do $\vee$-introdução:
  - **"P P $\vee$ Q"** é um axioma seqüencial do primeiro esquema do $\vee$-introdução,
  - **"Q P $\vee$ Q"** é um axioma seqüencial do segundo esquema do $\vee$-introdução;

- Regra da Prova por Casos: "$\dfrac{\Gamma\ P \vee Q,\ \Gamma\ P\ R,\ \Gamma\ Q\ R}{\Gamma\ R}$" é uma aplicação da regra da prova por casos;

- Regras da Redução ao Absurdo:
  - "$\dfrac{\Gamma\ P\ Q,\ \ \Gamma\ P \neg Q}{\Gamma \neg P}$" é uma aplicação da primeira regra da redução ao absurdo;
  - "$\dfrac{\Gamma \neg P\ Q,\ \ \Gamma \neg P \neg Q}{\Gamma\ P}$" é uma aplicação da segunda regra da redução ao absurdo;

Os axiomas e regras de seqüentes primitivos para o cálculo de seqüentes quantificacional clássico são todos os que foram dados acima para o cálculo de seqüentes proposicional, mais os seguintes:

- Regra da Generalização: Se x não é livre em Γ, então "$\dfrac{\Gamma\ P}{\Gamma\ \forall x P}$" é uma aplicação da regra da generalização;

- Esquema do $\forall$-eliminação: "$\forall x$ **P** **P**(x/**t**)" é um axioma seqüencial do esquema do $\forall$-eliminação;

- Esquema do $\exists$-introdução: "**P**(x/**t**) $\exists x$ **P**" é um axioma seqüencial do esquema do $\exists$-introdução;

- Regra do $\exists$-eliminação: Se y não é livre em $\Gamma \cup \{\mathbf{Q}\}$, então "$\dfrac{\Gamma\ \exists x \mathbf{P},\ \ \Gamma\ \mathbf{P}(x/y)\ \mathbf{Q}}{\Gamma\ \mathbf{Q}}$" é uma aplicação da regra do $\exists$-eliminação.

Os axiomas e regras de seqüentes primitivos para o cálculo de seqüentes equacional clássico são todos os que foram dados acima, mais os seguintes:

- Esquema da Reflexividade da Igualdade: "$\mathbf{t = t}$" é um axioma seqüencial do esquema da reflexividade da igualdade.

- Esquema da Substituição em Sinais Funcionais:
  "$\mathbf{t_1 = t_1' \ldots \ t_n = t_n' \ \ f(t_1,\ldots, t_n) = f(t_1',\ldots, t_n')}$" é uma axioma seqüencial do esquema da substituição em sinais funcionais.

- Esquema da Substituição em Sinais Predicativos:
  "$\mathbf{t_1 = t_1' \ldots \ t_n = t_n' \ \ p(t_1,\ldots, t_n) \rightarrow p(t_1',\ldots, t_n')}$" é uma axioma seqüencial do esquema da substituição em sinais predicativos.

3.5 Notação: Usamos as seguintes siglas:

- **CPC** = cálculo de seqüentes proposicional clássico;
- **CQC** = cálculo de seqüentes quantificacional clássico;
- **CEC** = cálculo de seqüentes equacional clássico;

3.6 Teorema: Se $\Gamma \cup \{\mathbf{P}\}$ é uma coleção de fórmulas não contendo quantificadores, então:

- $\Gamma \vdash_{\mathbf{CPC}} \mathbf{P}$ sss $\Gamma \vdash_{\mathbf{CQC}} \mathbf{P}$;
- $\Gamma \vdash_{\mathbf{CPC}} \mathbf{P}$ sss $\Gamma \vdash_{\mathbf{CEC}} \mathbf{P}$.

3.7 Teorema: Se $\Gamma \cup \{\mathbf{P}\}$ é uma coleção de fórmulas não contendo o sinal de "=", então:

- $\Gamma \vdash_{\mathbf{CQC}} \mathbf{P}$ sss $\Gamma \vdash_{\mathbf{CEC}} \mathbf{P}$.

Os dois teoremas anteriores dizem que os cálculos quantificacional equacional clássicos e são extensões conservativas do cálculo proposicional clássico, e que o

cálculo equacional clássico é uma extensão conservativa do cálculo quantificacional clássico.

Devido a este fato e pelos Teoremas 2.17 e 2.18, exporemos, de agora em diante, fatos adicionais sobre a lógica clássica sem nos preocuparmos em geral se estamos em um dos níveis proposicional, quantificacional ou equacional.

3.8 Convenção: Como estamos trabalhando exclusivamente com a lógica clássica, de agora em diante, para dizer que **P** é conseqüência sintática de Γ no cálculo de seqüentes para a lógica clássica, notaremos isto por $\Gamma \vdash \mathbf{P}$.

A seguir damos uma lista das principais regras e esquemas do cálculo de seqüentes para a lógica clássica.

3.9 Esquemas e Regras Estruturais:

- Reflexividade:

  Se $\mathbf{P} \in \Gamma$, então $\Gamma \vdash \mathbf{P}$.

- Transitividade:

  Se $\begin{cases} \Gamma \vdash \mathbf{P}_1 \\ \dots \\ \Gamma \vdash \mathbf{P}_n \end{cases}$ e $\mathbf{P}_1, .., \mathbf{P}_n \vdash \mathbf{Q}$, então $\Gamma \vdash \mathbf{Q}$.

- Monotonicidade:

  Se $\Gamma \vdash \mathbf{P}$ e $\Gamma \subseteq \Gamma'$, então $\Gamma' \vdash \mathbf{P}$.

- Compacidade:

  Se $\Gamma \vdash \mathbf{P}$, então existe Γ' finito tal que $\Gamma' \subseteq \Gamma$ e $\Gamma' \vdash \mathbf{P}$.

3.10 Esquemas e Regras de Introdução e Eliminação de Conetivos:

- Regra da Dedução:

  Se $\Gamma, \mathbf{P} \vdash \mathbf{Q}$, então $\Gamma \vdash \mathbf{P} \rightarrow \mathbf{Q}$.

- Modus Ponens:

  $\mathbf{P}, \mathbf{P} \rightarrow \mathbf{Q} \vdash \mathbf{Q}$

- $\wedge$-introdução:

  $\mathbf{P}, \mathbf{Q} \vdash \mathbf{P} \wedge \mathbf{Q}$

- $\wedge$-eliminação:

  i) $\mathbf{P} \wedge \mathbf{Q} \vdash \mathbf{P}$

  ii) $\mathbf{P} \wedge \mathbf{Q} \vdash \mathbf{Q}$

- $\vee$-introdução:

  i) $\mathbf{P} \vdash \mathbf{P} \vee \mathbf{Q}$

  ii) $\mathbf{Q} \vdash \mathbf{P} \vee \mathbf{Q}$

- Esquema da Prova por Casos:

  $\mathbf{P} \vee \mathbf{Q}, \mathbf{P} \rightarrow \mathbf{R}, \mathbf{Q} \rightarrow \mathbf{R} \vdash \mathbf{R}$

- Regras da Redução ao Absurdo:

  i) Se $\Gamma, \mathbf{P} \vdash \mathbf{Q}$ e $\Gamma, \mathbf{P} \vdash \neg\mathbf{Q}$, então $\Gamma \vdash \neg\mathbf{P}$.

  ii) Se $\Gamma, \neg\mathbf{P} \vdash \mathbf{Q}$ e $\Gamma, \neg\mathbf{P} \vdash \neg\mathbf{Q}$, então $\Gamma \vdash \mathbf{P}$.

3.11 Esquemas e Regras Complementares para Conetivos:

- Princípio da Reflexividade:

  $\vdash \mathbf{P} \rightarrow \mathbf{P}$

- Princípio da Não Contradição:

  i) $\mathbf{P}, \neg\mathbf{P} \vdash \mathbf{Q}$

  ii) $\vdash \neg(\mathbf{P} \wedge \neg\mathbf{P})$

- Princípio do Terceiro Excluído:

  $\vdash \mathbf{P} \vee \neg\mathbf{P}$

- Dupla Negação:

  i) $\mathbf{P} \vdash \neg\neg\mathbf{P}$

  ii) $\neg\neg\mathbf{P} \vdash \mathbf{P}$

- Modus Tollens:

  $\mathbf{P} \rightarrow \mathbf{Q}, \neg\mathbf{Q} \vdash \neg\mathbf{P}$

- Contraposição:

  i) $\mathbf{P} \rightarrow \mathbf{Q} \vdash \neg\mathbf{Q} \rightarrow \neg\mathbf{P}$

  ii) $\neg\mathbf{Q} \rightarrow \neg\mathbf{P} \vdash \mathbf{P} \rightarrow \mathbf{Q}$

- Leis de De Morgan:

  i) $\neg (\mathbf{P} \vee \mathbf{Q}) \vdash \neg\mathbf{P} \wedge \neg\mathbf{Q}$

  ii) $\neg\mathbf{P} \wedge \neg\mathbf{Q} \vdash \neg (\mathbf{P} \vee \mathbf{Q})$

  iii) $\neg(\mathbf{P} \wedge \mathbf{Q}) \vdash \neg\mathbf{P} \vee \neg\mathbf{Q}$

  iv) $\neg\mathbf{P} \vee \neg\mathbf{Q} \vdash \neg (\mathbf{P} \wedge \mathbf{Q})$

- Conseqüente da Implicação:

  $\mathbf{Q} \vdash \mathbf{P} \rightarrow \mathbf{Q}$

- Antecedente da Implicação:

  $\neg\mathbf{P} \vdash \mathbf{P} \rightarrow \mathbf{Q}$

- Implicação Material:
  - i) $\mathbf{P \rightarrow Q \vdash \neg P \vee Q}$
  - ii) $\mathbf{\neg P \vee Q \vdash P \rightarrow Q}$

- Silogismo Disjuntivo:
  - i) $\mathbf{\neg P, P \vee Q \vdash Q}$
  - ii) $\mathbf{\neg Q, P \vee Q \vdash P}$

- Redução da Disjunção:
  - i) $\mathbf{P \vee Q \vdash \neg P \rightarrow Q}$
  - ii) $\mathbf{\neg P \rightarrow Q \vdash P \vee Q}$
  - iii) $\mathbf{P \vee Q \vdash \neg Q \rightarrow P}$
  - iv) $\mathbf{\neg Q \rightarrow P \vdash P \vee Q}$

- Negação da Implicação:
  - i) $\mathbf{\neg (P \rightarrow Q) \vdash P \wedge \neg Q}$
  - ii) $\mathbf{P \wedge \neg Q \vdash \neg (P \rightarrow Q)}$

- Negação da Conjunção:
  - i) $\mathbf{\neg (P \wedge Q) \vdash P \rightarrow \neg Q}$
  - ii) $\mathbf{P \rightarrow \neg Q \vdash \neg (P \wedge Q)}$
  - iii) $\mathbf{\neg (P \wedge Q) \vdash Q \rightarrow \neg P}$
  - iv) $\mathbf{Q \rightarrow \neg P \vdash \neg (P \wedge Q)}$

- Silogismo Hipotético:

  $\mathbf{P \rightarrow Q, Q \rightarrow R \vdash P \rightarrow R}$

3.12 Lema da Substituição para Conetivos:

i) $\mathbf{P_1, P_1 \leftrightarrow P_2 \vdash P_2}$

ii) $\mathbf{\neg P_1, P_1 \leftrightarrow P_2 \vdash \neg P_2}$

iii) $\mathbf{P_1 \rightarrow Q, P_1 \leftrightarrow P_2 \vdash P_2 \rightarrow Q}$

iv) $\mathbf{P \rightarrow Q_1, Q_1 \leftrightarrow Q_2 \vdash P \rightarrow Q_2}$

v) $\mathbf{P_1 \wedge Q, P_1 \leftrightarrow P_2 \vdash P_2 \wedge Q}$

vi) $\mathbf{P \wedge Q_1, Q_1 \leftrightarrow Q_2 \vdash P \wedge Q_2}$

vii) $\mathbf{P_1 \vee Q, P_1 \leftrightarrow P_2 \vdash P_2 \vee Q}$

viii) $\mathbf{P \vee Q_1, Q_1 \leftrightarrow Q_2 \vdash P \vee Q_2}$

3.13 Escólio: Os esquemas e regras da Dupla Negação, Contraposição, De Morgan, Implicação Material, Redução da Disjunção, Negação da Implicação e Negação da Conjunção podem ser reescritos como equivalências.

3.14 Esquemas Complementares para Conetivos:

- Idempotência:

  i) $\mathbf{\vdash (P \wedge P) \leftrightarrow P}$

  ii) $\mathbf{\vdash (P \vee P) \leftrightarrow P}$

- Lei dos Membros da Equivalência:

  i) $\mathbf{P, Q \vdash P \leftrightarrow Q}$

  ii) $\mathbf{\neg P, \neg Q \vdash P \leftrightarrow Q}$

  iii) $\mathbf{P, \neg Q \vdash \neg(P \leftrightarrow Q)}$

  iv) $\mathbf{\neg P, Q \vdash \neg(P \leftrightarrow Q)}$

- Transitividade da Equivalência:

  $\mathbf{P \leftrightarrow Q, Q \leftrightarrow R \vdash P \leftrightarrow Q}$

- Equivalência Material:

  i) $\vdash (P \leftrightarrow Q) \leftrightarrow (P \wedge Q) \vee (\neg P \wedge \neg Q)$

  ii) $\vdash \neg (P \leftrightarrow Q) \leftrightarrow (P \wedge \neg Q) \vee (\neg P \wedge Q)$

- Negação da Equivalência:

  i) $\vdash \neg (P \leftrightarrow Q) \leftrightarrow (\neg P \leftrightarrow Q)$

  ii) $\vdash \neg (P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow \neg Q)$

- Comutatividade:

  i) $\vdash P \wedge Q \leftrightarrow Q \wedge P$

  ii) $\vdash P \vee Q \leftrightarrow Q \vee P$

  iii) $\vdash (P \leftrightarrow Q) \leftrightarrow (Q \leftrightarrow P)$

- Associatividade:

  i) $\vdash P \wedge (Q \wedge R) \leftrightarrow (P \wedge Q) \wedge R$

  ii) $\vdash P \vee (Q \vee R) \leftrightarrow (P \vee Q) \vee R$

  iii) $\vdash (P \leftrightarrow (Q \leftrightarrow R)) \leftrightarrow ((P \leftrightarrow Q) \leftrightarrow R)$

- Distributividade:

  i) $\vdash P \wedge (Q \vee R) \leftrightarrow (P \wedge Q) \vee (P \wedge R)$

  ii) $\vdash (P \wedge Q) \vee R \leftrightarrow (P \vee R) \wedge (Q \vee R)$

  iii) $\vdash P \vee (Q \wedge R) \leftrightarrow (P \vee Q) \wedge (P \vee R)$

  iv) $\vdash (P \vee Q) \wedge R \leftrightarrow (P \wedge R) \vee (Q \wedge R)$

- Importação/Exportação:

  $\vdash P \rightarrow (Q \rightarrow R) \leftrightarrow (P \wedge Q) \rightarrow R$

3.15 Esquemas e Regras de Introdução e Eliminação para Quantificadores:

- Generalização:

$$\text{Se } \begin{cases} \Gamma \vdash \mathbf{P} \\ x \text{ não é livre em } \Gamma \end{cases}, \text{ então } \Gamma \vdash \forall x\, \mathbf{P}.$$

- $\forall$-eliminação:

$$\forall x\, \mathbf{P} \vdash \mathbf{P}(x/t).$$

- $\exists$-introdução:

$$\mathbf{P}(x/t) \vdash \exists x\, \mathbf{P}.$$

- $\exists$-eliminação:

$$\text{Se } \begin{cases} \Gamma, \exists x \mathbf{P}, \mathbf{P}(x/y) \vdash \mathbf{Q} \\ y \text{ não é livre em } \Gamma \cup \{\mathbf{Q}\} \end{cases}, \text{ então } \Gamma, \exists x\, \mathbf{P} \vdash \mathbf{Q}.$$

3.16 Esquemas e Regras Complementares para Quantificadores:

- Negação de Fórmula Universal:

$$\vdash \neg \forall x\, \mathbf{P} \leftrightarrow \exists x\, \neg \mathbf{P}$$

- Negação de Fórmula Existencial:

$$\vdash \neg \exists x\, \mathbf{P} \leftrightarrow \forall x\, \neg \mathbf{P}$$

- Instanciação:

$$\text{Se } \begin{cases} \Gamma \vdash \mathbf{P} \\ x \text{ não é livre em } \Gamma \end{cases}, \text{ então } \Gamma \vdash \mathbf{P}(x/t).$$

- Lema da Substituição para Quantificadores:

  i) Se $\begin{cases} \Gamma \vdash \forall x \mathbf{P} \\ \Gamma \vdash \mathbf{P} \leftrightarrow \mathbf{Q} \\ x \text{ não é livre em } \Gamma \end{cases}$, então $\Gamma \vdash \forall x\, \mathbf{Q}$;

  ii) Se $\begin{cases} \Gamma \vdash \exists x \mathbf{P} \\ \Gamma \vdash \mathbf{P} \leftrightarrow \mathbf{Q} \\ x \text{ não é livre em } \Gamma \end{cases}$, então $\Gamma \vdash \exists x\, \mathbf{Q}$.

- Princípio da Substituição:

  Se $\begin{cases} \Gamma \vdash \mathbf{Q}(\mathbf{R}/\mathbf{P_1}) \\ \Gamma \vdash \mathbf{P_1} \leftrightarrow \mathbf{P_2} \\ \mathbf{R} \text{ não está no escopo de nenhuma variável livre em } \Gamma \cap \{\mathbf{P_1}, \mathbf{P_2}\} \end{cases}$,

  então $\Gamma \vdash \mathbf{Q}(\mathbf{R}/\mathbf{P_2})$.

- Vacuidade:

  Se x não é livre em $\mathbf{P}$, então $\begin{cases} \vdash \mathbf{P} \leftrightarrow \forall x \mathbf{P} \\ \vdash \mathbf{P} \leftrightarrow \exists x \mathbf{P} \end{cases}$

- $\exists$-importação:

  $\vdash \exists x\, \forall y\, \mathbf{P} \rightarrow \forall y\, \exists x\, \mathbf{P}$

- Relações entre Quantificadores e Conetivos:

  i) $\vdash \forall x\, (\mathbf{P} \rightarrow \mathbf{Q}) \rightarrow (\forall x\, \mathbf{P} \rightarrow \forall x\, \mathbf{Q})$

  ii) $\vdash \forall x\, (\mathbf{P} \wedge \mathbf{Q}) \leftrightarrow \forall x\, \mathbf{P} \wedge \forall x\, \mathbf{Q}$

  iii) $\vdash \forall x\, \mathbf{P} \vee \forall x\, \mathbf{Q} \rightarrow \forall x\, (\mathbf{P} \vee \mathbf{Q})$

  iv) $\vdash (\exists x\, \mathbf{P} \rightarrow \exists x\, \mathbf{Q}) \rightarrow \exists x\, (\mathbf{P} \rightarrow \mathbf{Q})$

  v) $\vdash \exists x\, (\mathbf{P} \wedge \mathbf{Q}) \leftrightarrow \exists x\, \mathbf{P} \wedge \exists x\, \mathbf{Q}$

  vi) $\vdash \exists x\, (\mathbf{P} \vee \mathbf{Q}) \leftrightarrow \exists x\, \mathbf{P} \vee \exists x\, \mathbf{Q}$

  vii) $\vdash \forall x\, (\mathbf{P} \rightarrow \mathbf{Q}) \rightarrow (\exists x\, \mathbf{P} \rightarrow \exists x\, \mathbf{Q})$

  viii) $\vdash \forall x\, (\mathbf{P} \leftrightarrow \mathbf{Q}) \rightarrow (\forall x\, \mathbf{P} \leftrightarrow \forall x\, \mathbf{Q})$

  ix) $\vdash \forall x\, (\mathbf{P} \leftrightarrow \mathbf{Q}) \rightarrow (\exists x\, \mathbf{P} \leftrightarrow \exists x\, \mathbf{Q})$

- Comutatividade:

  i) $\vdash \forall x\, \forall y\, \mathbf{P} \leftrightarrow \forall y\, \forall x\, \mathbf{P}$

  ii) $\vdash \exists x\, \exists y\, \mathbf{P} \leftrightarrow \exists y\, \exists x\, \mathbf{P}$

- Formas Congruentes:

  i) Se y não é livre em **P**, então $\begin{cases} \vdash \forall x\, \mathbf{P} \leftrightarrow \forall y\, \mathbf{P}(x/y) \\ \vdash \exists x\, \mathbf{P} \leftrightarrow \exists y\, \mathbf{P}(x/y) \end{cases}$

  ii) Se **P** e **P'** são fórmulas congruentes, então $\vdash \mathbf{P} \leftrightarrow \mathbf{P'}$

- Transporte de Quantificadores:

  i) Se x não é livre em **P**, então

  $$\begin{cases} \text{i)} \vdash \forall x\, (\mathbf{P} \rightarrow \mathbf{Q}) \leftrightarrow \mathbf{P} \rightarrow \forall x\, \mathbf{Q} \\ \text{ii)} \vdash \exists x\, (\mathbf{P} \rightarrow \mathbf{Q}) \leftrightarrow \mathbf{P} \rightarrow \exists x\, \mathbf{Q} \\ \text{iii)} \vdash \forall x\, (\mathbf{P} \wedge \mathbf{Q}) \leftrightarrow \mathbf{P} \wedge \forall x\, \mathbf{Q} \\ \text{iv)} \vdash \exists x\, (\mathbf{P} \wedge \mathbf{Q}) \leftrightarrow \mathbf{P} \wedge \exists x\, \mathbf{Q} \\ \text{v)} \vdash \forall x\, (\mathbf{P} \vee \mathbf{Q}) \leftrightarrow \mathbf{P} \vee \forall x\, \mathbf{Q} \\ \text{vi)} \vdash \exists x\, (\mathbf{P} \vee \mathbf{Q}) \leftrightarrow \mathbf{P} \vee \exists x\, \mathbf{Q} \end{cases}$$

  ii) Se x não é livre em **Q**, então

  $$\begin{cases} \text{i)} \vdash \forall x\, (\mathbf{P} \rightarrow \mathbf{Q}) \leftrightarrow \exists x\, \mathbf{P} \rightarrow \mathbf{Q} \\ \text{ii)} \vdash \exists x\, (\mathbf{P} \rightarrow \mathbf{Q}) \leftrightarrow \forall x\, \mathbf{P} \rightarrow \mathbf{Q} \\ \text{iii)} \vdash \forall x\, (\mathbf{P} \wedge \mathbf{Q}) \leftrightarrow \forall x\, \mathbf{P} \wedge \mathbf{Q} \\ \text{iv)} \vdash \exists x\, (\mathbf{P} \wedge \mathbf{Q}) \leftrightarrow \exists x\, \mathbf{P} \wedge \mathbf{Q} \\ \text{v)} \vdash \forall x\, (\mathbf{P} \vee \mathbf{Q}) \leftrightarrow \forall x\, \mathbf{P} \vee \mathbf{Q} \\ \text{vi)} \vdash \exists x\, (\mathbf{P} \vee \mathbf{Q}) \leftrightarrow \exists x\, \mathbf{P} \vee \mathbf{Q} \end{cases}$$

3.17 Regras Básicas da Igualdade:

- Reflexividade da Igualdade:

  $\vdash \mathbf{t} = \mathbf{t}$

- Substituição em Sinais Funcionais:

  $t_1 = t_1', \ldots, t_n = t_n' \vdash f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')$

- Substituição em Sinais Predicativos:

  $t_1 = t_1', \ldots, t_n = t_n' \vdash p(t_1, \ldots, t_n) \leftrightarrow p(t_1', \ldots, t_n')$

3.18 Esquemas e Regras Complementares da Igualdade:

- Simetria da Igualdade:

  $t = t' \vdash t' = t$

- Transitividade da Igualdade:

  $t = u, u = v \vdash t = v.$

- Princípio da Substituição para Igualdade em termos:

  Se $\Gamma \vdash t_1 = t_2$, então $\Gamma \vdash u(x/t_1) = u(x/t_2)$.

- Princípio da Substituição para Igualdade em fórmulas:

  Se $\begin{cases} \Gamma \vdash t_1 = t_2 \\ x \text{ não está no escopo de nenhuma variável livre em } \Gamma \cap \{t_1, t_2\}' \end{cases}$

  então $\Gamma \vdash Q(x/t_1) \leftrightarrow Q(x/t_2)$.

O conceito sintático correspondente ao conceito semântico de insatisfatibilidade é o conceito de *trivialidade*, dado na definição a seguir.

3.19 Definição: Dizemos que Γ é *trivial* se, para toda fórmula P, $\Gamma \vdash P$.

3.20 Definição: Dizemos que Γ é *consistente* se não existe **P** tal que $\Gamma \vdash \mathbf{P}$ e $\Gamma \vdash \neg\mathbf{P}$. Caso contrário dizemos que Γ é *inconsistente*.

O resultado abaixo fornece uma condição necessária e suficiente, com respeito à lógica clássica, para que uma coleção de fórmulas seja trivial.

3.21 Teorema: Γ é trivial se, e somente se, Γ é inconsistente[1].

Finalmente, encerramos este capítulo dando alguns resultados que relacionam conceitos semânticos e conceitos sintáticos.

3.22 Teorema: Γ é insatisfatível se, e somente se, Γ é trivial.

3.23 Teorema: Γ é satisfatível se, e somente se, Γ é não trivial.

3.24 Teorema da Correção e da Completude (do cálculo clássico com respeito à semântica clássica): $\Gamma \vdash \mathbf{P}$ se, e somente se, $\Gamma \models \mathbf{P}$.

[1] Esta propriedade não é compartilhada pelas lógicas ditas paraconsistentes, apresentadas, por exemplo, em [Buchsbaum 1988], [Buchsbaum & Pequeno 1991] e [Buchsbaum 1995].

3 – O MÉTODO DOS TABLEAUX

1 – Introdução

O sistema de tableaux, cujos precursores foram Beth e, independentemente, Hintikka [Reeves 1983], é um método de prova por refutação, no qual prova-se um teorema pelo insucesso na tentativa de construção sistemática de um modelo para a sua negação. Originalmente, o método verifica a impossibilidade da satisfação da negação de uma determinada fórmula.

O método dos tableaux consiste na geração de uma árvore de fórmulas (tableau), a partir de um tableau inicial, o qual é constituído de um único ramo no qual são listadas todas as premissas da base de conhecimento considerada e a negação do possível teorema que se deseja provar.

A partir deste tableau inicial, através de aplicações de regras de expansão, uma árvore é progressivamente expandida. Tal expansão é feita escolhendo-se um nó ainda não usado, para o qual existe uma regra de expansão correspondente, dos quais obtemos subárvores.

A proliferação de um ramo pára quando este satisfaz a condição de fechamento do sistema de tableaux considerado. Tal ramo será considerado fechado, não recebendo novas folhas.

O objetivo do procedimento é fechar todos os ramos, provando, desta forma, que a fórmula inicialmente negada é conseqüência lógica da base de conhecimento considerada. O fato de todos os ramos do tableau fecharem significa, semanticamente, a insatisfatibilidade da base de conhecimento, juntamente com a negação da fórmula considerada.

2 – Conceitos Gerais

Para a definição formal de um sistema de tableaux precisa-se, inicialmente, dos seguintes elementos: uma linguagem inicial, uma linguagem de trabalho, uma função de inicialização, responsável pela criação do tableau inicial, um conjunto de regras de expansão, responsável pela proliferação dos nós e um critério de fechamento.

2.1 Definição: Um *tableau em uma linguagem* **L** é uma árvore finita de nós cujo conteúdo é, no mínimo, uma fórmula de **L** e uma marca, a qual, por convenção, pertence ao conjunto {**0**, **1**}. Se **N** é um nó, denotamos a fórmula de **N** por **fórmula(N)**. Dizemos que um nó está marcado se sua marca é **1**. Dizemos que um nó está em **L** se sua fórmula é de **L**.

2.2 Definição: Cada nó de um tableau pertence a um único *nível*, o qual é rotulado por algum número natural. Cada nó de nível **n+1** é *filho* de um único nó de nível **n**. Existe somente um nó de nível **0**, que é chamado *nó raiz* ou *nó inicial* do tableau.Um nó é dito *folha* se este não possui mais sucessores. Se **p** é o nível de um nó do tableau e não existe outro nó de nível maior que **p**, então **p** é chamado *profundidade* do tableau.

2.3 Definição: Um *ramo de um tableau* é uma seqüência finita de nós $\mathbf{N_0}$... $\mathbf{N_k}$, tal que $\mathbf{N_0}$ é o nó inicial do tableau e, para $\mathbf{i} = 1,..., k$, $\mathbf{N_i}$ é um nó sucessor de $\mathbf{N_{i-1}}$, e $\mathbf{N_k}$ é um nó folha, isto é, o ramo termina em $\mathbf{N_k}$.

2.4 Definição: Sejam **L** uma linguagem formal, **L'** uma extensão de **L**, τ a coleção de todos os tableaux em **L'**, **PF**(τ) a coleção de todos o subconjuntos finitos de τ, θ a coleção de todos os ramos em **L'**, **I** uma função de **PF(L)** em τ, **F** uma função de θ em {**aberto**, **fechado**}, e finalmente **R** uma coleção de funções parciais de **L'** x θ em **PF**(τ), tal que o domínio de cada uma dessas funções parciais é da forma **L''** x θ, onde **L''** é um subconjunto de **L'**, podendo variar para cada função parcial. Um *sistema de tableaux* é uma quíntupla ordenada **S** = < **L, L', I, F, R**>, onde **L** é dita a *linguagem inicial* de **S**, **L'** é dita a *linguagem de trabalho* de **S**, **I** é dita a *função de inicialização* de **S**, **F** é dito o *critério de fechamento* de **S** e **R** é dita a *coleção de regras* de **S**.

2.5 Definição: Se r é uma regra de S e o seu domínio como função parcial é **L''** x θ, dizemos então que **L''** é o seu domínio como regra.

2.6 Definição: Sejam **S** = < **L, L', I, F, R**> um sistema de tableaux, **P** uma fórmula em **L'**, **N** um nó em **L'** e **r** uma regra de **S**. A *regra* **r** é dita *ser aplicável à fórmula* **P** se **P** pertence ao domínio da regra **r**. A *regra* **r** *é* dita *ser aplicável ao nó* **N** se **r** é aplicável a **fórmula(N)**.

2.7 Definição: Sejam **S** = < **L, L', I, F, R**> um sistema de tableaux, **P** uma fórmula em **L'** e **r** uma regra de **S**. **P** é dita uma *fórmula excluída* de **S** caso não exista regra **r** de **S** que seja aplicável a **P**.

2.8 Definição: Sejam **S** = < **L, L', I, F, R**> um sistema de tableaux e Γ uma coleção de fórmulas em **L**. **I**(Γ) é dito ser o *tableau inicial* para Γ em **S**.

2.9 Definição: Seja **S** = < **L, L', I, F, R**> um sistema de tableaux. Se ρ é um ramo em **L'** e **F**(ρ) = **fechado**, ρ é dito *fechado em* **S**; caso contrário, ρ é dito *aberto em* **S**.

2.10 Definição: Seja $\mathbf{S} = \langle \mathbf{L}, \mathbf{L'}, \mathbf{I}, \mathbf{F}, \mathbf{R} \rangle$ um sistema de tableaux. Um ramo em **L'** é dito *exaurido em* **S** caso todos os seus nós que possuem fórmulas não excluídas estejam marcados.

2.11 Definição: Sejam $\mathbf{S} = \langle \mathbf{L}, \mathbf{L'}, \mathbf{I}, \mathbf{F}, \mathbf{R} \rangle$ um sistema de tableaux e **T,T'** dois tableaux em **L'**. **T'** é dito uma *extensão imediata* de **T** em **S** se **T** é não exaurido e existe um nó **N** em **T** e uma regra **r** em **S**, tal que **r** seja aplicável a **N**, e **T'** pode ser obtido de **T** marcando-se **N** e acrescentando-se **r** (fórmula(**N**), ρ) em cada ramo aberto ρ de **T** onde **N** figure.

2.12 Definição: Sejam **S** um sistema de tableaux e $(\mathbf{T}_i)_{i \in \mathbf{I}}$ uma seqüência de tableaux de **S**, onde **I** é $\mathbb{N}$ ou **i** é da forma $\{1, \ldots, n\}$, onde $\mathbf{n} \in \mathbb{N}$, de modo que para cada $\mathbf{i} \in \mathbf{I}$, se $\mathbf{i} > 0$, então $\mathbf{T}_i$ é uma extensão imediata de $\mathbf{T}_{i-1}$ em **S**. Dizemos então que $(\mathbf{T}_i)_{i \in \mathbf{I}}$ é dita *uma seqüência de desenvolvimento de tableaux em* **S**. Se $(\mathbf{T}_i)_{i \in \mathbf{I}}$ é uma seqüência de desenvolvimento de tableaux em **S**, tal que $\mathbf{T}_0$ é o tableaux inicial para Γ, então $(\mathbf{T}_i)_{i \in \mathbf{I}}$ é dita ser *uma seqüência de desenvolvimento de tableaux em* **S** *para* Γ.

2.13 Definição: Dizemos que uma *seqüência* **T** *de desenvolvimento de tableaux* em **S** é *completa* se as seguintes condições forem satisfeitas:

- se **I** é finito, então $(\mathbf{T}_i)_{i \in \mathbf{I}}$ termina com uma confutação ou com um tableau possuindo um ramo exaurido aberto;
- se **I** é infinito, então $(\mathbf{T}_i)_{i \in \mathbf{I}}$ tende para uma árvore-limite possuindo um ramo infinito exaurido aberto.

2.14 Escólio: Dado um tableau $\mathbf{T}_0$ sempre é possível obter uma seqüência de desenvolvimento completa iniciando com $\mathbf{T}_0$ utilizando, por exemplo, algum dos procedimentos conhecidos de busca em largura ou de busca em profundidade[2].

[2] Os procedimentos de busca em largura e busca em profundidade são descritos, por exemplo, em [Horowitz & Sahni 1976].

2.15 Definição: Sejam S = < L, L', I, F, R> um sistema de tableaux e T,T' dois tableaux em **L'**. **T'** é dito *um desenvolvimento de* **T** *em* **S** caso exista uma seqüência de tableaux $T_0,\ldots, T_n$, tal que T_0 é T, e T_n é T'.

2.16 Definição: Sejam S = < L, L', I, F, R> um sistema de tableaux, **T** um tableau em **L'** e Γ uma coleção de fórmulas em **L'**. **T** é dito *um tableau para uma coleção de fórmulas* Γ *em* **S** se **T** é um desenvolvimento do tableau inicial para Γ.

2.17 Definição: Sejam **S** um sistema de tableaux e Γ uma coleção de fórmulas na linguagem inicial de **S**. Uma *confutação em* **S** é um tableau em **S** no qual todos os seus ramos estão fechados. Uma *confutação para* Γ *em* **S** é uma confutação em **S** que é um tableau para Γ em **S**.

2.18 Definição: Dizemos que *um nó* **N** *gera um nó* **N'** em um tableau **T**, *com respeito a um sistema de tableaux* **S**, se uma das seguintes condições for satisfeita:

- existe um tableau T_1, tal que **N** é um nó não marcado de T_1, e existe um tableau T_2, obtido de T_1 através da aplicação de uma das regras de **S** sobre **N** (em particular, T_2 é uma extensão imediata de T_1 em **S**), de modo que **N'** é um dos novos nós de T_2 obtidos por esta aplicação, e **T** é um desenvolvimento de T_2 em **S**;
- existe um nó **N''**, tal que **N** gera **N''** em **T** com respeito a **S**, e **N''** gera **N'** em **T** com respeito a **S**.

2.19 Definição: Dizemos que *uma fórmula* **P** *gera uma fórmula* **Q**, *em um tableaux* **T** *com respeito a um sistema* **S**, se existem nós **N** e **N'** tais que **P** é a fórmula de **N**, **Q** é a fórmula de **N'**, e **N** gera **N'** em **T**.

Conforme visto, o princípio da definição de um sistema de tableaux dá-se pela escolha de sua linguagem inicial e de trabalho. Para que se possa fazer uma descrição adequada de premissas e possíveis conclusões, define-se uma linguagem inicial do sistema de tableaux. Posteriormente, pode haver necessidade de novos sinais a serem adicionados a linguagem inicial ou até de uma linguagem para alguma outra

relacionada, de modo a permitir definições de certas regras de expansão, originando-se daí a linguagem de trabalho. Por exemplo, quando o sistema de tableaux trabalha com lógica proposicional não modal, então a linguagem de trabalho e linguagem inicial em geral coincidem porém, em lógica quantificacional clássica, a linguagem de trabalho é, em geral, igual a linguagem inicial acrescida de uma infinidade de novas constantes, devido ao tratamento que as regras devem dispensar aos quantificadores.

O conjunto de regras é elaborado a partir da definição adequada de cada uma das regras, as quais dependem do nó a que estão sendo aplicadas, bem como do ramo a que este nó pertence. Em lógica de primeira ordem não modal, por exemplo, o tratamento dado a fórmulas quantificadas universalmente e existencialmente depende do ramo em que estes nós figuram. Porém, para fórmulas não quantificadas o ramo a que o nó pertence não é relevante.

A função de fechamento varia com respeito à lógica a que o tableau se destina. Usando, mais uma vez, a lógica clássica como exemplo, a função de fechamento trata da ocorrência de contradições, isto é, um ramo é dito fechado caso ocorram duas fórmulas contraditórias.

A função de inicialização fornece, a partir de uma dada coleção de fórmulas, uma árvore que dá início ao processo de desenvolvimento ou expansão do tableau. Por exemplo, quando queremos verificar, na lógica clássica, se **P** é conseqüência de Γ, então o tableau inicial é um ramo contendo as fórmulas de Γ e a negação de **P**.

4 – O SISTEMA DE TABLEAUX TRADICIONAL PARA A LÓGICA QUANTIFICACIONAL CLÁSSICA

Neste capítulo definiremos o sistema de tableaux tradicional para a lógica quantificacional clássica tal como encontrado habitualmente na literatura existente, como, por exemplo, em [Revees 1983] e [Buchsbaum & Pequeno 1990]. De agora em diante chamaremos o sistema de tableaux que iremos progressivamente especificar de $\mathbf{S_0}$.

1 – Especificação do Sistema

A linguagem inicial de $\mathbf{S_0}$ é uma linguagem de primeira ordem fixa, que notamos, de agora em diante, por $\mathbf{L_0}$. A linguagem de trabalho de $\mathbf{S_0}$ é uma extensão de $\mathbf{L_0}$, a qual notamos por $\mathbf{L_1}$, obtidá acrescentando-se uma infinidade de novas constantes ao alfabeto de $\mathbf{L_0}$.

A função de inicialização de $\mathbf{S_0}$, a qual notamos por $\mathbf{I_0}$, associa a cada coleção Γ de fórmulas de $\mathbf{L_1}$ um tableau com um único ramo sem fórmulas marcadas, cujas fórmulas são obtidas de Γ substituindo todas as variáveis livres de Γ por novas constantes e retirando os quantificadores vácuos das fórmulas de Γ.

A coleção de regras de S_0, a qual notamos por $\mathbf{R_0}$, é composta das onze regras dadas pela Figura 1.

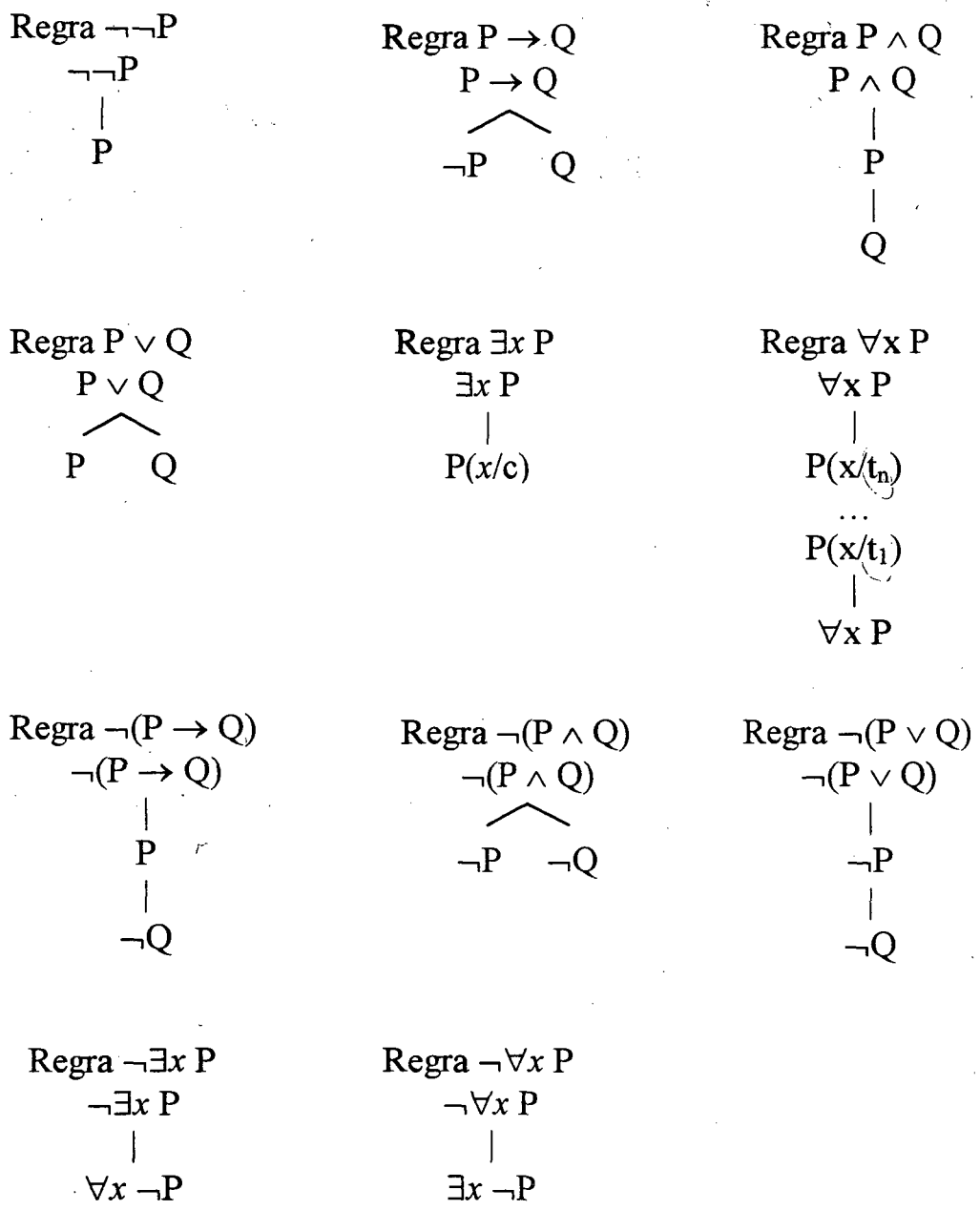

Figura 1: Regras de Expansão dos Ramos para o Sistema de Tableaux Tradicional.

O critério de fechamento de S_0 é uma função, a qual notamos por **F**, que associa a um dado ramo ρ em L_1 a palavra **fechado** se ρ possui duas fórmulas **P** e $\neg$**P'**, onde **P** é congruente a **P'**, e a palavra **aberto** caso contrário.

A constante **c**, apresentada na expressão $\mathbf{P}(x/\mathbf{c})$ escrita na definição da regra para fórmulas existenciais na Figura 1, é a primeira constante em L_1 que não figura no ramo considerado.

Os termos $t_1,\ldots, t_n$, apresentados nas expressões $\mathbf{P}(x/t_1)$, ..., $\mathbf{P}(x/t_n)$, escritos na definição da regra para fórmulas universais na Figura 1, são todos os termos fechados t_i ($i = 1, \ldots,0$) do ramo considerado, tais que $\forall x$ **P** ainda não tenha sido instanciado para t_i.

2 – Correção e Completude do Sistema S_0 com Respeito a Lógica Quantificacional Clássica

A idéia por trás dos procedimentos de prova automática por refutação, tais como os métodos da resolução e dos tableaux, é dada na proposição seguinte, a qual reduz o conceito de conseqüência lógica ao conceito de insatisfatibilidade.

2.1 Teorema: $\Gamma \models \mathbf{P}$ se, e somente se, $\Gamma \cup \{\neg\mathbf{P}\}$ é insatisfatível.

Um caso particular e importante do teorema acima dá-se quando a coleção de premissas é vazia.

2.2 Corolário: $\models \mathbf{P}$ se, e somente se, $\neg\mathbf{P}$ é insatisfatível.

Assim, o problema de mostrar que uma dada fórmula é conseqüência lógica de uma coleção de premissas reduz-se ao problema de mostrar que o conjunto de todas as premissas consideradas e da negação da fórmula considerada é insatisfatível.

Temos que o sistema $\mathbf{S_0}$ é correto e completo com respeito à lógica quantificacional clássica, conforme enunciamos a seguir.

2.3 Teorema: As seguintes proposições são equivalentes:

- **P** é insatisfatível;
- toda seqüência de desenvolvimento completa para **P** em $\mathbf{S_0}$ termina com uma confutação em **P**;
- existe uma seqüência de desenvolvimento completa para **P** em $\mathbf{S_0}$ que termina com uma confutação em **P**.

A partir dos Teoremas 2.1 e 2.3 obtemos de imediato o resultado abaixo.

2.4 Corolário: As seguintes proposições são equivalentes:

- $\Gamma \vdash \mathbf{P}$;
- toda seqüência de desenvolvimento completa para $\Gamma \cup \{\neg\mathbf{P}\}$ em $\mathbf{S_0}$ termina com uma confutação para $\Gamma \cup \{\neg\mathbf{P}\}$;
- existe uma seqüência de desenvolvimento completa para $\Gamma \cup \{\neg\mathbf{P}\}$ em $\mathbf{S_0}$ que termina com uma confutação para $\Gamma \cup \{\neg\mathbf{P}\}$.

2.5 Corolário: As seguintes proposições são equivalentes:

- $\Gamma \nvdash \mathbf{P}$;
- toda seqüência de desenvolvimento completa para $\Gamma \cup \{\neg\mathbf{P}\}$ em $\mathbf{S_0}$ termina com um tableau possuindo um ramo exaurido aberto ou tende para uma árvore-limite possuindo um ramo infinito exaurido aberto;
- existe uma seqüência de desenvolvimento completa para $\Gamma \cup \{\neg\mathbf{P}\}$ em $\mathbf{S_0}$ que termina com um tableau possuindo um ramo exaurido aberto ou tende para uma árvore-limite possuindo um ramo infinito exaurido aberto.

5 – REFINAMENTOS PARA O SISTEMA DE TABLEAUX TRADICIONAL

Nos capítulos anteriores descrevemos as idéias mais importantes relacionadas à lógica clássica e ao método dos tableaux. No capítulo 4 especificamos um sistema de tableaux para a lógica quantificacional clássica, o qual serve, neste trabalho, como suporte básico para três refinamentos, apresentando progressivas vantagens quanto ao desempenho em relação ao primeiro sistema.

Dizemos que uma proposição ou problema relacionando um certo número de dados é decidível se existe um algoritmo que, em um número finito de passos, fornece uma resposta positiva ou negativa, dependendo da veracidade ou falsidade dessa proposição. Um problema é semidecidível se existir pelo menos um algoritmo que dê uma resposta positiva se os dados fornecidos satisfizerem a proposição considerada.

O problema fundamental da automatização do raciocínio possui como dados uma coleção Γ finita de fórmulas e uma fórmula adicional **P**, e como proposição o enunciado "**P** é conseqüência lógica de Γ". Em [Church 1936], Alonzo Church mostrou que este problema é indecidível[3], porém existem algoritmos de prova automática, tais como os métodos da resolução e dos tableaux, que fornecem uma solução semidecidível

[3] Isto é, não existe nenhum algoritmo que sempre dê uma resposta positiva quando $\Gamma \vdash \mathbf{P}$, e dê uma resposta negativa quando $\Gamma \nvdash \mathbf{P}$.

para esta questão[4], e , além disso também fornecem respostas negativas para um grande número de casos particulares.

A automatização do raciocínio busca lidar com dois obstáculos básicos, os quais direcionam a evolução deste campo de pesquisa:

1º) todo algoritmo de automatização, no decorrer do seu processamento, provoca, em geral, um crescimento explosivo do espaço de busca, daí torna-se imperiosa a descoberta de certos refinamentos que diminuam ao máximo a proliferação de dados intermediários desnecessários;

2º) como o problema da prova automática de teoremas é indecidível, todo algoritmo de automatização está sujeito a entrar em um processamento perpétuo para certos dados que deveriam fornecer respostas negativas, daí são necessários também refinamentos que ampliem ao máximo os espaços para os quais obtemos soluções decidíveis.

Atendendo a estas duas demandas, especificamos, neste trabalho, três formas de refinamento para o método dos tableaux, visando assim diminuir ao máximo a proliferação de dados intermediários, bem como otimizar as possibilidades de obtenção de respostas negativas.

1 – Primeiro Refinamento

A primeira adaptação a ser feita no sistema de tableaux tradicional para a lógica quantificacional clássica visa eliminar repetições desnecessárias de nós na árvore de refutação. Por exemplo, dada uma árvore de refutação, se um determinado nó dessa árvore, cuja fórmula é $\mathbf{P} \wedge \mathbf{Q}$, está sendo expandido, ocorrerá o acréscimo do tableau,

[4] Isto é, dada uma coleção finita de fórmulas Γ e uma fórmula **P**, se $\Gamma \vdash \mathbf{P}$, então os métodos de prova automática de teoremas que gozam da propriedade de completude (tais como os métodos da resolução e dos tableaux) confirmam que **P** é conseqüência lógica de Γ.

mostrado pela Figura 2, a cada ramo aberto descendente daquele nó. Porém, se um dos ramos, que sofrerá o acréscimo deste tableau, contiver um nó com a fórmula **P'** congruente a **P**, por exemplo, então não existe a necessidade que um novo nó contendo P seja acrescido ao ramo em questão.

Figura 2: Ramo Acrescentado pela Expansão da Fórmula $P \wedge Q$.

Assim, as regras do sistema de tableaux tradicional sofrem alterações quanto aos tableaux que deverão ser acrescentados ao final de cada ramo descendente do nó em expansão.

Portanto, a quíntupla $\mathbf{S_0} = \langle \mathbf{L_0}, \mathbf{L_1}, \mathbf{I_0}, \mathbf{F}, \mathbf{R_0} \rangle$, especificada no capítulo anterior, é alterada, neste refinamento, para $\mathbf{S_1} = \langle \mathbf{L_0}, \mathbf{L_1}, \mathbf{I_0}, \mathbf{F}, \mathbf{R_1} \rangle$, onde **P'** e **Q'** denotam, respectivamente, fórmulas congruentes a **P** e **Q**, e $\mathbf{R_1}$ é o novo conjunto de regras, dado a seguir:

Regra $\neg\neg\mathbf{P}$ - Se **P'** ocorre no ramo, então nada é acrescentado ao ramo, caso contrário o acréscimo é realizado segundo a forma tradicional.

Regra $\mathbf{P \rightarrow Q}$ - Se $\neg$**P'** ou **Q'** ocorrem no ramo, então nada é acrescentado;

- Se $\neg$**P'** e **Q'** não ocorrem no ramo e $\neg$**P** é congruente a **Q**, então o acréscimo é feito segundo a Figura 3;
- Se $\neg$**P'** e **Q'** não ocorrem no ramo e $\neg$**P** não é congruente **Q**, então o acréscimo é feito da forma tradicional.

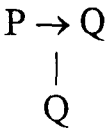

Figura 3: Regra $P \rightarrow Q$ modificada.

Regra **P** $\wedge$ **Q**

- Se **P'** e **Q'** ocorrem no ramo, então nada é acrescentado;
- Se **P'** ocorre no ramo e **Q'** não ocorre no ramo, então o acréscimo é feito segundo a Figura 4;
- Se **P'** não ocorre no ramo e **Q'** ocorre no ramo, então o acréscimo é feito segundo a Figura 5;
- Se nem **P'** e nem **Q'** ocorrem no ramo, então o acréscimo é feito da forma tradicional.

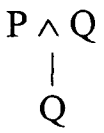

Figura 4: Primeira Regra P $\wedge$ Q modificada.

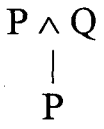

Figura 5: Segunda Regra P $\wedge$ Q modificada.

Regra **P** $\vee$ **Q**

- Se **P'** ou **Q'** ocorrem no ramo, então nada é acrescentado;
- Se **P'** e **Q'** não ocorrem no ramo e **P** é congruente a **Q**, então o acréscimo é feito segundo a Figura 6;
- Se **P'** e **Q'** não ocorrem no ramo e **P** não é congruente a **Q**, então o acréscimo é feito da forma tradicional.

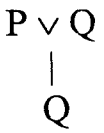

Figura 6: Regra P $\vee$ Q modificada.

Regra $\neg(\mathbf{P} \rightarrow \mathbf{Q})$ - Se **P'** e $\neg$**Q'** ocorrem no ramo, então nada é acrescentado;

- Se **P'** ocorre no ramo e $\neg$**Q'** não ocorre no ramo, então o acréscimo é feito segundo a Figura 7;
- Se **P'** não ocorre no ramo e $\neg$**Q'** ocorre no ramo, então o acréscimo é feito segundo a Figura 8;
- Se nem **P'** e nem $\neg$**Q'** ocorrem no ramo, então o acréscimo é feito da forma tradicional.

$$\neg(P \rightarrow Q)$$
$$|$$
$$\neg Q$$

Figura 7: Primeira Regra $\neg(\mathbf{P} \rightarrow Q)$ modificada.

$$\neg(P \rightarrow Q)$$
$$|$$
$$P$$

Figura 8: Segunda Regra $\neg(P \rightarrow Q)$ modificada.

Regra $\neg(\mathbf{P} \wedge \mathbf{Q})$ - Se $\neg$**P'** ou $\neg$**Q'** ocorrem no ramo, então nada é acrescentado;

- Se $\neg$**P'** e $\neg$**Q'** não ocorrem no ramo e **P** é congruente a **Q**, então o acréscimo é feito segundo a Figura 9;
- Se $\neg$**P'** e $\neg$**Q'** não ocorrem no ramo e **P** não é congruente a **Q**, então o acréscimo é feito da forma tradicional.

$$\neg(P \wedge Q)$$
$$|$$
$$\neg P$$

Figura 9: Regra $\neg(P \wedge Q)$ modificada.

Regra $\neg(\mathbf{P} \vee \mathbf{Q})$ - Se $\neg\mathbf{P}$' e $\neg\mathbf{Q}$' ocorrem no ramo, então nada é acrescentado;

- Se $\neg\mathbf{P}$' ocorre no ramo e $\neg\mathbf{Q}$' não ocorre no ramo, então o acréscimo é feito segundo a Figura 10;
- Se $\neg\mathbf{P}$' não ocorre no ramo e $\neg\mathbf{Q}$' ocorre no ramo, então o acréscimo é feito segundo a Figura 11;
- Se nem $\neg\mathbf{P}$' e nem $\neg\mathbf{Q}$' ocorrem no ramo, então o acréscimo é feito da forma tradicional.

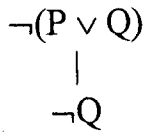

Figura 10: Primeira Regra $\neg(P \vee Q)$ modificada.

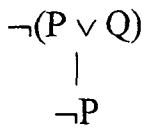

Figura 11: Segunda Regra $\neg(P \vee Q)$ modificada.

Regra $\exists x\ \mathbf{P}$ - Utiliza-se a regra a Regra $\exists x\ \mathbf{P}$ de **R**.

Regra $\forall x\ \mathbf{P}$ - Utiliza-se a Regra $\forall x\ \mathbf{P}$ de **R**, porém os nós com as fórmulas $\mathbf{P}(x/t_i)$, para $\mathbf{i} = 1,\ldots, \mathbf{n}$, onde $\mathbf{P}(x/t_i)$, a menos de congruência, já ocorre no ramo, não devem ser acrescentados.

Regra $\neg\exists x\ \mathbf{P}$ - Se uma fórmula congruente a $\forall x\ \neg\mathbf{P}$ ocorre no ramo, então nada é acrescentado, caso contrário utiliza-se a Regra $\neg\exists x\ \mathbf{P}$ de **R.**

Regra $\neg\forall x\ \mathbf{P}$ - Se uma fórmula congruente a $\exists x\ \neg\mathbf{P}$ ocorre no ramo, então nada é acrescentado, caso contrário utiliza-se a Regra $\neg\forall x\ \mathbf{P}$ de **R**.

Para este sistema de tableaux valem resultados análogos aos descritos em 4.2.3, 4.2.4 e 4.2.5.

2 – Segundo Refinamento

O refinamento anterior não se preocupa com a repetição, eventualmente desnecessária, das fórmulas quantificadas universalmente. Já no refinamento atual é dado um tratamento a esta questão, considerando-se o fato de que não existe a necessidade de que se repita uma fórmula universal, instanciada para todos os termos fechados de um determinado ramo, se não houver a possibilidade de que novos termos surjam neste ramo. Esta possibilidade é acusada pela não ocorrência futura no ramo de fórmulas quantificadas existencialmente, bem como pela não ocorrência futura de fórmulas que contenham variáveis sob o escopo de sinais funcionais.

A modificação, realizada neste segundo refinamento, altera somente a regra para fórmulas universais, que deverá ser como segue:

Regra $\forall x\ \mathbf{P}$ - Acrescentam-se os nós com as fórmulas $\mathbf{P}(x/t_i)$, para $\mathbf{i} = 1,\ldots, \mathbf{n}$, para as quais $\mathbf{P}(x/t_i)$, a menos de congruência, não ocorre no ramo: Repete-se o nó com a fórmula $\forall x\ \mathbf{P}$, somente quando existir no ramo uma fórmula **Q**, de um nó não marcado, tal que **Q** propague novos termos no ramo em questão, do modo como será definido a seguir.

Assim, após a inserção das fórmulas obtidas pela instanciação da fórmula universal para os termos fechados do ramo, o procedimento varre este ramo em busca

de fórmulas que possibilitem o aparecimento de novos termos. Se existir no ramo pelo menos uma fórmula que propague novos termos, a fórmula quantificada universalmente deve ser repetida neste ramo.

Para que possamos classificar as fórmulas pertencentes a um ramo quanto à propagação de novos termos, definiremos uma série de funções, as quais destinam-se a especificar este procedimento.

2.1 Definição: As cláusulas abaixo definem recursivamente as funções $f_\forall$ e $f_\exists$:

- $f_\forall(x, \mathbf{p}(t_1,..., t_n)) = 0$
- $f_\exists(x, \mathbf{p}(t_1,..., t_n)) = 0$
- $f_\forall(x, \neg\mathbf{P}) = f_\exists(x, \mathbf{P})$
- $f_\exists(x, \neg\mathbf{P}) = f_\forall(x, \mathbf{P})$
- $f_\forall(x, \mathbf{P} \rightarrow \mathbf{Q}) = \max \{f_\exists(x, \mathbf{P}), f_\forall(x, \mathbf{Q})\}$
- $f_\exists(x, \mathbf{P} \rightarrow \mathbf{Q}) = \max \{f_\forall(x, \mathbf{P}), f_\exists(x, \mathbf{Q})\}$
- $f_\forall(x, \mathbf{P} \wedge \mathbf{Q}) = \max \{f_\forall(x, \mathbf{P}), f_\forall(x, \mathbf{Q})\}$
- $f_\exists(x, \mathbf{P} \wedge \mathbf{Q}) = \max \{f_\exists(x, \mathbf{P}), f_\exists(x, \mathbf{Q})\}$
- $f_\forall(x, \mathbf{P} \vee \mathbf{Q}) = \max \{f_\forall(x, \mathbf{P}), f_\forall(x, \mathbf{Q})\}$
- $f_\exists(x, \mathbf{P} \vee \mathbf{Q}) = \max \{f_\exists(x, \mathbf{P}), f_\exists(x, \mathbf{Q})\}$
- $f_\forall(x, \forall x\, \mathbf{P}) = 0$
- $f_\exists(x, \forall x\, \mathbf{P}) = 0$
- $f_\forall(x, \exists x\, \mathbf{P}) = 0$
- $f_\exists(x, \exists x\, \mathbf{P}) = 0$
- $f_\forall(x, \forall y\, \mathbf{P}) = 1$ se, e somente se, x é livre em $\mathbf{P}$
- $f_\exists(x, \forall y\, \mathbf{P}) = f_\exists(x, \mathbf{P})$
- $f_\forall(x, \exists y\, \mathbf{P}) = f_\forall(x, \mathbf{P})$
- $f_\exists(x, \exists y\, \mathbf{P}) = 1$ se, e somente se, x é livre em $\mathbf{P}$

A função $f_\forall$ foi construída de modo que $f_\forall(x, \mathbf{P}) = 1$ se, e somente se, $\mathbf{P}$ possui uma subfórmula fora do escopo de x, que gera uma fórmula universal, cuja variável quantificada é distinta de x e em cuja matriz x é livre. Nos outros casos, $f_\forall(x, \mathbf{P}) = 0$.

Da mesma forma, $\mathbf{f}_{\exists}(x, \mathbf{P}) = 1$ se, e somente se, **P** possui uma subfórmula fora do escopo de x, que gera uma fórmula existencial, cuja variável quantificada é distinta de x e em cuja matriz x é livre. Nos outros casos, $\mathbf{f}_{\exists}(x, \mathbf{P}) = 0$.

2.4 Definição: A função **esf** (sigla tirada da expressão "escopo de sinal funcional") é definida como segue:

- $\mathbf{esf}(x, \mathbf{P}) = \begin{cases} 1, \text{ caso } x \text{ possua uma ocorrência livre em } \mathbf{P} \text{ no escopo de um} \\ \text{sinal funcional;} \\ 0, \text{ caso contrário.} \end{cases}$

2.5 Definição: As cláusulas abaixo definem recursivamente as funções **tu** (sigla tirada da expressão "tipo universal") e **te** (sigla tirada da expressão "tipo existencial"):

- $\mathbf{tu}(\mathbf{p}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) = 0$
- $\mathbf{te}(\mathbf{p}(\mathbf{t}_1, \ldots, \mathbf{t}_n)) = 0$
- $\mathbf{tu}(\neg\mathbf{P}) = \mathbf{te}(\mathbf{P})$
- $\mathbf{te}(\neg\mathbf{P}) = \mathbf{tu}(\mathbf{P})$
- $\mathbf{tu}(\mathbf{P} \rightarrow \mathbf{Q}) = \max \{\mathbf{te}(\mathbf{P}), \mathbf{tu}(\mathbf{Q})\}$
- $\mathbf{te}(\mathbf{P} \rightarrow \mathbf{Q}) = \max \{\mathbf{tu}(\mathbf{P}), \mathbf{te}(\mathbf{Q})\}$
- $\mathbf{tu}(\mathbf{P} \wedge \mathbf{Q}) = \max \{\mathbf{tu}(\mathbf{P}), \mathbf{tu}(\mathbf{Q})\}$
- $\mathbf{te}(\mathbf{P} \wedge \mathbf{Q}) = \max \{\mathbf{te}(\mathbf{P}), \mathbf{te}(\mathbf{Q})\}$
- $\mathbf{tu}(\mathbf{P} \vee \mathbf{Q}) = \max \{\mathbf{tu}(\mathbf{P}), \mathbf{tu}(\mathbf{Q})\}$
- $\mathbf{te}(\mathbf{P} \vee \mathbf{Q}) = \max \{\mathbf{te}(\mathbf{P}), \mathbf{te}(\mathbf{Q})\}$
- $\mathbf{tu}(\forall x\, \mathbf{P}) = \max \{\mathbf{f}_{\exists}(x, \mathbf{P}) + 1, \mathbf{esf}(x, \mathbf{P}) + 1, \mathbf{tu}(\mathbf{P})\}$
- $\mathbf{te}(\forall x\, \mathbf{P}) = \mathbf{te}(\mathbf{P})$
- $\mathbf{tu}(\exists x\, \mathbf{P}) = \mathbf{tu}(\mathbf{P})$
- $\mathbf{te}(\exists x\, \mathbf{P}) = \max \{\mathbf{f}_{\forall}(x, \mathbf{P}) + 1, \mathbf{esf}(x, \mathbf{P}) + 1, \mathbf{te}(\mathbf{P})\}$

Interpretando os valores obtidos para a função **tu**, observamos que $\mathbf{tu}(\mathbf{P}) = 2$ se, e somente se, **P** gera uma fórmula da forma $\forall x\, \mathbf{Q}$, tal que **Q** possui uma subfórmula fora do escopo de x em **Q**, a qual gera uma fórmula existencial, cuja variável

quantificada é distinta de x e em cuja matriz x é livre, ou **P** gera uma fórmula da forma $\forall x$ **Q**, tal que x possui uma ocorrência livre em **Q** no escopo de um sinal funcional. **tu(P)** = 1 se, e somente se, **P** gera uma fórmula universal que não se enquadra nos dois casos anteriores. **tu(P)** = 0 se, e somente se, **P** não gera fórmula universal.

Através da definição de **te**, podemos observar que **te(P)** = 2 se, e somente se, **P** gera uma fórmula da forma $\exists x$ **Q**, tal que **Q** possui uma subfórmula fora do escopo de x em **Q**, a qual gera uma fórmula universal, cuja variável quantificada é distinta de x e em cuja matriz x é livre, ou **P** gera uma fórmula da forma $\exists x$ **Q**, tal que x possui uma ocorrência livre em **Q** no escopo de um sinal funcional. **te(P)** = 1 se, e somente se, **P** gera uma fórmula existencial que não se enquadra nos dois casos anteriores. **te(P)** = 0 se, e somente se, **P** não gera fórmula existencial.

2.8 Definição: As clausulas abaixo definem a função de propagação de termos **pt**:

- $\mathbf{pt(P)} = \begin{cases} 2, \text{ caso } \mathbf{tu(P)} = 2; \\ 1, \text{ caso } \mathbf{tu(P)} \neq 2 \text{ e } \mathbf{te(P)} \neq 0; \\ 0, \text{ caso } \mathbf{tu(P)} \neq 2 \text{ e } \mathbf{te(P)} = 0. \end{cases}$

A busca por fórmulas que propaguem termos ocasiona o aparecimento de uma classificação destas fórmulas em três categorias: fórmulas que não propagam termos (**pt(P)** = 0), fórmulas que propagam termos temporariamente (**pt(P)** = 1), e fórmulas que propagam termos indefinidamente (**pt(P)** = 2).

2.9 Definição: Dizemos que uma fórmula **P** *propaga novos termos* se, e somente se, **pt(P)** $\neq 0$.

Propomos abaixo um teorema relacionando o conceito dado pela última definição com o conceito de geração de fórmulas por fórmulas em um tableaux.

2.10 Teorema: Sendo **P** uma fórmula de um nó de um tableau **T**, temos que **P** propaga novos termos se, e somente se, existe uma fórmula **Q** possuindo um termo que não

figure em **T**, tal que **P** gere **Q** em **T** com respeito a **S**, onde **S** poderá ser tanto o sistema de tableaux tradicional como o primeiro refinamento ao sistema de tableaux tradicional.

Logo, em ramos que possuem ao menos uma fórmula que propague termos indefinidamente, a repetição das fórmulas quantificadas universalmente sempre ocorrerá. Em ramos que não possuem fórmulas que propaguem termos indefinidamente, e possuem uma ou mais fórmulas que propaguem termos temporariamente, a repetição das fórmulas quantificadas universalmente será realizada até que aquelas fórmulas sejam suficientemente expandidas, deixando de existir no ramo fórmulas que propaguem termos temporariamente, cessando nesta altura a propagação de novos termos. Em ramos que possuem somente fórmulas que não propaguem termos, não se efetuará a repetição das fórmulas quantificadas universalmente.

Este segundo refinamento permitirá conclusões negativas para uma série de casos anteriormente indecidíveis pelos dois algoritmos anteriormente descritos.

Acreditamos, mas ainda não provamos, que valem resultados análogos para este refinamento àqueles descritos em 4.2.3, 4.2.4 e 4.2.5.

3 - Terceiro Refinamento

Após a modificação anterior, o passo seguinte é trazer ao método um pouco mais da intuição humana, que aparece quando o mesmo é executado com papel e lápis. Mais uma vez voltamos as atenções sobre a regra para expansão de fórmulas universais. O que qualquer pessoa sensata, que utilize o método dos tableaux, faz, ao pôr o mesmo em prática, é realizar instanciações de fórmulas universais somente com aqueles termos interessantes ao fechamento do ramo em questão. Algo com esse intuito foi sugerido por [Oppacher & Suen 1988], porém em nenhum momento ficou claro qual a forma para a execução deste tipo de procedimento. Além disso, na literatura consultada não foi encontrado algo que implementasse ou descrevesse o raciocínio proposto.

Nesta modificação final, realizada sobre o primeiro refinamento ao método dos tableaux, incluímos no algoritmo em questão um procedimento de unificação, o qual é uma expansão do algoritmo de unificação usado tradicionalmente nos métodos de resolução, tal como apresentado, por exemplo, em [Lloyd 1987]; outras apresentações deste algoritmo para o método da resolução podem ser encontradas em [Chang & Lee 1973], [Loveland 1978] e [Fitting 1990].

Encontramos em [Fitting 1990] um tipo de associação entre o método dos tableaux e o procedimento de unificação diferente da maneira proposta aqui. Tal método substitui variáveis quantificadas universalmente por variáveis livres, as quais, ao serem substituídas por termos apropriados ao fechamento de determinados ramos, geram novos tableaux, havendo uma relação de dependência entre distintos ramos possuindo a mesma variável livre. Propomos a associação da unificação ao método dos tableaux sem criar dependência entre os seus ramos, e sem necessidade de geração de mais de uma árvore de refutação durante o procedimento de prova, mantendo, desta forma, a característica básica do procedimento, isto é, a geração de uma árvore de refutação a partir de um tableau inicial através da aplicação de regras.

O terceiro refinamento altera também o processo de construção do tableau inicial, onde a função de inicialização, além substituir todas as variáveis livres do conjunto inicial de fórmulas por novas constantes e retirar os quantificadores vácuos de cada uma destas fórmulas, também realiza uma renomeação de variáveis, de modo que variáveis sob escopo de quantificadores diferentes possam ser mais facilmente identificadas. Isto é feito para facilitar o processo de unificação.

O processo de unificação modificado trata do ponto crucial desta alteração. No momento da expansão de uma fórmula universal obtemos, através deste processo, uma determinada lista de termos, que nos dirá para quais termos a fórmula universal deve ser instanciada, e se existe a necessidade ou não de repetição desta fórmula no ramo em expansão.

A seguir apresentamos uma série de conceitos relativos ao processo de unificação. Utilizamos como referência básica sobre este assunto [Lloyd 1987].

3.1 Definição: Uma *substituição* θ é um conjunto finito da forma $\{v_1/t_1,..., v_n/t_n\}$, onde cada v_i é uma variável, cada t_i é um termo distinto de v_i e as variáveis v_1, .., v_n são distintas. Cada elemento v_i/t_i é chamado *ligação para* v_i *em* θ. θ é chamado uma *substituição fechada* se os t_i's são termos fechados. θ é chamado uma *substituição de variáveis pura* se os t_i's são todos variáveis.

3.2 Definição: Uma *expressão* é um termo ou uma fórmula atômica.

3.3 Definição: Seja $\theta = \{v_1/t_1,..., v_n/t_n\}$ uma substituição e **E** uma expressão. Especificamos o que vem a ser $E\theta$, dita a *instância* de **E** para θ, a expressão obtida de **E** pela substituição simultânea de cada ocorrência da variável v_i em **E** pelo termo t_i ($i = 1,..., n$), pelas seguintes cláusulas:

- $c\theta = c$;
- $x_i\theta = t_i$, onde $x_i = v_i$ para algum $i = 1, ..., n$;
- $x\theta = x$, onde $x \neq v_i$ para todo $i = 1, ..., n$;
- $f(x_1, ..., x_n)\theta = f(x_1\theta, ..., x_n\theta)$;
- $p(x_1, ..., x_n)\theta = p(x_1\theta, ..., x_n\theta)$;

3.4 Definição: Se $E\theta$ é fechado, então $E\theta$ é chamado *instância fechada de* **E**. Se $S = \{E_1,..., E_n\}$ é um conjunto finito de expressões e θ é uma substituição, então $S\theta$ denota o conjunto $\{E_1\theta,..., E_n\theta\}$.

3.5 Definição: Sejam $\theta = \{ u_1/s_1,..., u_m/s_m\}$ e $\sigma = \{v_1/t_1,..., v_n/t_n\}$ substituições. Então a *composição* $\theta\sigma$ de θ e σ é a substituição obtida do conjunto

$$\{u_1/s_1\sigma,..., u_m/s_m\sigma, v_1/t_1,..., v_n/t_n\},$$

removendo-se qualquer ligação $u_i/s_i\sigma$, para a qual $u_i = s_i\sigma$, e, da mesma forma, removendo-se qualquer ligação v_j/t_j, para a qual $v_j \in \{u_1,..., u_m\}$.

3.6 Definição: A substituição dada pelo conjunto vazio é chamada *substituição identidade*. ε denotará a substituição identidade. Note que $\mathbf{E}\varepsilon = \mathbf{E}$, para todas as expressões **E**.

3.7 Teorema: Sejam θ, σ e γ substituições e **E** uma expressão, então:

- $\theta\varepsilon = \varepsilon\theta = \theta$;
- $(\mathbf{E}\theta)\sigma = \mathbf{E}\,(\theta\sigma)$, para todo **E**;
- $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

3.8 Definição: Seja **S** um conjunto formado por uma coleção de expressões. Uma substituição θ é chamada de *unificador* para **S** se $\mathbf{S}\theta$ é um objeto único.

3.9 Definição: Seja **S** uma coleção de expressões. O *conjunto de discordância* de **S** é definido pelo seguinte procedimento. Localize a posição do símbolo mais à esquerda, tal que nem toda expressão de **S**, naquela posição, tem o mesmo símbolo, e finalmente extraia de cada expressão correspondente em **S** a subexpressão iniciando nesta posição. O conjunto de todas as subexpressões obtidas é o conjunto de discordância.

3.10 Definição: O *método da unificação* é definido pela seqüência de passos abaixo:

1°) Sejam um contador $\mathbf{k} = 0$, uma substituição $\sigma_0 = \varepsilon$, e um conjunto de expressões **S**;

2°) Se $\mathbf{S}\sigma_k$ é um objeto único, então pare; σ_k é um unificador de **S**. Senão, encontre o conjunto de discordância $\mathbf{D_k}$ de $\mathbf{S}\sigma_k$;

3°) Se existe **v** e **t** em $\mathbf{D_k}$, tais que **v** é uma variável que não ocorre no termo **t**, então faça $\sigma_{k+1} = \sigma_k\,\{\mathbf{v/t}\}$, incremente **k** e volte ao passo 2. Senão, **S** não é unificável.

3.11 Definição: Uma *renomeação* $\theta = \{v_1//w_1, \ldots, v_m//w_m\}$ é um conjunto finito da forma $\{v_1//w_1, \ldots, v_n//w_n\}$, onde os v_i's e w_i's são variáveis distintas duas a duas. Dado um designador **D**, definimos o que vem ser $\mathbf{D}\theta$:

- $\mathbf{c}\theta = \mathbf{c}$;
- $x\theta = \begin{cases} \mathbf{t_i}, \text{ onde } x = v_i \text{ para algum } \mathbf{i} = 1, \ldots, \mathbf{n}; \\ x, \text{ onde } x \neq v_i \text{ para todo } \mathbf{i} = 1, \ldots, \mathbf{n}; \end{cases}$

- $\mathbf{f}(t_1,..., t_n)\,\theta = \mathbf{f}(t_1\theta,..., t_n\theta)$;
- $\mathbf{p}(t_1,..., t_n)\,\theta = \mathbf{p}(t_1\theta,..., t_n\theta)$;
- $(\neg\mathbf{P})\,\theta = \neg\mathbf{P}\theta$;
- $(\mathbf{P} \rightarrow \mathbf{Q})\,\theta = \mathbf{P}\theta \rightarrow \mathbf{Q}\theta$;
- $(\mathbf{P} \wedge \mathbf{Q})\,\theta = \mathbf{P}\theta \wedge \mathbf{Q}\theta$;
- $(\mathbf{P} \vee \mathbf{Q})\,\theta = \mathbf{P}\theta \vee \mathbf{Q}\theta$;
- $(\forall x\ \mathbf{P})\,\theta = \begin{cases} \forall w_i\ \mathbf{P}\theta, \text{ onde } x = v_i \text{ para algum } \mathbf{i} = 1, \ldots, \mathbf{n}; \\ \forall x\ \mathbf{P}\theta, \text{ onde } x \neq v_i \text{ para todo } \mathbf{i} = 1, \ldots, \mathbf{n}; \end{cases}$
- $(\exists x\ \mathbf{P})\,\theta = \begin{cases} \exists w_i\ \mathbf{P}\theta, \text{ onde } x = v_i \text{ para algum } \mathbf{i} = 1, \ldots, \mathbf{n}; \\ \exists x\ \mathbf{P}\theta, \text{ onde } x \neq v_i \text{ para todo } \mathbf{i} = 1, \ldots, \mathbf{n}; \end{cases}$

$\mathbf{D}\theta$ é dito uma renomeação de $\mathbf{D}$ para $\{w_1,..., w_n\}$ a partir de $\{v_1,..., v_n\}$. $\mathbf{D}\theta$ é dita uma renomeação completa de $\mathbf{D}$ para $\{w_1,..., w_n\}$ se $\mathbf{D}\theta$ é uma renomeação de $\mathbf{D}$ para $\{w_1,..., w_n\}$ a partir da coleção de todas as variáveis que ocorrem em $\mathbf{D}$.

A seguir apresentamos algumas definições relativas à obtenção das expressões sobre as quais será realizado o processo de unificação.

3.12 Definição: Dizemos quando uma dada primeira fórmula é uma *subfórmula positiva* ou uma *subfórmula negativa* de uma dada segunda fórmula através das seguintes cláusulas:

- $\mathbf{P}$ é subfórmula positiva de $\mathbf{P}$, $\mathbf{Q} \rightarrow \mathbf{P}$, $\mathbf{P} \wedge \mathbf{Q}$, $\mathbf{P} \vee \mathbf{Q}$, $\mathbf{Q} \wedge \mathbf{P}$, $\mathbf{Q} \vee \mathbf{P}$, $\forall x\ \mathbf{P}$ e $\exists x\ \mathbf{P}$;
- $\mathbf{P}$ é subfórmula negativa de $\neg\mathbf{P}$ e $\mathbf{P} \rightarrow \mathbf{Q}$;
- Se $\mathbf{P}$ é subfórmula positiva de $\mathbf{Q}$ e $\mathbf{Q}$ é sufórmula positiva de $\mathbf{R}$, então $\mathbf{P}$ é subfórmula positiva de $\mathbf{R}$;
- Se $\mathbf{P}$ é subfórmula positiva de $\mathbf{Q}$ e $\mathbf{Q}$ é sufórmula negativa de $\mathbf{R}$, então $\mathbf{P}$ é subfórmula negativa de $\mathbf{R}$;

- Se **P** é subfórmula negativa de **Q** e **Q** é sufórmula positiva de **R**, então **P** é subfórmula negativa de **R**;
- Se **P** é subfórmula negativa de **Q** e **Q** é sufórmula negativa de **R**, então **P** é subfórmula positiva de **R**.

3.13 Definição: x é uma *variável universal* em uma fórmula **P** se uma das seguintes condições for satisfeita:

- existe uma subfórmula positiva $\forall x$ **Q** de **P** tal que x é livre em **Q**;
- existe uma subfórmula negativa $\exists x$ **Q** de **P** tal que x é livre em **Q**.

Em ambos os casos, todas as ocorrências livres de x em **Q** são ditas ocorrências universais de x em **P**.

3.14 Definição: x é uma *variável existencial* em uma fórmula **P** se uma das seguintes condições for satisfeita:

- existe uma subfórmula positiva $\exists x$ **Q** de **P** tal que x é livre em **Q**;
- existe uma subfórmula negativa $\forall x$ **Q** de **P** tal que x é livre em **Q**.

Em ambos os casos, todas as ocorrências livres de x em **Q** são ditas ocorrências existenciais de x em **P**.

3.15 Notação: Usamos a letra **n** seguida de um numeral inteiro positivo como notação para aquilo que denominamos *constantes existenciais*, as quais usamos no processo de unificação para substituir as variáveis existenciais de uma fórmula.

3.16 Definição: *Fórmula assinalada* é um par cujo primeiro componente é um dos sinais "+" ou "-", e o segundo componente é uma fórmula.

3.17 Definição: Obtemos o que chamamos de *lista de fórmulas atômicas assinaladas de uma fórmula* **P** através do seguinte processo:

1º) separamos o conjunto **A** de todas as fórmulas atômicas assinaladas (**s**, **Q**) tais que:

- se **s** = "+", então **Q** é subfórmula atômica positiva de **P**;
- se **s** = "-", então **Q** é subfórmula atômica negativa de **P**;

2º) substituímos cada uma das ocorrências existenciais de variáveis em **P** por novas constantes existenciais em todas as fórmulas assinaladas de **A**, onde ocorrências de uma mesma variável existencial são substituídas por uma mesma constante existencial, e ocorrências de diferentes variáveis existenciais são substituídas por diferentes constantes existenciais, obtendo assim a lista desejada.

3.18 Definição: Obtemos o que chamamos de *lista de fórmulas atômicas assinaladas de uma dada coleção* Γ *de fórmulas com respeito a uma primeira lista* ζ *de fórmulas assinaladas* através do seguinte processo:

1º) separamos o conjunto **A** de todas as fórmulas atômicas assinaladas (**s**, **Q**) tais que:

- se **s** = "+", então **Q** é subfórmula atômica positiva de alguma fórmula de Γ;
- se **s** = "-", então **Q** é subfórmula atômica negativa de alguma fórmula de Γ;

2º) substituímos cada uma das ocorrências de variáveis existenciais de alguma fórmula de Γ por novas constantes existenciais, distintas das constantes existenciais que ocorrem em ζ, em todas as fórmulas assinaladas de **A**, onde ocorrências de uma mesma variável existencial são substituídas por uma mesma constante existencial, e ocorrências de diferentes variáveis existenciais são substituídas por diferentes constantes existenciais, obtendo assim a lista desejada.

3.19 Definição: Sejam Γ e Γ' duas coleções de fórmulas atômicas assinaladas. A *coleção de substituições obtidas de* Γ e Γ', *nesta ordem*, é o conjunto de todas as substituições que unificam fórmulas componentes de fórmulas atômicas assinaladas de Γ e Γ' com sinais diferentes.

Assim, para o terceiro refinamento, conservam-se as regras expostas no primeiro refinamento, com exceção da regra para fórmulas universais, substituída pelos passos enumerados a seguir.

Os passos abaixo especificam os procedimentos a serem seguidos na primeira aplicação da regra para quantificadores universais a uma determinada fórmula $\forall x$ **P**.

1º) Obtemos a coleção Γ de fórmulas atômicas assinaladas de $\forall x\ \mathbf{P}$, tal que x é livre nessas fórmulas;

2º) obtemos a coleção Γ' de fórmulas atômicas assinaladas de todas as fórmulas do ramo, distintas de $\forall x\ \mathbf{P}$, dos nós que ainda não foram marcados, com respeito a Γ;

3º) obtemos a coleção de substituições Σ, obtida de Γ e Γ';

4º) reunimos todos os termos $\mathbf{t_1},..., \mathbf{t_n}$, tais que $x/\mathbf{t_i}$ ($\mathbf{i} = 1,..., \mathbf{n}$) figura em algum elemento de Σ, onde os $\mathbf{t_i}$'s são termos fechados sem constantes existenciais;

5º) separamos de $\mathbf{t_1},..., \mathbf{t_n}$ os termos $\mathbf{t_{i1}},..., \mathbf{t_{ik}}$, tais que $\mathbf{P}(x/\mathbf{t_{i1}}),..., \mathbf{P}(x/\mathbf{t_{ik}})$ não figuram no ramo, a menos de congruência;

6º) caso k = 0, acrescentamos no ramo o tableau da Figura 12;

7º) caso k > 0, acrescentamos no ramo o tableau da Figura 13.

Na Figura 12, $\mathbf{P}(x/\mathbf{c_1})$' é uma renomeação completa de $\mathbf{P}(x/\mathbf{c_1})$ para uma coleção de variáveis que não ocorrem no ramo. Na Figura 13 e na Figura 14, $\mathbf{P}(x/\mathbf{t_{i1}})',..., \mathbf{P}(x/\mathbf{t_{ik}})'$ são, respectivamente, renomeações completas de $\mathbf{P}(x/\mathbf{t_{i1}}),..., \mathbf{P}(x/\mathbf{t_{ik}})$ para coleções de variáveis disjuntas duas a duas que não ocorrem no ramo.

$$\forall x\ P$$
$$|$$
$$P(x/c_1)'$$

Figura 12: Primeiro tableau inserido pela expansão de uma fórmula $\forall x\ P$.

$$\forall x\ P$$
$$|$$
$$P(x/t_{i1})'$$
$$...$$
$$P(x/t_{ik})'$$

Figura 13: Segundo tableau inserido pela expansão de uma fórmula $\forall x\ P$.

$$P(x/t_{i1})'$$
$$...$$
$$P(x/t_{ik})'$$

Figura 14: Terceiro tableau inserido pela expansão de uma fórmula $\forall x\ \mathbf{P}$.

Os passos abaixo especificam os procedimentos a serem seguidos na aplicação da regra para quantificadores universais a uma determinada fórmula $\forall x$ **P**, anteriormente instanciada.

Os passos 1, 2, 3, 4 e 5 são idênticos aos passos dados no primeira aplicação de regra à fórmula $\forall x$ **P**.

6º) Caso exista algum x/**t** figurando em algum elemento de Σ (descrito no passo 3), tal que t possui uma constante existencial, então acrescentamos no ramo o tableau da Figura 13;

7º) caso não exista nenhum x/**t** figurando em algum elemento de Σ, tal que **t** possui uma constante existencial ou uma variável, então acrescentamos o tableau da Figura 14;

8º) caso não exista nenhum x/**t** figurando em algum elemento de Σ, tal que **t** possui uma constante existencial, então procuramos por algum x/**t** figurando em algum elemento de Σ, tal que **t** possui uma variável, e por duas fórmulas assinaladas, com sinais distintos, **P** e **Q** de Γ', tal que exista uma variável y em **t** e uma substituição θ que unifica **P** e **Q**, tal que y/**t'** pertence a θ, para algum termo **t'**, onde **t'** possui uma constante existencial ou é um termo que não possui nenhuma instância fechada que já tenha sido usada pela fórmula do ramo que contém y;

9º) se a busca executada no passo anterior foi bem sucedida, então acrescentamos ao ramo considerado o tableau dado pela Figura 13;

10º) se a busca executada no 8º passo não foi bem sucedida, então acrescentamos ao ramo considerado o tableau dado pela Figura 14.

Acreditamos, mas ainda não provamos, que valem resultados análogos para este refinamento àqueles descritos em 4.2.3, 4.2.4 e 4.2.5.

6 - DESCRIÇÃO DA IMPLEMENTAÇÃO

Apresentamos abaixo a descrição e a listagem do programa (escrito em ALLEGRO CL PERSONAL EDICTION 5.0) que implementa os 4 sistemas de tableaux descritos nos capítulos anteriores.

Este programa foi construído de modo a resolver, considerando a lógica quantificacional clássica, qualquer uma das questões abaixo:

- verificar se uma determinada fórmula é teorema ou não;
- verificar se uma dada coleção de fórmulas é satisfatível ou não;
- verificar se uma determinada fórmula é conseqüência lógica ou não, de uma dada coleção de fórmulas.

O programa pode também indicar que, diante dos dados de entrada, nada pode ser respondido.

Os dados de entrada são de quatro tipos:

- Algoritmo: é uma escolha entre o sistema de tableaux tradicional e um de seus refinamentos;
- Número de nós: número máximo de nós que a árvore de refutação alcançará;
- Teorema: fórmula da lógica equacional;
- Coleção de Fórmulas: conjunto de fórmulas da lógica equacional.

O programa utiliza uma representação interna para fórmulas da lógica quantificacional clássica, onde cada fórmula é uma lista cujo o primeiro elemento pode ser um dos operadores da lista abaixo:

- **ent** - para formar implicação entre duas fórmulas;
- **e** - para formar conjunção entre duas fórmulas;
- **ou** - para formar disjunção entre duas fórmulas;
- **eq** - para formar equivalência entre duas fórmulas;
- **n** - para formar negação de fórmula;
- **qs** - para formar quantificação universal de uma fórmula;
- **ex** - para formar quantificação existencial de uma fórmula;
- qualquer outro símbolo diferente dos apresentados acima – para formar uma fórmula atômica.

Desta forma, consideramos as palavras acima escritas em negrito reservadas à operação do programa, não podendo ser utilizadas de outra forma, que não a descrita acima. Outras palavras reservadas são:

- **n** seguido de índice, pois o mesmo é utilizado como constante especial no processo de unificação;
- **c** seguido de índice, pois este é utilizado como nova constante para instanciações de fórmulas existenciais.

Outra palavra especial é **i**, a qual utilizamos para representar o predicado de igualdade. Assim, cada vez que **i** é utilizado como sinal predicativo em uma fórmula, este predicado será considerado o predicado especial de igualdade, recebendo por isso um tratamento diferenciado com respeito ao fechamento do ramo. Isto é feito porque se tivermos em um ramo, por exemplo, uma fórmula $\neg$ (**t** = **t**), internamente representada por (n (i t t)), este ramo apresenta uma inconsistência, e portanto deve ser fechado. Este predicado especial permite que o sistema implementado seja testado para um número maior de problemas.

A partir do conjunto de fórmulas e do possível teorema, obtidos como entrada, o programa inicia seu processamento através da construção de uma árvore inicial

contendo n nós, onde n é o número de fórmulas do conjunto de fórmulas acrescido de um, devido ao possível teorema. Cada nó, representado por um átomo, contém os seguintes atributos:

- **form**: fórmula correspondente ao nó.
- **consist**: indica se o ramo o qual aquele nó pertence é fechado (consist = sim) ou não (consist = não);
- **n_cte**: índice para criação de uma nova constante no ramo;
- **l_fec**: lista com os termos fechados presentes no ramo;
- **inst**: lista termos instanciados para fórmula universal;
- **pai**: nó imediatamente anterior ao nó em consideração;
- **filhos**: lista de nós imediatamente posteriores ao nó em consideração;
- **rep**: indicação se a fórmula causa propagação de termos e qual o tipo da propagação.

"**form**" é uma fórmula da lógica equacional clássica sem variáveis livres, sem quantificadores vácuos (quantificadores cuja variável quantificada não ocorre na matriz) e sem o conetivo de equivalência, que são eliminados no momento da inicialização de cada nó, para uma maior simplicidade nas operações de expansão de nó.

As informações "**consist**", "**n_cte**" e "**l_fec**", são válidas apenas nas folhas da árvore, para os demais nós são nulas. "**inst**" é informação relevante apenas para fórmulas universais, para os demais tipos de fórmulas esta informação é nula. "**rep**" é utilizado apenas pelo algoritmo 3, nos demais algoritmos recebe valor nulo.

No momento em que a árvore é inicializada, cria-se uma lista de auxílio, chamada "**lista_arvore**". Esta lista contém inicialmente o nó raiz, e é responsável pela realização do caminhamento em amplitude que o algoritmo realiza para expansão da árvore. Após a criação do tableau inicial e da lista auxiliar, inicia-se um laço onde o primeiro nó da lista é selecionado para expansão, se este nó gera sucessores, então estes são acrescentado como folhas do tableau e no final da lista auxiliar. Isto é feito sucessivamente até que todos os ramos da árvore estejam fechados, ou um ramo exaurido aberto seja encontrado, ou o número máximo de nós da árvore seja alcançado.

As variáveis globais utilizadas pelo programa são:

- **lista_arvore** - lista contendo todas as formulas não expandidas;
- **n_nos** - número de nós da árvore;
- **n_ex** - número de constantes utilizadas no processo de unificação.
- **l_atomicas** - lista contendo todas as fórmulas atômicas de um determinado ramo do tableaux;
- **l_a1** - lista de fórmulas atômicas de formulas que serão unificadas em um determinado momento.
- **list_var** - lista contendo as variáveis das fórmulas componentes do tableux, é usada para eliminar repetição de variáveis no algoritmo4.

A seguir apresentamos uma listagem da funções componentes do programa e uma breve explicação da utilidade das mesmas.

As funções **principal-executa-on-change**, **read-integer**, **texto-to-list**, **tokens**, **constituent** e **principal-cancel-button-1-on-change** são responsáveis obtenção dos dados da interface. **principal-executa-on-change** também chama a função tableaux, responsável pelo inicio do processamento dos dados extraídos da interface.

```
(defun principal-executa-on-change (widget new-value old-value)
  (declare (ignore-if-unused widget new-value old-value))
  (let ((algoritmo (find-sibling :algoritmo widget))
        (limite (find-sibling :limite widget))
        (teoria (find-sibling :teoria widget))
        (teorema (find-sibling :teorema widget)) )
       (let (resultado v-limite v-teoria v-teorema)
         (setf v-limite (read-integer (value limite)))
         (setf v-teoria (texto-to-list (value teoria)))
         (if (eq (value teorema) "")
             (setf v-teorema nil)
             (setf v-teorema (read-from-string (value teorema))))
```

```
        (if (not (eq v-teorema nil))
        (if (and (eq (length v-teorema)1)(atom (car v-teorema)))
          (setf v-teoria (append v-teoria (list (cons 'n v-teorema))))
          (setf v-teoria (append v-teoria (list (list 'n v-teorema)))) ))
       (if (not (eq v-teoria nil))
         (setf resultado (tableaux v-limite v-teoria (value algoritmo))))
         (print 'resultado=)(princ resultado)
         (print 'n_nos=)(princ n_nos)  ))
  t) ; Accept the new value

(defun read-integer (str)
  ;transforma em inteiro o valor limite lido da caixa de texto
  (if (every #'digit-char-p str)
    (let ((accum 0))
      (dotimes (pos (length str))
        (setf accum (+ (* accum 10)
                  (digit-char-p (char str pos))))))  accum)  nil))

(defun texto-to-list (texto)
  ;varre a caixa de texto para converter em uma lista
 (let (ret)
 (setf texto (tokens texto #'constituent 0))
  (while texto
    (setf ret (append ret (list (read-from-string (car texto)))))
    (setf texto (cdr texto)))
  ret))

(defun tokens (str test start)
  ;extrai os item da caixa de texto
  (let ((p1 (position-if test str :start start)))
    (if p1
      (let ((p2 (position-if #'(lambda (c)
```

```
                    (not (funcall test c)))
            str :start p1)))
        (cons (subseq str p1 p2)
         (if p2
           (tokens str test p2)  nil)))  nil)))

(defun constituent (c)
  ;elimina os caracteres de nova linha
  (and (graphic-char-p c)
     (not (char= c  #\Newline))))

(defun principal-cancel-button-1-on-change (widget new-value old-value)
  (declare (ignore-if-unused widget new-value old-value))
  t) ; Accept the new value
```

A função **tableaux** contém o laço principal de execução do algoritmo e recebe três parâmetros. O primeiro parâmetro é **limite**, que se refere ao número máximo de nós que a árvore deve alcançar. O segundo é **teoria**, que é a lista das fórmulas da base de conhecimento, mais a negação do possível teorema. Esta lista é enviada para função que constrói o tableau inicial. O terceiro parâmetro é **algoritmo**, que é uma alternativa entre algoritmo1 (sistema de tableaux tradicional), algoritmo2 (primeiro refinamento), algoritmo3 (segundo refinamento) e algoritmo4 (terceiro refinamento), de acordo com esta variável o procedimento que executa a alternativa selecionada é chamado. A função **tableaux** retorna à interface **sai**, que é um flag informando se uma confutação foi ou não encontrada.

```
(defun tableaux (limite teoria algoritmo)
  (let (lista filhos par sai pai)
; lista - lista contendo todas os nós inicializados com as formulas obtidas da interface
; filhos - lista de filhos de um nó, que serão adicionados no final de lista_arvore
; sai - flag que sinaliza um ramo aberto exaurido
```

```
(setf n_nos (length teoria))
(setf lista (inicializa_arvore teoria algoritmo))
(if (not (eq lista 'inconsistente))(let()
(setf par 'sim)
(setf lista_arvore (list (car lista)))
(loop
 (if (equal algoritmo ':algoritmo4)
    (setf sai (processa_no))
    (if (equal algoritmo ':algoritmo3)
       (setf sai (processa_no_pu))
       (if (equal algoritmo ':algoritmo2)
          (setf sai (processa_no_apu))
          (if (equal algoritmo ':algoritmo1)
          (setf sai (processa_no_aapu))))))
  (setf pai (car lista_arvore))
  (setf lista_arvore (cdr lista_arvore))
  (setf filhos (get pai 'filhos))
  (setf filhos (retira_inconsistencia filhos))
  (if (eq  (get pai 'consistencia)'sim)
     (setf lista_arvore (append lista_arvore filhos)))
  (if (or (eq lista_arvore nil)(eq sai nil))
     (let ()
        (if (eq sai nil) (print '(nao eh mesmo!))
        (print '(eh uma confutacao2)))
        (return) ))
 (if (> n_nos limite)(let () (setf sai nil)(print '(não sei)) (return))))
  (if (eq sai nil)'(nao eh uma confutacao)'(eh uma confutacao))
'sim))))
```

A função **inicializa_arvore** constrói o tableau inicial. Recebe como parâmetro uma lista de fórmulas contendo as fórmulas da teoria e a negação do possível teorema. Retorna um ponteiro para o tableau inicial ou uma mensagem indicando que o tableau inicial é inconsistente.

```
(defun inicializa_arvore (l algoritmo)
(let (a no b m con tipo form termos_fechados)
    (setf list_var nil)
    (setf b 1)
    (setf a (length l))
    (loop
      (setf no (newatom "no" b))
      (if (> b 1)  (let(vf)
        (setf vf (list no))
        (putt (car m) vf 'filhos)))
      (if (= b 1) (putt no nil 'pai))
      (if (> b 1) (putt no (car m) 'pai))
      (setf form (come_parenteses (car l)))
      (setf form (retira_equivalencia form))
      (setf tipo (ver_tipo form))
      (setf form (retira_vacuo form))
      (if (atom form) (setf form (list form)))
      (setf form (come_parenteses form))
      (if (equal algoritmo ':algoritmo3)
        (putt no (propaga_termos form tipo)'rep)
        (putt no 0 'rep))
      (if (equal algoritmo ':algoritmo4)
        (setf form (marca form 'p)))
        (putt no form 'formula)
      (putt no nil 'n_cte)
      (putt no nil 'l_fec)
      (if (= b a) (putt no nil 'filhos))
```

```
        (putt no nil 'inst)
       (if (= b 1)
        (putt con 'sim 'consist)
        (let()
           (setf con (ver_consist form tipo (car m) algoritmo))))
           (putt no (get con 'consist) 'consistencia)
       (if (eq (get con 'consist) 'nao)
          (return-from inicializa_arvore 'inconsistente))
       (setf m (cons no m))
       (setf l (cdr l))
       (if (= b a) (let ()
           (putt (car m) 1 'n_cte)
           (setf termos_fechados (acha_termos_fechados))
           (putt (car m) termos_fechados 'l_fec)
              (return)))
       (setf b (1+ b)))
(setf m (reverse m))
m))
```

As funções **processa_no_aapu**, **processa_no_apu**, **processa_no_pu** e **processa_no_u**, listadas a seguir, são responsáveis, respectivamente, pela execução do sistema de tableaux tradicional, do primeiro, segundo e terceiro refinamentos ao sistema de tableaux tradicional. Estas funções não recebem parâmetros pois atuam sobre **lista_arvore**, que é uma variável global, cujo primeiro elemento é extraído para expansão. As funções **analisa** e **insere_no_a**, chamadas em cada um destes procedimentos, são responsáveis, respectivamente, pela obtenção de novas fórmulas e inserção destas em ramos abertos descendentes da fórmula expandida. Cada função retorna ao procedimento **tableaux** a variável **sai**, que é um flag informando se uma confutação foi ou não encontrada.

```
(defun processa_no_aapu ()
   (let (var pai tipo folhas ant pos t1 t2 sai nc tf const formu g_tf inst confu tmp
repe1    repe2)
   (setf confu 0)
   (setf tmp 0)
      ;loop para retirar folhas se filhos da lista de processamento dos nos
   (if (eq (get (car lista_arvore) 'filhos) nil)
      (setf folhas (list (car lista_arvore)))
      (setf folhas (acha_folhas lista_arvore)))
   (setf folhas (retira_inconsistencia folhas))
   (if (eq folhas nil)(return-from processa_no_aapu t))
   (setf var (analisa (car lista_arvore)))
   (setf tipo (get var 'tipo))
   (setf pai (car lista_arvore))
   ; se fórmula resulta numa inclusão tipo and
   (if (eq tipo 'a)
      (let (xx xx2 yy yy2 n1 n2)
       (setf ant (get var 'ant))
       (setf pos (get var 'pos))
       (setf t1 (ver_tipo ant))
       (setf t2 (ver_tipo pos))
       (setf repe1 0)
       (setf repe2 0)
      (while folhas
       (setf xx (ver_consist ant t1 (car folhas) ':algoritmo1))
       (setf xx2 (get xx 'consist))
       (setf nc (get (car folhas) 'n_cte))
       (setf tf (get (car folhas) 'l_fec))
       (setf yy nil)
       (setf n1 (insere_no_a ant xx2 tf nc nil (car folhas)repe1))
       (if (eq xx2 'sim)  (let()
             (setf yy (ver_consist pos t2 n1 ':algoritmo1))
```

```
            (setf yy2 (get yy 'consist))
            (setf n2 (insere_no_a pos yy2 tf nc nil n1 repe2))
            (if (and (and (eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
                     (and (eq yy2 'sim)(or (eq t2 'g)(eq t2 'p))))
              (setf sai (ha_formula_expandir (car folhas) pai))
              (setf sai t)))
          (setf sai t))
        (if (eq sai nil)  (let ()
            (print '(nao eh uma confutacao1))
            (return)))
      (setf folhas (cdr folhas)) ) )
  ; se inclusão do tipo or
(if (eq tipo 'b)
      (let (xx xx2 yy yy2)
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (setf t1 (ver_tipo ant))
      (setf t2 (ver_tipo pos))
      (setf repe1 0)
      (setf repe2 0)
     (while folhas
      (setf xx (ver_consist ant t1 (car folhas)':algoritmo1))
      (setf xx2 (get xx 'consist))
      (setf nc (get (car folhas) 'n_cte))
      (setf tf (get (car folhas) 'l_fec))
      (setf yy (ver_consist pos t2 (car folhas)':algoritmo1))
      (setf yy2 (get yy 'consist))
      (insere_no_a ant xx2 tf nc nil (car folhas) repe1)
      (insere_no_a pos yy2 tf nc nil (car folhas)repe2)
      (if (or (and (eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
              (and (eq yy2 'sim)(or (eq t2 'g)(eq t2 'p))))
              (setf sai (ha_formula_expandir (car folhas) pai))
```

```
            (setf sai t))
        (if (eq sai nil) (let ()
            (print '(nao eh uma confutacao1))
            (return)))
        (setf folhas (cdr folhas))) )
    ;se inclusão do tipo (not quantif)
    (if (or (eq tipo 'm)(eq tipo 'n))(let (xx xx2)
        (setf ant (get var 'ant))
        (setf pos (get var 'pos))
        (if (eq tipo 'm) (let ()
            (setf formu (list 'ex (car ant) pos))
            (setf t1 'f))
          (let ()
            (setf formu (list 'qs (car ant) pos))
            (setf t1 'e)))
        (setf repe1 0)
          (while folhas
            (setf xx (ver_consist ant t1 (car folhas)':algoritmo1))
            (setf xx2 (get xx 'consist))
            (setf nc (get (car folhas) 'n_cte))
            (setf tf (get (car folhas) 'l_fec))
            (insere_no_a formu xx2 tf nc nil (car folhas)repe1)
            (setf folhas (cdr folhas)))
          (setf sai t))
    ; se inclusão tipo universal
    (if (eq tipo 'e)(let (n1 ttf xx xx2 atual)
        (setf ant (get var 'ant))
        (setf pos (get var 'pos))
        (setf inst (get pai 'inst))
        (while folhas
            (setf confu 0)
            (setf nc (get (car folhas) 'n_cte))
```

```
            (setf tf (get (car folhas) 'l_fec))
            (if (eq tf nil) (setf tf (list (newatom "c" 0))))
            (setf g_tf (set-difference tf inst))
            (setf tmp nil)
            (setf atual (car folhas))
          (while g_tf
            (setf formu (substitui (car ant) pos (car g_tf)))
            (setf t1 (ver_tipo formu))
            (setf repe1 0)
            (setf xx (ver_consist formu t1 atual ':algoritmo1))
            (setf xx2 (get xx 'consist))
            (setf n1 (insere_no_a formu xx2 nil nc  nil atual repe1))
            (setf tmp (union tmp (acha_fec_f formu t1)))
            (setf atual n1)
            (if (eq xx2 'nao)
              (let ()(setf confu 1)(return))
              (setf confu 0))
            (setf g_tf (cdr g_tf)))
          (if (eq confu 0)(let(inst_new)
            (setf ttf tf)
            (setf tf (union tmp tf))
            (setf tf (reverse tf))
            (setf inst_new (union inst ttf))
   (insere_no_a (get pai 'formula) 'sim tf nc inst_new atual (get pai 'rep))))
          (setf folhas (cdr folhas)) )
        (setf sai t))
   ; se inclusão tipo existencial
  (if (eq tipo 'f)(let ( )
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (if (eq folhas nil) (setf sai t))
      (while folhas
```

```
            (setf nc (get (car folhas) 'n_cte))
            (setf const (newatom "c" nc))
            (setf tf (get (car folhas) 'l_fec))
            (setf formu (substitui (car ant) pos const))
            (setf nc (+ nc 1))
            (setf tf (cons const tf))
            (setf t1 (ver_tipo formu))
            (setf repe1 0)
            (setf tmp (acha_fec_f formu t1))
            (setf tf (union tf tmp))
            (setf tf (reverse tf))
            (insere_no_a formu  'sim tf nc nil (car folhas) repe1)
            (if (or (eq t1 'g)(eq t1 'p))
               (setf sai (ha_formula_expandir (car folhas) pai))
               (setf sai t))
            (setf folhas (cdr folhas))))
 ; se nao inclui ou inclusão tipo not not
(if (or (eq tipo 'g) (eq tipo 'p)(eq tipo 'o))  (let ()
    (setf ant (get var 'ant))
    (if (eq tipo 'o); se inclusão tipo not not
     (let (xx xx2)
        (setf ant (get var 'ant))
        (setf t1 (ver_tipo ant))
        (setf repe1 0)
        (while folhas
         (setf xx (ver_consist ant t1 (car folhas)':algoritmo1))
         (setf xx2 (get xx 'consist))
         (setf nc (get (car folhas) 'n_cte))
         (setf tf (get (car folhas) 'l_fec))
         (insere_no_a ant xx2 tf nc nil(car folhas) repe1)
          (if (and (eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
             (setf sai (ha_formula_expandir (car folhas) pai))
```

```
                (setf sai t))
              (if (eq sai nil)  (let ()
                 (print '(nao eh uma confutacao3))
                 (return)))
              (setf folhas (cdr folhas))))
           (setf sai t);caso de erro
        ) )))))))
    (return-from processa_no_aapu sai);sai t continua o processamento
                                      ;sai nil acho ramo aberto (nao é confutacao)
))


(defun processa_no_apu ()
    (let (var pai  tipo folhas ant pos t1 t2 sai nc tf  const  formu  g_tf  inst confu  tmp
repe1 repe2)
    (setf confu 0)
    (setf tmp 0)
       ;loop para retirar folhas se filhos da lista de processamento dos nos
    (if (eq (get (car lista_arvore) 'filhos) nil)
        (setf folhas (list (car lista_arvore)))
        (setf folhas (acha_folhas lista_arvore)))
    (setf folhas (retira_inconsistencia folhas))
    (if (eq folhas nil)(return-from processa_no_apu t))
    (setf var (analisa (car lista_arvore)))
    (setf tipo (get var 'tipo))
    (setf pai (car lista_arvore))
     ; se fórmula resulta numa inclusão tipo and
    (if (eq tipo 'a)
       (let (xx xx1 xx2 yy yy1 yy2 inseriu n1 n2)
        (setf ant (get var 'ant))
        (setf pos (get var 'pos))
        (setf t1 (ver_tipo ant))
```

```
 (setf t2 (ver_tipo pos))
 (setf repe1 0)
 (setf repe2 0)
(while folhas
 (setf inseriu 0)
 (setf xx (ver_consist ant t1 (car folhas) ':algoritmo2))
 (setf xx1 (get xx 'igual))
 (setf xx2 (get xx 'consist))
 (setf nc (get (car folhas) 'n_cte))
 (setf tf (get (car folhas) 'l_fec))
 (setf yy nil)
 (if (eq xx1 'nao) (let ()
     (setf n1 (insere_no_a ant xx2 tf nc nil (car folhas)repe1))
     (setf inseriu 1)
     (if (eq xx2 'sim)
       (setf yy (ver_consist pos t2 n1 ':algoritmo2))))
    (setf yy (ver_consist pos t2 (car folhas)':algoritmo2)))
 (if (eq xx2 'sim)(let ()
 (setf yy1 (get yy 'igual))
 (setf yy2 (get yy 'consist))
 (if (eq yy1 'nao)(let()
   (if (eq inseriu 1)
     (setf n2 (insere_no_a pos yy2 tf nc nil n1 repe2))
     (setf n2 (insere_no_a pos yy2 tf nc nil (car folhas)repe2)))
     (setf inseriu 2)))))
 (if (and (eq inseriu 1)
       (eq xx2 'sim)
       (or (eq t1 'g)(eq t1 'p)))
          (setf sai (ha_formula_expandir (car folhas) pai))
          ;else
           (if (and (eq inseriu 2)
                 (eq yy2 'sim)
```

```
                    (or (eq t2 'g)(eq t2 'p)))
                (setf sai (ha_formula_expandir (get n2 'pai) pai))
                    ;else
                    (if (eq inseriu 0)
                (setf sai (ha_formula_expandir (car folhas) pai))
                    (setf sai t))))
        (if (eq sai nil)  (let ()
            (print '(nao eh uma confutacao1))
            (return)))
        (setf folhas (cdr folhas))))
  ; se inclusão do tipo or
(if (eq tipo 'b)
     (let (xx xx1 xx2 yy yy1 yy2 inseriu1 inseriu2 )
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (setf t1 (ver_tipo ant))
      (setf t2 (ver_tipo pos))
      (setf repe1 0)
      (setf repe2 0)
     (while folhas
      (setf xx (ver_consist ant t1 (car folhas)':algoritmo2))
      (setf xx1 (get xx 'igual))
      (setf xx2 (get xx 'consist))
      (setf inseriu1 0)
      (setf inseriu2 0)
      (setf nc (get (car folhas) 'n_cte))
      (setf tf (get (car folhas) 'l_fec))
      (if (not (eq ant pos)) (let ()
          (setf yy (ver_consist pos t2 (car folhas)':algoritmo2))
          (setf yy1 (get yy 'igual))
          (setf yy2 (get yy 'consist))
      (if (and (eq xx1 'nao)(eq yy1 'nao)) (let ()
```

```
            (setf inseriu1 1)
            (insere_no_a ant xx2 tf nc nil (car folhas)repe1)
            (setf inseriu2 1)
            (insere_no_a pos yy2 tf nc nil (car folhas)repe2) )))
          (if (eq xx1 'nao)  (let ()
                (setf inseriu1 1)
                (insere_no_a ant xx2 tf nc nil (car folhas)repe1))))
        (if (or (and (eq inseriu1 1)(eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
              (and (eq inseriu2 1)(eq yy2 'sim)(or (eq t2 'g)(eq t2 'p))))
              (setf sai (ha_formula_expandir (car folhas) pai))
              ;else
              (if (or (eq inseriu1 0)(eq inseriu2 0))
                (setf sai (ha_formula_expandir (car folhas) pai))
                ;else
                (setf sai t)))
         (if (eq sai nil)  (let ()
              (print '(nao eh uma confutacao1))
              (return)))
        (setf folhas (cdr folhas)) ))
;se inclusão do tipo (not quantif)
   (if (or (eq tipo 'm)(eq tipo 'n))(let (xx xx1 xx2)
        (setf ant (get var 'ant))
        (setf pos (get var 'pos))
        (if (eq tipo 'm)  (let ()
              (setf formu (list 'ex (car ant) pos))
              (setf t1 'f))
            (let ()
              (setf formu (list 'qs (car ant) pos))
              (setf t1 'e)))
        (setf repe1 0)
          (while folhas
              (setf xx (ver_consist ant t1 (car folhas)':algoritmo2))
```

```
            (setf xx1 (get xx 'igual))
            (setf xx2 (get xx 'consist))
            (setf nc (get (car folhas) 'n_cte))
            (setf tf (get (car folhas) 'l_fec))
            (if (eq xx1 'nao)
              (insere_no_a formu xx2 tf nc nil (car folhas)repe1))
            (setf folhas (cdr folhas)))
        (setf sai t))
; se inclusão tipo universal
(if (eq tipo 'e)(let (n1 ttf xx xx1 xx2 atual)
    (setf ant (get var 'ant))
    (setf pos (get var 'pos))
    (setf inst (get pai 'inst))
    (while folhas
            (setf confu 0)
            (setf nc (get (car folhas) 'n_cte))
            (setf tf (get (car folhas) 'l_fec))
            (if (eq tf nil) (setf tf (list (newatom "c" 0))))
            (setf g_tf (set-difference tf inst))
            (setf tmp nil)
            (setf atual (car folhas))
        (while g_tf
            (setf formu (substitui (car ant) pos (car g_tf)))
            (setf t1 (ver_tipo formu))
            (setf repe1 0)
            (setf xx (ver_consist formu t1 atual ':algoritmo2))
            (setf xx1 (get xx 'igual))
            (setf xx2 (get xx 'consist))
            (if (eq xx1 'nao)(let ()
             (setf n1 (insere_no_a formu xx2 nil nc  nil atual repe1))
              (setf tmp (union tmp (acha_fec_f formu t1)))
              (setf atual n1)))
```

```
            (if (eq xx2 'nao)
              (let ()(setf confu 1)(return))
              (setf confu 0))
            (setf g_tf (cdr g_tf)))
          (if (eq confu 0)(let(inst_new)
            (setf ttf tf)
            (setf tf (union tmp tf))
            (setf tf (reverse tf))
            (setf inst_new (union inst ttf))
            (insere_no_a (get pai 'formula) 'sim tf nc inst_new atual (get pai 'rep))))
          (setf folhas (cdr folhas)))
        (setf sai t))
  ; se inclusão tipo existencial
  (if (eq tipo 'f)(let ()
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (if (eq folhas nil) (setf sai t))
      (while folhas
            (setf nc (get (car folhas) 'n_cte))
            (setf const (newatom "c" nc))
            (setf tf (get (car folhas) 'l_fec))
            (setf formu (substitui (car ant) pos const))
            (setf nc (+ nc 1))
            (setf tf (cons const tf))
            (setf t1 (ver_tipo formu))
            (setf repe1 0)
            (setf tmp (acha_fec_f formu t1))
            (setf tf (union tf tmp))
            (setf tf (reverse tf))
            (insere_no_a formu  'sim tf nc nil (car folhas) repe1)
            (if (or (eq t1 'g)(eq t1 'p))
              (setf sai (ha_formula_expandir (car folhas) pai))
```

```
            ;else
            (setf sai t))
          (setf folhas (cdr folhas))))
; se nao inclui ou inclusão tipo not not
(if (or (eq tipo 'g) (eq tipo 'p)(eq tipo 'o))
  (let ()
    (setf ant (get var 'ant))
    (if (eq tipo 'o); se inclusão tipo not not
     (let (xx xx1 xx2 inseriu)
       (setf ant (get var 'ant))
       (setf t1 (ver_tipo ant))
       (setf repe1 0)
       (while folhas
        (setf xx (ver_consist ant t1 (car folhas)':algoritmo2))
        (setf xx1 (get xx 'igual))
        (setf xx2 (get xx 'consist))
        (setf inseriu 0)
        (setf nc (get (car folhas) 'n_cte))
        (setf tf (get (car folhas) 'l_fec))
        (if (eq xx1 'nao)(let ()
           (insere_no_a ant xx2 tf nc nil(car folhas) repe1)
           (setf inseriu 1)))
       (if (and (eq inseriu 1)(eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
           (setf sai (ha_formula_expandir (car folhas) pai))
           ;else
           (if (eq inseriu 0)
              (setf sai (ha_formula_expandir (car folhas) pai))
              (setf sai t)))
        (if (eq sai nil)  (let ()
           (print '(nao eh uma confutacao3))
           (return)))
        (setf folhas (cdr folhas))))
```

```
        (setf sai t);caso de erro
      ))))))))
(return-from processa_no_apu sai);sai t continua o processamento
                        ;sai nil acho ramo aberto (nao é confutacao)
))


(defun processa_no_pu ()
   (let (var  pai tipo folhas ant pos t1 t2 sai  nc tf  const formu g_tf  inst confu tmp
repe1 repe2)
  (setf confu 0)
   (setf tmp 0)
     ;loop para retirar folhas se filhos da lista de processamento dos nos
   (if (eq (get (car lista_arvore) 'filhos) nil)
      (setf folhas (list (car lista_arvore)))
      (setf folhas (acha_folhas lista_arvore)))
   (setf folhas (retira_inconsistencia folhas))
   (if (eq folhas nil)(return-from processa_no_pu t))
   (setf var (analisa (car lista_arvore)))
   (setf tipo (get var 'tipo))
   (setf pai (car lista_arvore))
    ; se fórmula resulta numa inclusão tipo and
   (if (eq tipo 'a)
     (let (xx xx1 xx2 yy yy1 yy2 inseriu n1 n2)
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (setf t1 (ver_tipo ant))
      (setf t2 (ver_tipo pos))
      (setf repe1 (propaga_termos ant t1))
      (setf repe2 (propaga_termos pos t2))
     (while folhas
      (setf inseriu 0)
```

```
(setf xx (ver_consist ant t1 (car folhas) ':algoritmo3))
(setf xx1 (get xx 'igual))
(setf xx2 (get xx 'consist))
(setf nc (get (car folhas) 'n_cte))
(setf tf (get (car folhas) 'l_fec))
(setf yy nil)
(if (eq xx1 'nao)  (let ()
    (setf n1 (insere_no_a ant xx2 tf nc nil (car folhas)repe1))
    (setf inseriu 1)
    (if (eq xx2 'sim)
      (setf yy (ver_consist pos t2 n1 ':algoritmo3))))
   (setf yy (ver_consist pos t2 (car folhas)':algoritmo3)))
(if (eq xx2 'sim)(let ()
(setf yy1 (get yy 'igual))
(setf yy2 (get yy 'consist))
(if (eq yy1 'nao)(let()
  (if (eq inseriu 1)
     (setf n2 (insere_no_a pos yy2 tf nc nil n1 repe2))
     (setf n2 (insere_no_a pos yy2 tf nc nil (car folhas)repe2)))
     (setf inseriu 2)))))
(if (and (eq inseriu 1)
       (eq xx2 'sim)
       (or (eq t1 'g)(eq t1 'p)))
          (setf sai (ha_formula_expandir (car folhas) pai))
          ;else
           (if (and (eq inseriu 2)
                    (eq yy2 'sim)
                    (or (eq t2 'g)(eq t2 'p)))
              (setf sai (ha_formula_expandir (get n2 'pai) pai))
                    ;else
                    (if (eq inseriu 0)
              (setf sai (ha_formula_expandir (car folhas) pai))
```

```
                    (setf sai t))))
        (if (eq sai nil)  (let ()
            (print '(nao eh uma confutacao1))
            (return)))
        (setf folhas (cdr folhas)) ))
  ; se inclusão do tipo or
(if (eq tipo 'b)
      (let (xx xx1 xx2 yy yy1 yy2 inseriu1 inseriu2)
       (setf ant (get var 'ant))
       (setf pos (get var 'pos))
       (setf t1 (ver_tipo ant))
       (setf t2 (ver_tipo pos))
       (setf repe1 (propaga_termos ant t1))
       (setf repe2 (propaga_termos pos t2))
      (while folhas
       (setf xx (ver_consist ant t1 (car folhas)':algoritmo3))
       (setf xx1 (get xx 'igual))
       (setf xx2 (get xx 'consist))
       (setf inseriu1 0)
       (setf inseriu2 0)
       (setf nc (get (car folhas) 'n_cte))
       (setf tf (get (car folhas) 'l_fec))
       (if (not (eq ant pos))  (let ()
            (setf yy (ver_consist pos t2 (car folhas)':algoritmo3))
            (setf yy1 (get yy 'igual))
            (setf yy2 (get yy 'consist))
       (if (and (eq xx1 'nao)(eq yy1 'nao)) (let ()
            (setf inseriu1 1)
            (insere_no_a ant xx2 tf nc nil (car folhas)repe1)
            (setf inseriu2 1)
            (insere_no_a pos yy2 tf nc nil (car folhas)repe2))))
           (if (eq xx1 'nao)  (let ()
```

```
                (setf inseriu1 1)
                (insere_no_a ant xx2 tf nc nil (car folhas)repe1))))
          (if (or (and (eq inseriu1 1)(eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
                (and (eq inseriu2 1)(eq yy2 'sim)(or (eq t2 'g)(eq t2 'p))))
                (setf sai (ha_formula_expandir (car folhas) pai))
                ;else
                (if (or (eq inseriu1 0)(eq inseriu2 0))
                  (setf sai (ha_formula_expandir (car folhas) pai))
                  ;else
                  (setf sai t)))
           (if (eq sai nil)  (let ()
                (print '(nao eh uma confutacao1))
                (return)))
          (setf folhas (cdr folhas))))
;se inclusão do tipo (not quantif)
    (if (or (eq tipo 'm)(eq tipo 'n))(let (xx xx1 xx2
          (setf ant (get var 'ant))
          (setf pos (get var 'pos))
          (if (eq tipo 'm)  (let ()
              (setf formu (list 'ex (car ant) pos))
              (setf t1 'f))
            (let ()
              (setf formu (list 'qs (car ant) pos))
              (setf t1 'e)))
          (setf repe1 (propaga_termos formu t1))
            (while folhas
                (setf xx (ver_consist ant t1 (car folhas)':algoritmo3))
                (setf xx1 (get xx 'igual))
                (setf xx2 (get xx 'consist))
                (setf nc (get (car folhas) 'n_cte))
                (setf tf (get (car folhas) 'l_fec))
                (if (eq xx1 'nao)
```

```
                (insere_no_a formu xx2 tf nc nil (car folhas)repe1))
            (setf folhas (cdr folhas)))
        (setf sai t))
; se inclusão tipo universal
(if (eq tipo 'e)(let (n1 ttf xx xx1 xx2 atual)
    (setf ant (get var 'ant))
    (setf pos (get var 'pos))
    (setf inst (get pai 'inst))
    (while folhas
        (setf confu 0)
        (setf nc (get (car folhas) 'n_cte))
        (setf tf (get (car folhas) 'l_fec))
        (if (eq tf nil) (setf tf (list (newatom "c" 0))))
        (setf g_tf (set-difference tf inst))
        (setf tmp nil)
        (setf atual (car folhas))
      (while g_tf
        (setf formu (substitui (car ant) pos (car g_tf)))
        (setf t1 (ver_tipo formu))
        (setf repe1 (propaga_termos formu t1))
        (setf xx (ver_consist formu t1 atual ':algoritmo3))
        (setf xx1 (get xx 'igual))
        (setf xx2 (get xx 'consist))
        (if (eq xx1 'nao)(let ()
          (setf n1 (insere_no_a formu xx2 nil nc  nil atual repe1))
          (setf tmp (union tmp (acha_fec_f formu t1)))
          (setf atual n1)))
        (if (eq xx2 'nao)
          (let ()(setf confu 1)(return))
          (setf confu 0))
        (setf g_tf (cdr g_tf)))
      (if (eq confu 0)(let(inst_new v_rep)
```

```
        (setf v_rep (ver_repeticao atual pai))
        (setf ttf tf)
        (setf tf (union tmp tf))
        (setf tf (reverse tf))
        (setf inst_new (union inst ttf))
        (if (or (eq v_rep t)(not (eq (get pai 'rep) 0))) (let()
          (insere_no_a (get pai 'formula) 'sim tf nc inst_new atual (get pai 'rep)))
          (let ()
             (putt atual tf 'l_fec)
             (putt atual nc 'n_cte)))))
      (setf folhas (cdr folhas)))
    (setf sai t))
; se inclusão tipo existencial
(if (eq tipo 'f)(let ()
    (setf ant (get var 'ant))
    (setf pos (get var 'pos))
    (if (eq folhas nil) (setf sai t))
    (while folhas
        (setf nc (get (car folhas) 'n_cte))
        (setf const (newatom "c" nc))
        (setf tf (get (car folhas) 'l_fec))
        (setf formu (substitui (car ant) pos const))
        (setf nc (+ nc 1))
        (setf tf (cons const tf))
        (setf t1 (ver_tipo formu))
        (setf repe1 (propaga_termos formu t1))
        (setf tmp (acha_fec_f formu t1))
        (setf tf (union tf tmp))
        (setf tf (reverse tf))
        (insere_no_a formu  'sim tf nc nil (car folhas) repe1)
        (if (or (eq t1 'g)(eq t1 'p))
          (setf sai (ha_formula_expandir (car folhas) pai))
```

```
            ;else
            (setf sai t))
          (setf folhas (cdr folhas))))
; se nao inclui ou inclusão tipo not not
(if (or (eq tipo 'g) (eq tipo 'p)(eq tipo 'o))  (let ()
    (setf ant (get var 'ant))
    (if (eq tipo 'o); se inclusão tipo not not
     (let (xx xx1 xx2 inseriu )
       (setf ant (get var 'ant))
       (setf t1 (ver_tipo ant))
       (setf repe1 (propaga_termos ant t1))
       (while folhas
        (setf xx (ver_consist ant t1 (car folhas)':algoritmo3))
        (setf xx1 (get xx 'igual))
        (setf xx2 (get xx 'consist))
        (setf inseriu 0)
        (setf nc (get (car folhas) 'n_cte))
        (setf tf (get (car folhas) 'l_fec))
        (if (eq xx1 'nao)(let ()
          (insere_no_a ant xx2 tf nc nil(car folhas) repe1)
          (setf inseriu 1)))
       (if (and (eq inseriu 1)(eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
            (setf sai (ha_formula_expandir (car folhas) pai))
            ;else
            (if (eq inseriu 0)
              (setf sai (ha_formula_expandir (car folhas) pai))
              (setf sai t)))
         (if (eq sai nil) (let ()
            (print '(nao eh uma confutacao3))
            (return)))
         (setf folhas (cdr folhas))  ))
      (setf sai t);caso de erro
```

```
        ))))))))
    (return-from processa_no_pu sai);sai t continua o processamento
                        ;sai nil acho ramo aberto (nao é confutacao)
))


(defun processa_no ()
    (let (var pai tipo folhas ant pos t1 t2 sai nc tf  const formu g_tf  inst confu  tmp repe1
repe2     flag tf_todos  inst_new)
    (setf confu 0)
    (setf tmp 0)
      ;loop para retirar folhas se filhos da lista de processamento dos nos
    (if (eq (get (car lista_arvore) 'filhos) nil)
       (setf folhas (list (car lista_arvore)))
       (setf folhas (acha_folhas lista_arvore)))
    (setf folhas (retira_inconsistencia folhas))
    (if (eq folhas nil)(return-from processa_no t))
    (setf var (analisa (car lista_arvore)))
    (setf tipo (get var 'tipo))
    (setf pai (car lista_arvore))
     ; se fórmula resulta numa inclusão tipo and
    (if (eq tipo 'a)
      (let (xx xx1 xx2 yy yy1 yy2 inseriu n1 n2)
       (setf ant (get var 'ant))
       (setf pos (get var 'pos))
       (setf t1 (ver_tipo ant))
       (setf t2 (ver_tipo pos))
       (setf repe1 0)
       (setf repe2 0)
      (while folhas
       (setf inseriu 0)
       (setf xx (ver_consist ant t1 (car folhas) ':algoritmo4))
```

```
(setf xx1 (get xx 'igual))
(setf xx2 (get xx 'consist))
(setf nc (get (car folhas) 'n_cte))
(setf tf (get (car folhas) 'l_fec))
(setf yy nil)
(if (eq xx1 'nao)  (let ()
    (setf n1 (insere_no_a ant xx2 tf nc nil (car folhas)repe1))
    (setf inseriu 1)
    (if (eq xx2 'sim)
      (setf yy (ver_consist pos t2 n1 ':algoritmo4))))
   (setf yy (ver_consist pos t2 (car folhas)':algoritmo4)))
(if (eq xx2 'sim)(let ()
  (setf yy1 (get yy 'igual))
  (setf yy2 (get yy 'consist))
(if (eq yy1 'nao) (let()
   (if (eq inseriu 1)
     (setf n2 (insere_no_a pos yy2 tf nc nil n1 repe2))
     (setf n2 (insere_no_a pos yy2 tf nc nil (car folhas) repe2)))
     (setf inseriu 2)))))
(if (and (eq inseriu 1)
       (eq xx2 'sim)
       (or (eq t1 'g)(eq t1 'p)))
          (setf sai (ha_formula_expandir (car folhas) pai))
          ;else
           (if (and (eq inseriu 2)
                    (eq yy2 'sim)
                    (or (eq t2 'g)(eq t2 'p)))
              (setf sai (ha_formula_expandir (get n2 'pai) pai))
                    ;else
                    (if (eq inseriu 0)
               (setf sai (ha_formula_expandir (car folhas) pai))
                    (setf sai t))))
```

```
        (if (eq sai nil)  (let ()
              (print '(nao eh uma confutacao1))
              (return)))
        (setf folhas (cdr folhas))))
  ; se inclusão do tipo or
(if (eq tipo 'b)
      (let (xx xx1 xx2 yy yy1 yy2 inseriu1 inseriu2)
       (setf ant (get var 'ant))
       (setf pos (get var 'pos))
       (setf t1 (ver_tipo ant))
       (setf t2 (ver_tipo pos))
       (setf repe1 0)
       (setf repe2 0)
      (while folhas
       (setf xx (ver_consist ant t1 (car folhas)':algoritmo4))
       (setf xx1 (get xx 'igual))
       (setf xx2 (get xx 'consist))
       (setf inseriu1 0)
       (setf inseriu2 0)
       (setf nc (get (car folhas) 'n_cte))
       (setf tf (get (car folhas) 'l_fec))
       (if (not (eq ant pos)) (let ()
            (setf yy (ver_consist pos t2 (car folhas)':algoritmo4))
            (setf yy1 (get yy 'igual))
            (setf yy2 (get yy 'consist))
       (if (and (eq xx1 'nao)(eq yy1 'nao)) (let ()
            (setf inseriu1 1)
            (insere_no_a ant xx2 tf nc nil (car folhas)repe1)
            (setf inseriu2 1)
            (insere_no_a pos yy2 tf nc nil (car folhas)repe2))))
          (if (eq xx1 'nao) (let ()
               (setf inseriu1 1)
```

```
            (insere_no_a ant xx2 tf nc nil (car folhas)repe1))))
      (if (or (and (eq inseriu1 1)(eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
            (and (eq inseriu2 1)(eq yy2 'sim)(or (eq t2 'g)(eq t2 'p))))
            (setf sai (ha_formula_expandir (car folhas) pai))
            ;else
            (if (or (eq inseriu1 0)(eq inseriu2 0))
               (setf sai (ha_formula_expandir (car folhas) pai))
               ;else
               (setf sai t)))
       (if (eq sai nil)  (let ()
            (print '(nao eh uma confutacao1))
            (return)))
      (setf folhas (cdr folhas))))
;se inclusão do tipo (not quantif)
(if (or (eq tipo 'm)(eq tipo 'n))(let (xx xx1 xx2)
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (if (eq tipo 'm) (let ()
            (setf formu (list 'ex (car ant) pos))
            (setf t1 'f))
         (let ()
            (setf formu (list 'qs (car ant) pos))
            (setf t1 'e)))
      (setf repe1 0)
        (while folhas
            (setf xx (ver_consist formu t1 (car folhas)':algoritmo4))
            (setf xx1 (get xx 'igual))
            (setf xx2 (get xx 'consist))
            (setf nc (get (car folhas) 'n_cte))
            (setf tf (get (car folhas) 'l_fec))
            (if (eq xx1 'nao)
               (insere_no_a formu xx2 tf nc nil (car folhas)repe1))
```

```
            (setf folhas (cdr folhas)))
        (setf sai t))
  ;se inclusão tipo universal
 (if (eq tipo 'e)(let (n1 n2 ttf xx xx1 xx2 atual prim_rep la1 flag_aqf
         lcs3_tmp lcs0 lcs1 lcs3); listas de constantes para substituição
      (setf ant (get var 'ant))
      (setf pos (get var 'pos))
      (setf inst (get pai 'inst))
      (setf prim_rep inst)
      (acha_atomicas (get pai 'formula))
      (setf la1 l_a1)
      (setf l_a1 nil)
      ;unifica com fórmula já marcadas somente na primeira repetição
      (if (eq prim_rep nil) (let ()
          (setf lcs1 (unific_sup pai la1 ant))
          (setf lcs0 (extrai_const lcs1 ant)))
         (setf lcs0 nil))
      (while folhas
            (setf lcs3_tmp (unific_inf pai (car folhas) la1 ant))
            (setf lcs3 (car (cdr lcs3_tmp)))
            (setf flag_aqf (car lcs3_tmp))
            (setf lcs0 (append lcs0 (extrai_const lcs3 ant)))
            (setf lcs0 (remove-duplicates lcs0 :test #'equal))
            (setf confu 0)
            (setf flag (rep_universal lcs3; lista de unificações
                  lcs0 ; constantes que unificam
                  inst ;constantes já instanciadas
                  flag_aqf ; unificação secundária
                  ))
            (setf nc (get (car folhas) 'n_cte))
            (setf tf_todos (get (car folhas) 'l_fec))
            (setf tf lcs0)
```

```
   (if (and (eq tf nil) (eq prim_rep nil))  (let()
         (setf tf (list (newatom "c" 1)))
         (if (eq nc 1)(setf nc (+ nc 1)))))
   (setf g_tf (set-difference tf inst))
   (setf inst_new (union g_tf inst))
   (setf inst_new (remove-duplicates inst_new :test #'equalp))
   (setf tmp nil)
   (setf atual (car folhas))
  (if (eq flag t)  (let()
     (setf n2 (insere_no_a (get pai 'formula) 'sim tf_todos
          nc inst_new atual (get pai 'rep)))
       (setf atual n2)))
(while g_tf
   (setf formu (substitui (car ant) pos (car g_tf)))
   (setf formu (troca_var formu))
   (setf t1 (ver_tipo formu))
   (setf repe1 0)
   (setf xx (ver_consist formu t1 atual ':algoritmo4))
   (setf xx1 (get xx 'igual))
   (setf xx2 (get xx 'consist))
   (if (eq xx1 'nao)(let ()
     (setf n1 (insere_no_a formu xx2 nil nc  nil atual repe1))
      (setf tmp (union tmp (acha_fec_f formu t1)))
      (setf atual n1)))
   (if (eq xx2 'nao)
     (let ()(setf confu 1)(return))
     (setf confu 0))
   (setf g_tf (cdr g_tf)))
(if (eq confu 0)(let()
   (setf ttf (union tmp tf_todos))
   (setf ttf (reverse ttf))
   (setf ttf (remove-duplicates ttf :test #'equalp))
```

```
            (putt atual ttf 'l_fec)
            (putt atual nc 'n_cte)))
          (setf folhas (cdr folhas)))
        (setf sai t))
 ; se inclusão tipo existencial
(if (eq tipo 'f)(let ()
    (setf ant (get var 'ant))
    (setf pos (get var 'pos))
    (if (eq folhas nil) (setf sai t))
    (while folhas
          (setf nc (get (car folhas) 'n_cte))
          (setf const (newatom "c" nc))
          (setf tf (get (car folhas) 'l_fec))
          (setf formu (substitui (car ant) pos const))
          (setf nc (+ nc 1))
          (setf tf (cons const tf))
          (setf t1 (ver_tipo formu))
          (setf repe1 0)
          (setf tmp (acha_fec_f formu t1))
          (setf tf (union tf tmp))
          (setf tf (reverse tf))
          (if (or (eq t1 'g)(eq t1 'p))
             (setf sai (ha_formula_expandir (car folhas) pai))
             ;else
             (setf sai t))
          (setf folhas (cdr folhas))))
 ; se nao inclui ou inclusão tipo not not
(if (or (eq tipo 'g) (eq tipo 'p)(eq tipo 'o))  (let ()
    (setf ant (get var 'ant))
    (if (eq tipo 'o); se inclusão tipo not not
     (let (xx xx1 xx2 inseriu)
        (setf ant (get var 'ant))
```

```
        (setf t1 (ver_tipo ant))
        (setf repe1 0)
        (while folhas
         (setf xx (ver_consist ant t1 (car folhas)':algoritmo4))
         (setf xx1 (get xx 'igual))
         (setf xx2 (get xx 'consist))
         (setf inseriu 0)
         (setf nc (get (car folhas) 'n_cte))
         (setf tf (get (car folhas) 'l_fec))
         (if (eq xx1 'nao)(let ()
             (insere_no_a ant xx2 tf nc nil(car folhas) repe1)
            (setf inseriu 1)))
        (if (and (eq inseriu 1)(eq xx2 'sim)(or (eq t1 'g)(eq t1 'p)))
             (setf sai (ha_formula_expandir (car folhas) pai))
             ;else
             (if (eq inseriu 0)
                (setf sai (ha_formula_expandir (car folhas) pai))
                (setf sai t)))
          (if (eq sai nil) (let ()
              (print '(nao eh uma confutacao3))
              (return)))
          (setf folhas (cdr folhas)) ))
       (setf sai t);caso de erro
    ) )))))))
  (return-from processa_no sai);sai t continua o processamento
                    ;sai nil acho ramo aberto (nao é confutacao)
))
```

A função **retira_inconsistencia** retira os nós que são folhas de ramos fechados da lista das folhas que receberão novos filhos.

```
(defun retira_inconsistencia (folhas)
(let (ret)
  (while folhas
    (if (eq (get (car folhas) 'consistencia) 'sim)
      (setf ret (append ret (list (car folhas)))))
    (setf folhas (cdr folhas)))
  (return-from retira_inconsistencia ret)))
```

A função **newatom** cria um novo átomo, composto de uma parte alfabética (**no**) e uma parte numérica (**b**).

```
(defun newatom (no b)
   (intern (concatenate 'simple-string no (write-to-string b))))
```

A função **putt** atribui valor (**value**) a uma propriedade (**property**) de um determinado átomo (**atom**).

```
(defun putt (atom value property)
        (setf (get atom property) value))
```

A função **come_parenteses** retira os parêntese excedentes de uma determinada fórmula.

```
(defun come_parenteses (exp)
 (let (tam)
  (if (atom exp)
   (return-from come_parenteses exp))
  (setf tam (length exp))
  (while (and (= tam 1) (listp (nth 0 exp)) (not (atom (nth 0 exp))))
```

```
    (let ()
      (setf exp (car exp))
      (setf tam (length exp))))
  (return-from come_parenteses exp)))
```

A função **retira_equivalência** elimina das fórmulas, no momento da construção do tableaux inicial, o conetivo de bi-implicação.

```
(defun retira_equivalencia (formula)(let (a b forma sn)
    (if (listp formula)(let()
        (setf formula (come_parenteses formula))
        (setf forma (ver_tipo formula)))
      (let()
        (setf formula (list formula))
        (setf forma (ver_tipo formula))))
    (if (eq forma 'o)
        (list 'n (list 'n (retira_equivalencia (cdr (car (cdr formula))))))
    (if (eq forma 'p)
      (list 'n (retira_equivalencia(cdr formula)))
    (if (eq forma 'g)(let (tam)
      (setf tam (length formula))
      (if (= tam 1)(car formula) formula))
    (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c))(let ()
      (setf a (second formula))
      (setf b (third formula))
      (setf sn (first formula))
      (list sn (retira_equivalencia a) (retira_equivalencia b)))
   (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k)(eq forma 'm) (eq forma
'n))
      (list 'n (retira_equivalencia (car (cdr formula))))
   (if (or (eq forma 'e) (eq forma 'f))(let(x)
```

```
    (setf a (third formula))
    (setf x (second formula))
    (setf sn (first formula))
    (list sn x (retira_equivalencia a )))
  (if (eq forma 'd)(let(aa bb aaa bbb)
    (setf a (second formula))
    (setf aa (retira_equivalencia a))
    (if  (and (listp aa)(eq (length aa) 1)
        (not (eq (car aa) 'e))
        (not (eq (car aa) 'ou))
        (not (eq (car aa) 'qs))
        (not (eq (car aa) 'ex))
        (not (eq (car aa) 'n))
        (not (eq (car aa) 'ent))
        (not (eq (car aa) 'eq)))
         (setf aaa (cons 'n aa))(setf aaa (list 'n aa)))
    (setf b (third formula))
    (setf bb (retira_equivalencia b))
    (if  (and (listp bb)(eq (length bb) 1)
        (not (eq (car bb) 'e))
        (not (eq (car bb) 'ou))
        (not (eq (car bb) 'qs))
        (not (eq (car bb) 'ex))
        (not (eq (car bb) 'n))
        (not (eq (car bb) 'ent))
        (not (eq (car bb) 'eq)))
        (setf bbb (cons 'n bb))
         (setf bbb (list 'n bb)))
    (list 'ou (list 'e aa bb)(list 'e aaa bbb)))))))))))
```

A função **ver_tipo** retorna tipo da formula, que é recebida como parâmetro. O tipo pode ser um dos valores abaixo:

- a – fórmula é uma conjunção;
- b – fórmula é uma disjunção;
- c – fórmula é uma implicação;
- d – fórmula é uma bi-implicação;
- e – fórmula é quantificada universalmente;
- f – fórmula é quantificada existencialmente;
- g – fórmula é atômica;
- h – fórmula é uma negação de conjunção;
- i – fórmula é uma negação de disjunção;
- j – fórmula é uma negação de implicação;
- k – fórmula é uma negação de bi-implicação;
- m – fórmula é uma negação de fórmula universal;
- n – fórmula é uma negação de fórmula existencial;
- o – fórmula é uma dupla negação;
- p – fórmula é uma negação de fórmula atômica.

```
(defun ver_tipo (exp)
 (let (tam tipo seg)
   (if (atom exp)(let ()(setf tipo 'g)  (return-from ver_tipo tipo)))
   (setf tam (length exp))
   ;1 se começa com n
    (if (eq (car exp) 'n)
;entao1  se tem tamanho 2 e o elemento 2 eh lista
       (if (and (eq tam 2)(listp (car (cdr exp))))
          ;entao2
          (let ()
             (setf seg (car (cdr exp)))
             (if (eq (car seg) 'e)
                (setf tipo 'h)
                (if (eq (car seg) 'ou)
```

```
                (setf tipo 'i)
                (if (eq (car seg) 'ent)
                   (setf tipo 'j)
                   (if (eq (car seg) 'eq)
                      (setf tipo 'k)
                      (if (eq (car seg) 'qs)
                         (setf tipo 'm)
                         (if (eq (car seg) 'ex)
                            (setf tipo 'n)
                            (if (eq (car seg) 'n)
                               (setf tipo 'o)(setf tipo 'p)))))))))
        ;senao2
        (setf tipo 'p))
     ;senao1
     (if (eq tam 3)
        ;entao4
        (if (eq (car exp) 'e)
           (setf tipo 'a)
           (if (eq (car exp) 'ou)
              (setf tipo 'b)
              (if (eq (car exp) 'ent)
                 (setf tipo 'c)
                 (if (eq (car exp) 'eq)
                    (setf tipo 'd)
                    (if (eq (car exp) 'qs)
                       (setf tipo 'e)
                       (if (eq (car exp) 'ex)
                          (setf tipo 'f) (setf tipo 'g)))))))
        ;senao4
        (setf tipo 'g)))
(return-from ver_tipo tipo)))
```

A função **retira_vacuo** elimina quantificadores vácuos em uma fórmula (**fórmula**).

```
(defun retira_vacuo (formula)
 (let(forma a b sn x)
    (if (listp formula)(let()
        (setf formula (come_parenteses formula))
        (setf forma (ver_tipo formula)))
      (let()
         (setf formula (list formula))
         (setf forma (ver_tipo formula))))
     (if (eq forma 'o)
         (list 'n (list 'n (retira_vacuo (cdr (car (cdr formula))))))
     (if (eq forma 'p)
       (list 'n (retira_vacuo(cdr formula)))
     (if (eq forma 'g)(let (tam)
       (if (not (member formula l_atomicas :test #'equal))
             (setf l_atomicas (cons formula l_atomicas)))
             (setf tam (length formula))
             (if (= tam 1)
                (return-from retira_vacuo (car formula))
                (return-from retira_vacuo formula)))
     (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c)(eq forma 'd))(let ()
       (setf a (second formula))
       (setf b (third formula))
       (setf sn (first formula))
       (list sn (retira_vacuo a) (retira_vacuo b)))
     (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k))(let()
       (setf a (second (car (cdr formula))))
       (setf b (third (car (cdr formula))))
       (setf sn (first (car (cdr formula))))
```

```
        (list 'n (list sn (retira_vacuo a)(retira_vacuo b))))
    (if (or (eq forma 'e) (eq forma 'f))(let(ver)
      (setf a (third formula))
      (setf x (second formula))
      (setf sn (first formula))
      (setf ver (ver_var_livre x a))
      (if (not (eq ver nil))
        (list sn x (retira_vacuo a))
       (let (chato)
        (setf chato (retira_vacuo a))
        (princ (list chato))
        (if (atom chato)chato (list chato)))))
   (if (or (eq forma 'm) (eq forma 'n))(let(ver)
      (setf a (third (car (cdr formula))))
      (setf x (second (car (cdr formula))))
      (setf sn (first (car (cdr formula))))
      (setf ver (ver_var_livre x a))
      (if (not (eq ver nil))
        (list 'n (list sn x (come_parenteses (retira_vacuo a))))
        (list 'n (come_parenteses (retira_vacuo a))))))))))))))
```

A função **propaga_termos** verifica qual a classe de propagação de termos a que uma fórmula (**formula**) pertence. Pode retornar os valores abaixo:

- 0 se não propaga;
- 1 se propaga temporariamente;
- 2 se propaga indefinidamente.

```
(defun propaga_termos (formula tipo)
  (let (a b)
    (setf a (t_universal formula tipo))
   (setf b (t_existe formula tipo))
```

```
(if (eq a 2) 2
  (if (eq b 0) 0 1))))
```

A função **marca** insere os atributos p (indicando ocorrência positiva) ou n (indicando ocorrência negativa) nos sinais de predicado de uma fórmula.

```
(defun marca (formula sinal)(let(forma a b sn)
    (if (listp formula)(let()
       (setf formula (come_parenteses formula))
       (setf forma (ver_tipo formula)))
     (let()
        (setf formula (list formula))
        (setf forma (ver_tipo formula))))
    (if (eq forma 'o)
        (list 'n (list 'n (marca (cdr (car (cdr formula))) sinal)))
    (if (eq forma 'p)(let (aa)
       (setf aa (troca_sinal sinal))
      (list 'n (marca (cdr formula) aa)))
    (if (eq forma 'g)(let (tam simb bbb aa aaa ai ss)
      (setf simb (car formula))
      (setf ss simb)
      (setf aa (string  ss))
      (setf ai aa)
      (setf aaa (intern (concatenate 'string  ai  '(#\.))) )
      (setf aaa (string aaa))
      (if (eq sinal 'p)
         (setf bbb (intern(concatenate 'string aaa '(#\p))))
         (setf bbb (intern(concatenate 'string aaa '(#\n)))))
     (setf tam (length formula))
      (if (eq tam 1)
         (return-from marca bbb)
```

```
            (let ()
               (setf formula (cdr formula))
               (setf bbb (cons bbb formula))
               (return-from marca bbb))))
      (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c)(eq forma 'd))(let (s1 s2)
        (setf a (second formula))
        (setf b (third formula))
        (setf sn (first formula))
        (if (or (eq forma 'd)(eq forma 'c))(setf s1 (troca_sinal sinal))(setf s1 sinal))
        (if (eq forma 'd)(setf s2 (troca_sinal sinal))(setf s2 sinal))
        (list sn (marca a s1) (marca b s2)))
     (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k)(eq forma 'm) (eq forma
'n))(let()
        (list 'n (marca (car (cdr formula))(troca_sinal sinal))))
     (if (or (eq forma 'e) (eq forma 'f))(let(x xx xxx tipo temp nova_f)
        (setf a (third formula))
        (setf x (second formula))
        (setf sn (first formula))
        (if (or (and (eq sn 'ex)(eq sinal 'p))
              (and (eq sn 'qs)(eq sinal 'n)))(setf tipo 'ex))
        (if (or (and (eq sn 'ex)(eq sinal 'n))
              (and (eq sn 'qs)(eq sinal 'p)))(setf tipo 'un))
        (setf temp (marca a sinal))
        (if (not (member x list_var :test #'equal))
           (let ()
            ;xx variável da lista
            ;xxx nova variável
            (setf xx x)
            (putt xx 1 'num)
            (setf list_var (cons xx list_var))
            (setf xxx x))
           (let (l_aux ss aa ai aaa nn)
```

```
(setf l_aux (member x list_var))
(setf nn (get (car l_aux) 'num))
(setf ss x)
(setf aa (string  ss))
(setf ai aa)
(setf aaa (concatenate 'string  ai  '(#\.)))
(setf xxx (newatom aaa nn))
(setf nn (+ nn 1))
(putt (car l_aux) nn 'num)  ))
(putt xxx tipo 'tipo)
(putt xxx 1 'num)
(setf list_var (cons xxx list_var))
(setf nova_f (substitui x temp xxx))
(setf nova_f (list xxx nova_f))
(setf nova_f (cons sn nova_f))) ))))))))
```

A função **ver_consist** varre o ramo, no momento imediatamente anterior à inserção de uma nova fórmula (**formula**), verificando se esta inserção provoca uma contradição, ocasionando o fechamento do ramo. **ver_consist** também verifica se a fórmula a ser inserida já faz parte do ramo (exceto quando algoritmo = algoritmo1, por se tratar do sistema de tableaux tradicional).

ver_consist devolve a variável **rep**, que possui dois atributos, o primeiro indica se o ramo será, após a inserção da fórmula, aberto ou fechado, o segundo indica se a fórmula a ser inserida ocorre anteriormente no ramo.

```
(defun ver_consist (formula tipo pai algoritmo)
 (let (tf fp consistencia igual ret flag t1 t2 ai ai1)
 (setf flag 0)
    (if (equal algoritmo ':algoritmo4)
       (setf tf (retira_atributos formula))
       (setf tf formula))
```

```
    (if (atom tf)(setf tf (list tf)))
    ; utilizado para formulas com igualdade, quando um not(x=x) deve causar
fechamento do ramo
    (if (eq tipo 'p)
        (if (and (listp (second tf))(eq (car (second tf)) 'i))
            (if (equal (second (second tf))(third (second tf)))
                (setf flag 1))))
     (if (or (eq tipo 'p)(eq tipo 'h)
        (eq tipo 'i)(eq tipo 'j)
        (eq tipo 'k)(eq tipo 'm)
        (eq tipo 'n)(eq tipo 'o))
        (setf tf (cdr tf))
     (if  (or (eq tipo 'g)(eq tipo 'a)
          (eq tipo 'b)(eq tipo 'c)
          (eq tipo 'd)(eq tipo 'e)
          (eq tipo 'f))
       (let ()
         (if (= (length tf) 1)
           (setf tf (append '(n) tf))
           (setf tf (list 'n tf))))))
  (setf tf (come_parenteses tf))
  (setf consistencia 'sim)
  (setf igual 'nao)
  (setf t1 (ver_tipo tf))
  (while (not (eq pai nil))
       (setf fp (get pai 'formula))
       (setf t2 (ver_tipo fp))
       (if (or (eq t1 t2)(eq t2 tipo))(let(fp_aux)
         (setf fp_aux fp)
         (if (equal algoritmo ':algoritmo4)
             (setf fp (retira_atributos fp)))
         (if (atom fp)(setf fp (list fp)))
```

```
        (setf ai(ver_igualdade fp tf))
        (if (eq ai t) (setf consistencia 'nao))
        (setf ai1(ver_igualdade fp_aux formula))
        (if (eq ai1 t)
          (setf igual 'sim))))
    (setf pai (get pai 'pai)))
  (if (eq flag 1)(setf consistencia 'nao))
  (putt ret consistencia 'consist)
  (putt ret igual 'igual)
ret))
```

A função **acha_termos_fechados** retorna uma lista de termos fechados da lista de formulas atômicas contidas na árvore de inicialização do tableaux. Utiliza a função **acha_fec**.

```
(defun acha_termos_fechados ()
   (let (tam ret ai)
      (while l_atomicas
          (setf tam (length (car l_atomicas)))
          (if (> tam 1)(let()
            (setf ai (acha_fec  (car l_atomicas)))
            (setf ret (append ret (get ai 'lista)))))
          (setf l_atomicas (cdr l_atomicas)))
    (setf ret (remove-duplicates ret :test #'equal))
    ret))
```

A função **acha_folhas** encontra folhas de um determinado ramo. Recebe como parâmetro a lista auxiliar do tableaux. Retorna uma lista com as folhas descendentes do primeiro nó da lista auxiliar do tableaux.

```
(defun acha_folhas (lista)
 (let (topo ffi folhas)
  (setf topo (car lista))
  (if (eq (get topo 'filhos) nil)
     (return-from acha_folhas (list topo))
     (let ()
        (setf ffi (get topo  'filhos))
        (while ffi
         (let (vv)
            (setf vv (acha_folhas ffi))
            (if (eq (member vv folhas) nil)
               (setf folhas (append folhas vv)))
            (setf ffi (cdr ffi))))))
  (return-from acha_folhas folhas)))
```

A função **analisa** retorna a(s) fórmula(s) que resulta(m) da aplicação de uma determinada regra a uma dada fórmula.

```
(defun analisa (no)
(let (exp ant ana pos tipo forma)
    (setf exp (get no 'formula))
   (setf exp (come_parenteses exp))
   (setf forma (ver_tipo exp))
   ;se forma and ou or
   (if (or (eq forma 'a) (eq forma 'b))(let ()
      (setf ant (list (car (cdr exp))))
      (setf pos (cdr (cdr exp)))
      (setf tipo forma))
      ;else se forma n-and ou n-or
      (if (or (eq forma 'h)(eq forma 'i))(let ()
         (setf ant (list 'n (car (cdr (car (cdr exp))))))
```

```
        (setf pos (cdr (cdr (car (cdr exp)))))
        (push 'n pos)
        (if (eq forma 'h)(setf tipo 'b)(setf tipo 'a)))
        ;else se forma if
        (if (eq forma 'c)(let()
          (setf ant (list 'n (car (cdr exp))))
          (setf pos (cdr (cdr exp)))
          (setf tipo 'b))
          ;else se forma n-if
          (if (eq forma 'j)(let()
            (setf ant (list (car (cdr (car (cdr exp))))))
            (setf pos (cdr (cdr (car (cdr exp)))))
            (push 'n pos)
            (setf tipo 'a))
;else se forma atômica ou n-atômica
                  (if (or (eq forma 'g)(eq forma 'p))(let()
                    (setf ant exp)
                    (setf tipo forma))
                    ;else  se forma dupla negação
                    (if (eq forma 'o)(let()
                      (setf ant (cdr (car (cdr exp))))
                      (setf tipo forma))
;else se forma n-para todo ou n-existe
                     (if (or (eq forma 'm)(eq forma 'n))(let()
                     (setf ant (list (car (cdr (car (cdr exp))))))
                        (setf pos (cdr (cdr (car(cdr exp)))))
                        (push 'n pos)
                        (setf tipo forma))
;else se forma para todo ou existe
                        (if (or (eq forma 'e)(eq forma 'f))(let()
                          (setf ant (list (car (cdr exp))))
                          (setf pos (cdr (cdr exp)))
```

```
                    (setf tipo forma)))))))))
  (setf ant (come_parenteses ant))
  (setf pos (come_parenteses pos))
  (putt ana tipo 'tipo)
  (putt ana ant 'ant)
  (putt ana pos 'pos)
  (return-from analisa ana)))
```

A função **insere_no_a** é responsável pela criação dos nós da árvore de refutação. Recebe os seguintes parâmetros:

- **formula** – fórmula a ser inserida na árvore;
- **fecha** – indicação se o ramo é fechado ou não (fecha = sim indica ramo aberto, fecha = não, indica ramo fechado);
- **tf** – lista de termos fechados do ramo;
- **nc** – índice da próxima constante que poderá ser criada no ramo;
- **i** – lista termos instanciados até o momento para fórmulas universais, nas outras formulas é um valor nulo;
- **folha** - nó que será pai do novo nó sendo criado;
- **rep** - indica se a fórmula causa propagação de termos e qual o tipo da propagação (rep = 0, 1 ou 2, se algoritmo = algoritmo3; rep = 0, caso contrário) ;

Função retorna **fl**, que é um ponteiro para o nó inserido.

```
(defun insere_no_a (formula fecha tf nc i folha rep)
; função que realiza inserção de um nó na arvore
  (let (fl fil)
       (setf n_nos (1+ n_nos))
       (setf fl (newatom "no" n_nos))
       (putt fl formula 'formula)
       (putt fl nil 'filhos)
       (putt folha nil 'n_cte)
```

```
        (putt folha nil 'l_fec)
        (setf fil (get folha 'filhos))
        (setf fil (append fil (list fl)))
        (putt folha fil 'filhos)
        (putt fl folha 'pai)
        (putt fl nc 'n_cte)
        (putt fl tf 'l_fec)
        (putt fl i 'inst)
        (putt fl fecha 'consistencia)
        (putt fl rep 'rep)
fl))
```

ha_formula_expandir trata-se de uma função que verifica de há fórmulas a serem expandidas no ramo. Retorna **nil** se não existem mais fórmulas a serem expandidas, ou retorna **t**, caso contrário.

```
(defun ha_formula_expandir (no lim)
(let (pai exp forma)
   (setf pai no)
   (while (not (eq pai lim))
      (setf exp (get pai 'formula))
      (setf forma (ver_tipo exp))
      (if (or (eq forma 'g)(eq forma 'p))
         (+ 1 1)
        (return-from ha_formula_expandir t))
     (setf pai (get pai 'pai)))
   (return-from ha_formula_expandir nil)))
```

A função **substitui**, que é utilizada na instanciação de fórmulas universais e existenciais, substitui uma variável (**x**) por um termo (**termo**) em uma fórmula

(**fórmula**).

```
(defun substitui (x formula termo)
  (let (forma)
   (if (listp formula)(let()
       (setf formula (come_parenteses formula))
       (setf forma (ver_tipo formula)))
     (let()
        (setf formula (list formula))
        (setf forma (ver_tipo formula))))
    (setf termo(come_parenteses termo))
    (if (and (listp termo)
          (eq (length termo) 1))
            (setf termo(car termo)))
    (if (eq x termo) formula
    (if (eq forma 'o)
        (list 'n (list 'n (substitui x (cdr (car (cdr formula))) termo)))
    (if (eq forma 'p)
      (list 'n (substitui x (cdr formula) termo))
    (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c)(eq forma 'd))
      (list (first formula)
          (substitui x (second formula) termo)
          (substitui x (third formula) termo))
     (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k))
       (list 'n (list (first (car (cdr formula)))
                 (substitui x (second (car (cdr formula))) termo)
                 (substitui x (third (car (cdr formula))) termo)))
    (if (or (eq forma 'e) (eq forma 'f))
      (if (eq (second formula) x) formula
        (list (first formula)
           (second formula)
           (substitui x (third formula) termo)))
```

```
(if (or (eq forma 'm) (eq forma 'n))
   (if (eq (second (car (cdr formula))) x) formula
      (list 'n (list (first (car (cdr formula)))
                     (second (car (cdr formula)))
                     (come_parenteses
                    (substitui x (third (car (cdr formula))) termo)))))
 (if (eq forma 'g)
    (if (eq formula x) termo
       (if (atom formula)   formula
          (if (and (listp formula)(eq (length formula) 1))(car formula)
              (subst termo x formula :test #'equal)))))))))))))
```

A função **acha_fec_f** constrói uma lista contendo os termos fechado de uma determinada fórmula. Utiliza a função **acha_fec**.

```
(defun acha_fec_f (formula tipo)
  (let (fec)
  (setf formula(come_parenteses formula))
  (if (eq tipo 'g)(let()
     (setf fec (acha_fec formula))
     (setf fec (get fec 'lista)))
     (if (eq tipo 'p)
        (setf fec (append fec (acha_fec_f (cdr formula)(ver_tipo (cdr formula)))))
        (if (or (eq tipo 'a)(eq tipo 'b)(eq tipo 'd)(eq tipo 'c))(let(a b)
           (if (atom (second formula))
              (setf a (list (second formula)))
              (setf a (second formula)))
           (if (atom (third formula))
              (setf b (list (third formula)))
              (setf b (third formula)))
           (setf fec (append fec (acha_fec_f a(ver_tipo a))))
```

```
        (setf fec append fec (acha_fec_f b(ver_tipo b)))))
      (if (or (eq tipo 'h)(eq tipo 'i)(eq tipo 'j)(eq tipo 'k))(let(a b fo)
        (setf fo (car (cdr formula)))
        (if (atom (second fo))
          (setf a (list (second fo)))
          (setf a (second fo)))
        (if (atom (third fo))
          (setf b (list (third fo)))
          (setf b (third fo)))
        (setf fec (append fec (acha_fec_f a(ver_tipo a))))
        (setf fec (append fec (acha_fec_f b(ver_tipo b)))))
      (if (or (eq tipo 'e)(eq tipo 'f))(let(a )
        (if (atom (third formula))
          (setf a (list (third formula)))
          (setf a (third formula)))
        (setf fec (append fec (acha_fec_f a(ver_tipo a)))))
      (if (or (eq tipo 'm)(eq tipo 'n))(let(a fo)
         (setf fo (second formula))
        (if (atom (third fo))
          (setf a (list (third fo)))
          (setf a (third fo)))
        (setf fec (append fec (acha_fec_f a(ver_tipo a)))))
       (if (eq tipo 'o)(let(a fo)
         (setf fo (second formula))
        (if (atom (second fo))
          (setf a (list (second fo)))
          (setf a (second fo)))
        (setf fec (append fec (acha_fec_f a(ver_tipo a)))))
      (setf fec nil))))))))
(setf fec (remove-duplicates fec :test #'equal ))
fec))
```

A função **ver_repeticao** retorna **nil** se não existem fórmulas entre o nó folha (**no**) e nó sendo expandido (**lim**) para serem expandidas no ramo. Caso contrário, retorna **t**.

```
(defun ver_repeticao (no lim)
 (let (pai forma)
   (setf pai no)
   (while (not (eq pai lim))
      (setf forma (get pai 'rep))
      (if (or (eq forma 2)(eq forma 1)) (return-from ver_repeticao t))
      (setf pai (get pai 'pai)))
   (return-from ver_repeticao nil)))
```

A função **unific_inf** dispara o processo de unificação da fórmula universal com as fórmulas anteriores a ela no ramo. Esta função também dispara o processo de unificação indireta (**unif2**). Retorna uma lista onde o primeiro elemento é um flag indicando a ocorrência de unificação indireta e o segundo elemento é a lista de unificações entre a fórmula universal e a parte inferior do ramo.

```
(defun unific_inf (pai f_arv la1 var)(let(lc ant f2 la2 lc_aux unif2 flag_universal ret)
  (setf ant f_arv)
 (setf unif2 nil)
  (while (not (eq ant pai))
    (setf f2 (get ant 'formula))
    (acha_atomicas f2)
    (setf la2 l_a1)
    (setf l_a1 nil)
    (setf lc_aux (unific_form la1 la2 var))
    (setf flag_universal (ver_unif_universal lc_aux var))
```

```
    (if (not (eq flag_universal nil))
      ;se existem unificações universais, então verifico se
      ; essas  fórmulas unificam com outras existencialmente
      (if (eq unif2 nil)
        (setf unif2 (unificacao2
        pai ;fórmula universal que deu origem a unificação universal
        ant ; no atual
        la2 ;lista de form. atômicas da fórmula atual
        flag_universal ;lista de unificações universais da fórmula atual com a form. pai
        f_arv ; folha do ramo em questão
            ))))
    (setf lc (append lc lc_aux))
    (setf ant (get ant 'pai)))
  (setf lc (remove-duplicates lc :test #'equal ))
(setf ret (list unif2 lc))
ret ))
```

A função **acha_atomicas** constrói uma lista com as fórmulas atômicas de uma determinada fórmula.

```
(defun acha_atomicas(formula)(let(forma a b)
    (if (listp formula)(let()
      (setf formula (come_parenteses formula))
      (setf forma (ver_tipo formula)))
    (let()
       (setf formula (list formula))
       (setf forma (ver_tipo formula))))
    (if (eq forma 'o)
      (acha_atomicas (cdr (car (cdr formula))))
    (if (eq forma 'p)
      (acha_atomicas(cdr formula))
```

```
(if (eq forma 'g)(let ()
    (setf l_a1 (cons formula l_a1)))
(if (or (eq forma 'a)(eq forma 'b)(eq forma 'c))(let ()
  (setf a (second formula))
  (setf b (third formula))
  (acha_atomicas a) (acha_atomicas b))
(if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)
        (eq forma 'k)(eq forma 'm) (eq forma 'n))
 (acha_atomicas (car (cdr formula)))
(if (or (eq forma 'e) (eq forma 'f))(let()
 (setf a (third formula))
 (acha_atomicas a ))))))))))
```

A função **unific_sup** dispara o processo de unificação da fórmula universal em expansão com as fórmulas anteriores a ela no ramo. Retorna uma lista de unificações entre a fórmula universal e a parte superior do ramo.

```
(defun unific_sup (pai la1 var)
(let (ant lc la2 f2 lc_aux)
  (setf ant (get pai 'pai))
  (if (eq ant nil) return)
  (while (not (eq ant nil))
    (setf f2 (get ant 'formula))
    (acha_atomicas f2)
    (setf la2 l_a1)
    (setf l_a1 nil)
     (setf lc_aux  (unific_form la1 la2 var))
    (setf lc (append lc lc_aux))
    (setf ant (get ant 'pai)))
     (setf lc (remove-duplicates lc :test #'equal ))
lc ))
```

A função **rep_universal** retorna **t** se um nó, cuja fórmula possui a lista de unificações (**lcs3**), deve ser repetido no ramo. Retorna **nil**, caso contrário.

```
(defun rep_universal (lcs3 lcs0 inst flag)
(let(lcs3_aux tem_exist nome n r_inst)
 (setf lcs3_aux lcs3)
(setf r_inst (set-difference lcs0 inst))
(while (not (eq lcs3_aux nil))
  (setf nome (string (car (cdr (car lcs3_aux)))))
  (setf n (char nome 0))
  (if (char-equal n #\n)
     (setf tem_exist t))
  (setf lcs3_aux (cdr lcs3_aux)))
(if (eq inst nil) t
  (if (eq flag 1)t
  (if(not (eq tem_exist t)) nil
   (if (and (eq lcs0 inst)(eq (length lcs3)(length inst))) nil
      (if (not (eq lcs3 nil))t)))))))
```

A função **extrai_const** retorna uma lista de termos fechados que devem ser usados para instanciação de uma formula universal. Esta lista é retirada da lista de unificações entre a fórmula universal e o ramo ao qual ela pertence.

```
(defun extrai_const (lc var)
  (let (var1 l_ret)
  (while (not (eq lc nil))
    (setf var1 (car (car lc)))
    (if (equalp (string var1) (string (car var)))
      (let(con)
```

```
(setf con (come_parenteses(cdr (car lc))))
  (if (eq (ocorre_var con) t)
      (setf con  (list (newatom "c" 0)))
    (let()
     (if (not (eq (ocorre_n con)t))(let()
       (setf con (come_parenteses con))
       (if (eq (length con) 1)
          (let()
            (setf con (car con))
            (if (eq l_ret nil)
              (setf l_ret (list con))
            (setf l_ret (cons con l_ret))))
          (let()
            (if (eq l_ret nil)
              (setf l_ret (list con))
          (setf l_ret (append l_ret con)))))))))))
  (setf lc (cdr lc)))
  (if (not (eq l_ret nil))
    (let()
      (setf l_ret (remove-duplicates l_ret :test #'equal ))))
  l_ret ))
```

A função **unific_self** dispara o processo de unificação da fórmula universal com ela mesma.

```
(defun unific_self (la1 la2 var)(let(lc)
  (setf lc (unific_form la1 la2 var))
   (setf lc (remove-duplicates lc :test #'equal ))
lc ))
```

A função **t_universal** retorna 2, se **fórmula** gera uma fórmula da forma $\forall x$ **Q**, tal que **Q** possui uma subfórmula fora do escopo de x em **Q**, a qual gera uma fórmula existencial cuja variável quantificada é distinta de x e em cuja matriz x é livre, ou **fórmula** gera uma fórmula da forma $\forall x$ **Q**, tal que x possui uma ocorrência livre em **Q** no escopo de um sinal funcional. Retorna 1, se **fórmula** gera uma fórmula universal que não se enquadra nos dois casos anteriores. Retorna 0 se **fórmula** não gera fórmula universal.

```
(defun t_universal (formula tipo)
  (let (a b c d)
  (if (eq tipo 'g)(setf a 0)
  (if (or (eq tipo 'p)
        (eq tipo 'h)
        (eq tipo 'i)
        (eq tipo 'j)
        (eq tipo 'k)
        (eq tipo 'm)
        (eq tipo 'n))(let ()
        (setf a (t_existe (car (cdr formula)) (ver_tipo (car (cdr formula))))))
  (if (eq tipo 'o)
     (setf a (t_universal (second (car (cdr formula))) (ver_tipo (second (car (cdr
formula))))))
  (if (eq tipo 'c)(let()
     (setf b (t_existe (second formula)(ver_tipo (second formula))))
     (setf c (t_universal (third formula)(ver_tipo (third formula))))
     (setf a (max b c)))
  (if (eq tipo 'a)(let()
     (setf b (t_universal (second formula)(ver_tipo (second formula))))
     (setf c (t_universal (third formula)(ver_tipo (third formula))))
     (setf a (max b c)))
  (if (eq tipo 'b)(let()
     (setf b (t_universal (second formula)(ver_tipo (second formula))))
```

```
    (setf c (t_universal (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'd)(let (tmp)
    (setf tmp (list 'e (list 'ent (second formula)(third formula))
                  (list 'ent (third formula)(second formula))))
    (setf a (t_universal tmp 'a)))
  (if (eq tipo 'e)(let()
                (setf b (+ 1 (f_existe (second formula)
                             (third formula)
                             (ver_tipo (third formula)))))
                (setf c (+ 1 (esf (second formula) (third formula))))
                (setf d (t_universal (third formula)(ver_tipo (third formula))))
                (setf a (max b c d)))
  (if (eq tipo 'f)
    (setf a (t_universal (third formula)(ver_tipo (third formula))))
   ))))))))) a))
```

A função **t_existe** retorna 2 se **fórmula** gera uma fórmula da forma $\exists x$ **Q**, tal que **Q** possui uma subfórmula fora do escopo de x em **Q**, a qual gera uma fórmula universal cuja variável quantificada é distinta de x e em cuja matriz x é livre, ou se **fórmula** gera uma fórmula da forma $\exists x$ **Q**, tal que x possui uma ocorrência livre em **Q** no escopo de um sinal funcional. Retoma 1 se **fórmula** gera uma fórmula existencial que não se enquadra nos dois casos anteriores. Retorna 0 se **fórmula** não gera fórmula existencial.

```
(defun t_existe (formula tipo)
  (let (a b c d)
  (if (eq tipo 'g)(setf a 0)
  (if (or (eq tipo 'p)
        (eq tipo 'h)
        (eq tipo 'i)
        (eq tipo 'j)
```

```
        (eq tipo 'k)
        (eq tipo 'm)
        (eq tipo 'n))
        (setf a (t_universal (car (cdr formula)) (ver_tipo (car (cdr formula)))))
  (if (eq tipo 'o)
    (setf a (t_existe (second (car (cdr formula))) (ver_tipo (second (car (cdr
formula))))))
  (if (eq tipo 'c)(let ()
    (setf b (t_universal (second formula)(ver_tipo (second formula))))
    (setf c (t_existe (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'a)(let ()
    (setf b (t_existe (second formula)(ver_tipo (second formula))))
    (setf c (t_existe (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'b)(let()
    (setf b (t_existe (second formula)(ver_tipo (second formula))))
    (setf c (t_existe (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'd)(let (tmp)
    (setf tmp (list 'e (list 'ent (second formula)(third formula))
                    (list 'ent (third formula)(second formula))))
    (setf a (t_existe tmp 'a)))
  (if (eq tipo 'e)
    (setf a (t_existe (third formula)(ver_tipo(third formula))))
  (if (eq tipo 'f)(let ()
    (setf b (+ 1 (f_universal (second formula)
                        (third formula)
                        (ver_tipo (third formula)))))
    (setf c (+ 1 (esf (second formula) (third formula))))
    (setf d (t_existe (third formula)(ver_tipo (third formula))))
    (setf a (max b c d))))))))))))))))a))
```

A função **troca_sinal** inverte o parâmetro recebido. Se recebe o sinal **p** (positivo), retorna **n** (negativo), e vice versa.

```
(defun troca_sinal (sinal)
  (if (eq sinal 'p)'n 'p))
```

A função **retira_atributos** elimina de uma fórmula os sinais que indicam se suas fórmulas atômicas são ocorrências positivas ou negativas. Utiliza a função **ret_sinal**, que retorna o sinal de predicado sem atributos.

```
(defun retira_atributos (formula)(let(forma simb a b sn)
    (if (listp formula)(let()
        (setf formula (come_parenteses formula))
        (setf forma (ver_tipo formula)))
      (let()
          (setf formula (list formula))
          (setf forma (ver_tipo formula))))
     (if (eq forma 'o)
         (list 'n (list 'n (retira_atributos (cdr (car (cdr formula))))))
    (if (eq forma 'p)
       (list 'n (retira_atributos(cdr formula)))
     (if (eq forma 'g)(let (tam)
        (if (listp formula)
          (let()
            (setf simb (car formula))
            (setf simb (ret_sinal simb))
            (setf formula (cdr formula))
            (setf formula (cons (intern simb) formula)))
          (let()
```

```
            (setf formula (ret_sinal formula))))
        (setf tam (length formula))
        (if (= tam 1)
            (intern (car formula))
            formula))
      (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c))(let ()
        (setf a (second formula))
        (setf b (third formula))
        (setf sn (first formula))
        (list sn (retira_atributos a) (retira_atributos b)))
     (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k)(eq forma 'm) (eq forma
'n))(let()
         (list 'n (retira_atributos (car (cdr formula)))))
     (if (or (eq forma 'e) (eq forma 'f))(let(x)
        (setf a (third formula))
        (setf x (second formula))
        (setf sn (first formula))
        (list sn x (retira_atributos a)))
))))))))
```

A função **ver_igualdade** verifica se duas fórmula são iguais. Utiliza a função **retira_variaveis** para eliminar substituições puras de variáveis.

```
(defun ver_igualdade (f1 f2)(let(t1 t2)
    (setf t1 (ver_tipo f1))
    (setf t2 (ver_tipo f2))
    (if (not (eq t1 t2))nil
      (let (r1 r2 termo var1 var2)
        (if (eq t1 'e) (let()
                    (setf var1 (second f1))
                    (setf var2 (second f2))
```

```
                (setf termo 'x))
        (if (eq t1 'f) (let()
                (setf var1 (second f1))
                (setf var2 (second f2))
                (setf termo 'y))
            (let()
              (setf var1 nil)
              (setf var2 nil)
               (setf termo nil))))
        (setf r1 (retira_variaveis var1 f1 termo))
        (setf r2 (retira_variaveis var2 f2 termo))
    (if (equal r1 r2)t nil)))))
```

A função **acha_fec** retorna uma lista de termos de uma fórmula atômica (**termo**).

```
(defun acha_fec (termo) (let (lt nome n aux vol aux1 aux2 inc guarda)
(setf guarda 'sim)
   (setf termo (cdr termo))
   (while termo
       (if (atom (car termo))
          (let()
             (setf nome (string (car termo)))
             (setf n (char nome 0))
             (if (or (char-equal n #\x)
                   (char-equal n #\y)
                   (char-equal n #\w))
               (let () (setf guarda 'nao)(setf inc 'nao))
               (setf inc 'sim))
             (if (eq inc 'sim)
             (setf lt (append (list (car termo)) lt))))
```

```
        (if (listp (car termo)) (let()
            (setf aux (acha_fec (car termo)))
            (setf aux1 (get aux 'lista))
            (setf aux2 (get aux 'flag))
            (setf lt (append aux1 lt))
            (if (eq aux2 'sim) (setf lt (cons (car termo)lt)))
             (if (eq aux2 'nao) (setf guarda aux2)))))
          (setf termo (cdr termo)))
(putt vol lt 'lista)
(putt vol guarda 'flag)
vol))
```

A função **unif** substitui as variáveis existenciais de fórmulas por constantes especiais para que o processo de unificação continue.

```
(defun unif (f1 f2 var)(let(l_v p1 p2)
  (setf n_ex 1)
  (setf f1 (substitui_var_exist f1))
  (setf f2 (substitui_var_exist f2))
   (setf p1 (cdr f1))
    (setf p2 (cdr f2))
  (setf l_v (acha_dif f1 f2 p1 p2 var))
  l_v ) )
```

A função **ver_unif_universal** recebe uma lista de unificações, e verifica, nesta lista, se existem unificações entre variáveis e termos, os quais contenham outras variáveis universais.

```
(defun ver_unif_universal (aux var)
  (let ( var_aux ret atrib)
  (while (not (eq aux nil))
```

```
  (setf var_aux (car (car aux)))
  (if (eq (car var) var_aux)(let(ll)
     (setf ll (car (cdr (car aux))))
     (setf atrib (get ll 'tipo))
     (if (eq atrib 'un)(setf ret (cons ll ret)))))
  (setf aux (cdr aux)))
 ret))
```

A função **unific_form** busca por fórmulas onde uma destas deve possuir uma ocorrência negativa e a outra positiva (ou vice versa) nas listas de fórmulas atômicas **la1** e **la2**, de modo a enviá-las ao processo de unificação.

```
(defun unific_form (la1 la2 var)
  (let(lc len1 len2 proc n1 nome nome1 nome2 pred1 pred2 rabo g_la2)
   (while (not (eq la1 nil))
    (setf pred1 (car la1))
    (setf nome1 (string (car pred1)))
    (setf len1 (length nome1))
    (setf nome nome1)
    (setf n1 (char nome (- len1 1)))
   (if (char-equal n1 #\p)(let()
       (setf nome (ret_sinal nome))
       (setf nome  (concatenate 'string  (string nome)  '(#\.)))
       (setf nome  (concatenate 'string  nome  '(#\n))))
      (if  (char-equal n1 #\n)(let()
       (setf nome (ret_sinal nome))
       (setf nome (concatenate 'string  (string nome)  '(#\.)))
       (setf nome (concatenate 'string  nome  '(#\p))))))
      (setf rabo (cdr pred1))
      (setf proc (cons nome rabo))
      (setf g_la2 la2)
```

```
    (while (not (eq la2 nil))
      (setf pred2 (car la2))
     (setf nome2  (string (car pred2)))
     (setf len2 (length nome2))
     (if (and (equal len1 len2)(equal (string (car pred2))(car proc)))
       (let(l_tmp)
          (setf l_tmp (unif pred1 pred2 var))
         (setf lc (append lc l_tmp))))
     (setf la2 (cdr la2)))
    (setf la2 g_la2)
    (setf la1 (cdr la1)))
lc ))
```

A função **ocorre_var** recebe um termo como parâmetro e verifica se ocorrem variáveis neste termo.

```
(defun ocorre_var (f)
  (let(ret nome n)
    (if (not (listp f))(setf f (list f)))
   (while f
    (if (listp (car f))
       (setf ret (ocorre_var (car f)))
      (let()
          (setf nome (string (car f)))
          (setf n (char nome 0))
          (if (or(char-equal n #\x)
                (char-equal n #\y)
                (char-equal n #\w))
              (setf ret t))))
   (if (eq ret t)(return-from ocorre_var t))
   (setf f (cdr f)))
```

```
ret))
```

A função **ocorre_n** recebe um termo e verifica se ocorrem constantes especiais neste termo. Constantes especiais, formadas pela letra **n** seguida de um índice, são originalmente variáveis existenciais, as quais foram substituídas para o funcionamento do algoritmo de unificação.

```
(defun ocorre_n (f)
  (let(ret nome n)
    (if (not (listp f))(setf f (list f)))
    (while f
     (if (listp (car f))
        (setf ret (ocorre_n (car f)))
       (let()
           (setf nome (string (car f)))
           (setf n (char nome 0))
           (if (char-equal n #\n)
              (setf ret t))))
    (if (eq ret t)(return-from ocorre_n t))
    (setf f (cdr f)))
ret))
```

A função **f_existe** retorna 1, se **fórmula** possui uma subfórmula fora do escopo de **var** que gera uma fórmula existencial cuja variável quantificada é distinta de **var** e em cuja matriz **var** é livre. No outros casos, retorna 0.

```
(defun f_existe (var formula tipo)
  (let (a b c)
  (if (eq tipo 'g)(setf a 0)
  (if (or (eq tipo 'p)
```

```
        (eq tipo 'h)
        (eq tipo 'i)
        (eq tipo 'j)
        (eq tipo 'k)
        (eq tipo 'm)
        (eq tipo 'n))
        (setf a(f_universal var (car (cdr formula)) (ver_tipo (car (cdr formula)))))
  (if (eq tipo 'o)
    (setf a (f_existe var (second (car (cdr formula))) (ver_tipo (second (car (cdr
formula))))))
  (if (eq tipo 'c)(let ()
    (setf b (f_universal var (second formula)(ver_tipo (second formula))))
    (setf c (f_existe var (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'a)(let ()
    (setf b (f_existe var (second formula)(ver_tipo (second formula))))
    (setf c (f_existe var (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'b)(let()
    (setf b (f_existe var (second formula)(ver_tipo (second formula))))
    (setf c (f_existe var (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'd)(let (tmp)
    (setf tmp (list 'e (list 'ent (second formula)(third formula))
                (list 'ent (third formula)(second formula))))
    (setf a (f_existe var tmp 'a)))
  (if (eq tipo 'e)
    (if (eq var (second formula))
      (setf a 0)
      (setf a (f_existe var (third formula) (ver_tipo (third formula)))))
  (if (eq tipo 'f)
    (if (eq var (second formula))
```

```
    (setf a 0)
    (if (ver_var_livre var (third formula)) (setf a 1)(setf a 0)))
))))))))a))
```

A função **f_universal** retorna 1, se **fórmula** possui uma subfórmula fora do escopo de **var** que gera uma fórmula universal cuja variável quantificada é distinta de **var** e em cuja matriz **var** é livre. Nos outros casos, retorna 0.

```
(defun f_universal (var formula tipo)
  (let (a b c)
  (if (eq tipo 'g)(setf a 0)
  (if (or (eq tipo 'p)
        (eq tipo 'h)
        (eq tipo 'i)
        (eq tipo 'j)
        (eq tipo 'k)
        (eq tipo 'm)
        (eq tipo 'n))
        (setf a (f_existe var (car (cdr formula)) (ver_tipo (car (cdr formula)))))
  (if (eq tipo 'o)
     (setf a (f_universal var (second (car (cdr formula))) (ver_tipo (second (car (cdr
formula))))))
  (if (eq tipo 'c)(let ()
     (setf b (f_existe var (second formula)(ver_tipo (second formula))))
     (setf c (f_universal var (third formula)(ver_tipo (third formula))))
     (setf a (max b c)))
  (if (eq tipo 'a)(let ()
     (setf b (f_universal var (second formula)(ver_tipo (second formula))))
     (setf c (f_universal var (third formula)(ver_tipo (third formula))))
     (setf a (max b c)))
  (if (eq tipo 'b)(let()
```

```
    (setf b (f_universal var (second formula)(ver_tipo (second formula))))
    (setf c (f_universal var (third formula)(ver_tipo (third formula))))
    (setf a (max b c)))
  (if (eq tipo 'd)(let (tmp)
    (setf tmp (list 'e (list 'ent (second formula)(third formula))
                    (list 'ent (third formula)(second formula))))
    (setf a (f_universal var tmp 'a)))
  (if (eq tipo 'e)
    (if (eq var (second formula))
      (setf a 0)
      (if (ver_var_livre var (third formula)) (setf a 1)(setf a 0)))
  (if (eq tipo 'f)
    (if (eq var (second formula))
      (setf a 0)
      (setf a (f_universal var (third formula) (ver_tipo (third formula)))))
))))))))a))
```

A função **esf** retorna 0 caso uma variável (**var**) possua uma ocorrência livre em uma fórmula (**form**) no escopo de um sinal funcional. Caso contrário, retorna 1.

```
(defun esf (var form)
  (let (ret a)
    (setf ret (ver_var_livre var form))
    (if (eq ret t)(setf a 1)(setf a 0))a))
```

A função **ret_sinal** retira do símbolo de predicado o sinal que indica se a fórmula atômica, onde este predicado ocorre, é uma ocorrência positiva ou negativa.

```
(defun ret_sinal (sim)(let(ls i tam simb)
  (setf simb (string sim))
```

```
  (setf tam (length simb))
  (setf i 0)
  (while (not (eq (- tam 2) i))
   (setf ls (cons (char simb i)ls))
   (setf i (+ 1 i)))
(setf ls (list-to-delimited-string ls ""))
(setf simb nil)
ls ))
```

A função **retira_variaveis** elimina substituições puras de variáveis.

```
(defun retira_variaveis (x formula termo)
  (let()
   (let (forma)
     (if (listp formula)
       (setf formula (come_parenteses formula))
       (setf formula (list formula)))
       (setf forma (ver_tipo formula))
     (if (eq forma 'o)
      (list 'n (list 'n (retira_variaveis x (cdr (car (cdr formula))) termo)))
     (if (eq forma 'p)
         (list 'n (retira_variaveis x (cdr formula) termo))
    (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c)(eq forma 'd))
         (list (first formula)
              (retira_variaveis x (second formula) termo)
              (retira_variaveis x (third formula) termo))
     (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k))
        (list 'n (list (first (car (cdr formula)))
                  (retira_variaveis x (second (car (cdr formula))) termo)
                  (retira_variaveis x (third (car (cdr formula))) termo)))
    (if (eq forma 'e)
```

```
        (list (first formula) 'x
            (retira_variaveis (second formula)(third formula) 'x))
    (if (eq forma 'f)
        (list (first formula)  'y
            (retira_variaveis (second formula) (third formula) 'y))
    (if (or (eq forma 'm) (eq forma 'n))
      (list 'n (retira_variaveis x (second formula) termo))
    (if (eq forma 'g)
      (if (eq formula x) termo
        (if (atom formula)   formula
          (if (and (listp formula)(eq (length formula) 1))(car formula)
            (subst termo x formula :test #'equal))))))))))))))
```

A função **substitui_var_exist** substitui as variáveis existenciais de uma fórmula por constante especiais.

```
(defun substitui_var_exist (formula )(let(termo lve)
  (setf  lve  (acha_var_exist formula))
  (while (not (eq lve nil))
    (setf termo (newatom "n" n_ex))
    (setf n_ex (+ n_ex 1))
    (setf formula (substitui (car lve) formula (come_parenteses termo)))
    (setf lve (cdr lve)))
formula))
```

A função **acha_dif** percorre os termos que seguem dois símbolos de predicados iguais, de sinais diferentes, de modo a realizar o processo de unificação. Inicia este processo procurando diferença entre esses dois termos. Quando encontra esta diferença, se um dos termos é uma variável que não ocorre no segundo termo, compõe este par a lista de unificações. O primeiro elemento do par indica a variável que deve ser

substituída pelo segundo. O procedimento realiza esta substituição e recomeça a busca pela próxima diferença, até que os dois termos se igualem, ou até que se verifique a impossibilidade de torná-los iguais.

```
(defun acha_dif (f1 f2 p1 p2 var)
 (let (nome1 nome2 n1 n2 l_ret sub dif tem_var)
(setf dif nil)
 (while (not (and (eq p1 nil)(eq p2 nil)))
 (if (and (listp (car p1))(listp (car p2))(not (eq (car p1) (car p2))))
    (let(c1 c2)
      (setf c1 (cdr (car p1)))
      (setf c2 (cdr (car p2)))
      (setf dif (acha_dif f1 f2 c1 c2 var))
      (setf l_ret (composicao_subs l_ret dif))
       (setf f1 (sub_novas f1 dif))
      (setf f2 (sub_novas f2 dif)))
  (let(prob1 prob2)
    (if (not (listp (car p1)))(setf prob1 (string (car p1))))
    (if (not (listp (car p2)))(setf prob2 (string (car p2))))
  (if (not (equalp  prob1 prob2))
     (let(vv tt)
       (if (not (listp (car p1)))
          (let()
           (setf nome1 (string (car p1)))
           (setf n1 (char nome1 0)))
           (setf n1 nil))
       (if (not (listp (car p2)))
          (let()
           (setf nome2 (string (car p2)))
           (setf n2 (char nome2 0)))
           (setf n2 nil))
       (if (and (not (eq n1 nil))
```

```
   (or (char-equal n1 #\x)
      (char-equal n1 #\y)
      (char-equal n1 #\w)))
      (if (eq (ver_livre_atomica  nome1 (list (car p2))) nil)
       (let(chata)
        (setf tem_var 1)
        (setf chata (string (car var)))
        (if (equalp nome2 chata)
         (let()
           (setf vv (car p2))
           (setf tt (intern nome1)))
         (let()
           (setf vv (intern nome1))
           (setf tt (car p2))))))
(if (and (not (eq n2 nil))
   (or (char-equal n2 #\x)
      (char-equal n2 #\y)
      (char-equal n2 #\w)))
      (if (eq (ver_livre_atomica  nome2 (list (car p1))) nil)
          (let(chata)
            (setf tem_var 1)
            (setf chata (string (car var)))
          (if (equalp chata nome1)
           (let()
             (setf vv (car p1))
             (setf tt (intern nome2)))
           (let()
             (setf vv (intern nome2))
             (setf tt (car p1))))
           ))))
(if (eq tem_var 1)(let(nova_f1 nova_f2)
   (setf sub (cons tt sub))
```

```
        (setf sub (cons vv sub))
        (setf l_ret (cons sub l_ret))
        (setf tem_var 0)
        (setf sub nil)
        (setf nova_f1 (substitui vv f1 (come_parenteses tt)))
        (setf nova_f2 (substitui vv f2 (come_parenteses tt)))
        (setf f1 nova_f1)
        (setf f2 nova_f2)
        (if (not (equalp (cdr f1) (cdr f2)))(let(nn)
            (setf nn (acha_dif nova_f1  nova_f2 (cdr nova_f1)
                        (cdr nova_f2) var))
            (setf l_ret (composicao_subs l_ret nn))
            (setf f1 (sub_novas f1 nn))
            (setf f2 (sub_novas f2 nn))))
      (if (equalp (cdr f1) (cdr f2))(let () ; (print 'a****)(return))))))
       (let(b)
         (setf b(ver_livre_atomica (car var) p1))
         (if (eq b t)(let()
          (setf sub (cons (car var) sub))
          (setf sub (cons (car var) sub))
           (setf sub nil)))))))
 (if (eq (cdr f1)(cdr f2))
      (let(b)
       (setf b (ver_livre_atomica (car var) f1))
       (if (eq b t)(let()
         (setf sub (cons (car var) sub))
         (setf sub (cons (car var) sub))
         (setf l_ret (cons sub l_ret))
         (setf sub nil)))
      (return)))
(if (not (eq p1 nil))
 (setf p1 (cdr p1)))
```

```
 (if (not (eq p2 nil))
  (setf p2 (cdr p2))))
l_ret ))
```

A função **unific_inf2** obtém uma lista de unificações entre as fórmulas atômicas de uma determinada fórmula, a qual unifica através de variáveis universais com a fórmula universal em expansão, e as fórmulas atômicas do ramo que a sucedem.

```
(defun unific_inf2 (pai la1 var folha)
(let(lc lc_aux ant la2 f2)
(setf ant folha)
(while (not (eq ant pai))
     (setf f2 (get ant 'formula))
     (acha_atomicas f2)
     (setf la2 l_a1)
     (setf l_a1 nil)
     (setf lc_aux (unific_form la1 la2 var))
     (setf lc (append lc lc_aux))
     (setf ant (get ant 'pai)))
   (setf lc (remove-duplicates lc :test #'equal ))
lc))
```

A função **ver_unif_existencial** verifica, em uma lista de pares de termos unificantes (**aux**), se existem substituições entre variáveis e termos contendo variáveis existenciais.

```
(defun ver_unif_existencial (aux)
(let (tem_const nome n ll constantes)
  (setf constantes nil)
  (while (not (eq aux nil))
```

```
(setf ll (car (cdr (car aux))))
(setf nome (string ll))
(setf n (char nome 0))
(if ;(not (char-equal n #\n))
   (or (char-equal n #\x)
      (char-equal n #\y)
      (char-equal n #\w))
      (setf tem_const nil)
      (let()
         (setf tem_const t)
         (if (char-equal n #\n)
            (setf constantes 1))))
(setf aux (cdr aux)))
 constantes))
```

A função **composicao_subs** faz a composição de duas listas de substituições.

```
(defun composicao_subs (l1 l2)
  (let(g_l2 li_var1 f22 f21  f11 f12 s1 s2 n_list n_sub)
 (if (eq l1 nil) (setf n_list l2)
    (if (eq l2 nil) (setf n_list l1)
     (let()
     (if (not (listp l1)) (setf l1 (list l1)))
     (setf g_l2 l2)
     (while l1
      (setf l2 g_l2)
      (setf s1 (car l1))
      (setf f11 (car s1))
      (setf li_var1 (cons f11 li_var1))
      (setf f12 (cdr s1))
      (while l2
```

```
      (setf s2 (car l2))
      (setf f21 (car s2))
      (setf f22 (cdr s2))
      (if (not (listp f12))(setf f12 (list f12)))
      (if (not (listp f12))(setf f22 (list f22)))
      (setf n_sub (substitui f21 f12 (come_parenteses f22)))
      (if (not (eq n_sub f11))(let(sub)
        (setf sub (cons n_sub sub))
        (setf sub (cons f11 sub))
        (setf n_list (cons sub n_list))))
      (setf l2 (cdr l2)))
    (setf l1 (cdr l1)))
  (while g_l2
    (if (eq (member (car (car g_l2)) li_var1)nil)
       (setf n_list (cons (car g_l2)n_list)))
    (setf g_l2 (cdr g_l2)))
  )))
 (if (not (listp (car n_list)))(setf n_list (list n_list)))
n_list))
```

A função **ver_var_livre** verifica se uma variável (**x**) é livre em uma fórmula (**fórmula**). Retorna **t** se variável é livre na formula, caso contrário retorna **nil**.

```
(defun ver_var_livre (x formula)(let (a y forma)
  (if (listp formula)
     (setf forma (ver_tipo formula))
     (let()
       (setf formula (list formula))
       (setf forma (ver_tipo formula))))
  (setf formula (come_parenteses formula))
  (if (or (eq forma 'e) (eq forma 'f))(let()
```

```
    (setf a (third formula))
    (setf y (second formula))
     (and (not (eq x y)) (ver_var_livre x a)))
  (if (or (eq forma 'm) (eq forma 'n))(let()
    (setf a (fourth formula))
    (setf y (third formula))
    (and (not (eq x y)) (ver_var_livre x a)))
  (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c)(eq forma 'd))
     (or (ver_var_livre x (second formula))(ver_var_livre x (third formula)))
  (if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k))
     (or (ver_var_livre x (second (car (cdr formula))))
     (ver_var_livre x (third (car (cdr formula)))))
  (if (eq forma 'o)
     (ver_var_livre x (cdr (car (cdr formula))))
  (if (eq forma 'p)
     (ver_var_livre x (cdr formula))
  (if (eq forma 'g)
     (if (member x formula) t
     (ver_livre_atomica x formula)))))))))))
```

A função **acha_var_exist** encontra as variáveis existenciais em uma fórmula.

```
(defun acha_var_exist (formula)(let(f n nome l_var_ex)
 ; (print 'formula=)(princ formula)
 (setf f (cdr formula))
 (while f
  (if (listp (car f))
    (setf l_var_ex (append l_var_ex (acha_var_exist (car f))))
    (let()
      (setf nome (string (car f)))
      (setf n (char nome 0))
```

```
      (if (or (char-equal n #\x)
              (char-equal n #\y)
              (char-equal n #\w))
        (if (eq (get (car f) 'tipo) 'ex)
          (setf l_var_ex (cons (car f) l_var_ex))))))
   (setf f (cdr f)))
  (setf l_var_ex (remove-duplicates l_var_ex :test #'equal ))
  l_var_ex))
```

A função **sub_novas** dispara o processo de substituição das variáveis de uma fórmula atômica, utilizando uma lista de substituições obtida através do processo de unificação.

```
(defun sub_novas (fl l_ret)
  (while l_ret
    (setf fl (substitui (car (car l_ret)) fl
                        (come_parenteses (cdr (car l_ret)))))
    (setf l_ret (cdr l_ret)))
 fl)
```

A função **ver_livre_atomica** verifica se uma variável (**x**) ocorre em uma fórmula atômica (**formula**). Retorna t se variável ocorre na formula, ou retorna **nil**, caso contrário.

```
(defun ver_livre_atomica (x formula)
  (let (f ret)
  (setf f formula)
  (while f
    (if (listp (car f))(setf ret (ver_livre_atomica x (car f)))
    (if (equal (car f) x)
```

```
    (setf ret t)))
   (if (eq ret t)(return-from ver_livre_atomica t))
   (setf f (cdr f)))
ret))
```

A função **troca_var** substitui as variáveis de **fórmula** por outras que não ocorram no tableaux ao qual a **fórmula** pertence.

```
(defun troca_var (formula)(let(forma a b sn)
    (if (listp formula)(let()
       (setf formula (come_parenteses formula))
       (setf forma (ver_tipo formula)))
     (let()
        (setf formula (list formula))
        (setf forma (ver_tipo formula))
        (setf formula (car formula))))

    (if (eq forma 'g)formula

    (if (eq forma 'o)
        (list 'n (list 'n (troca_var (cdr (car (cdr formula))))))

    (if (eq forma 'p)
      (list 'n (troca_var (cdr formula)))

    (if (or (eq forma 'a)(eq forma 'b)(eq forma 'c)(eq forma 'd))(let ()
      (setf a (second formula))
      (setf b (third formula))
      (setf sn (first formula))
      (list sn (troca_var a) (troca_var b)))
```

```
(if (or (eq forma 'h)(eq forma 'i)(eq forma 'j)(eq forma 'k)(eq forma 'm) (eq forma 'n))
    (list 'n (troca_var (car (cdr formula))))

(if (or (eq forma 'e) (eq forma 'f))(let(x xx xxx tipo temp nova_f)
   (setf a (third formula))
   (setf x (second formula))
   (setf sn (first formula))
   (setf tipo (get x 'tipo))
   (setf temp (troca_var a))
   (if (not (member x list_var :test #'equal))
      (let ()
       (setf xx x)
       (putt xx 1 'num)
       (setf list_var (cons xx list_var))
       (setf xxx x))
      (let (l_aux ss aa ai aaa nn tamind ind)
       (setf l_aux (member x list_var))
       (setf nn (get (car l_aux) 'num))
       (setf ss x)
       (setf aa (string  ss))
       (setf ai aa)
       (setf aaa (concatenate 'string  ai  '(#\.)))
       (setf xxx (newatom aaa nn))
       (setf nn (+ nn 1))
       (putt (car l_aux) nn 'num)
       (putt xxx 1 'num)
       (setf list_var (cons xxx list_var)) ))
       (putt xxx tipo 'tipo)
       (setf nova_f (substitui x temp xxx))
       (setf nova_f (list xxx nova_f))
       (setf nova_f (cons sn nova_f)))))))))))
```

7 – RESULTADOS OBTIDOS PELO SISTEMA IMPLEMENTADO

Este capítulo destina-se a exibir os resultados encontrados pelo sistema implementado através do presente trabalho. Para isto, enumeramos uma série de problemas utilizados nos testes e os resultados obtidos pela execução dos quatro sistemas de tableaux sobre os mesmos, bem como uma breve análise desses resultados.

Abaixo são apresentados os problemas testados, os quais foram extraídos de [Pelletier 1986].

(P1) $\vdash (p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$

(P2) $\vdash \neg\neg p \leftrightarrow p$

(P3) $\vdash \neg (p \rightarrow q) \rightarrow (q \rightarrow p)$

(P4) $\vdash (\neg p \rightarrow q) \leftrightarrow (\neg q \rightarrow p)$

(P5) $\vdash ((p \vee q) \rightarrow (p \vee r)) \rightarrow (p \vee (q \rightarrow r))$

(P6) $\vdash p \vee \neg p$

(P7) $\vdash p \vee \neg\neg\neg p$

(P8) $\vdash ((p \rightarrow q) \rightarrow p) \rightarrow p$

(P9) $\vdash ((p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)) \rightarrow \neg(\neg p \vee \neg q)$

(P10) $p \rightarrow r$

$r \rightarrow (p \wedge q)$

$p \rightarrow (q \vee r)$

$\vdash p \leftrightarrow q$

(P11) $\vdash p \leftrightarrow p$

(P12) $\vdash ((p \leftrightarrow q) \leftrightarrow r) \leftrightarrow (p \leftrightarrow (q \leftrightarrow r))$

(P13) $\vdash (p \vee (q \wedge r)) \leftrightarrow (p \vee q) \wedge (p \vee r))$

(P14) $\vdash (p \leftrightarrow q) \leftrightarrow ((q \vee \neg p) \wedge (\neg q \vee p))$

(P15) $\vdash (p \rightarrow r) \leftrightarrow (\neg q \vee q)$

(P16) $\vdash (p \rightarrow q) \vee (q \rightarrow p)$

(P17) $\vdash ((p \wedge (q \rightarrow r)) \rightarrow s) \leftrightarrow ((\neg p \vee q \vee s) \wedge (\neg p \vee \neg r \vee s))$

(P18) $\vdash \exists x \, \forall y \, (p(y) \rightarrow p(x))$

(P19) $\vdash \exists x \, \forall y \, \forall z \, (p(y) \rightarrow q(z)) \rightarrow (p(x) \rightarrow q(x))$

(P20) $\vdash (\forall x \forall y \exists z \forall w((p(x) \wedge q(y)) \rightarrow (r(z) \wedge s(w))) \rightarrow (\exists x \exists y(p(x) \wedge q(y)) \rightarrow \exists z \, r(z))$

(P21) $\exists x\ (p \rightarrow q(x))$

$\exists x\ (q(x) \rightarrow p)$

$\vdash \exists x\ (p \leftrightarrow q(x))$

(P22) $\vdash \forall x\ (p \leftrightarrow q(x)) \rightarrow (p \leftrightarrow \forall x\ q(x))$

(P23) $\vdash \forall x\ (p \vee q(x)) \leftrightarrow (p \vee \forall x\ q(x))$

(P24) $\neg\exists x\ (s(x) \wedge q(x))$

$\forall x\ (p(x) \rightarrow (q(x) \vee r(x)))$

$\neg\exists x\ p(x) \rightarrow \exists x\ q(x)$

$\forall x((q(x) \vee r(x)) \rightarrow s(x))$

$\vdash \exists x\ (p(x) \wedge r(x))$

(P25) $\exists x\ p(x)$

$\forall x\ (p1(x) \rightarrow (\neg q1(x) \wedge r(x))),\ \forall x\ (p(x) \rightarrow (q1(x) \wedge p1(x)))$

$\forall x\ (p(x) \rightarrow q(x)) \vee \exists x\ (p(x) \wedge r(x))$

$\vdash \exists x\ (q(x) \wedge p(x))$

(P26) $\exists x\ p(x) \leftrightarrow \exists x\ q(x)$

$\forall x\ \forall y\ ((p(x) \wedge q(y)) \rightarrow (r(x) \leftrightarrow s(y)))$

$\vdash \forall x\ (p(x) \rightarrow r(x)) \leftrightarrow \forall x\ (q(x) \rightarrow s(x))$

(P27) $\exists x\ (p1(x) \wedge \neg q1(x))$

$\forall x\ (p1(x) \rightarrow r1(x))$

$\forall x\ ((q2(x) \wedge p2(x)) \rightarrow p1(x))$

$\exists x\ (r1(x) \wedge \neg q1(x)) \rightarrow \forall x\ (q2(x) \rightarrow \neg\ r1(x))$

$\vdash \forall x\ (q2(x) \rightarrow \neg p2(x))$

(P28) $\forall x\ p(x) \rightarrow \forall x\ q(x)$

$\forall x\ (q(x) \vee r(x)) \rightarrow \exists x\ (q(x) \wedge s(x))$

$\exists x\ s(x) \rightarrow \forall x\ (p1(x) \rightarrow q1(x))$

$\vdash \forall x\ ((p(x) \wedge p1(x)) \rightarrow q1(x))$

(P29) $\exists x\ p1(x) \wedge \exists x\ q1(x)$

$\vdash (\forall x\ (p1(x) \rightarrow r1(x)) \wedge \forall x\ (q1(x) \rightarrow q2(x))) \leftrightarrow \forall x\ \forall y\ ((p1(x) \wedge q1(y)) \rightarrow (r1(x) \wedge q2(y)))$

(P30) $\forall x\ (p1(x) \vee q1(x)) \rightarrow \neg r1(x))$

$\forall x\ ((q1(x) \rightarrow \neg p2(x)) \rightarrow (p1(x) \wedge r1(x)))$

$\vdash \forall x\ p2(x)$

(P31) $\neg\exists x\ ((p1(x) \wedge (q1(x) \vee r1(x)))$

$\exists x\ (p2(x) \wedge p1(x))$

$\forall x\ (\neg r1(x) \rightarrow q2(x))$

$\vdash \exists x\ (p2(x) \wedge q2(x))$

(P32) $\forall x\ ((p1(x) \wedge (q1(x) \vee r1(x))) \rightarrow p2(x))$, $\forall x\ ((p2(x) \wedge r1(x)) \rightarrow q2(x))$, $\forall x\ (r2(x) \rightarrow r1(x)) \vdash \forall x\ ((p1(x) \wedge r2(x)) \rightarrow q2(x))$

(P33) $\vdash \forall x\ ((p(a) \wedge (p(x) \rightarrow p(b))) \rightarrow p(c)) \leftrightarrow (\forall x\ ((\neg p(a) \vee (p(x) \vee p(c))) \wedge (\neg p(a) \vee (p(b) \vee p(c))))$

(P34) $\vdash (\exists x\ \forall y\ (p(x) \leftrightarrow p(y)) \leftrightarrow (\exists x\ q(x) \leftrightarrow \forall y\ p(y))) \leftrightarrow (\exists x\ \forall y\ (q(x) \leftrightarrow q(y)) \leftrightarrow (\exists x\ p(x) \leftrightarrow \forall y\ p(y)))$

(P35) $\vdash \exists x\ \exists y\ (p(x, y) \rightarrow \forall x\ \forall y (p(x,y))$

(P36) $\forall x\, \exists y\, p1(x,y)$

$\forall x\, \exists y\, q1(x,y)$

$\forall x\, \forall y\, ((p1(x,y) \vee q1(x,y)) \rightarrow \forall z\, ((p1(y,z) \vee q1(y,z)) \rightarrow r1(x,z))$

$\vdash\ \forall x\, \exists y\, r1(x,y)$

(P37) $\forall z\, \exists w\, \forall x\, \exists y\, ((p(x,z) \rightarrow p(y,w)) \wedge p(y,z) \wedge (p(y,w) \rightarrow \exists y1\ q(y1,w)))$

$\forall x\, \forall z\, (\neg p(x,z) \rightarrow \exists y\, q(y,z))$

$(\exists x\, \exists y\, q(x,y)) \rightarrow \forall x\, r(x,x))$

$\vdash\ \forall x\, \exists y\, r(x,y)$

(P38) $\vdash\ \forall x\, ((p(a) \wedge (p(x) \rightarrow \exists y\, ((p(y) \wedge r(x,y))))) \rightarrow \exists y\, \exists w\, (p(z) \wedge r(x,w) \wedge r(w,z)))$
$\leftrightarrow \forall x((\neg p(a) \vee p(x) \vee \exists z\, \exists w\, (p(z) \wedge r(x,w) \wedge r(w,z)) \wedge (\neg p(a) \vee \neg \exists y\, (p(y) \wedge r(x,y)) \vee$
$\exists z\, \exists w\, (p(z) \wedge r(x,w) \wedge r(w,z))))$

(P39) $\vdash \exists x\, \forall y\, (p1(x, y) \leftrightarrow \neg p1(y,y))$

(P40) $\vdash \exists y\, \forall x\, (p1(x, y) \leftrightarrow p1(x,x)) \rightarrow \neg \forall x\, \exists y\, \forall z\, (p1(x,y) \leftrightarrow \neg p1(z,x))$

(P41) $\forall z\, \exists y\, \forall x\, (p1(x,y) \leftrightarrow (p1(x,z) \wedge \neg p1(x,x)))$

$\vdash\ \neg \exists z\, \forall x\, p1(x,z)$

(P42) $\vdash \forall z\, \exists y\, \forall x\, (p1(x, y) \leftrightarrow\ \neg \exists z\, (p1(x,z) \wedge \neg p1(z,x)))$

(P43) $\forall x\, \forall y\, (q(x,y) \leftrightarrow \forall z\, (p1(z,x) \leftrightarrow p1(z,y)))$

$\vdash\ \forall x\, \forall y\, (q(x,y) \leftrightarrow q(y,x))$

(P44) $\forall x\, (p1(x) \rightarrow \exists x\, (q1(y) \wedge r1\ (x,y)) \wedge \exists y\, (q1(y) \wedge \neg\ r1(x,y)))$

$\exists x\, (q2(x) \wedge \forall y\, (q1(y) \rightarrow r1(x,y)))$

(P45) $\forall x\,(p1(x) \wedge \forall y\,(q1(y) \wedge r1(x,y) \rightarrow q2(x,y)) \rightarrow \forall y\,(q1(y) \wedge r1(x,y) \rightarrow r2(y)))$

$\neg\, \exists y\,(p3(y) \wedge r2(y))$

$\exists x\,(p1(x) \wedge \forall y\,(r1(x,y) \rightarrow p3(y)) \wedge \forall y\,(q1(y) \wedge r1(x,y) \rightarrow q2(x,y)))$

$\vdash\!\!-\ \exists x\,(p1(x) \wedge \neg\, \exists y\,(q1(y) \wedge r1(x,y)))$

A tabela a seguir mostra os resultados obtidos pela aplicação de cada um dos sistemas de tableaux apresentados no presente trabalho sobre os problemas acima propostos. O número máximo de nós permitidos para a árvore de refutação foi 10.000 (dez mil). Assim, as lacunas da tabela que estão preenchidas com um travessão representam os problemas para os quais o sistema correspondente não obteve resposta diante do limite estabelecido acima.

Problema	Número de nós da árvore de refutação obtidos pelo			
	Sistema de Tableaux Tradicional	Primeiro Refinamento	Segundo Refinamento	Terceiro Refinamento
P1	20	20	20	20
P2	12	12	12	12
P3	6	6	6	6
P4	19	19	19	19
P5	16	15	15	15
P6	3	3	3	3
P7	4	4	4	4
P8	6	6	6	6
P9	24	18	18	18
P10	32	28	28	28
P11	9	9	9	9
P12	174	144	144	144
P13	39	39	39	35
P14	54	48	48	48

P15	18	16	16	16
P16	6	6	6	6
P17	60	60	60	60
P18	17	17	17	14
P19	139	139	139	121
P20	-	-	-	91
P21	60	48	42	49
P22	51	48	43	43
P23	31	31	30	31
P24	284	173	111	145
P25	66	66	52	59
P26	-	-	-	6818
P27	234	95	74	81
P28	-	-	172	142
P29	6985	4534	2660	110
P30	72	72	57	57
P31	37	37	26	30
P32	597	596	426	583
P33	879	226	182	184
P34	-	-	2290	971
P35	43	38	38	35
P36	-	-	-	-
P37	-	-	-	-
P38	254	208	208	191
P39	13	13	10	11
P40	2483	2207	2207	445
P41	-	-	-	-
P42	45	44	44	39
P43	-	-	-	-
P44	3475	2867	2867	223
P45	-	-	-	4525

A seguir realizamos uma série de observações com base nos dados expostos acima.

Nos problemas de P1 até P18 o número de nós permaneceu praticamente inalterado para qualquer um dos sistemas de tableaux. Isto ocorre devido ao fato desses problemas possuírem somente fórmulas da lógica proposicional, onde apenas o segundo refinamento tem efeito na redução do número de nós.

É possível perceber em alguns problemas (P21, P23, P24 e outros) um leve acréscimo no número de nós obtidos pelo terceiro refinamento em relação ao segundo; isto ocorre devido a uma questão de implementação na regra para fórmulas quantificadas universalmente. No terceiro refinamento, no momento de expansão de uma fórmula universal, repetimos esta fórmula antes de introduzir suas instâncias no tableau. Este procedimento faz com que a fórmula universal seja analisada novamente antes que suas instâncias, que podem ocasionar o fechamento do ramo, tenham sido expandidas.

Verificamos também a superioridade do terceiro refinamento, em relação aos demais sistemas, quando executado sobre fórmulas com símbolos predicativos de aridade superior a um, e em fórmulas com o conetivo "$\leftrightarrow$".

Os casos onde nenhum dos sistemas de tableaux encontraram resultados são aqueles onde temos fórmulas quantificadas universalmente que geram fórmulas quantificadas existencialmente.

De modo geral, podemos observar que os refinamentos introduzidos no sistema de tableaux produziram árvores com um número menor de nós, bem como solucionaram uma quantidade maior de problemas.

8 - CONCLUSÃO

O presente trabalho teve início com o estudo da lógica quantificacional clássica e do método dos tableaux de uma maneira generalizada. Posteriormente descrevemos um sistema de tableaux tradicional para a lógica quantificacional clássica, o qual serviu de ponto de partida para o desenvolvimento de outros três sistemas de tableaux. Cada um dos quatro sistemas possui uma descrição e uma implementação, as quais podem ser observadas nos capítulos 4, 5 e 6, respectivamente, e obedecem à árvore genealógica da Figura 15.

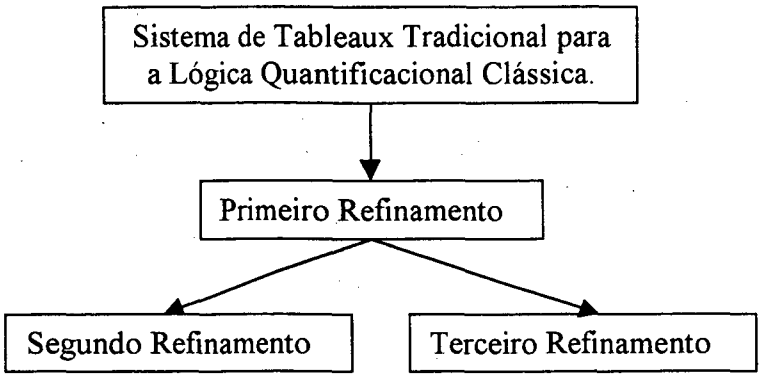

Figura 15: Árvore Genealógica dos Sistemas Implementados

O fluxograma, apresentado na Figura 16, mostra o esqueleto da implementação final, a qual integra os quatro sistemas de tableaux aqui descritos.

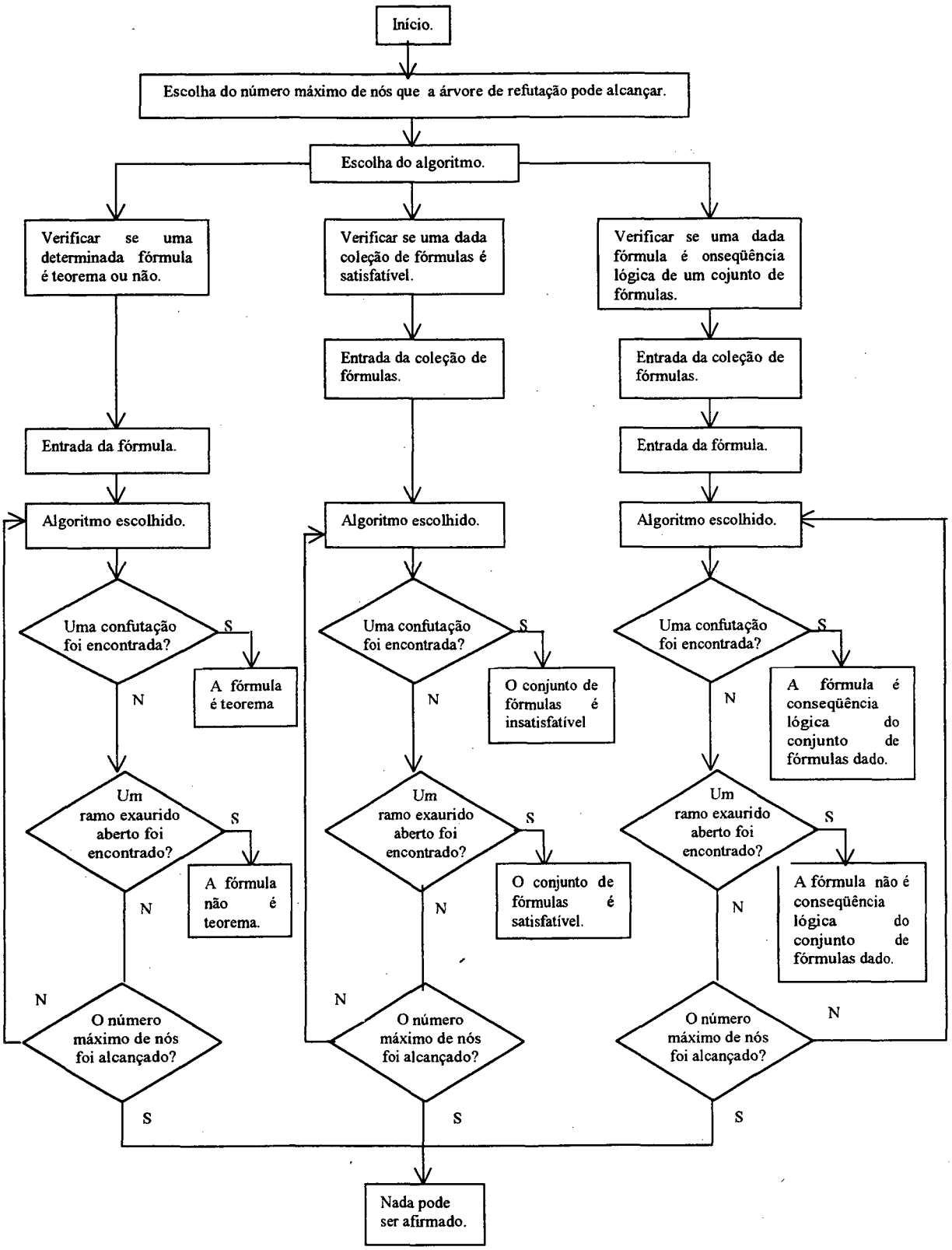

Figura 16: Esquema de Implementação

O intuito deste trabalho foi que cada modificação sucessiva, realizada sobre o sistema de tableaux tradicional para a lógica quantificacional clássica, resultasse na construção de árvores com um número cada vez menor de nós, amenizando o problema da explosão combinatória, bem como em soluções para um número maior de problemas, tornando decidíveis vários casos que eram originalmente indecidíveis pelo método dos tableaux tradicional. Também testou-se a viabilidade da utilização do processo de unificação associado ao método dos tableaux, conservando as principais características deste método, o que não ocorre, por exemplo, no sistema de tableaux descrito em [Fiitting 1990].

Após a realização de uma série de testes e da análise dos mesmos, como pode ser observado no capítulo 7, foi possível verificar, na maioria dos casos, a obtenção do propósito acima descrito. Assim, de modo geral, a cada refinamento a árvore de refutação sofreu sucessivas quedas no número de nós e, também, as técnicas empregadas para evitar a repetição de fórmulas universais contribuíram para que obtivéssemos solução para um número maior de problemas. Além disso, foi possível verificar a viabilidade da associação do procedimento de unificação para o método dos tableaux.

A seguir enumeramos uma série de possíveis extensões para o presente trabalho:

- uniformização do tratamento dado a fórmulas universais no terceiro refinamento;
- expansão do método de unificação para tableaux aqui descrito para atender às necessidades específicas da lógica equacional clássica;
- adaptação do método de unificação aqui descrito para construção de máquinas de inferência baseadas em seqüentes;
- experimentação de novos métodos de busca sistemática em tableaux, priorizando fórmulas que a médio prazo geram expansões menores, preferindo, por exemplo, a expansão de fórmulas conjuntivas sobre as disjuntivas;
- prova de correção e completude para os sistemas aqui descritos;
- construção de uma interface amigável.

9 – REFERÊNCIAS

[Barreto 1997]
Barreto, Jorge Muniz. *Inteligência Artificial no Limiar do Século XXI*. ρρρ Edições. Florianópolis, 1997.

[Bittencourt 1997]
Bittencourt, Guilherme. *Inteligência Artificial: Ferramentas e Teorias*. Ed. da UFSC, Florianópolis, 1998.

[Bell & Machover 1993]
Bell, John Lane & Machover, Moshé. *A Course in Mathematical Logic*. Elsevier Science Publishers B.V., Amsterdam, 1993.

[Buchsbaum 1988]
Buchsbaum, Arthur. *Um Método Automático de Prova para a Lógica Paraconsistente*. Dissertação de Mestrado do Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1988.

[Buchsbaum 1995]
Buchsbaum, Arthur. *Lógicas da Inconsistência e da Incompletude: Semântica e Axiomática*. Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1988.

[Buchsbaum & Pequeno 1990]
Buchsbaum, Arthur & Pequeno Tarcisio. *O Método dos Tableaux Generalizado e sua Aplicação ao Raciocínio Automático em Lógicas Não Clássicas*. O que nos faz pensar – Cadernos do Departamento de Filosofia da Pontifícia Universidade Católica do Rio de Janeiro, nº 3, setembro de 1990.

[Buchsbaum & Pequeno 1991]
Buchsbaum, Arthur & Pequeno Tarcisio. Uma família de Lógicas Paraconsistente e/ou Partacompletas com Semânticas Recursivas. Monografias em Ciência da Computação nº 5/91. Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1991. Republicado por Coleção Documentos Série de Lógica e Teoria da Ciência nº 14. Instituto de Estudos Avançados, Universidade de São Paulo, 1993.

[Buchsbaum & Pequeno 1994]
Buchsbaum, Arthur & Pequeno Tarcisio. *Automated Deduction with Non Classical Negations*. Proceedings of 3rd Workshop on Theorem Proving with Analytic Tableaux and Related Methods - 1994 - pp. 51-64.

[Chang & Lee 1973]
Chang, Chin-Liang & Lee, Richard Char-Tung. *Simbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Church 1936]
Church, Alonzo. *A Note on the Entscheidungsproblem*. The Journal of Symblic Logic, vol.1, pp 40-41, 101-102, 1936.

[Copi 1978]
Copi, Irving M.. *Introdução à Lógica*. Editora Mestre Jou, 2ª edição, 1978.

[da Costa 1994]
da Costa, Newton C. A.. *Ensaio sobre os Fundamentos da Lógica*, Editora Hucitec, 2ª edição, 1994.

[Ebbinghaus & Flum & Thomas 1989]
Ebbinghaus, H. D. & Flum, J. & Thomas, W.. *Mathemathical Logic*. Springer-Verlag, New York, 1989.

[Enderton 1972]
Enderton, Herbert B.. *A Mathematical Introduction to Logic*, Academic Press, 1972.

[Fiitting 1990]
Fitting, Melvin. *First-Order Logic and Automated Theorem Proving*. Springer Verlag, 1990.

[Graham 1996]
Graham, Paul. *ANSI Common Lisp*. Prentice Hall, 1996.

[Horowitz & Sahni 1976]
Horowitz, Ellis & Sahni, Sartaj. *Fundamentals of Data Structures*. Computer Science Press, 1976.

[Lloyd 1987]
Lloyd, J. W.. *Foundations of Logic Programming*. Springer Verlag, 1987.

[Loveland 1978]
Loveland, Donald W.. *Automated Theorem Proving*. North-Holland Publishing Company, 1978.

[Margaris 1990]
Margaris, Angelo. *First Order Mathematical Logic*. Dover Publications, New York, 1990.

[Nolt & Rohatyn 1991]
Nolt, John & Rohatyn, Dennis. *Lógica*. McGraw-Hill & Makron Books, 1991.

[Oppacher & Suen 1988]
Oppacher, F. & Suen E.. *Harp: A Tableau-Based Theorem Prover*. Journal of Automated Reasoning. Kluner Academic Publishers, 1988.

[Prawitz 1989]
Prawitz, Dag. *Natural Deduction – A Proof-Theoretical Stydy*. Almqvist & Wiksell, 1965.

[Pelletier 1986]
Pelletier, Francis J.. *Seventy-Five Problems for Testing Automatic Theorem Provers*. Journal of Automated Reasoning. D. Reidel Publishing Company, 1986.

[Rabuske 1995]
Rabuske, Renato Antônio. *Inteligência Artificial*. Ed. Da UFSC, Florianópolis, 1995.

[Reeves 1983]
Reeves, S. V.. *An Introduction to Semantic Tableaux. Departmente of Computer Science*. CMS-55, University of Essex, 1983.

[Reeves]
Reeves, S. V.. *Semantic Tableaux as Framework for Automated Theorem-Proving*. University of London.

[Rich 1993]
Rich, Elaine. *Inteligência Artificial*. Makron Books, São Paulo, 1993.

[Smullyan 1995]
Smullyan, Raymond M.. *First-Order Logic*. Dover Publications, 1995

[Suppes 1972]
Suppes, Patrick, *Axiomatic Set Theory*. Dover Publications, 1972.

[van Dalen 1989]
van Dalen, D.. *Logic and Structure*. Springer-Verlag, 1989.