EMERSON FARIA NOBRE

UMA IMPLEMENTAÇÃO DO MÉTODO DAS CONEXÕES DE BIBEL PARA UMA LÓGICA PARACONSISTENTE ANOTADA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre, Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Décio Krause
Co-orientador: Prof. Dr. Martin A. Musicante

CURITIBA
2001

UFPR
Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática do aluno ***Emerson Faria Nobre***, avaliamos o trabalho intitulado ***"Uma Implementação do Método das Conexões de Bibel para uma Lógica Paraconsiente Anotada"***, cuja defesa foi realizada no dia 31 de agosto de 2001, às dezesseis horas, no anfiteatro B, do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 31 de agosto de 2001.

Prof. Dr. Décio Krause
DINF/UFPR - Orientador

Prof. Dr. Martin Alejandro Musicante
CO-ORIENTADOR - DINF/UFPR

Prof. Dr. Celso A. A. Kaestner
PPGIA-PUC/PR

Prof. Dr. Marcos Alexandre Castilho
DINF/UFPR

SUMÁRIO

RESUMO

O método das conexões de Bibel é uma alternativa ao método de resolução e vem sendo aplicado a vários sistemas lógicos, incluindo fragmentos da lógica linear, lógica intuicionista e várias lógicas modais.

Em um trabalho anterior, o método das conexões de Bibel foi estendido para um tipo de lógica paraconsistente, chamada de lógica anotada. Neste trabalho o método do trabalho anterior é estendido tornando as definições mais adequadas à implementação. Um procedimento que melhora a eficiência do método das conexões de Bibel é apresentado e uma implementação do método é desenvolvida na linguagem SML.

Os agentes inteligentes e sistemas especialistas podem utilizar métodos de prova em seus módulos raciocinadores para derivar conclusões de uma base de conhecimentos. A lógica anotada trata uma base de conhecimentos inconsistente de modo adequado. Um exemplo ilustrativo de uso de uma base de conhecimentos inconsistente em um sistema médico é automatizado através da implementação. Por fim, uma comparação de eficiência é apresentada.

ABSTRACT

Bibel´s connection method is an alternative to the well know resolution method and has been applied in several logical systems, including fragments of linear logic, intuitionistic logic and various modal logics.

In a previous work, Bibel´s connection method was extended to a kind of paraconsistent logic, called annotated logic. In this work, the previous work is extended, making the definitions well suited for an implementation. A procedure that improve the efficiency is showed and an implementation in the Standard ML is developed.

The intelligent agents and expertise systems can make use of proof methods in it´s reasoning modules to derive conclusions of an knowledge base. The annotated logic can manage inconsistent knowledge bases in an adequate way. An illustrative example of the use of an inconsistent knowledge base in a medical system is automatized through the implementation. Finally, an efficiency comparation is showed.

CAPÍTULO 1

INTRODUÇÃO

Tradicionalmente as pesquisas na área da Ciência da Computação são realizadas levando em conta que algo deve ser exclusivamente verdadeiro ou falso. Porém, de modo intuitivo, parece que nós humanos não raciocinamos somente desta forma. Muitas vezes decidimos algo a partir de informações contraditórias. Neste sentido, os sistemas de Inteligência Artificial [33], como por exemplo os Agentes Inteligentes, provavelmente poderão ter um campo mais amplo de aplicação se puderem derivar conclusões tendo por base conhecimentos que possuam inconsistências.

Neste trabalho são definidas a teoria e a implementação de um método de prova de teoremas que consegue, de modo adequado, implicar conclusões a partir de um conjunto de informações inconsistentes. Esta implementação pode, por exemplo, fazer parte do módulo raciocinador de um sistema especialista que admita contradições, ou como ferramenta de auxílio ao estudo formal de teorias inconsistentes [21].

O método de prova utilizado na implementação foi o método das conexões de Bibel [5, 6, 7]. Este método de prova foi adaptado para uma lógica paraconsistente anotada proposicional, a qual, trata contradições de forma adequada.

Embora o assunto desta dissertação seja especificamente o dito no parágrafo acima, para se ter uma visão um pouco mais abrangente do assunto em questão, mais adiante neste capítulo, será apresentado um breve histórico da lógica paraconsistente, uma descrição de suas peculiaridades e uma visão geral de provadores de teoremas e métodos de prova. Por fim, será vista a organização dos próximos capítulos.

1.1 Objetivos

Esta dissertação tem como objetivos definir e implementar o método das conexões de Bibel para a lógica paraconsistente anotada proposicional.

1.2 Um breve histórico da lógica paraconsistente

Entre 1910 e 1913, o polonês Jan Lukasiewicz (1878-1956) e o russo Nicolai Vasiliev (1880-1940), de modo independente, salientaram a importância da revisão de algumas leis da lógica aristotélica. Ambos argumentaram que a eliminação do princípio da não-contradição poderia de alguma forma conduzir a lógicas não-aristotélicas. Vasiliev embora não tivesse proposto nenhum sistema lógico formal acreditava que sua lógica, além da interpretação não-aristotélica, também poderia manter a interpretação aristotélica.

Porém, somente trinta e oito anos mais tarde que, influenciado por idéias de Lukasiewicz, Stanislaw Jaskowski (1906-1965) construiu, com base na lógica discursiva, um tipo específico de lógica paraconsistente. Foi em 1948 que, Jaskowiski propôs o primeiro cálculo proposicional paraconsistente. Este cálculo foi desenvolvido em linhas gerais, de modo a preencher três motivações básicas: (1) oferecer alguns conceitos básicos para possibilitar a sistematização dedutiva de teorias que contém contradições, em particular, (2) aquelas contradições que são geradas por vaguidade; e, finalmente, (3) estudar algumas teorias empíricas que contenham axiomas contraditórios [21].

Entretanto, não desmerecendo o importante trabalho desenvolvido por Jaskowiski, quem mais contribuiu para a evolução da lógica paraconsistente foi o brasileiro Newton Carneiro Afonso da Costa, que, desde 1954 até atualmente, vem elaborando cálculos e aplicações para esta lógica. A lógica paraconsistente, tal como é conhecida hoje, em grande parte deve-se a ele.

Newton C.A. da Costa elaborou e tem elaborado cálculos proposicionais, cálculos de predicados (com e sem igualdade), cálculos de descrições e numerosas aplicações à teoria de conjuntos[18, 19, 20].

Vários anos após sua criação, em 1976, o nome *lógica Paraconsistente* foi sugerido por Francisco Miró Quesada durante o terceiro Congresso Latino Americano de Lógica Matemática. Desde então este passou a ser o nome oficial desta lógica [20].

Mais tarde, em 1991, na *Mathematics Subject Classification*, onde são relacionadas as áreas nas quais se subdivide a matemática contemporânea, e que é revista a cada cinco anos pela American Mathematical Society, foi introduzido, na seção *Logic and Fondations*,

o verbete *03B53: Paraconsistent Logic* [20].

Dentre várias outras lógicas paraconsistentes, em 1987, V.S. Subrahmanian [36] introduziu o conceito de lógica anotada. Enfatizando que, neste trabalho faremos uma adaptação de um método de prova, originalmente desenvolvido para a lógica clássica, para a lógica anotada.

Nos últimos anos, como um campo de pesquisa autônomo, a lógica paraconsistente tem crescido muito - tanto do ponto de vista teórico, como em termos de diversas aplicações (inteligência artificial, matemática, filosofia, etc.) A título de exemplo, pode-se mencionar, no domínio dos sistemas especialistas, o emprego da lógica paraconsistente na manipulação de informações inconsistentes, bem como na programação lógica com cláusulas contraditórias (Paralog - Um Prolog paraconsistente).

1.3 Princípios da lógica paraconsistente

Uma teoria dedutiva T, cuja linguagem contenha um símbolo de negação $\neg$, é dita *inconsistente* se no conjunto de seus teoremas contém ao menos dois deles, um dos quais é a negação do outro.

Por exemplo, se F e $\neg F$ são teoremas de T, então T é inconsistente.

A teoria T é *trivial* se todas as fórmulas de sua linguagem forem teoremas, ou seja, todas as sentenças sintaticamente corretas da linguagem de T puderem ser provadas. Assim sendo, podemos concluir que uma teoria trivial não apresenta interesse algum, pois, não é possível distinguir o verdadeiro do falso.

Por exemplo, sendo F e G fórmulas sintaticamente corretas, é possível derivar G de $F \wedge \neg F$, o que, intuitivamente, não faz sentido, pois, somente fazem parte das premissas as fórmulas F e $\neg F$.

A lógica clássica e muitas outras são triviais se e somente se são inconsistentes. Deste modo, a lógica clássica é adequada para implicar conclusões somente de premissas consistentes.

Uma lógica é paraconsistente se pode ser utilizada como lógica subjacente a teorias inconsistentes mas não triviais. Isto implica, dentre outras coisas, que o princípio da

não-contradição deve de alguma forma ser restringido, a fim de que possam aparecer contradições, mas deve-se evitar que de duas premissas contraditórias se possa deduzir uma fórmula qualquer. Mais especificamente, os cálculos apresentados pelo Prof. da Costa [20] foram erigidos para satisfazer basicamente as seguintes condições:

- o princípio da não-contradição na forma $\neg(F \wedge \neg F)$ não deve ser válido em geral.
- não se deve ter $F, \neg F \vdash G$, ou seja, de duas premissas contraditórias não deve ser possível, em geral, deduzir-se qualquer proposição, e
- todos os esquemas e regras da lógica clássica que forem compatíveis com estas duas condições devem em princípio ser mantidas.

Dentre as lógicas paraconsistentes, estão os cálculos C_n de da Costa [20]. Em cada um deste cálculos, existem sentenças de dois tipos:

- as *bem comportadas*, onde toda fórmula válida do cálculo clássico também o será nos cálculos da lógica paraconsistente;
- as *mau comportadas*, onde, se F for mal comportada, pode-se ter $F \wedge \neg F$.

Deste modo, a lógica clássica permanece válida para as sentenças bem comportadas, enquanto que, para as mau comportadas a lógica paraconsistente permite certas investigações que não seriam possíveis na lógica clássica.

As lógicas paraconsistentes não pretendem substituir a lógica clássica e sim serem complementares.

Para maiores detalhes veja [20] e suas referências.

1.4 Provadores de teorema

Um problema central em Inteligência Artificial é como fazer que seja possível um computador inferir conclusões. Um dos meios, utilizado originalmente para auxiliar matemáticos, é através da prova de teoremas, onde, partindo de um conjunto de proposições é possível derivar conclusões.

Não é de agora que pesquisadores vêm tentando utilizar provadores de teoremas como módulo raciocinador em Inteligência Artificial. Segundo o apresentado em [11], na primeira conferência sobre Inteligência Artificial, no Dartmouth College, no verão de 1956, Newell e Simon discutiam um sistema dedutivo para a lógica proposional. Misky, paralelamente, estava desenvolvendo as idéias que, em 1963, foram utilizadas no provador de teoremas para geometria elementar de Gelernter [13]. Em 1960, Wang produziu a primeira implementação, razoavelmente eficiente, de um algoritmo completo para provar teoremas em lógica proposicional.

Seguindo estes esforços, o próximo passo importante no desenvolvimento de técnicas de dedução automática, foi um método, relativamente simples e logicamente completo, para provar teoremas do cálculo de predicados de primeira ordem [34]. O procedimento, e aqueles derivados deste, são geralmente referidos como *procedimento de resolução* [34] porque a regra básica é o princípio de resolução.

Com o surgimento do método de resolução, acreditou-se que poderia tornar possível a construção de um provador automático de teoremas geral que pudesse resolver qualquer problema axiomatizável. Devido a esta motivação, poucos anos mais tarde, em 1969, Green conduziu experimentos extensivos com sistemas solucionadores de problemas baseados em resolução.

O resultado dos experimentos de Green e muitos outros projetos similares foram desapontadores. A dificuldade era que, no caso geral, o espaço de busca gerado pelo método de resolução aumentava exponencialmente em relação ao número de fórmulas usadas para descrever um problema. Deste modo, problemas de complexidade moderada não podiam ser resolvidos em tempo aceitável. Várias heurísticas, independentes do domínio, foram propostas para tratar deste problema, mas elas provaram ser muito fracas para produzir resultados satisfatórios.

Embora até hoje não se tenha conseguido obter sucesso na solução de problemas em geral através de um único provador automático de teoremas, muitos resultados foram obtidos na solução de problemas específicos.

O programa AURA respondeu questões abertas em várias áreas da matemática (1983).

O provador de Boyer-Moore(1979) vem sendo usado e estendido durante muitos anos, e foi usado por Natarajan Shankar para fazer a primeira prova completa, rigorosa e formal do teorema da incompletude de Gödel(1986). O programa OTTER é um dos provadores de teoremas mais robustos; ele tem sido usado para resolver várias questões abertas em lógica combinatorial [33].

Além do uso em matemática, dentre diversas aplicações, os provadores automáticos de teoremas podem ser aplicados na síntese (construção) de programas de computadores. Primeiramente, a especificação é transformada em um teorema a ser provado. Então, uma prova construtiva, supondo que o teorema é verdadeiro, é gerada. A idéia básica é que cada passo da prova construtiva corresponde a um passo de computação [12]. Assim, o programa é extraído através da prova construtiva. A lógica subjacente a este tipo de prova é a lógica intuicionista de Brouwer [21, 11].

Os provadores de teoremas também tem sido usados na área da Inteligência Artificial como módulos de raciocínio de sistemas especialistas e agentes inteligentes. São nestes sistemas que se torna interessante ter como lógica subjacente ao método de prova uma lógica paraconsistente. Deste modo, o raciocinador pode, além de produzir raciocínios consistentes, manter também os raciocínios inconsistentes em sua base de conhecimentos.

1.5 Métodos de prova automática

O núcleo de um provador de teoremas é o método de prova. Além do método de resolução, foram criados outros métodos de prova automática de teoremas, dentre eles, o método do tableau [35] e o método das conexões de Bibel [5, 6, 7]. A prova no método dos tableaux tem por base a construção de uma árvore a partir da fórmula a ser provada, enquanto que no método das conexões de Bibel uma matriz é construída a partir da fórmula a ser provada e então a prova é realizada tendo por base a matriz.

Com o passar dos anos, foram desenvolvidas várias versões destes métodos. O objetivo de algumas versões era o uso de lógicas não-clássicas, enquanto que, em outras o objetivo principal era melhorar a eficiência do método.

Por exemplo, em [29] é descrito o método das conexões de Bibel para fragmentos das

lógicas lineares, intuicionista e modal, em [10] é descrito o método do tableau para a lógica paraconsistente C_1. Em [39] foi realizada uma comparação de eficiência entre o método das conexões de Bibel, tableaux analíticos, tableaux analíticos otimizados, resolução linear, resolução linear com seleção e procedimento de Davis Putman.

Para maiores detalhes sobre métodos de prova, veja [11, 39] e suas referências.

1.6 Organização da dissertação

Esta dissertação está organizada da seguinte forma: no próximo capítulo o método das conexões de Bibel para a lógica clássica é apresentado; no capítulo 3 é apresentada uma lógica anotada; no capítulo 4 o método das conexões de Bibel é adequadamente adaptado para a lógica anotada; no capítulo 5 um algoritmo para melhorar a eficiência do método das conexões de Bibel, uma breve comparação com o método tableaux e uma visão geral sobre a complexidade dos métodos de prova são vistos; no capítulo 6 as características, estruturas e operações da linguagem Standard ML são vistas; são apresentados no capítulo 7, uma implementação do método das conexões de Bibel para a lógica anotada em Standard ML, um exemplo de aplicação e um teste empírico de eficiência da implementação; Por fim, no capítulo 8, é dada uma conclusão ao trabalho e sugestões de pesquisas são apresentadas.

CAPÍTULO 2

O MÉTODO DAS CONEXÕES DE BIBEL PARA A LÓGICA CLÁSSICA PROPOSICIONAL

Este método de prova automática de teoremas foi desenvolvido pelo alemão Wolfgang Bibel em 1982 [5], que a seguir será apresentado de um modo menos formal tal como em [7].

2.1 Conceitos e sintaxe

Supondo que as expressões da linguagem sejam as da lógica proposicional clássica, o método de Bibel é definido da seguinte forma.

Definição 2.1.1 Seja L uma proposição da linguagem da lógica clássica proposicional. O átomo L é um *literal positivo* e o átomo negado $\neg L$ é um *literal negativo.*

Definição 2.1.2 Uma *cláusula* é um conjunto finito de literais.

Definição 2.1.3 Uma *matriz* é um conjunto finito de cláusulas.

A matriz de Bibel pode pode ser representada de duas formas (figuras 2.1 e 2.2):

Na *representação positiva* um literal representa ele próprio, uma cláusula representa a conjunção de seus literais e uma matriz representa a disjunção de suas cláusulas. Esta forma de representação é adequada para prova direta.

Na *representação negativa* um literal representa sua negação, uma cláusula representa a disjunção de seus literais e uma matriz representa a conjunção de suas cláusulas. Esta forma de representação é adequada para prova indireta.

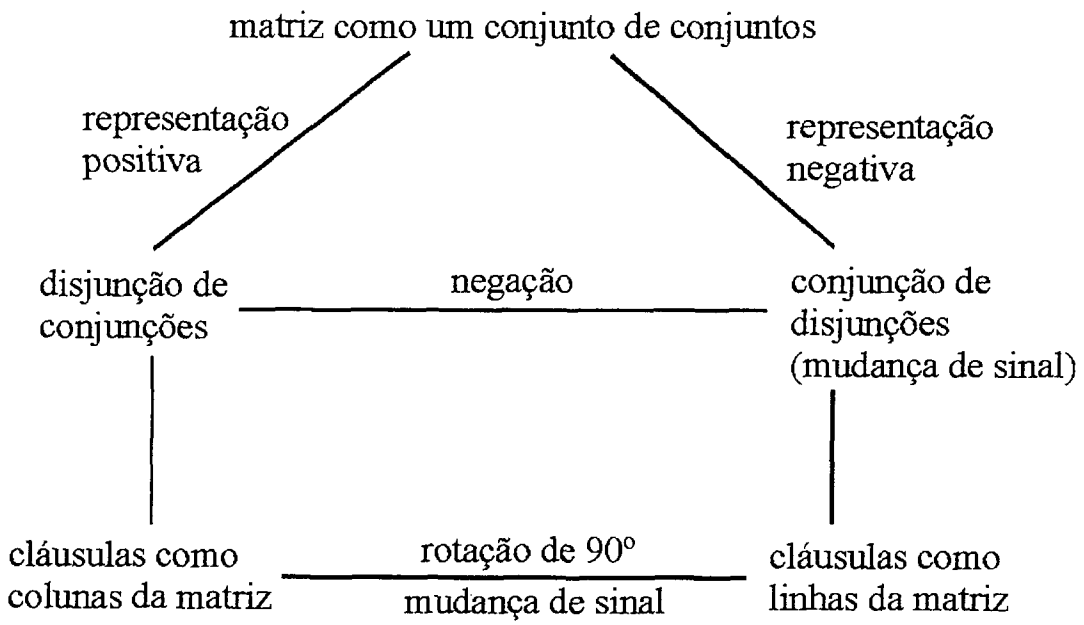


Figura 2.1: Representação positiva e negativa

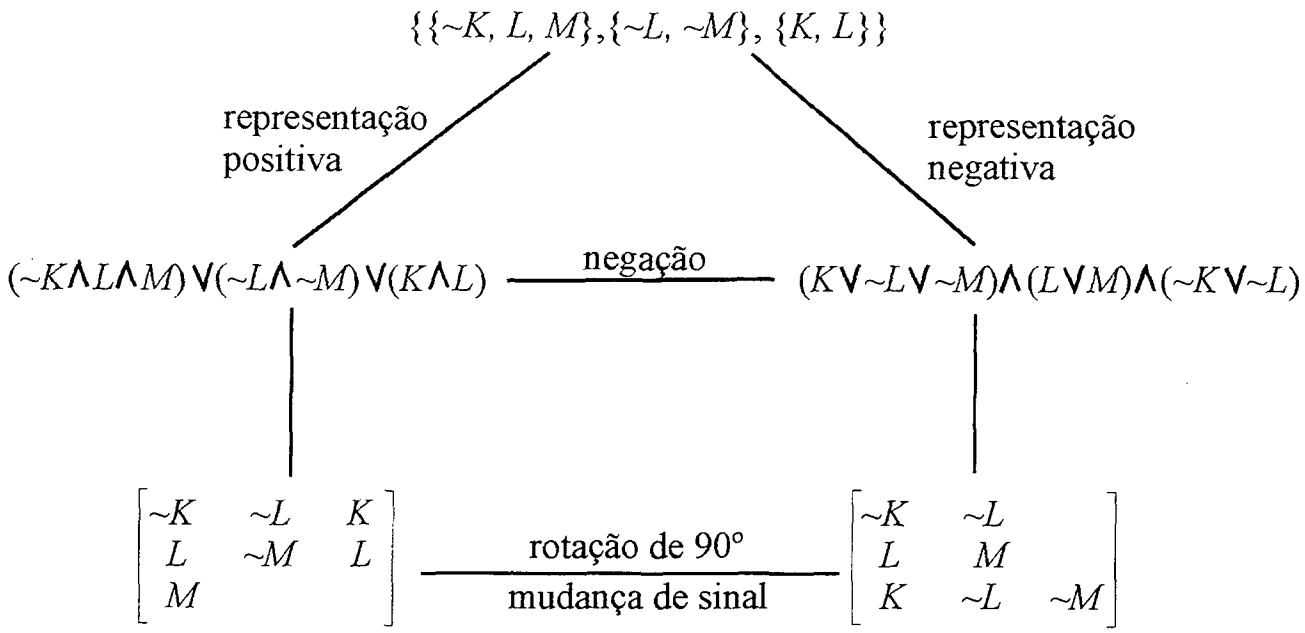


Figura 2.2: Matrizes e fórmulas

2.2 Semântica

Definição 2.2.1 Seja A um conjunto de símbolos proposicionais. Uma *interpretação* $\mathcal{M}$ é um subconjunto finito de A.

A valoração da interpretação $\mathcal{M}$ na representação positiva de uma matriz é:

- seja L um átomo, L é verdadeiro se $L \in \mathcal{M}$, senão L é falso.
- seja $\neg L$ um literal, $\neg L$ é verdadeiro se $\neg L \notin \mathcal{M}$, senão $\neg L$ é falso.
- seja C uma cláusula, C é verdadeira se todos seus literais são verdadeiros, senão C é falsa.
- seja F uma matriz, F é verdadeira se alguma cláusula é verdadeira, senão F é falsa.

Na representação negativa da matriz é só substituir o verdadeiro por falso e o falso por verdadeiro.

Definição 2.2.2 A interpretação $\mathcal{M}$ é um *modelo* de F se o valor de F em $\mathcal{M}$ é verdadeiro.

Definição 2.2.3 F é *satisfatível* se existe um modelo para F.

Definição 2.2.4 F é *válida* se toda interpretação é um modelo de F.

2.3 Caminhos, conexões e caracterização de validade

Definição 2.3.1 Seja C_n uma cláusula, um *caminho* através da matriz $\{C_1, ..., C_n\}$ é um conjunto da ocorrências do literal na cláusula C, um literal de cada cláusula.

Definição 2.3.2 Uma *conexão* é um subconjunto de um caminho na forma $\{L, \neg L\}$.

Definição 2.3.3 Uma *união* é um conjunto de conexões de uma matriz.

Definição 2.3.4 Uma matriz F possui uma *ponte* , se cada caminho através de F contém pelo menos uma conexão.

Temos que uma fórmula é *válida* se e somente se a matriz que representa esta fórmula possui um ponte de uniões.

2.4 Outras características do método

Bibel também define o cálculo proposicional, onde é apresentado o procedimento de extensão que tem por objetivo aumentar a eficiência do método. [6]

Algumas vantagens do método de Bibel são:

- A matriz não é alterada durante a prova.
- A matriz fornece uma visão ampla da prova.
- O procedimento do método é simples.
- Se a implementação for realizada através de uma linguagem imperativa, então o método de Bibel é mais compacto que a Resolução e o Tableau, ocupando assim um espaço menor na memória do computador durante a prova. Isto é devido à não necessidade de fazer cópias de partes da matriz durante a prova.

Algumas desvantagens são:

- Ao contrário do método tableau, que é mais próximo do modo de pensar humano, o método de Bibel é voltado para o modo de funcionamento das máquinas.
- A quantidade de caminhos percorridos na matriz aumenta exponencialmente, tornando certas provas intratáveis. Todos os provadores de teoremas apresentam este problema.
- Existem poucas fontes de informação sobre este método de prova em comparação ao que existe para o método de resolução.

No próximo capítulo será apresentada uma lógica anotada.

CAPÍTULO 3

A LÓGICA ANOTADA PROPOSICIONAL

Como foi mencionado no primeiro capítulo, a lógica anotada pertence a classe de lógicas paraconsistentes. Deste modo, esta lógica admite contradições mas não é trivial.

Neste capítulo será apresentado um breve histórico das lógicas anotadas em geral, uma descrição da lógica anotada que usaremos, sua sintaxe, semântica, postulados e algumas conclusões.

3.1 Histórico

Em 1987, V.S. Subrahmanian [36] introduziu o conceito de lógica anotada. No mesmo ano, H.A. Blair e V.S. Subrahmanian [8] apresentaram um trabalho no qual foram descritas uma teoria de modelos e teoria de prova baseadas em tal lógica.

Mais tarde, em 1988, H.A. Blair e V.S. Subrahmanian [9] estenderam os resultados anteriores para permitir a programação em lógica anotada. A teoria de prova foi então adequadamente estendida.

M. Kifer e Lozinskii [26] demonstraram que a lógica clássica está de certa forma embutida na lógica anotada. Eles mostraram também ligações entre lógicas anotadas e lógicas não monotônicas.

Devido à crescente importância da lógica anotada, da Costa, V.S. Subrahmanian e Vago [17] fizeram um estudo de seus fundamentos. Eles desenvolveram uma família de lógicas anotadas proposicionais, chamada $P\mathcal{T}$, e de primeira ordem, chamada $Q\mathcal{T}$.

Da Costa, L.J. Henschen, J.J. Lu e V.S. Subrahmaniam [15], em 1989, apresentaram a teoria e aspectos práticos de implementação de um provador de teoremas, baseado no método de Resolução, para a lógica $Q\mathcal{T}$.

Em 1991, da Costa, J.M. Abe e V.S. Subrahmanian [14] apresentaram uma axiomatização para a lógica $Q\mathcal{T}$, e desenvolveram a lógica $Q\mathcal{T}^2$. Por fim, demonstraram que a

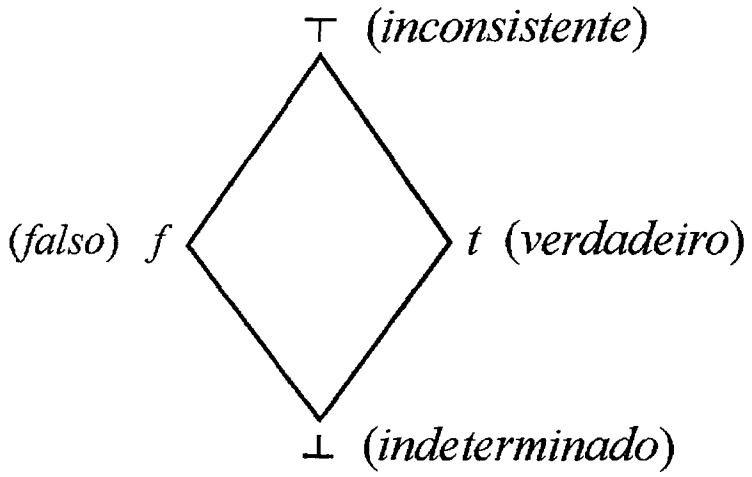


Figura 3.1: Reticulado FOUR

lógica QT^2 é correta e completa.

M. Kifer e V.S. Subrahmanian [27], em 1992, apresentaram uma teoria generalizada para a programação lógica anotada e suas aplicações. Também neste ano, V.S. Subrahmanian [37] apresentou uma teoria para banco de dados dedutivos paraconsistentes. Dando continuidade, em 1994, J. Grant e V.S. Subrahamanian [23] mostraram como é possível raciocinar sobre bases de conhecimentos inconsistentes.

Dentre os trabalhos mais recentes, da Costa e D. Krause [16] apresentaram uma lógica anotada indutiva.

3.2 Visão geral da lógica anotada proposicional

De modo informal, na linguagem da lógica clássica uma proposição é verdadeira ou falsa. Já na linguagem da lógica anotada, pelo menos em uma de suas versões, podemos dizer que acreditamos em uma proposição com "certo grau de crença". Este grau de crença é estimado pelos elementos (constantes de anotação) de um reticulado $\mathcal{T}$ completo na ordem $\geq$. De maneira mais precisa, seja k uma proposição e μ uma constante de anotação. Escrevemos na forma k_μ para denotar uma proposição anotada. Lê-se k_μ como "creio em k com grau de crença μ".

O reticulado mais comum é o FOUR (veja o diagrama de Hasse que representa este reticulado na figura 3.1), $\top \geq t$, $f \geq \bot$, onde, $\top$, t, f e $\bot$ denotam respectivamente "in-

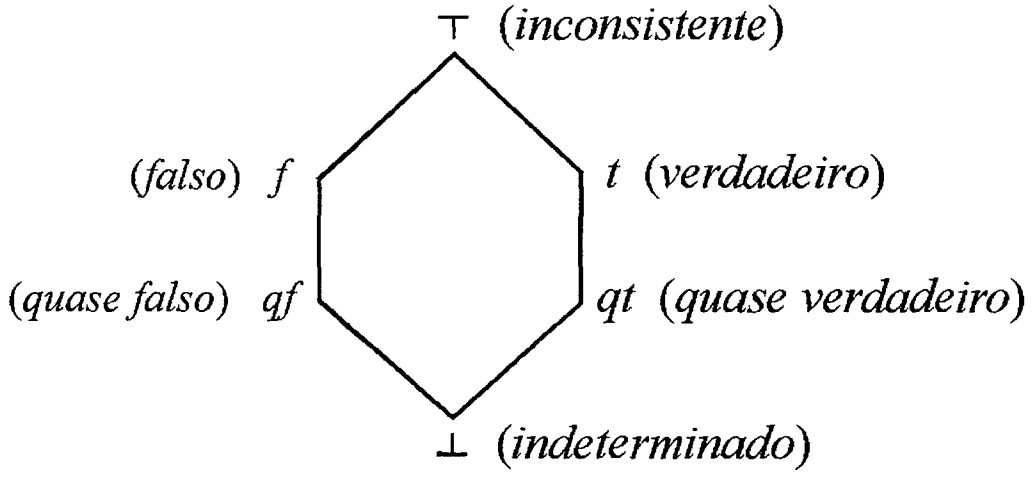


Figura 3.2: Reticulado SIX

consistente", "verdadeiro", "falso"e "indeterminado". Note que t e f são incomparáveis, ou seja, não é possível comparar t e f para saber qual possui maior grau de crença.

Seja $\mathcal{T}$ o reticulado FOUR e $k_\top$ uma proposição anotada, onde k significa "vai chover hoje". Lê-se $k_\top$ como "creio que vai chover hoje com grau de crença inconsistente".

Outro reticulado é o SIX (veja figura 3.2), $\top \geq t \geq qt \geq \bot$, $\top \geq f \geq qf \geq \bot$, onde qt e qf denotam respectivamente "quase verdadeiro"e "quase falso". Por exemplo, seja $\mathcal{T}$ o reticulado SIX, e k_{qf} uma proposição anotada, onde k significa "vamos ao cinema". Lê-se k_{qt} como "creio que vamos ao cinema com grau de crença quase verdadeiro".

Existem lógicas anotadas com duas anotações, ou seja, uma proposição é anotada por duas constantes de anotação, como por exemplo a lógica estudada em [3].

Além do reticulado, a lógica anotada possui a função $\neg : \mathcal{T} \to \mathcal{T}$, onde $\neg$ denota a negação (fraca) própria das lógicas paraconsistentes [18].

Em particular, se $\mathcal{T}$ é o intervalo $[0,1] \subseteq \Re$ e $\neg(x) = 1 - x$ então a lógica anotada pode desempenhar os modos de raciocínio da lógica fuzzy [16].

Os conectivos da lógica anotada são denotados por $\neg$ (negação), $\wedge$ (conjunção), $\vee$ (disjunção) e $\to$ (implicação). A precedência e os parênteses são usados de modo usual.

Por exemplo, seja $\mathcal{T}$ o reticulado SIX, $k_{qt} \wedge k_f \to k_\top$ uma fórmula onde a proposição k significa "vai chover". Lê-se "se creio que vai chover com grau de crença quase verdadeiro e creio que vai chover com grau de crença falso então creio que vai chover com grau de

crença inconsistente". O conceito de fórmula será apresentado mais adiante de modo formal.

A seguir será apresentada a sintaxe e a semântica de uma lógica paraconsistente anotada proposicional.

3.3 Sintaxe da lógica anotada proposicional

A linguagem da lógica anotada proposicional (LAP) possui os seguintes símbolos primitivos:

- símbolos proposicionais $k, l, m, ...$;
- conectivo unário $\neg$ (negação);
- conectivos binários $\wedge$ (conjunção), $\vee$ (disjunção) e $\rightarrow$ (implicação);
- cada elemento μ, ν, ... de $\mathcal{T}$ é uma *constante de anotação*, onde $\mathcal{T}$ é um reticulado completo na ordem $\geq$;
- símbolos auxiliares: parênteses.

Definição 3.3.1 Uma *expressão* é qualquer seqüência finita de símbolos de seu vocabulário.

Exemplo 3.3.1 São expressões:

- $\neg\vee)l\neg mk$
- $\rightarrow$
- $\neg(k_\mu \vee \neg m_\mu)$

Parece que somente a terceira "quer dizer algo". Precisamos assim, caracterizar as expressões relevantes para nosso discurso. Tais expressões são descritas pela gramática da LAP.

Definição 3.3.2 Um *literal proposicional* ou *literal* é uma proposição anotada por uma constante de anotação escrito na forma k_μ. Literais serão denotados por K, L, M, etc.

Definição 3.3.3 Para qualquer literal L, a fórmula $\neg^n L$ é chamada de *hiperliteral*; se L é um hiperliteral, então $\neg^n L = k_{\neg^n \mu}$, onde $\neg : \mathcal{T} \to \mathcal{T}$ denota uma função fixa (que fornece o significado da negação), e n é o fator multiplicador (um número natural $\neq 0$).

Exemplo 3.3.2 Seja $\mathcal{T}$ o reticulado FOUR e a função de negação $\neg : \mathcal{T} \to \mathcal{T}$ definida da seguinte forma:

$\neg : \top \to \top$

$\neg : t \to f$

$\neg : f \to t$

$\neg : \bot \to \bot$

assim, por exemplo $\neg k_t = k_{\neg t} = k_f$, $\neg k_f = k_{\neg f} = k_t$, $\neg k_\bot = k_{\neg \bot} = k_\bot$.

Definição 3.3.4 As *fórmulas* são obtidas pela seguinte definição indutiva generalizada:

- Qualquer literal ou hiperliteral é uma fórmula (atômica).
- se F e G são fórmulas quaisquer, então $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \to G)$ são fórmulas;
- uma expressão constitui uma fórmula se e somente se foi obtida por aplicação de uma das regras anteriores.

A fórmula $\neg F$ é lida negação de F; $(F \wedge G)$, conjunção de F e G; $(F \vee G)$, disjunção de F e G; $(F \to G)$, implicação de G por F.

Definição 3.3.5 Uma *fórmula complexa* é qualquer fórmula que não seja um literal ou hiperliteral.

A precedência dos conectivos em ordem decrescente é $\neg$, $\wedge$, $\vee$ e $\to$. Os parênteses serão usados de modo usual.

Definição 3.3.6 Seja F uma fórmula qualquer, a *negação forte* de F é definida por $\neg^* F = F \to (\neg(F \to F) \wedge (F \to F))$.

3.4 Semântica da lógica anotada proposicional

Introduzimos o conceito de interpretação da LAP:

Definição 3.4.1 Seja A o conjunto de símbolos proposicionais da LAP e B o conjunto de fórmulas da LAP. Uma interpretação para LAP é uma função $I : A \to \mathcal{T}$. Dada uma interpretação I, podemos associar uma valoração $V_I : B \to \{0, 1\}$ assim definida:

- Se $k \in A$ e $\mu \in \mathcal{T}$, então $V_I(k_\mu) = 1$ se e somente se $I(k) \geq \mu$, $V_I(k_\mu) = 0$ se e somente se não é o caso que $I(k) \geq \mu$.
- Se A é da forma $\neg^n k_\mu$ $(n \geq 1)$, então $V_I(\neg^n(k_\mu) = V_I(\neg^{n-1}(k_{\neg\mu}))$.
- Sejam F e G fórmulas quaisquer. Então,
- $V_I(F \wedge G) = 1$ se e somente se $V_I(F) = V_I(G) = 1$;
- $V_I(F \vee G) = 1$ se e somente se $V_I(F) = 1$ ou $V_I(G) = 1$;
- $V_I(F \to G) = 1$ se e somente se $V_I(F) = 0$ ou $V_I(G) = 1$;
- $V_I(\neg F) = 1 - V_I(F)$ se e somente se F não é um hiperliteral.

Temos que $V_I(k_\mu) = 1$ se e somente se $I(k) \geq \mu$, ou seja, k_μ, é verdadeira segundo a interpretação I se a interpretação dada a $k, I(k)$, for maior ou igual a "meu valor de crença"μ com respeito à proposição k. Ela é falsa em caso contrário.

Pode-se mostrar que há interpretações I e proposições k_μ tais que $V_I(k_\mu) = 1$ e $V_I(\neg k_\mu) = 1$, ou seja, temos contradições verdadeiras nesta lógica. Isto é intuitivo se considerarmos proposições do tipo $k_\top$. Sua negação $\neg k_\top$ equivale a $k_{\neg\top}$ que é também $k_\top$.Assim, se k_μ for verdadeira, então é claro que sua negação também é verdadeira. Se ela for falsa, sua negação também é falsa.

3.5 Postulados da LAP

Os postulados (esquemas de axiomas e regras de inferência) da LAP são os seguintes: F, G e H são fórmulas quaisquer, I e J são fórmulas complexas, k é um símbolo proposicional e μ, ν são constantes de anotação.

$(\rightarrow_1)$ $F \rightarrow (G \rightarrow F)$

$(\rightarrow_2)$ $F \rightarrow (G \rightarrow H) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$

$(\rightarrow_3)$ $((F \rightarrow G) \rightarrow F \rightarrow F)$

$(\rightarrow_4)$ $\frac{F, F \rightarrow G}{G}$ (Modus Ponens MP)

$(\wedge_1)$ $F \wedge G \rightarrow F$

$(\wedge_2)$ $F \wedge G \rightarrow G$

$(\wedge_3)$ $F \rightarrow (G \rightarrow (F \rightarrow G))$

$(\vee_1)$ $F \rightarrow F \vee G$

$(\vee_2)$ $G \rightarrow F \vee G$

$(\vee_3)$ $(F \rightarrow H) \rightarrow ((G \rightarrow H) \rightarrow ((F \vee G) \rightarrow H))$

$(\neg_1)$ $(I \rightarrow J) \rightarrow ((I \rightarrow \neg J) \rightarrow \neg I)$

$(\neg_2)$ $I \rightarrow (\neg I \rightarrow F)$

$(\neg_3)$ $I \vee \neg I$

$(\mathcal{T}_1)$ $k_\perp$

$(\mathcal{T}_2)$ $\neg^n k_\mu \rightarrow \neg^{n-1} k_{\neg\mu}, n \geq 1$

$(\mathcal{T}_3)$ $k_\mu \rightarrow k_\nu, \mu \geq \nu$

$(\mathcal{T}_4)$ $k_{\mu 1} \wedge ... \wedge k_{\mu n} \rightarrow k_\mu$, onde $\mu = \vee_{i=1}^{n}$, $n \geq 2$. Se o reticulado é finito, então $\vee$ denota o supremo.

Definição 3.5.1 Uma seqüência finita de fórmulas $(F_1, F_2, ..., F_n)(n \in N, n \geq 1)$ chama-se *demonstração* ou *prova* se para cada $i, 1 \leq i \leq n$ temos que:

- ou F_i é um axioma;
- ou F_i foi obtida de duas fórmulas anteriores da seqüência pela aplicação da regra de *Modus Ponens.*

Definição 3.5.2 Uma fórmula F denomina-se *teorema* se existir uma demonstração $(F_1, F_2,$ $..., F_n)$ tal que $F_n = F$, ou seja, F é a última fórmula da referida seqüência.

A seqüência $(F_1, F_2, ..., F_n)$ da definição acima denomina-se *prova* ou *demonstração* do teorema F. Neste caso simbolizamos por $\vdash F$.

Teorema 3.5.1 Em LAP, $\neg^*$ possui todas as propriedades da negação clássica. Por exemplo temos:

- $\vdash F \vee \neg^* F$
- $\vdash \neg^*(F \wedge \neg^* F)$
- $\vdash (F \rightarrow G) \rightarrow ((F \rightarrow \neg^* G) \rightarrow \neg^* F)$
- $\vdash F \rightarrow \neg^* \neg^* F$
- $\vdash \neg^* F \rightarrow (F \rightarrow G)$
- $\vdash (F \rightarrow \neg^* F) \rightarrow G$

Corolário 3.5.1 Em LAP, os conectivos $\neg^*, \wedge, \vee$ e $\rightarrow$ possuem todas as propriedades da negação, conjunção, disjunção e implicação clássicas, respectivamente.

Corolário 3.5.2 Seja B um conjunto de fórmulas complexas, C um conjunto de literais e $A = B \cup C$. Em A, os conectivos $\neg, \wedge, \vee$ e $\rightarrow$ possuem todas as propriedades da negação, conjunção, disjunção e implicação clássicas, respectivamente. Note que os hiperliterais não pertencem ao conjunto A.

Corolário 3.5.3 Nas fórmulas complexas e nos literais da LAP, intuitivamente, os conectivos $\neg$ e $\neg^*$ possuem o mesmo significado.

Corolário 3.5.4 O cálculo proposicional clássico está contido em LAP e este constitui um sub-cálculo estrito do primeiro.

Definição 3.5.3 Uma teoria T é *trivial* se qualquer sentença da linguagem de T for um teorema; em hipótese contrária, T é não trivial.

Teorema 3.5.2 LAP é inconsistente mas não trivial.

Na LAP as contradições somente aparecem entre os literais atômicos. Assim, percebemos, a modo de resumo, que as fórmulas complexas obedecem aos postulados da lógica

clássica com respeito aos conectivos primitivos, e se considerarmos todas as fórmulas da linguagem da LAP, considerando-se a negação forte e demais conectivos (excetuando-se, obviamente, a negação fraca), teremos também a lógica clássica.

Teorema 3.5.3 A LAP é *correta* com respeito à semântica discutida, isto é, se A é um teorema, então A constitui uma fórmula logicamente válida de LAP.

Teorema 3.5.4 A LAP é *completa* com respeito à semântica discutida, isto é, se A é uma fórmula logicamente válida, então A constitui um teorema de LAP.

Veja as demonstrações em [17].

3.6 Conclusão

No início deste capítulo foi apresentado um histórico, onde foram brevemente citados alguns dos trabalhos sobre lógica anotada. Em seguida foi dada uma visão geral de uma determinada lógica anotada, sua sintaxe, semântica, axiomas e por fim algumas conseqüências que podemos inferir de tal lógica. No próximo capítulo será apresentado o método das conexões para esta lógica anotada.

CAPÍTULO 4

O MÉTODO DAS CONEXÕES DE BIBEL PARA UMA LÓGICA ANOTADA PROPOSICIONAL

Nos últimos capítulos foram apresentados uma lógica anotada e o método das conexões de Bibel para a lógica clássica. Agora será apresentada a adaptação do método de Bibel para a lógica anotada.

Em 1996, C.A.A. Kaestner e D. Krause [25] estenderam pela primeira vez o método das conexões de Bibel para a lógica anotada proposicional. Foram definidas a sintaxe abstrata de uma matriz, a semântica e os conceitos de caminhos e conexões. O método de Bibel [5] foi adequadamente modificado para ser aplicado a tais lógicas, porém, não perdendo sua essência. As principais mudanças feitas foram na sintaxe dos literais, inclusão da definição de hiperliterais, na semântica e na definição de literais complementares. Se mantiveram inalterados a definição de matrizes, fórmulas (não atômicas), caminhos e caracterização de validade.

O que será apresentado neste capítulo é a continuação de [25]. A definição de literais foi alterada para simplificar a implementação do método. A noção de polaridade do literal é equivalente à negação (das fórmulas complexas) da lógica anotada, enquanto que em [25] a polaridade do literal tinha um significado independente. E em [5], o significado da polaridade do literal é o mesmo da negação clássica. Devido a esta alteração na definição de literais, algumas alterações foram necessárias na semântica e na definição de literais complementares. Foi também incluída a definição de decomposição, sem a qual o método não seria completo, pois, como será exemplificado mais adiante, o método não conseguiria provar todos os teoremas.

A principal mudança em relação ao método de Bibel para a lógica clássica [5] foi na noção de literal complementar. De modo semelhante, C. Kreitz and J. Otten [29] concluíram que, para adaptar o método de Bibel para as lógicas intuicionista e modal, a

principal mudança seria também na noção de literal complementar.

De modo intuitivo, possivelmente para adequar o método de Bibel para outras lógicas não clássicas será necessário fazer outras alterações na noção de literal complementar.

A seguir é descrito em detalhes o método das conexões de Bibel para a lógica anotada.

4.1 Conceitos e sintaxe abstrata de uma matriz

No capítulo anterior, a negação dos hiperliterais, literais e fórmulas complexas foi denotada por $\neg$. Para tornar as definições mais claras, neste capítulo a negação dos hiperliterais será denotada por $\neg$ enquanto dos literais e fórmulas complexas por $\sim$.

Seja A um conjunto finito, ordenado e não vazio de símbolos proposicionais. Para cada elemento $a \in A$, um reticulado $\mathcal{T}_a$ não vazio e finito é associado. Os elementos de $\mathcal{T}_a$ serão chamados de *constantes de anotação* e denotados por μ, ν, etc.

Definição 4.1.1 Um *literal proposicional* é uma tripla (a, μ, p) onde $a \in A$, $\mu \in \mathcal{T}_a$ e $p \in \{0, 1\}$. p é a polaridade do literal. Nós iremos escrever $\sim L$ ou $\sim L_\mu$ para denotar o literal $(a, \mu, 1)$, e L ou L_μ para denotar o literal $(a, \mu, 0)$. Literais serão denotados por K, L, M, etc.

Seja R, um alfabeto de ocorrências ou posições. Os elementos de R são denotados por r.

Definição 4.1.2 Por indução, nós definimos o conceito de matrizes (proposicionais) sobre (A, R), denotado por D, E, F, sendo seu tamanho $\sigma(F)$, suas posições $\Omega(F) \in R$ e sua profundidade $\delta(r)$ de r em F para qualquer $r \in \Omega(F)$:

- Para qualquer literal L e para qualquer $r \in R$, o par $(L, r) = L^r$ é uma matriz com $\sigma(L^r) = 0, \Omega(L^r) = \{r\}$ e $\delta(L^r) = 0$.
- Se $F_1, ..., F_n$, $n \geq 0$ são matrizes tais que $\Omega(F_i) \cap \Omega(F_j) = 0$ para $i \neq j$ e $1 \leq i, j \leq n$, então o conjunto $F = \{F_1, ..., F_n\}$ é uma matriz onde:
  - $\sigma(\emptyset) = 0$ para $n = 0$ e $\sigma(F) = 1 + \sum_{i=1}^{n} \sigma(F_i)$ para $n > 0$;

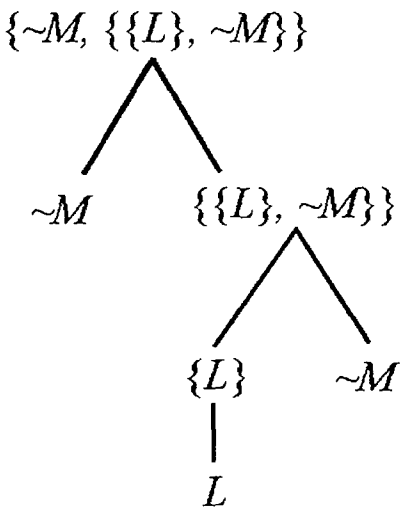


Figura 4.1: Árvore correspondente a matriz $\{\sim M^0, \{\{L^1\}, \sim M^2\}\}$

- $\Omega(F) = \Omega(F_1) \cup ... \cup \Omega(F_n)$;
- $\delta(r) = m + 1$ para qualquer $r \in \Omega(F_i), 1 \leq i \leq n$, onde m é a profundidade de r em F_i.

De acordo com esta definição, as partes atômicas de matrizes são proposições, e em geral uma matriz é um conjunto aninhado de ocorrências de literais.

Exemplo 4.1.1 Vamos considerar $A = (a, b, c, d)$ associado com os elementos do reticulado FOUR, e $R = \{0, 1, 2, 3\}$ sendo um alfabeto de posições. Então $L = (a, \perp, 0)$ e $\sim M = (c, f, 1)$ são proposições, enquanto $\{\{L^0\}, \{\sim M^1\}\}$ e $\{\sim M^0, \{\{L^1\}, \sim M^2\}\}$ são matrizes sobre (A, R).

Uma matriz pode também ser vista como uma *árvore* , onde algumas folhas são associadas com literais. A figura 4.1 mostra a árvore correspondente a segunda matriz do exemplo acima.

Definição 4.1.3 Seja F uma matriz e $l, m \in \{0, 1\}$. O conjunto de *fórmulas* $\tilde{F}$ representadas por F com respeito a (l, m) são indutivamente definidas a seguir:

- Se F é um literal $F = L^r$ e $l = 0$ então $\tilde{F} = L$;
- Se F é um literal $F = L^r$ e $l = 1$ então $\tilde{F} =\sim L$;

- Se $F = \{F_1, ..., F_n\}$, $n \geq 0$ e se $m = 1$, então $\tilde{F} = \wedge(\tilde{F}_1, ..., \tilde{F}_n)$, onde $\tilde{F}_i$ são fórmulas representadas por F_i com respeito a $(l, 0), i = 1, ..., n$;

- Se $F = \{F_1, ..., F_n\}$, $n \geq 0$ e se $m = 0$, então $\tilde{F} = \vee(\tilde{F}_1, ..., \tilde{F}_n)$, onde $\tilde{F}_i$ são fórmulas representadas por F_i com respeito a $(l, 1), i = 1, ..., n$.

Definição 4.1.4 Uma fórmula $\tilde{F}$ é *positivamente representada* por uma matriz F se ela é representada por F com respeito a $l = m = 0$; $\tilde{F}$ é *negativamente representada* se ela é representada por F com respeito a $l = m = 1$. Uma *fórmula proposicional* é qualquer fórmula representada por uma matriz. Fórmulas são também denotadas por D, E, F.

Nós utilizamos as seguintes notações:

- Se $n = 0$, $\wedge(F_1, ..., F_n) = \wedge()$ é abreviado por $\mathbf{T}$, e $\vee()$ por $\mathbf{F}$;

- Se $n = 1$, ambos $\wedge(F)$ e $\vee(F)$ são abreviados por F;

- Se $n \geq 2$, $\wedge(F_1, ..., F_n)$ é uma *conjunção* e $\vee(F_1, ..., F_n)$ é uma *disjunção* ;

- Para qualquer literal L, a fórmula $\neg^k L$ é chamada de *hiperliteral*; se L é um hiperliteral, então $\neg^k L = (\alpha, \neg^k(\mu), p)$, onde $\neg : \mathcal{T}_a \longrightarrow \mathcal{T}_a$ denota uma função fixa (que fornece o significado da negação), e k é o fator multiplicador (um número natural);

- Se $F = \wedge(F_1, ..., F_n)$, $n \geq 0$, então $\sim F = \vee(\sim F_1, ..., \sim F_n)$;

- Se $F = \vee(F_1, ..., F_n)$, $n \geq 0$, então $\sim F = \wedge(\sim F_1, ..., \sim F_n)$.

- Para qualquer fórmula F, $\sim\sim F = F$;

- Qualquer fórmula $\sim F \vee G$ pode ser escrita como $F \rightarrow G$;

- Qualquer fórmula $(F \rightarrow G) \wedge (G \rightarrow F)$ pode ser escrita como $F \leftrightarrow G$;

- Nós também convencionamos que a ordem de precedência diminui na seqüência $\sim, \wedge, \vee, \rightarrow, \leftrightarrow$. Qualquer parênteses que seja redundante a esta convenção pode ser excluído.

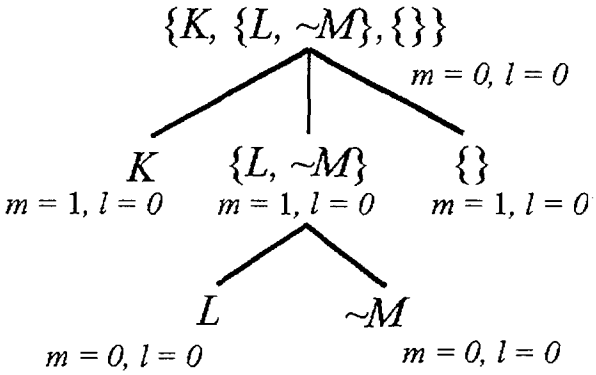


Figura 4.2: Representação positiva de $F = \{K, \{L, \sim M\}, \{\}\}$

Note que se F e G são fórmulas, então $\neg(F \rightarrow G)$ e $\neg \sim F$ por exemplo, não são fórmulas.

De acordo com as convenções acima, toda fórmula bem formada (definida do modo usual) determina uma única matriz; por outro lado, uma matriz pode representar mais de uma fórmula.

Exemplo 4.1.2 Seja $F = \{K, \{L, \sim M\}, \{\}\}$ uma matriz. A *árvore* na figura 4.2 é a representação positiva de F.

Exemplo 4.1.3 As fórmulas $K \wedge L \rightarrow M$, $K \rightarrow \sim L \vee M$, $L \wedge K \rightarrow M$, são todas representadas por $\{\sim K^1, \sim L^2, \sim M^3\}$.

Os resultados apresentados em [5], capítulo 2, continuam aplicados aqui, tal como se segue:

- Se a fórmula $\tilde{F}$ é positivamente representada pela matriz F, então $\sim \tilde{F}$ é negativamente representada por F;
- Se duas fórmulas $\tilde{F}_1$ e $\tilde{F}_2$ são positivamente representadas pela mesma matriz F, então $\tilde{F}_1$ e $\tilde{F}_2$ são logicamente equivalentes no senso da lógica anotada [9].

Estes resultados justificam o uso de matrizes em vez de fórmulas.

O exemplo a seguir justifica o nome *matriz* empregado primeiramente por Bibel [5], e também usado aqui (veja também [6]).

Exemplo 4.1.4 Seja $\tilde{F}$ a seguinte fórmula:

$$(K \wedge \sim L \rightarrow \sim N) \wedge M \wedge \neg L \rightarrow (\sim N \wedge \sim K)$$

onde K, L, M e N são literais; se nós colocarmos $\tilde{F}$ na forma normal disjuntiva (como usual), iremos obter:

$$(K \wedge \sim L \wedge N) \vee \sim M \vee \sim \neg L \vee (\sim N \wedge \sim K)$$

Esta fórmula pode ser apresentada em um arranjo bidirecional, onde os literais de uma determinada coluna são conectados por "$\wedge$", e as colunas são conectadas por "$\vee$", como se segue:

$$F = \begin{bmatrix} K & & & \sim N \\ \sim L & \sim M & \sim \neg L & \\ N & & & \sim K \end{bmatrix}$$

4.2 Semântica

Definição 4.2.1 Uma *interpretação* $\mathcal{M}$ é uma função que associa um elemento do reticulado com todos símbolos proposicionais. Através da denotação de $\mathcal{M}(a) = \mu_a$, nós podemos escrever $(a, b, c, ...) \mapsto (\mu_a, \mu_b, \mu_c, ...)$.

Vamos considerar as duas matrizes "especiais" introduzidas anteriormente: $\vee() = \mathbf{F} = \emptyset = \{\}$ e $\wedge() = \mathbf{T} = \{\emptyset\} = \{\{\}\}$.

Agora nós iremos definir o "valor verdade" $\mathcal{M}(F)$ da matriz F como se segue:

- Se F é um literal $(a, \mu, 0)$, então $\mathcal{M}(F) = \mathbf{T} = \{\emptyset\}$ see $\mathcal{M}(a) \geq \mu$, senão $\mathcal{M}(F) = \mathbf{F} = \emptyset$.

- Se F é um literal $(a, \mu, 1)$, então $\mathcal{M}(F) = \mathbf{F} = \emptyset$ see $\mathcal{M}(a) \geq \mu$, senão $\mathcal{M}(F) = \mathbf{T} = \{\emptyset\}$.

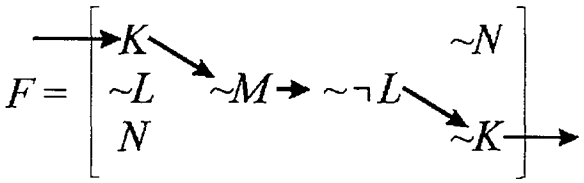


Figura 4.3: O caminho $\{K, \sim M, \sim \neg L, \sim K\}$ através da matriz F

- Se F é uma matriz $F = \{F_1, ..., F_n\}, n \geq 0$, então $\mathcal{M}(F) = \bigcup_{k=1}^{n} \mathcal{M}(F_k)$ quando $m = 0$ e $\mathcal{M}(F) = \bigcap_{k=1}^{n} \mathcal{M}(F_k)$ quando $m = 1$.

Nós escreveremos $\mathcal{M}$ sat $\tilde{F}$ (e também $\mathcal{M}$ sat F) see $\mathcal{M}(F) = \mathbf{T}$ para uma matriz F que representa $\tilde{F}$.

Definição 4.2.2 Uma matriz F é válida see $\mathcal{M}(F) = \mathbf{T}$ para todas interpretações. Ela é chamada contraditória see $\mathcal{M}(F) = \mathbf{F}$ para todas interpretações.

Nós podemos verificar que se F é válida, então $\sim F$ é contraditória, e o inverso também é verdadeiro.

4.3 Caminhos, conexões, e caracterização de validade

Definição 4.3.1 Um *caminho* através de uma matriz F é um conjunto de ocorrências de literais, definidos como a seguir:

- Se $F = 0$, então o único caminho através de F é $\emptyset$;
- Se $F = L^r$, então o único caminho através de F é o conjunto $\{L^r\}$;
- Se $F = \{F_1, ..., F_m, F_{m+1}, ..., F_{m+n}\}, m, n \geq 0, m+n \geq 1$ para m literais $F_1, ..., F_m$ e para n matrizes que não são literais $F_{m+1}, ..., F_{m+n}$, então para qualquer matriz $E_i \in F_{m+i}$ e para qualquer caminho p_i através de $E_i, 1 \leq i \leq n$, o conjunto $\bigcup_{j=1}^{m}\{F_j\} \cup \bigcup_{i=1}^{n} p_i$ é um caminho através de F.

Exemplo 4.3.1 Seja F, a matriz do exemplo 4.1.4; um caminho através de F são todos os caminhos da esquerda para a direita que passam através dos literais (interpretados como *ligações*) como mostrado na figura 4.3.

Definição 4.3.2 Os literais $L = (a, \mu, p)$ e $M = (a, \nu, q)$ são complementares see:

- $p = 0$, $q = 1$ and $\nu \geq \mu$, or
- $p = 1$, $q = 0$ and $\mu \geq \nu$.

Definição 4.3.3 Caminhos que possuem literais complementares como elementos são chamados de *conexões*.

Exemplo 4.3.2 Seja $\mathcal{T}_a$ o reticulado FOUR, mencionado acima, $\sim K = (a, t, 1)$, $\sim L = (a, f, 1)$, $M = (a, \top, 0)$, e a matriz $F = \{\sim K, \sim L, M\}$. O único caminho através de F não possui literais complementares. Para ver isto, observe que $\sim K$ e $\sim L$ não são complementares com M. Alternativamente observe que $\top \not\leq t$ e $\top \not\leq f$ e , logo, M não pode conectar com $\sim K$ nem $\sim L$.

Definição 4.3.4 Nós dizemos que uma n-tupla ordenada $(\mu_1, ..., \mu_n)$, $n > 1$, de elementos não comparáveis de $\mathcal{T}_a$ é uma *decomposição* de μ se $\mu = \sqcup\{\mu_1, ..., \mu_n\}$ e não existe elementos não comparáveis $\{\mu'_1, ..., \mu'_n\}$ of $\mathcal{T}_a$ tal que $\mu'_i < \mu_i$ e $\mu = \sqcup\{\mu'_1, ..., \mu'_n\}$.

Note que a definição de decomposição obedece o último postulado da lógica anotada apresentado no capítulo anterior. Note também que, em termos de implementação, esta definição é eficiente para pequenos reticulados discretos.

Exemplo 4.3.3 Considere o reticulado completo FOUR. Então, a única decomposição do elemento $\top$ são os elementos $\{t, f\}$ e logo, a matriz do último exemplo agora é $F = \{\sim K, \sim L, \{M_1, M_2\}\}$, onde $M_1 = (a, t, 0)$ e $M_2 = (a, f, 0)$. Observe que K é complementar com M_1 e L é complementar com M_2. Alternativamente, observe que $t \leq t$ e $f \leq f$.

O resultado acima nos mostra que o método das conexões de Bibel é completo para a lógica anotada somente quando fazemos uso da decomposição.

Proposição 4.3.1 (Correto e Completo): Uma matriz F é válida se e somente se todos os caminhos através F são conexões ou a matriz é da forma $\{L_\mu\}$, onde μ é o elemento ínfimo do reticulado $\mathcal{T}_a$.

4.4 Verificando a validade de uma fórmula

Prova de teoremas, em nosso caso, consiste de:

1. Construir uma matriz contendo todas as premissas (base de conhecimentos) e um objetivo (teorema).

2. Checagem da validade desta matriz.

Suponha que temos um conjunto de fórmulas $\Gamma = \{F_1, ..., F_n\}$ e uma questão G. Então, para investigar se G é uma conseqüência semântica do conjunto de fórmulas de Γ, nós devemos verificar se a matriz de $(\bigwedge_{i=1}^{n} F_i) \rightarrow G$ é válida. O caso paraconsistente está presente neste procedimento devido a nossa definição de literal complementar. De fato, neste caso a existência de um literal $L = (a, \mu, p)$ e sua "negação" $\neg L = (a, \neg(\mu), p)$ não é condição suficiente para assegurar literais complementares no caminho. Isto exemplifica perfeitamente as idéias de programa paraconsistente geral [18]. Note que para obter a prova de uma questão, é necessário que todos os caminhos da matriz $\sim (\bigwedge_{i=1}^{n} F_i) \vee G$ possuam conexões, que implica a existência de literais complementares em todos caminhos.

4.5 Conclusão

Neste capítulo foi apresentada a adaptação do método das conexões de Bibel para uma lógica anotada proposicional [28]. A checagem de validade de uma fórmula foi feita através da verificação de todos caminhos da matriz. No próximo capítulo será apresentado um algoritmo mais eficiente para verificar a validade de uma fórmula e também uma analogia entre o método das conexões de Bibel e o método tableau.

CAPÍTULO 5

OTIMIZAÇÃO DO MÉTODO DAS CONEXÕES DE BIBEL

Segundo o que foi apresentado no capítulo anterior, é necessário que todos os caminhos de uma matriz possuam literais complementares para concluir que uma fórmula é válida. Deste modo, um algoritmo deve percorrer todos os caminhos da matriz procurando por literais complementares.

Definição 5.0.1 Levando em conta somente matrizes não aninhadas, seja $M = \{|C_1|, ..., |C_n|\}$ uma matriz com n colunas, onde, $|C_i|$ é o número de literais da coluna i. O *número de caminhos* $|\beta|$ de M é calculado por $|\beta| = \prod_{i=1}^{n} C_i$.

Exemplo 5.0.1 Suponha que temos uma matriz de 10 linhas $\times$ 12 colunas. Aplicando a fórmula $|\beta| = \prod_{i=1}^{n} C_i$, teremos:

$$|\beta| = 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 = 10^{11}$$

Como facilmente é observado, o número de caminhos a serem percorridos aumenta exponencialmente no número de colunas adicionadas à matriz fazendo com que matrizes relativamente pequenas possuam um número grande de caminhos.

Um procedimento que primeiramente liste todos os caminhos (como acontece quando uma fórmula é transformada em sua forma normal conjuntiva) e então verifique as conexões em cada caminho provavelmente irá falhar na primeira fase devido à quantidade de recursos necessários.

Por outro lado, em uma matriz contendo n literais, será necessário verificar apenas n^2 conexões [6], sendo que cada literal pode somente ser conectado com n outros literais e no máximo n vezes (isto é uma estimativa genérica). Como existem muito mais caminhos que conexões, cada conexão determina vários caminhos complementares. Deste modo é mais adequado ter como base de busca as conexões e não os caminhos.

5.1 O procedimento de extensão

O procedimento de extensão [6, 7] busca conexões em uma matriz.

Considere a matriz (a) da figura 5.1. Inicialmente, selecionamos a primeira conexão envolvendo um elemento da cláusula indicada pela seta vertical, a K-conexão. Isto é suficiente para estabelecer todos os caminhos que possuem esta conexão (indicado na matriz (b) da figura 5.1 por um ponto (.) depois de $\sim K$). Neste caso existem exatamente dois caminhos. Agora nós iremos dividir os caminhos que ainda não foram verificados no conjunto de caminhos que contém K (indicado pelo retângulo ao redor de K) e o conjunto de caminhos que não contém K, que em nosso caso são exatamente os caminhos que possuem L como seu primeiro elemento. Nós iremos tratar o segundo conjunto mais tarde, como indicado pela seta diagonal apontando para o primeiro elemento na seqüência de literais abertos. Nós dizemos que esta matriz evoluiu da anterior através de um *passo de extensão.*

No segundo passo manteremos nossa atenção no subconjunto de K-caminhos ainda não verificados. Nós iremos selecionar uma conexão com um literal da cláusula seguinte que é marcada com uma seta vertical. Isto significa, que repetimos exatamente o que foi feito no primeiro passo, mas limitado ao conjunto de caminhos ainda não verificados. Este passo e também o conjunto de passos que o seguem é mostrado na figura 5.1. Considere, por exemplo, o resultado do terceiro passo de extensão que é a matriz (e) figura 5.1. Neste ponto, ambos caminhos (no subconjunto atualmente sobre consideração) passando através de L da segunda cláusula já foram verificados - esta é a razão do ponto colocado depois de L e antes de começar o próximo passo. Agora nós podemos nos concentrar no conjunto de caminhos que passam através do literal M da segunda cláusula, o qual, nós marcamos anteriormente com uma seta diagonal, continuado com as próximas extensões. Para ser possível fazer um passo de extensão, algumas vezes é necessário um passo de truncamento o qual é indicado pelo símbolo $\sim$. Este truncamento é feito quando já verificamos todos os elementos de uma coluna, voltando para o próximo elemento a ser verificado da coluna anterior. Procedendo assim até que todos elementos da primeira coluna tenham sido verificados.

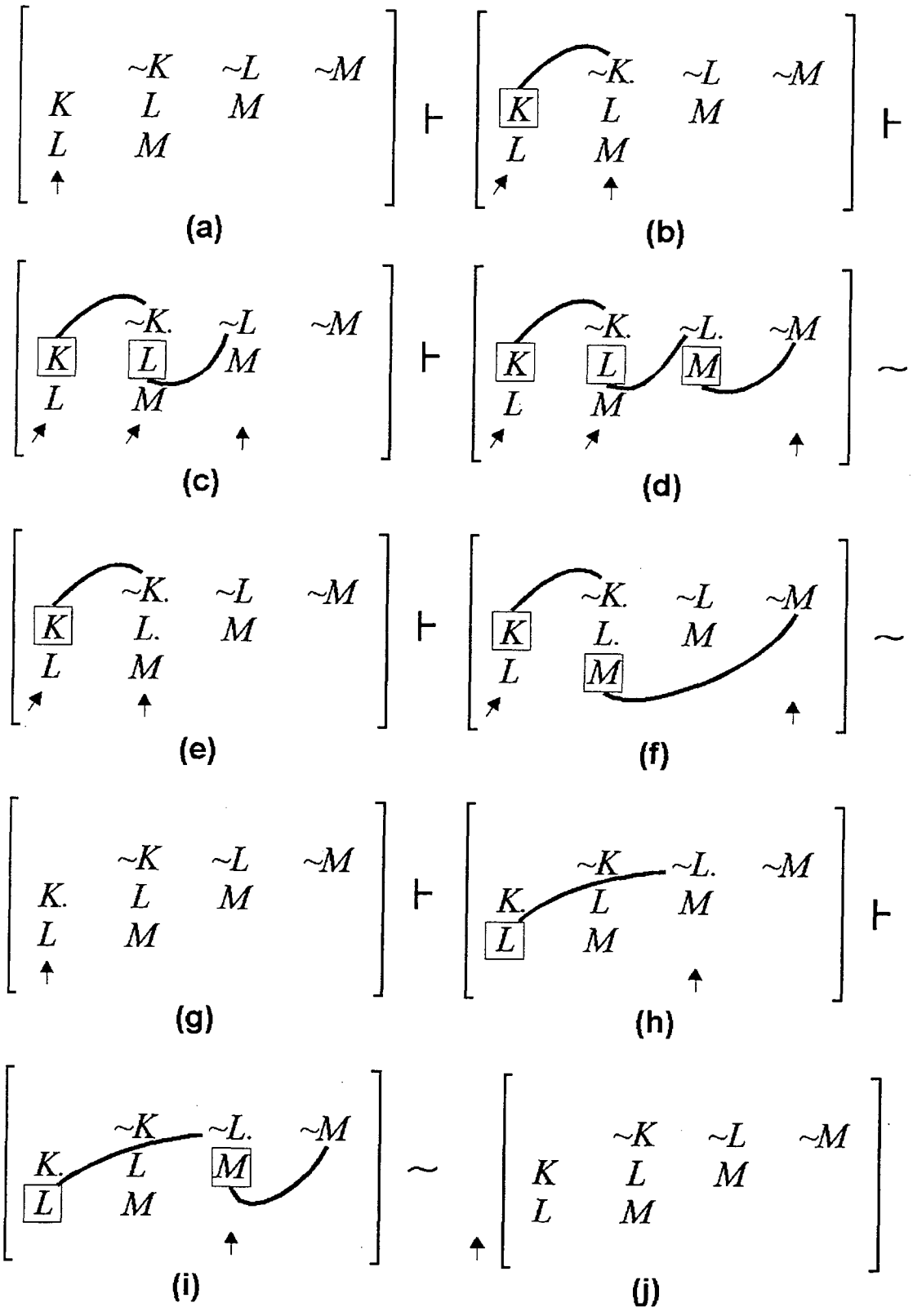


Figura 5.1: Passos do procedimento de extensão

De modo geral, começamos selecionando uma cláusula de início que contém somente literais positivos e selecionando um literal desta cláusula. Um passo de extensão é feito com este literal e os outros literais da cláusula de início são armazenados no conjunto de literais abertos. Outro passo de extensão é feito com um literal da cláusula que contém o literal que foi conectado com o primeiro, não podendo ser o mesmo literal que participou da primeira conexão. Este procedimento é repetido até não existirem literais disponíveis para fazer novas extensões. Então o primeiro elemento na seqüência de literais abertos é selecionado através de um passo de truncamento, e então uma nova série de passos de extensão é feito, e assim por diante. O procedimento termina com sucesso, ou seja, a fórmula é válida, uma vez que o conjunto de literais abertos esteja vazio. Esta seqüência de passos é chamada de derivação (de extensão).

Se este procedimento não possibilitar um passo de extensão devido a falta de uma conexão apropriada, então (assim como é usual em procedimentos de busca) ele faz um retrocesso *(Backtracking)*, voltando para o literal do caminho ativo (literal com um retângulo ao redor) que está envolvido na última conexão, e tenta achar um literal complementar alternativo. Se não consegue, então o procedimento termina com a prova de que a fórmula é inválida.

O requerimento de que a cláusula de início possua somente literais positivos não gera perda da generalidade do procedimento devido ao seguinte fato:

Teorema 5.1.1 Uma matriz válida (e não vazia) contém no mínimo uma cláusula que possui somente literais positivos e no mínimo uma cláusula que possui somente literais negativos.

Prova [6].

Além disto, o procedimento de extensão descrito acima é correto no senso de que o término com sucesso prova que a fórmula é válida e o término sem sucesso prova que a fórmula é inválida. Para fazer esta declaração de forma precisa e completa nós iremos definir agora algumas características relevantes de procedimentos de prova.

Definição 5.1.1 Quando uma fórmula F é provada ser válida por um procedimento de

prova $\mathcal{B}$, nós escrevemos $\vdash F$.

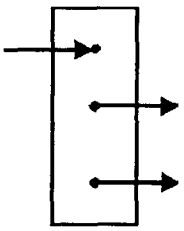
Um procedimento de prova $\mathcal{B}$ é correto quando cada fórmula F a qual o procedimento prove ser válida é de fato válida; isto é, quando $\vdash_{\mathcal{B}} F$ implica $\models F$. $\mathcal{B}$ é completa com respeito a classe de fórmulas quando cada fórmula válida F da classe pode ser provada ser válida em $\mathcal{B}$; isto é, quando $\models F$ implica $\vdash_{\mathcal{B}} F$.

Um procedimento de decisão para uma classe de fórmulas é um procedimento de prova que decide para cada fórmula da classe, em passos finitos, quando ela é válida ou não.

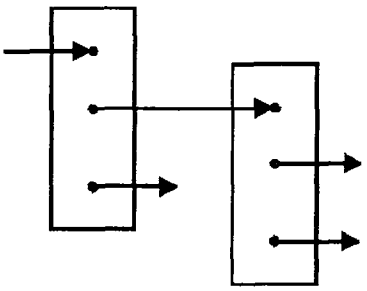
Teorema 5.1.2 O procedimento de extensão é correto e completo.

A declaração de correção é uma contrapartida precisa das afirmações informais feitas acima. A corretude e a completude podem ser vistas seguindo o raciocínio do exemplo dado [6].

Uma característica importante é o *encadeamento linear* (linear chaining). O encadeamento linear refere-se ao modo que as cláusulas são encadeadas por meio de seqüências lineares de conexões, como pode ser visto na figura 5.1. Note a seqüência de conexões $(\{K, \sim K\}, \{L, \sim L\}, \{M, \sim M\})$ que, junto com as quatro cláusulas envolvidas forma uma cadeia linear. Cada cláusula envolvida pode ser representada esquematicamente da seguinte maneira



Uma conexão chega em uma cláusula. No entanto, esta ligação não está presente na cláusula de início. Zero, uma ou mais conexões saem da cláusula, de acordo com o número de literais que ela possui. Contudo, somente uma destas faz parte de cada cadeia, formando a próxima ligação:

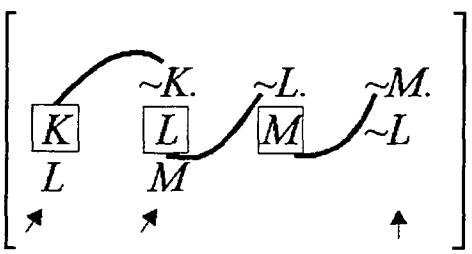


O fim da cadeia é alcançado quando não há mais conexões saindo da cláusula. Cada cláusula está presente no máximo uma vez em cada cadeia. Assim, o comprimento da cadeia é limitado ao número de cláusulas dadas. Uma prova completa pode ser representada como uma árvore formada de cláusulas em cadeias, onde as ramificações na árvore consistem em cadeias lineares.

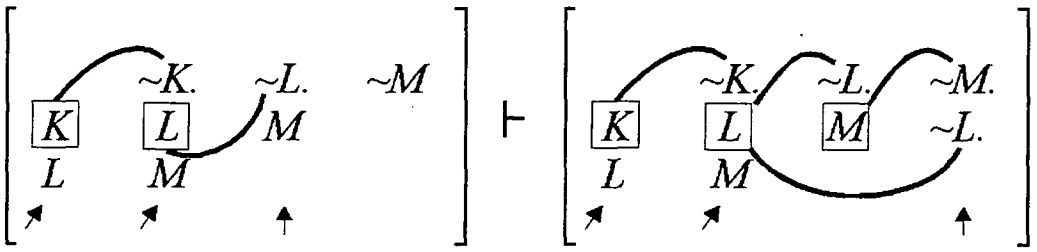
Baseado no que acabamos de apresentar, é possível afirmar que o procedimento de encadeamento é completamente local, ou seja, é determidado pela cláusula atual e suas conexões. Isto significa que ele pode ser implementado de um modo eficiente. Esta é a base da linguagem de programação Prolog. Um programa Prolog é um conjunto de cláusulas de Horn (da lógica de predicados). O interpretador do Prolog é basicamente um procedimento de prova, onde a parte proposicional tem seus fundamentos no procedimento de extensão.

5.2 O procedimento geral de extensão

O procedimento de extensão apresentado na seção anterior é completo somente para a classe de fórmulas de Horn. Para o tratamento de qualquer fórmula proposicional na forma normal nós precisamos fazer somente uma pequena adição no método discutido na seção anterior. Por exemplo, considere novamente a situação apresentada na matriz (d) da figura 5.1. Incluindo o literal $\sim L$ na quarta cláusula temos:



Não é possível fazer um passo de extensão neste ponto. Para adequar o procedimento para o caso geral nós precisamos verificar se existe um literal complementar no caminho ativo que em nosso caso são os literais K, L, M, onde L e $\sim L$ são complementares. No procedimento de extensão além de colocar um ponto (.) no literal da última ligação (o que é feito com $\sim M$ em nosso exemplo), nós também colocamos um ponto em todos literais da cláusula atual que possuírem um literal complementar no conjunto de literais do caminho ativo (literias com um retângulo ao redor). Veja este passo na figura a seguir:



O procedimento geral de extensão funciona exatamente como o procedimento de extensão visto na seção anterior, com a diferença que após cada extensão é aplicado o passo do procedimento geral que acabamos de apresentar. O retrocesso continua sendo necessário em alguns casos.

5.3 Comparação com o método do Tableau

O *tableau analítico* foi introduzido em [4] (veja também [24]) e foi mais elaborado em [35, 22]. Tableaux são árvores; no caso da matriz do exemplo acima, esta árvore tem a forma mostrada na figura 5.2.

A prova é por refutação, assim como na resolução o tableau tenta derivar uma contradição a partir da fórmula a ser provada. A raíz da árvore é a negação da fórmula. Os sucessores de cada nó são os literais de uma das cláusulas da matriz. Nós dizemos que uma ramificação de um tableau é *fechada* se contém dois literais complementares, isto é indicado por um ponto. No exemplo todas as ramificações são fechadas, sinalizando que a fórmula é válida. Para a lógica anotada verificamos se cada ramificação é fechada através da seguinte definição.

Definição 5.3.1 Os literais $L = (a, \mu, p)$ e $M = (a, \nu, q)$ são complementares see:

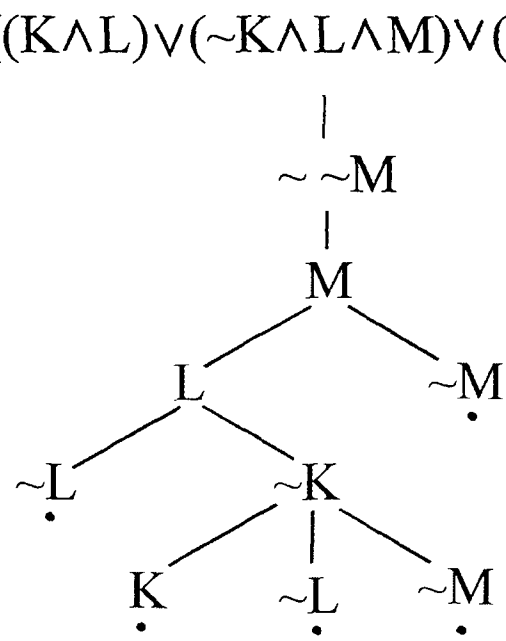


Figura 5.2: Um tableaux fechado da fórmula $(k \wedge L) \vee (\sim K \wedge L \wedge M) \vee (\sim L \wedge M) \vee \sim M$

- $p = 0$, $q = 1$ and $\mu \geq \nu$, ou
- $p = 1$, $q = 0$ and $\nu \geq \mu$.

α	α_1	α_2
$\sim\sim K$	K	K
$K \wedge L$	K	L
$\sim (K \vee L)$	$\sim K$	$\sim L$

Tabela 5.1: Regras α

β	β_1	β_2
$K \vee L$	K	L
$\sim (K \wedge L)$	$\sim K$	$\sim L$

Tabela 5.2: Regras β

As regras formais para construir um tableaux podem ser convenientemente classificadas em α (veja tabela 5.1) e β (veja tabela 5.2). Para toda fórmula pertencente a uma destas classes (por exemplo α), as tabelas fornecem as fórmulas correspondentes reduzidas (por exemplo α_1 e α_2). Através desta classificação, as regras para construir uma árvore são obtidas da seguinte forma:

Primeiramente criamos o primeiro nó da árvore com a cláusula $\sim\sim M$ (poderíamos ter escolhido qualquer disjunção da fórmula para ser o primeiro nó). Em seguida, aplicamos a regra α no primeiro nó para obter o nó M. Então escolhemos uma outra disjunção, diferente da primeira, e aplicamos a regra β. Nós escolhemos a disjunção $\sim (\sim L \wedge M)$ para obter os nós L e $\sim M$ (a regra β separa em duas ramificações). Aqui podemos observar que a ramificação que contém os literais M e $\sim M$ está fechada, ou seja, não precisamos expandir mais esta ramificação. Agora vamos continuar a expandir a ramificação do literal L. Escolhemos uma disjunção diferente da primeira e da segunda. Nós escolhemos a disjunção $\sim (K \wedge L)$ para obter os nós $\sim K$ e $\sim L$ através da regra β. Aqui encontramos mais uma ramificação fechada devido aos literais $\sim L$ e L. Vamos continuar a expandir a ramificação do literal K. Finalmente criamos os nós K, $\sim L$ e $\sim M$ aplicando a regra β na última disjunção que nos restou. Agora verificamos que fechamos os caminhos que possuem os literais K e $\sim K$, L e $\sim L$ e M e $\sim M$. Como todos os caminhos foram fechado, a fórmula é válida.

Proposição 5.3.1 Seja F uma fórmula proposicional qualquer. A quantidade de ramificações do tableaux (completo) de F é igual a quantidade de caminhos da matriz de F e a quantidade de ramificações fechadas do tableaux de F é igual a quantidade de conexões da matriz de F.

Exemplo 5.3.1 A quantidade de conexões da matriz da figura 5.1 é igual a quantidade de ramificações fechadas do tableaux da figura 5.2. Assim, podemos afirmar que no senso do método das conexões a busca da solução no tableaux é orientada a conexões e não a caminhos.

Enquanto que o tableaux apresenta uma visão mais clara da prova, o procedimento de extensão não dispersa as informações, mantendo-as em seus lugares.

Na tese [39] foi demonstrado que o método das conexões de Bibel é equivalente a um determinado tableaux analítico otimizado, dentre outras conclusões interessantes relativas ao método de resolução. Foi afirmado que o método das conexões de Bibel e o método tableaux somente diferem com respeito a representação visual.

5.4 Limitações com respeito à complexidade

Os métodos de prova para lógica proposicional testam quando um elemento do conjunto de todas as fórmulas é um elemento do subconjunto de tautologias, isto é, verifica se a fórmula é válida. Uma versão mais fraca deste problema é testar quando uma fórmula é um elemento do subconjunto de fórmulas satisfazíveis. Ambos problemas são de grande importância na teoria da complexidade. Do mesmo modo, a teoria da complexidade é muito importante para o desenvolvimento de procedimentos de prova [39].

A satisfatibilidade de uma fórmula é um problema *NP-completo*. A validade de uma fórmula é um problema *co-NP-completo*. Veja [39] e referências contidas nesta. Dizendo de modo informal, a conseqüência é que nenhum algoritmo polinomial determinístico para ambos problemas é conhecido. Todos os métodos mencionados (conexões de Bibel, resolução, tableaux,etc...) são exponenciais. Isto significa que existe uma função exponencial f e uma fórmula com n literais que a prova requer pelo menos $f(n)$ passos. Com o aumento linear do tamanho da fórmula, o número de passos da função cresce exponencialmente, rapidamente atingindo os limites dos mais poderosos computadores existentes.

5.5 Conclusão

Neste capítulo foi apresentado o procedimento de extensão e uma breve comparação com o método tableau, onde parece que o método das conexões de Bibel apenas difere do método tableaux em sua representação visual [39].

Porém, além do procedimento de extensão, Bibel definiu também regras de redução, onde são aplicadas várias regras que reduzem o tamanho de determinadas matrizes antes que a prova seja iniciada [7]. Estas reduções não foram levadas em conta na tese [39].

Nos dois próximos capítulos serão apresentadas a linguagem Standard ML e uma implementação do método visto no capítulo anterior em Standard ML, levando em conta o procedimento geral de extensão visto neste capítulo.

CAPÍTULO 6

A LINGUAGEM STANDARD ML

A linguagem Standard ML surgiu na década de 80 e teve sua origem no desenvolvimento de um sistema que pretendia o estudo matemático de sistemas computacionais. Não é por isso de se estranhar que esta linguagem se distingua por possuir um suporte teórico muito rigoroso, e que por isso seja particularmente apreciada pelos investigadores das áreas da Ciência da Computação. Do ponto de vista pedagógico, esta linguagem é atraente porque este rigor na formalização matemática do seu significado se traduz na prática por se apresentar como uma linguagem muito simples, onde os diversos conceitos podem ser entendidos como generalizações de conceitos matemáticos elementares.

6.1 Características

A seguir apresentamos as principais características da linguagem Standard ML:

- *Linguagem Funcional*: Em SML podemos criar funções de ordem superior. Funções podem ser passadas como argumentos, armazenadas em estruturas de dados, retornadas como resultado de chamadas de funções. Funções podem ser estaticamente aninhadas em outras funções. Funções também podem ser criadas em tempo de execução.

- *Definição Formal* : A linguagem SML foi especificada através de uma definição formal, escrita no formalismo relacional [30] [32].

- *Tipos Fortes* : Os tipos em SML seguem o mesmo estilo que o Algol e o Pascal. Se por exemplo, uma função é definida para um tipo inteiro, o interpretador não permite que se passe como parâmetro da função um tipo real. Sendo neste ponto, oposto ao que temos na linguagem Lisp, que não é baseada em tipos fortes.

- *Polimorfismo* : Podemos criar funções polimórficas em SML. Diferente das linguagens orientadas a objetos, onde, implementamos vários procedimentos com o mesmo nome, em SML implementamos uma única função que recebe e retorna parâmetros de diversos tipos.

- *Inferência de Tipos* : Uma das mais importantes características da linguagem SML é sua capacidade de inferir tipos, ou seja, quando declaramos uma função, a SML automaticamente infere os tipos mais adequados para os parâmetros e para o resultado da função.

- *Ligação(binding) Estática* : Por exemplo, se declararmos X com o valor 1, em seguida utilizarmos X na declaração de uma função, e finalmente declararmos novamente X com o valor 2, sempre que aplicarmos a função, esta usará o valor 1 para X, ou seja, os valores válidos para a função, são os valores que estão definidos no momento da definição da função, e não no momento de sua aplicação.

- *Casamento de Padrões* : Esta característica, juntamente com as funções recursivas, provavelmente é o que torna a linguagem SML tão simples e eficiente.

- *Tipos Estruturados e Recursivos* : É possível criar novos tipos a partir dos tipos atômicos, sendo que estes podem ser recursivos. Como exemplo de tipo estruturado recursivo temos o tipo FORMULA declarado no programa do próximo capítulo.

- *Tipos de Dados Imutáveis* : Em SML, as variáveis e estruturas de dados, uma vez criadas e inicializadas, são imutáveis. Isto significa que elas nunca são mudadas ou atualizadas. Podendo esta característica ser vista como vantagem, porque evita que uma parte do programa altere um valor que posteriormente será referenciado por outra parte do programa. Em linguagens funcionais como a SML, criamos novas estruturas a partir das existentes e deixamos as estruturas que não estão sendo mais utilizadas, para serem coletadas pelo *garbage collector*.

6.2 Tipos de dados atômicos

Os tipos de dados atômicos da linguagem SML, são os seguintes:

- *Unit* : possui um único valor que é representado por () ou {}.
- *Bool*: é composto pelos valores verdade *true* e *false* . Dentre as operações disponíveis para este tipo temos: not, andalso, orelse, if...then...else, dentre outras.
- *Int* : possui como domínio os números inteiros. As principais operações disponíveis são: $=, <=, >=, <, >, -$, div e mod.
- *Real* : possui como domínio os números reais. É importante notar, que embora na matemática estejamos habituados a considerar um número inteiro como sendo também um real, em SML, estes dois tipos têm conjuntos distintos. Devemos, por isso, distinguir entre *6* e *6.0* . Quando existe a necessidade de converter um número de real para inteiro, devemos fazer uso da função *real* . Dentre os operadores, temos $=, <=, >=, <, >, +, -$, sin, cos e sqrt.
- *String* : É um tipo de dado que possui como valores, seqüências de caracteres. Estas seqüências são delimitadas por aspas duplas ("). Caracteres podem ser dígitos, letras ou outros símbolos especiais e de controle. Algumas das operações sobre strings são: $=, <=, >=, <, >$, e size.

6.3 Tipos de dados estruturados pré-definidos

No estudo de tipos estruturados, é fundamental a caracterização de:

- *Construtores* : que nos permitem construir valores estruturados a partir de seus constituintes.
- *Destrutores* : Também chamados de seletores ou *pattern matching* , permitem obter as componentes isoladamente.

Os tipos estruturados pré-definidos na linguagem SML são os seguintes:

- *Tuplas* : Este tipo é uma estrutura formada por n elementos de tipos atômicos, sendo permitido elementos de mais de um tipo na mesma n-upla. O construtor deste tipo são os parênteses.

- Registros : Nas tuplas, os elementos são identificados através da posição que ocupam. O problema é que tal representação tende a ser pouco legível, principalmente se levarmos em conta estruturas complexas, formadas por vários elementos. A idéia do registro, é substituir o papel da posição como identificador por um rótulo explicitado pelo utilizador. Dizemos, por isso, que o registro é composto por vários campos, e cada um destes é identificado por um rótulo. Agora, podemos nos referenciar a um campo através de seu rótulo. O construtor deste tipo são as chaves.

- *Listas* : São tipos estruturados que nos permitem representar seqüências de valores de um determinado tipo. Em relação aos tuplos e aos registros, vêm limitada a possibilidade de agregar valores de diversos tipos, mas em contrapartida possuem a enorme flexibilidade de poder agregar um número indeterminado de elementos. O construtor deste tipo são os :: e nil, ou os colchetes. Como operações sobre listas temos hd, tl, size, e também o casamento de padrões.

6.4 Nomes e contextos

Em SML, podemos associar nomes a valores através de operações que nos permitem manipular o contexto onde as expressões são calculadas.

Estas operações sobre os contextos determinam uma segunda classe de frases admissíveis em SML. Até agora, dizíamos que o interpretador SML aceitava como frases, somente expressões, e como resultado apresentava o valor e o tipo desta expressão. Agora vamos considerar uma segunda classe de frases: as declarações, onde o interpretador SML simplesmente altera o contexto onde as próximas expressões serão calculadas.

6.4.1 Declarações globais

A forma mais simples que estas declarações poderão tomar é:

val <identificador>=<expressão>;

O resultado da <expressão> é associado ao <identificador>. É importante notar que a expressão é calculada antes da mudança do contexto.

6.4.2 Declarações locais

Através de declarações locais podemos restringir o contexto dos identificadores à parte do programa onde são estritamente necessárias.

Para isso temos disponível as seguintes formas:

let <declaração> in <expressão> end

local <declaração1> in <declaração2> end .

A diferença entre a forma *let* e *local* é que a primeira faz uma declaração para ser usada no cálculo da expressão e a segunda faz uma declaração para ser usada em outra declaração.

6.5 Casamento de padrões

O mecanismo que fazemos uso para a destruição de estruturas é o casamento de padrões *(pattern matching).*

Este mecanismo não é mais do que uma generalização das declarações já apresentadas onde se permite que o identificador seja substituído por um padrão. Um padrão é o termo onde surgem ocorrências de variáveis (com restrição que cada variável só poderá surgir uma única vez). Deste modo, temos a declaração:

val <padrão>=<expressão>

onde são associados os valores às variáveis intervenientes em <padrão> por forma de que este coincida com o resultado de <expressão>. Vejamos alguns exemplos:

Digitando a seguinte declaração:

> val(x,y) = (5+7,"ABC");

O resultado retornado pelo SML é:

val x = 12: int

val y = "ABC": string

Digitando a seguinte declaração:

> val {nome = x, idade = _} = {nome = "Ana", idade = 18};

O resultado retornado pelo SML é:

val x = "Ana": string

6.6 Funções

A forma mais geral de uma função em SML é:

val<id_função>=fn<padrão1>=><expressão1> | <padrão2>=><expressão2> |...

Exemplo:

Digitando a seguinte declaração:

val adiciona1 = fn x => x + 1;

O resultado retornado pelo SML é:

val adiciona = fn : int− > int

o que quer dizer que a função *adiciona1* é uma função que tem como domínio um inteiro e imagem outro inteiro. Agora para usarmos esta função podemos digitar:

adiciona1 5;

E o resultado retornado pelo SML é:

val it = 6 : int

Se no momento da aplicação da função, o valor passado como parâmetro, casar com <padrão1> então a <expressão1> é calculada, caso contrário, o valor tenta casar com o <padrão2>, e assim por diante. Caso o valor passado não consiga casar com nenhum padrão, é retornado o erro *nonexhaustive match failure* . Este erro significa que não existe um padrão que case com o valor passado, ou seja, a função é parcial, trata somente de parte do domínio dos parâmetros passados.

Existe também uma notação alternativa para a declaração de funções:

fun <id_função> <padrão1>=<expressão1> | <id_função> <padrão2>=<expressão2>

...

6.7 Funções recursivas

As formas de uma função recursiva em SML são:

val rec <id_função = fn <padrão1> => <expressão1> | <padrão2> => <expressão2>

|...

ou

fun <id_função> <padrão1>=<expressão1> | <id_função> <padrão2>=<expressão2>

...

Sendo que a única diferença entre a forma de uma função recursiva e uma função não recursiva, é que na função recursiva, adicionamos a palavra *rec* .

Exemplo:

$val\ rec\ fatorial = fn\ 0 => 1 \mid x => x * (fatorial(x-1));$

ou,

$fun\ fatorial\ 0 = 1 \mid fatorial\ x = x * (fatorial(x-1));$

6.8 Definição de novos tipos estruturados

Através do mecanismo de definição de novos tipos podemos declarar um nome representando um tipo, e ao mesmo tempo especificar quais os construtores deste tipo. Os construtores são funções como quaisquer outras, tendo apenas a particularidade de o seu contra-domínio ter de ser obrigatoriamente o tipo que se declara. O domínio de cada construtor terá de ser de um tipo já definido, ou seja, um tipo atômico ou estruturado pré-existente no SML, ou definido anteriormente pelo usuário.

A sintaxe de declaração de novos tipos é a seguinte:

$datatype < identificador >=< construtor1 > of <tipo1> \mid < construtor2 > of < tipo2 > ... \mid < construtorN > of < tipoN >;$

A primeira aplicação importante das declarações *datatype* é a que resulta da utilização de construtores constantes, funções cujo domínio é um conjunto vazio.

Uma outra classe de tipos que consideramos é constituída por tipos não constantes. Como por exemplo, podemos citar a declaração do tipo FORMULA no provador de teo-

remas do próximo capítulo.

6.9 Estruturas, estruturas parametrizadas e interfaces

O que vimos até agora oferece um ambiente confortável e seguro para o desenvolvimento de pequenos programas. Contudo, um grande programa pode conter muitas definições (de valores, funções e tipos) , deste modo, nós precisamos de um método para empacotar junto definições relacionadas; por exemplo, um tipo e uma coleção de funções que processem os elementos deste tipo. Standard ML possui módulos chamados *structures*(estrutura) [38]. AS estruturas facilitam a divisão de um grande programa em unidades menores, independentes, com conexões explicitas bem definidas. Assim, o desenvolvimento de um grande programa pode ser dividido entre vários membros da equipe de programação.

Outro motivo para agrupar coleções de definições em uma estrutura é que nós podemos passar estruturas como argumentos e também retorna-lás como resultado. Em Standard ML a entidade que desempenha o papel de mapear estruturas para estruturas é chamada de *functor* [38].

Finalmente, nós podemos querer que algumas das declarações de uma estrutura não sejam visíveis, ou seja, que somente possam ser acessadas por outras declarações internas da estrutura. Para isso, nós precisamos de uma interface para a nossa estrutura. A entidade que desempenha o papel de esconder declarações de uma estrutura é a *signature* [38].

As estruturas, functors e interfaces são usadas na implementação apresentada no próximo capítulo.

6.10 Outras características da linguagem SML

Além do que foi apresentado, a linguagem SML apresenta outras características que podem ser vistas de forma mais detalhada em [2] [40]. Dentre estas características podemos citar:

- *Tipos Abstratos* : Quando definimos tipos abstratos em SML através da palavra reservada *abstype* , temos que definir funções construtoras e destrutoras para manipular o tipo, não sendo possível acessar diretamente a estrutura do tipo.

- *Biblioteca Base* : O SML possui uma biblioteca base, chamada de *Basis Library* , com diversas operações disponíveis, dentre as operações temos, manipulação de vetores, arrays, entrada e saída, sistema operacional, etc.

Neste capítulo tivemos uma noção básica da linguagem Standard ML. No próximo capítulo veremos uma implementação do método das conexões nesta linguagem.

CAPÍTULO 7

IMPLEMENTAÇÃO

Neste capítulo será descrita uma implementação do método de Bibel para a lógica anotada na linguagem Standard ML. Também será visto um exemplo de um sistema especialista médico e, por fim, uma comparação empírica de eficiência.

7.1 Histórico da implementação

No início do desenvolvimento deste trabalho foi realizada uma implementação do método de Bibel para a lógica clássica proposicional. Após o estudo da lógica anotada, foi implementada uma versão para a lógica anotada proposicional. Na terceira versão, foram incluídas a estrutura e operações de reticulados discretos. Com esta mudança, como veremos mais adiante, podemos, de modo simples, declarar novos reticulados na implementação.

Até a terceira versão do programa a busca foi feita tendo por base os caminhos, causando assim, uma explosão combinatória. Como foi exposto no capítulo 5 e será exemplificado mais adiante, a prova de pequenas matrizes demorava muito, ou mesmo, na maioria dos casos toda a memória era rapidamente esgotada e a execução do programa abortada pelo computador.

Na quarta versão do programa a busca foi feita tendo por base as conexões (veja o capítulo 5). Como exemplificaremos mais adiante, o problema da explosão combinatória não foi resolvido, porém, amenizado. Muitas das matrizes, que nas versões anteriores do programa levavam vários minutos, ou mesmo, abortavam a prova, nesta versão do programa são provadas em menos de um segundo.

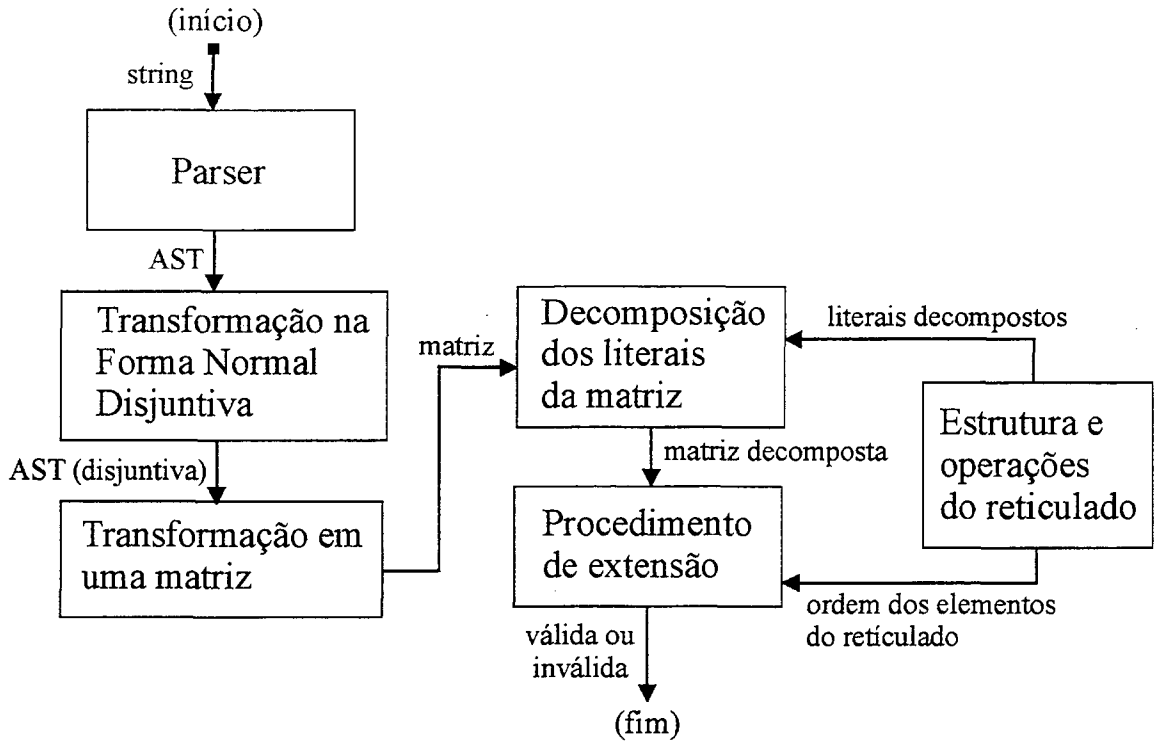


Figura 7.1: Diagrama de Blocos da Implementação

7.2 Visão Geral da Implementação

Em seguida, esboçaremos a quarta versão do programa, devido a esta ser a evolução das versões anteriores.

A implementação foi feita através de um parser [1], da transformação da fórmula em sua forma normal disjuntiva, da transformação da fórmula em uma matriz positiva e do cálculo proposicional, onde, é aplicado o procedimento de extensão visto no capítulo 5. Veja a figura 7.1.

O parser recebe como entrada uma seqüência de caracteres (que representam os literais, conectivos e parênteses) e gera uma árvore de sintaxe abstrata (AST) [1]. A AST representa a fórmula a ser provada.

Em seguida, a AST é transformada em uma nova AST que representa a fórmula na sua forma normal disjuntiva. Esta fórmula, então, é transformada em uma matriz positiva.

Finalmente, através do procedimento de extensão, o programa verifica se a matriz é válida ou não.

Para a lógica anotada, além do descrito acima, também foi implementado um módulo

com a estrutura e operações dos reticulados FOUR e SIX. Veremos mais adiante que é possível declarar outros reticulados sem a necessidade de modificar a estrutura do programa.

Veja no apêndice A a listagem completa da implementação.

7.3 Descrição informal da Implementação

Nesta seção será apresentada uma descrição informal das partes mais importantes da implementação.

As signatures [40] serão chamadas de *interface* e as structures [40] de *estrutura*.

7.3.1 O functor Bibel

A interface BIBEL declara a função *valido* que tem como domínio uma fórmula (seqüência de caracteres que representa os literais, conectivos e parênteses) e como imagem os valores verdadeiro e falso. Na aplicação da função valido, o resultado é verdadeiro se a fórmula passada como argumento é válida, em caso contrário o resultado é falso.

```
signature BIBEL =
sig
   val valido: string -> bool
end; (* BIBEL *)
```

A interface BIBEL está associada ao functor Bibel. A função *valido* é implementada no functor Bibel através da seguinte composição de funções:

```
val valido = spanning o remove_repetido o decompoe o matriz o
                                                normal_disj o ast;
```

Esta função aplica, direta ou indiretamente, todas as outras funções da implementação.

A função valido é a composição das seguintes funções: primeiramente a fórmula (seqüência de caracteres) é transformada em uma AST através da função *ast*, a AST resultante, é transformada em sua forma normal disjuntiva através da função *normal_disj*, o

resultado é transformado em uma matriz através da função *matriz*, a função *decompoe* faz a decomposição dos literais da matriz, a função *remove_repetido*, remove literais repetidos das colunas da matriz e a função *spanning* retorna verdadeiro se a matriz é válida. As implementações destas funções serão apresentadas mais adiante.

Além de implementar as funções mencionadas acima, o functor Bibel recebe como parâmetros uma interface ESTRUTURA_RETICULADO e uma interface OPERACOES_RETICULADO, que indicam qual é o reticulado a ser usado na verificação de literal complementar (veja a definição 4.3.2) e na decomposição (veja a definição 4.3.4) dos literais da matriz.

A estrutura *AnotadaFOUR* associa o functor Bibel ao reticulado FOUR, enquanto que a estrutura *AnotadaSIX* associa o functor Bibel ao reticulado SIX. Desta forma para provar uma fórmula com o reticulado FOUR, aplicamos a função *valido* através de AnotadaFOUR.valido ... e para provar uma fórmula com o reticulado SIX, aplicamos a função *valido* através de AnotadaSIX.valido ...

```
structure AnotadaFOUR = Bibel (Reticulado_FOUR) (Reticulado_Discreto);


structure AnotadaSIX = Bibel (Reticulado_SIX) (Reticulado_Discreto);
```

7.3.2 Declaração das estruturas e operações de reticulados

A interface ESTRUTURA_RETICULADO, declara os identificadores *entrutura_ordem* e *estrutura_ decomposicao*

```
signature ESTRUTURA_RETICULADO =
sig
   val estrutura_ordem: (string * string) list
   val estrutura_decomposicao: (string * string list) list
end; (*TIPO_RETICULADO*)
```

O identificador estrutura_ordem é associado a uma lista de pares de strings, onde, cada valor do par denota um elemento do reticulado. O primeiro valor de cada par deve

ser maior, na ordem do reticulado, que o segundo. Cada par corresponde a um arco do diagrama de Hasse do reticulado que está sendo representado.

Já o identificador estrutura_decomposicao é associado a uma lista de tuplas, onde, o primeiro valor da tupla denota um determinado elemento do reticulado e o segundo é uma lista de valores que denota os elementos que são a decomposição do primeiro valor. Veja a definição 4.3.4.

Podemos associar a esta interface uma ou mais estruturas de reticulado, onde, cada estrutura associada representa um determinado reticulado. A seguinte estrutura representa o reticulado FOUR, onde, os valores "1", "t", "f" e "0" denotam inconsistente, verdadeiro, falso e indeterminado respectivamente (veja o capítulo 3). De acordo com a definição 4.3.4, o reticulado FOUR tem somente um elemento que pode ser decomposto. O elemento inconsistente pode ser decomposto em verdadeiro e falso.

```
structure Reticulado_FOUR: ESTRUTURA_RETICULADO =
struct
   val estrutura_ordem = [("1","t"),("1","f"),("t","0"),("f","0")];
   val estrutura_decomposicao = [("1",["t","f"])];
end; (*Reticulado_FOUR*)
```

A seguinte estrutura representa o reticulado SIX, onde, os valores "1", "t", "qt", "f", "qf" e "0" denotam inconsistente, verdadeiro, quase verdadeiro, falso, quase falso e indeterminado respectivamente.

```
structure Reticulado_SIX: ESTRUTURA_RETICULADO =
struct
   val estrutura_ordem =
        [("1","t"),("1","f"),("t","qt"),("f","qf"),("qt","0"),("qf","0")];
   val estrutura_decomposicao = [("1",["qt","qf"])];
end; (*Reticulado_SIX*)
```

Além da interface que descreve a estrutura, também foi necessário declarar a interface OPERACOES_RETICULADO, a qual, declara as operações de ordem e decomposição do

reticulado. Nesta interface são declaradas as funções *maior_igual* e *decomposicao*.

```
signature OPERACOES_RETICULADO =
sig
   val maior_igual: (''a * ''a) list -> ''a * ''a -> bool
   val decomposicao: (''a * 'b list) list -> 'c * ''a * string ->
                     ('c * 'b * string) list
end; (*OPERACOES_RETICULADO*)
```

A função maior_igual possui como domínio um valor correspondente à estrutura_ordem e uma tupla de valores do reticulado, enquanto que, como resultado possui os elementos booleanos verdadeiro e falso.

Na aplicação da função maior_igual, o resultado é verdadeiro se o primeiro valor da tupla for maior ou igual ao segundo valor, de acordo com a ordem dos elementos associados ao identificador estrutura_ordem, caso contrário, o resultado da função é falso.

A função decomposição tem como domínio um valor correspondente à estrutura_ decomposicao e uma tripla que representa um literal (veja a definição 4.1.1) e como imagem uma lista de literais. Na aplicação, o resultado da função é a lista de literais obtida pela decomposição do literal passado como argumento da função.

As funções da interface OPERACOES_RETICULADO são implementadas pela estrutura Reticulado_Discreto. A implementação das operações descritas acima é dada a seguir.

```
structure Reticulado_Discreto: OPERACOES_RETICULADO =
struct
   fun maior_igual ((a,b)::xs) (x,y) = if (x = y)
                   then true
                   else if (a,b) = (x,y)
                        then true
                        else if (a = x) andalso
                                (maior_igual ((a,b)::xs) (b,y))
```

```
                                then true
                                else maior_igual xs (x,y)
    |   maior_igual _ _ = false;


    fun decomposicao2 nil (_,_,_)     = nil
    |   decomposicao2 (x::xs) (a,b,c) = (a,x,c) ::
                                        decomposicao2 xs (a,b,c);


    fun decomposicao _ (_,_,"~")          = nil
    |   decomposicao nil (_,_,_)          = nil
    |   decomposicao ((x,y)::xs) (a,b,c) = if (x = b)
                  then decomposicao2 y (a,b,c)
                  else decomposicao xs (a,b,c);
end; (*Reticulado_Discreto*)
```

7.3.3 Transformação de uma string em uma árvore de sintaxe abstrata

Para obter uma AST a partir de uma string de entrada, é necessário identificar e separar os tokens [1] desta string, e através de um parser, construir uma AST. Estas operações foram implementadas através da função composta *ast* e de outras funções que serão vistas a seguir.

```
val ast = primeiro o expressao o remove_espacos o explode;
```

A função *ast* é a composição das seguintes funções: a função *explode* separa em caracteres a string de entrada, a função *remove_espacos* remove todos os espaços em branco e a função *expressao* identifica os tokens e constroe uma AST.

Os tokens são extraídos da esquerda para a direita. Sendo que o primeiro token deve ser sempre uma negação, um parênteses ou um literal. Os tokens que a implementação

reconhece são os seguintes:

- qualquer seqüência de caracteres alfanumérico ($a..z, A..Z, 0..9$), seguindo por ":", seguido por qualquer seqüência de caracteres alfanuméricos é reconhecido como um literal;
- "(" é reconhecido como abertura de parênteses e ")" é reconhecido como fechamento de parênteses;
- "&&" é reconhecido como conectivo de conjunção;
- "||" é reconhecido como conectivo de disjunção;
- " => " é reconhecido como conectivo de implicação;
- " $\sim$ " é reconhecido como conectivo de negação;
- espaços em branco são ignorados.

Exemplo 7.3.1 Supondo o reticulado FOUR, a fórmula $K_t \wedge (K_t \rightarrow K_f) \rightarrow K_\top$ é representada pela string "$K : t$ &&$(K : t$ => $K : f)$ => $K : 1$" que tem os tokens "$K : t$", "&&", "(", "$K : t$", " => ", "$K : f$", ")", " => " e "$K : 1$".

A AST é declarada na implementação através do tipo recursivo FORMULA.

```
datatype FORMULA = E of FORMULA*FORMULA        (* conjuncao    &&        *)
|                  Ou of FORMULA*FORMULA      (* disjuncao    ||        *)
|                  Impl of FORMULA*FORMULA    (* implicacao   =>        *)
|                  Nao of FORMULA             (* negacao      ~         *)
|                  Prop of string*string;     (* literal                *)
                                               (* (Proposicao, Anotação) *)
```

Intuitivamente, a declaração do tipo FORMULA significa o seguinte:

- Uma fórmula é uma conjunção de fórmulas;
- Uma fórmula é uma disjunção de fórmulas;

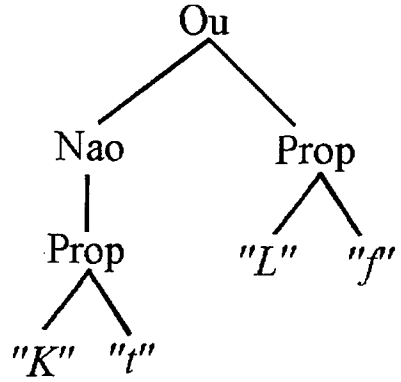


Figura 7.2: AST da fórmula $\sim K_t \vee L_f$

- Uma fórmula é uma implicação de uma fórmula por outra fórmula;
- Uma fórmula é uma negação de uma fórmula;
- Uma fórmula é um literal.

Sendo que, E é o construtor de conjunção, *Ou* é o construtor de disjunção, *Impl* é o construtor de implicação, *Nao* é o construtor de negação e *Prop* é o construtor de literal. Veja a definição de fórmula no capítulo 3.

Exemplo 7.3.2 A fórmula $\sim K_t \vee L_f$ é representada no tipo FORMULA por Ou(Nao(Prop("K","t")), Prop("L","f")). O tipo FORMULA deste exemplo pode também ser visto em forma de árvore na figura 7.2

Exemplo 7.3.3 A fórmula $\sim K_t \vee L_f$ é representada pela string " $\sim K : t \parallel L : f$" que tem os tokens " $\sim$ ", "$K : t$", "$\parallel$" e "$L : f$". Esta string, através do parser, é transformada em uma AST que é representada no tipo FORMULA por Ou(Nao(Prop("K","t")), Prop("L","f")).

As operações descritas acima são implementadas através das seguintes funções e de outras funções auxiliares que podem ser vistas na listagem completa do programa. As funções *expressao* e *expressao'* constroem a implicação de uma fórmula por outra, *termo* e *termo'* constroem a conjunção ou disjunção de duas fórmulas e *fator* e *fator'* constroem a negação de uma fórmula, constroem os literais e determinam a construção da árvore de acordo com a precedência imposta pelos parênteses.

```
fun expressao cs =  expressao'(termo cs)
and expressao' (l, nil) = (l, nil)
|   expressao' (l, c::nil) = (l, c::nil)
|   expressao' (l,b::c::cs) =
      if "=>" = ((str b) ^ (str c))
      then let val (r, cs) = termo cs
                     in expressao' (Impl(l,r),cs)
           end
      else (l,b::c::cs)
and termo cs = termo'(fator cs)
and termo' (l, nil) = (l, nil)
|   termo' (l,c::nil) = (l,c::nil)
|   termo' (l,b::c::cs) =
      if "&&" = ((str b) ^ (str c))
      then let val (r, cs) = fator cs
           in termo' (E(l,r),cs) end
      else if "||" = ((str b) ^ (str c))
           then let val (r, cs) = fator cs
                in termo' (Ou(l,r),cs) end
           else (l,b::c::cs)
and fator nil = raise sintaxerror
|   fator (c::cs) =
    if c = #"("
    then let val (t1, cs) = expressao cs
         in (t1, remove_cabeca (igual #")") cs)
         end
    else if c = #"~"
         then let val (t1, cs) = fator cs
              in (Nao(t1),cs)
```

```
            end
        else leProp(Prop("","),c::cs);
```

7.3.4 Transformação da AST em sua forma normal disjuntiva

Esta implementação trata somente matrizes não aninhadas, desta forma a fórmula é transformada na forma normal disjuntiva através da função *normal_disj*.

Esta função composta obtém a AST na forma normal disjuntiva aplicando primeiramente a função *implica*, a função *deMorgan* e por fim a função *distrib* na AST passada como parâmetro.

```
(* Transforma uma fórmula proposicional para a forma normal disjuntiva *)

val normal_disj = distrib o deMorgan o implica;
```

As funções *implica*, *deMorgan* e *distrib* foram implementadas da seguinte forma.

```
(*  Implicação                                  *)
(*  Impl(a,b)   <=>   Ou(Nao(a),b)              *)

fun implica (Impl(a, b)) = Ou(Nao(implica a), implica b)
|   implica (Ou(a, b))   = Ou(implica a, implica b)
|   implica (E(a, b))    = E(implica a, implica b)
|   implica (Nao(a))     = Nao(implica a)
|   implica (x)          = x;



(*  Leis de De Morgan                           *)
(*  Nao(Ou(a,b))  <=>   E(Nao(a),Nao(b))        *)
(*  Nao(E(a,b))    <=>  Ou(Nao(a),Nao(b))       *)

fun deMorgan (Nao(Nao(a)))  = deMorgan a   (* dupla negacao*)
```

```
|   deMorgan (Nao(Ou(a,b))) = E(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Nao(E(a,b)))  = Ou(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Ou(a,b))      = Ou(deMorgan a, deMorgan b)
|   deMorgan (E(a,b))       = E(deMorgan a, deMorgan b)
|   deMorgan (Nao(a))       = Nao(deMorgan a)
|   deMorgan (x) = x;


(*  Distributividade                          *)
(*  E(a,Ou(b,c))   <=>   Ou(E(a,b)),E(a,c))   *)

fun distrib1 (E(a,Ou(b,c)))  = Ou(E(distrib1 a,distrib1 b),
                                  E(distrib1 a,distrib1 c))
|   distrib1 (E(Ou(a,b),c))  = Ou(E(distrib1 a,distrib1 c),
                                  E(distrib1 b,distrib1 c))
|   distrib1 (Ou(a,b))       = Ou(distrib1 a,distrib1 b)
|   distrib1 (E(a,b))        = E(distrib1 a,distrib1 b)
|   distrib1 (x)             = x;

fun distrib x =
      let val f = distrib1 x
        in if (x = f) then x else distrib f
      end;
```

Intuitivamente, estas funções desempenham as seguintes operações:

1. *implica*: remoção da implicação. Toda fórmula na forma $K \rightarrow L$ é substituída por $\sim K \vee L$;

2. *deMorgan*: aplicação das leis de De Morgan. Toda fórmula na forma $\sim (K \vee L)$ é substituída por $(\sim K \wedge \sim L)$, enquanto que, toda fórmula na forma $\sim (K \wedge L)$ é

substituída por $(\sim K \vee \sim L)$;

3. *distrib*: aplicação de uma das regras de distributividade. Toda fórmula na forma $(K \wedge (L \vee M))$ é substituída por $((K \wedge L) \vee (K \wedge M))$.

7.3.5 Construção e decomposição da matriz

Em uma fórmula na forma normal disjuntiva, uma cláusula é uma conjunção de literais e uma fórmula é uma disjunção de cláusulas. Na matriz positiva que representa tal fórmula, cada cláusula é uma coluna da matriz. Na implementação, a matriz é representada por uma lista de listas, onde cada uma das listas representa uma coluna da matriz.

Foram implementadas as funções *matriz_coluna* e *matriz* para transformar conjunções de literais em colunas e disjunções de cláusulas em linhas, respectivamente.

Estas funções fazem o casamento de padrão (veja capítulo 6) com todas as combinações possíveis de uma AST na forma normal disjuntiva e constroem uma lista de listas.

A função *matriz* faz o casamento de padrão com uma AST que representa a disjunção de uma disjunção e um literal, a disjunção de um literal e uma disjunção, a disjunção de uma disjunção e um literal negado, a disjunção de um literal negado e uma disjunção, a disjunção de duas disjunções, a disjunção de uma disjunção e uma conjunção, a disjunção de uma conjunção e uma disjunção, etc.

A função *matriz_coluna* faz o casamento de padrão com uma AST que representa a conjunção de uma conjunção e um literal, a conjunção de uma conjunção e um literal negado, a conjunção de um literal e uma conjunção, a conjunção de um literal negado e uma conjunção, a conjunção de duas conjunções, a conjunção de dois literais, a conjunção de um literal negado e um literal, etc.

Exemplo 7.3.4 Supondo o reticulado FOUR. A fórmula $K_t \wedge (K_t \vee K_f) \rightarrow K_t$ é representada pela string "$K : t$ && $(K : t \parallel K : f)$ => $K : t$" que é representada pela AST $Impl(E(Prop(\text{“}K\text{”}, \text{“}t\text{”}), OU(Prop(\text{“}K\text{”}, \text{“}t\text{”}), Prop(\text{“}K\text{”}, \text{“}f\text{”}))))$.

Na forma normal disjuntiva, esta fórmula se torna $\sim K_t \vee (\sim K_t \wedge \sim K_f) \vee K_t$ que é representada pela string "$\sim K : t \parallel (\sim K : t$ && $\sim K : f) \parallel K : t$" que é representada pe-

la AST $Ou(Nao(Prop(\text{“}K\text{”}, \text{“}t\text{”})), Ou(E(Nao(Prop(\text{“}K\text{”}, \text{“}t\text{”})), Nao(Prop(\text{“}K\text{”}, \text{“}f\text{”}))))$, $(Prop(\text{“}K\text{”}, \text{“}t\text{”})))$.

Esta AST na forma normal disjuntiva faz o casamento de padrão com a seguinte linha da função *matriz* que é uma disjunção entre um literal negado $Nao(Prop(\text{“}K\text{”}, \text{“}t\text{”}))$ e uma disjunção $Ou(E(Nao(Prop(\text{“}K\text{”}, \text{“}t\text{”})), Nao(Prop(\text{“}K\text{”}, \text{“}f\text{”})))), (Prop(\text{“}K\text{”}, \text{“}t\text{”})))$.

```
|   matriz (Ou(Nao(Prop(a1,a2)),Ou(b,c)))= ((a1,a2,"~")::nil)::
                                                matriz (Ou(b,c))
```

Como pode ser visto no código acima, o literal negado é incluído na lista de listas $((\text{“}K\text{”}, \text{“}t\text{”}, \text{“} \sim \text{”}) :: nil) ::$ mas a função matriz é chamada recursivamente para tratar da disjunção. Nesta chamada recursiva, o casamento de padrão é feito com a seguinte linha da função *matriz* que é a disjunção de uma conjunção $E(Nao(Prop(\text{“}K\text{”}, \text{“}t\text{”})), Nao(Prop(\text{“}K\text{”}, \text{“}f\text{”})))$ com um literal $Prop(\text{“}K\text{”}, \text{“}t\text{”})$.

```
|   matriz (Ou(E(a,b),Prop(c1,c2)))       = ((c1,c2,"")::nil)::
                                              (matriz_coluna (E(a,b)))::nil
```

No código acima, o literal é incluído na lista que concatenada a lista de listas anterior se torna $((\text{“}K\text{”}, \text{“}t\text{”}, \text{“} \sim \text{”}) :: nil) :: ((\text{“}K\text{”}, \text{“}t\text{”}, \text{“”}) :: nil) ::$. Note que neste ponto a função *matriz_coluna* é chamada para tratar a conjunção.

Concluindo o exemplo, após a conjunção ser tratada, a lista de listas que representa a matriz da fórmula apresentada será:

$((\text{“}K\text{”}, \text{“}t\text{”}, \text{“} \sim \text{”}) :: nil) :: ((\text{“}K\text{”}, \text{“}t\text{”}, \text{“”}) :: nil) :: ((\text{“}K\text{”}, \text{“}t\text{”}, \text{“} \sim \text{”}) :: (\text{“}K\text{”}, \text{“}f\text{”}, \text{“} \sim \text{”}) :: nil) :: nil$

A seguir é apresentado o código completo das funções *matriz_coluna* e *matriz*. Este código possui todas as alternativas de casamento de padrão com a AST para construir uma lista de listas que representa uma matriz positiva.

```
fun matriz_coluna (E(E(a,b),Prop(c1,c2)))       = (c1,c2,"")::
                                                    (matriz_coluna (E(a,b)))
```

```
|   matriz_coluna (E(E(a,b),Nao(Prop(c1,c2)))) = (c1,c2,"~")::
                                                  (matriz_coluna (E(a,b)))
|   matriz_coluna (E(Prop(a1,a2),E(b,c)))      = (a1,a2,"")::
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(Nao(Prop(a1,a2)),E(b,c))) = (a1,a2,"~")::
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(E(a,b),E(c,d)))           = (matriz_coluna (E(a,b))) @
                                                  (matriz_coluna (E(c,d)))
|   matriz_coluna (E(Prop(a1,a2),Prop(b1,b2))) = (b1,b2,"")::
                                                  (a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Prop(b1,b2))) = (b1,b2,"")::
                                                       (a1,a2,"~")::nil
|   matriz_coluna (E(Prop(a1,a2),Nao(Prop(b1,b2)))) = (b1,b2,"~")::
                                                       (a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = (b1,b2,"~")::
                                                            (a1,a2,"~")::nil
|   matriz_coluna (Prop(a1,a2))                = (a1,a2,"")::nil
|   matriz_coluna (Nao(Prop(a1,a2)))           = (a1,a2,"~")::nil;

fun matriz (Ou(Ou(a,b),Prop(c1,c2)))     = ((c1,c2,"")::nil)::
                                             matriz (Ou(a,b))
|   matriz (Ou(Prop(a1,a2),Ou(b,c)))     = ((a1,a2,"")::nil)::
                                             matriz (Ou(b,c))
|   matriz (Ou(Ou(a,b),Nao(Prop(c1,c2))))= ((c1,c2,"~")::nil)::
                                             matriz (Ou(a,b))
|   matriz (Ou(Nao(Prop(a1,a2)),Ou(b,c)))= ((a1,a2,"~")::nil)::
                                             matriz (Ou(b,c))
|   matriz (Ou(Ou(a,b),Ou(c,d)))          = (matriz (Ou(a,b))) @
                                             (matriz (Ou(c,d)))
```

```
|   matriz (Ou(Ou(a,b),E(c,d)))          = (matriz_coluna (E(c,d)))::
                                              matriz (Ou(a,b))
|   matriz (Ou(E(a,b),Ou(c,d)))          = (matriz_coluna (E(a,b)))::
                                              matriz (Ou(c,d))
|   matriz (Ou(E(a,b),Prop(c1,c2)))      = ((c1,c2,"")::nil)::
                                              (matriz_coluna (E(a,b)))::nil
|   matriz (Ou(E(a,b),Nao(Prop(c1,c2)))) = ((c1,c2,"~")::nil)::
                                              (matriz_coluna (E(a,b)))::nil
|   matriz (Ou(Prop(a1,a2),E(b,c)))      = (matriz_coluna (E(b,c)))::
                                              ((a1,a2,"")::nil)::nil
|   matriz (Ou(Nao(Prop(a1,a2)),E(b,c))) = (matriz_coluna (E(b,c)))::
                                              ((a1,a2,"~")::nil)::nil
|   matriz (Ou(E(a,b),E(c,d)))          = (matriz_coluna (E(c,d)))::
                                              (matriz_coluna (E(a,b)))::nil
|   matriz (Ou(Prop(a1,a2),Prop(b1,b2))) = ((b1,b2,"")::nil)::
                                              ((a1,a2,"")::nil)::nil
|   matriz (Ou(Nao(Prop(a1,a2)),Prop(b1,b2))) = ((b1,b2,"")::nil)::
                                              ((a1,a2,"~")::nil)::nil
|   matriz (Ou(Prop(a1,a2),Nao(Prop(b1,b2)))) = ((b1,b2,"~")::nil)::
                                              ((a1,a2,"")::nil)::nil
|   matriz (Ou(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = ((b1,b2,"~")::nil)::
                                              ((a1,a2,"~")::nil)::nil
|   matriz a                            = (matriz_coluna a)::nil;
```

Após a construção da lista de listas, cada literal é verificado, e se for o caso, decomposto (veja a definição 4.3.4) através das seguintes funções.

Note que a decomposição de literais prejudica a eficiência do método, devido a matriz ficar com um número maior de literais.

```
fun decompoe2 nil          = nil
```

```
|   decompoe2 ((a,b,c)::cs) = let val x = Operacoes.decomposicao
                                    Estrutura.estrutura_decomposicao (a,b,c)
                              if (x = nil)
                                 then [(a,b,c)] @ decompoe2 cs
                                 else x @ decompoe2 cs
                           end;

fun decompoe nil        = nil
|   decompoe (xs::xss) = decompoe2 xs :: decompoe xss;
```

A função decompoe recebe uma lista de listas (matriz) e chama a função decompoe2 para decompor os literais de cada uma das listas (coluna). O resultado é uma lista de listas com literais decompostos.

7.3.6 Verificação de literais complementares

A definição 4.3.2 foi implementada através da função *complementar*. Esta função recebe como parâmetro dois literais e retorna verdadeiro se são complementares, caso contrário retorna falso.

```
fun complementar (a,b,c) (d,e,f) =
                    if (a=d) andalso (f="") andalso
                      (c="~") andalso (Operacoes.maior_igual
                       Estrutura.estrutura_ordem(b,e))
                    then true
                    else if (a=d) andalso (f="~") andalso
                           (c="") andalso (Operacoes.maior_igual
                            Estrutura.estrutura_ordem(e,b))
                          then true
                          else false;
```

Note que a função maior_igual do reticulado é utilizada.

7.3.7 O procedimento de extensão

O procedimento geral de extensão visto no capítulo 5 foi implementado aqui através das seguintes funções e de outras funções auxiliares que podem ser vistas na listagem completa da implementação.

Através da função *acha_coluna_positiva*, obtemos uma coluna da matriz que possui somente literais não negados.

A função auxiliar *coluna_positiva* retorna verdadeiro se a coluna passada como parâmetro possui somente literais não negados.

```
fun coluna_positiva nil = false
|   coluna_positiva ((_,_,"~")::nil) = false
|   coluna_positiva ((_,_,"")::nil) = true
|   coluna_positiva ((_,_,"~")::xs) = false
|   coluna_positiva ((_,_,"")::xs) = coluna_positiva xs;

fun acha_coluna_positiva  nil = nil
|   acha_coluna_positiva  (xs::nil) =
                    if (coluna_positiva xs)
                    then xs
                    else nil
|   acha_coluna_positiva (xs::xss) =
                    if (coluna_positiva xs)
                    then xs
                    else acha_coluna_positiva xss;
```

Como na implementação uma matriz é representada por uma lista de listas, foi necessário, além do procedimento de extensão exposto no capítulo 5, levar em conta operações sobre listas. Por exemplo, ao invés de marcar um literal complementar de uma cláusula com um ponto, é feita uma cópia da lista que representa a cláusula, onde, este literal é removido, ficando somente os literais abertos. Então, o próximo passo de extensão

é feito. Quando é feito um truncamento, a cópia sem o literal é ignorada e a cláusula com todos seus literais é utilizada novamente. A única diferença, é que ao invés de colocar um ponto no literal complementar, é feita uma manipulação de listas.

A seguinte função, que é a principal do procedimento de extensão, tem como parâmetros uma lista de literais, que representa o caminho ativo, e uma lista de listas que representa a matriz. Quando da aplicação desta função, o resultado é verdadeiro se a matriz passada como parâmetro é válida, caso contrário, o resultado é falso.

```
fun expande bs ((x::xs)::xss) =
        if (membro_complementar x bs)
        then if xs = nil
          then true
          else expande bs (xs::xss)
        else let
            val (col, col_menos, status) =
                 acha_coluna_complementar x xss
        in
            if status then
              let val status_subnivel =
                   if col_menos = nil then true
                   else expande (x::bs)
                     (col_menos::(menos_conjunto col xss))
              in if status_subnivel then
                   if xs = nil then true
                   else expande bs (xs::xss)
                 else false
              end
            else false
        end;
```

Seja n o número de cláusulas da matriz, $1 \leq i, j \leq n$. De modo intuitivo, a função

expande executa o seguinte procedimento recursivo:

- se existe somente um literal aberto na cláusula i, e este é complementar de algum literal do caminho ativo, então a função retorna verdadeiro.
- senão, se existe mais de um literal aberto na cláusula i, e o primeiro é complementar de algum literal do caminho ativo, então a função é chamada recursivamente para verificar o segundo literal aberto da cláusula i.
- senão, se existe um literal na cláusula j que é complementar ao próximo literal aberto da cláusula i e não existem mais literais abertos na cláusula j, então a função retorna verdadeiro.
- senão, se existe um literal na cláusula j que é complementar ao próximo literal aberto da cláusula i e existem mais literais abertos na cláusula j, então e a função é chamada recursivamente, incluindo o literal da cláusula i que participou da conexão no caminho ativo ,para verificar o próximo literal complementar da cláusula j.
- senão, a função retorna falso.

A função *spanning* recebe uma matriz como argumento e retorna verdadeiro se a matriz possui somente um literal anotado pelo elemento ínfimo do reticulado ou se a função expande retorna verdadeiro. Retorna falso se a matriz é vazia ou se a função expande retorna falso.

```
fun spanning xss = if (matriz_infimo xss)
                   then true
                   else let
                      val matriz = acha_coluna_positiva xss
                   in if (matriz <> nil)
                      then expande nil (matriz::(menos_conjunto matriz xss))
                      else false
                   end;
```

As principais funções da implementação foram descritas. Agora vamos ver um exemplo ilustrativo de uso desta implementação.

7.4 Exemplo ilustrativo

Considere a construção de um sistema médico simplificado, que faz o diagnóstico de três doenças denotadas por K, L, M. Existem dois sintomas denotados por N e O. O uso pretendido deste sistema é o seguinte:

- A parte central do sistema é o conhecimento fornecido pelo doutor DOC_1.
- Quando pretendemos aplicar este conhecimento para um determinado paciente, diga Paulo, então os patologistas, técnicos em raio-X e outros profissionais que conduzem os exames médicos do Paulo, adicionam os resultados destes exames na base de conhecimento.

Na vida real, tal sistema poderia funcionar recebendo a base de conhecimentos central em um arquivo, enquanto o registro de cada paciente é mantido em arquivo separado. Então a base de conhecimentos central e o registro de um paciente são unidos para formar a base de conhecimentos atual e usados para diagnosticar doenças do paciente.

Nós assumiremos que nosso sistema é escrito na forma de um conjunto finito de fórmulas anotadas do reticulado FOUR. Suponha agora que o doutor DOC_1 nos forneceu as seguintes regras (fórmulas).

- (F_1) $K_t \rightarrow L_f$
- (F_2) $L_t \rightarrow K_f$
- (F_3) $K_t \rightarrow M_t$
- (F_4) $N_t \rightarrow K_t$
- (F_5) $O_t \rightarrow L_t$

Intuitivamente, o doutor está dizendo que:

- um indivíduo não pode ter ambas doenças K e L. (F_1 e F_2)
- se um indivíduo tem a doença K, então ele tem a doença M. (F_3)
- se um indivíduo tem o sintoma N, então ele tem a doença K. (F_4)
- se um indivíduo tem o sintoma O, então ele tem o doença L. (F_5)

Para exemplificar o uso do programa, nós iremos descrever três situações. A primeira situação é similar a uma questão do Prolog, enquanto que as outras duas exploram a capacidade de tratar inconsistências:

Caso 1:

Suponha que o patologista disse que o exame de Paulo foi positivo para o sintoma N e nós desejamos saber, por exemplo, se Paulo tem a doença K mas não tem a doença L.

Para responder esta pergunta nós precisamos verificar se a matriz da fórmula
$F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge N_t \rightarrow (K_t \wedge L_f)$
é válida.

Para que nosso provador verifique se a fórmula acima é válida ou não, após carregar a implementação no interpretador SML, digitamos ou carregamos a seguinte linha no interpretador do SML.

AnotadaFOUR.valido "$(K : t \Rightarrow L : f)$ && $(L : t \Rightarrow K : f)$ && $(K : t \Rightarrow M : t)$ && $(N : t \Rightarrow K : t)$ && $(O : t \Rightarrow L : t)$ && $N : t \Rightarrow (K : t$ && $L : f)$";

Neste caso, como o provador retorna verdadeiro, podemos concluir que Paulo tem a doença K mas não a doença L.

Caso 2:

Vamos supor agora que o patologista disse que o exame de Paulo foi positivo para os sintomas N e O, e nós queremos saber se Paulo tem as doenças K e L.

Para verificar esta questão nós precisamos verificar se a matriz da fórmula
$F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge N_t \wedge O_t \rightarrow (K_t \wedge L_t)$
é válida.

Assim como no caso anterior, para que nosso provador verifique se a fórmula acima é válida ou não, digitamos ou importamos a seguinte linha no interpretador do SML.

AnotadaFOUR.valido "$(K : t => L : f)$ && $(L : t => K : f)$ && $(K : t => M : t)$ && $(N : t => K : t)$ && $(O : t => L : t)$ && $N : t$ && $O : t$ => $(K : t$ && $L : t)$"

Como é fácil de ver, o resultado é verdadeiro e deste modo podemos concluir que Paulo tem as doenças K e L, em contradição com o que o doutor DOC_1 determinou nas fórmulas F_1 e F_2. A razão para a nossa conclusão foi que a informação fornecida pelo patologista esta em contradição com a informação fornecida pelo doutor. Na lógica clássica, a contradição acima tornaria toda a base de conhecimentos inconsistente (todas fórmulas da linguagem poderiam ser derivadas da base de conhecimentos). Em nosso caso, a inconsistência é limitada aos literais K e L.

Caso 3:

O caso anterior mostrou uma base de conhecimentos inconsistente usada para derivar uma fórmula inconsistente. Contudo, o sistema pode tratar inconsistências de um modo não trivial. Deixe-nos exemplificar esta condição através do uso do mesmo conjunto de sintomas do caso 2 para verificar se a doença M. Esta questão pode ser escrita como M_f, assim a nova situação é descrita pela seguinte fórmula:

$F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge N_t \wedge O_t \rightarrow M_f$

convertendo a fórmula, mais uma vez, para o formato de nosso provador, teremos

AnotadaFOUR.valido "$(K : t => L : f)$ && $(L : t => K : f)$ && $(K : t => M : t)$ && $(N : t => K : t)$ && $(O : t => L : t)$ && $N : t => M : f$"

O provador ira retornar falso, ou seja, não é possível afirmar que Paulo não tem a doença M.

As duas últimas questões mostraram que nosso método pode tratar inconsistências na base de conhecimentos sem que toda fórmula possa ser derivada.

As fórmulas do exemplo que acabamos de apresentar não são complexas o suficiente para serem utilizadas em uma comparação de eficiência. Devido a isto, na próxima seção usaremos fórmulas mais complexas.

7.5 Uma comparação empírica de eficiência

Nesta seção veremos uma comparação de eficiência entre a terceira e quarta versão da implementação. Na terceira versão a busca tem por base os caminhos, enquanto que, na quarta versão a busca tem por base as conexões. Veja a listagem destas duas versões da implementação no apêndice A.

A configuração do equipamento utilizado foi:

- Processador Pentium III 450Mhz
- 192Mb de memória RAM (175Mb livres)
- Disco Rígido IDE de 8Gb (6Gb livres)
- Sistema Operacional Conectiva Linux 5.0, com partição de SWAP de 80Mb

Foram utilizadas quatro fórmulas válidas. Cada fórmula foi provada dez vezes em cada versão da implementação e o tempo médio foi calculado através de média simples.

As fórmulas estão escritas na sintaxe da implementação (veja o início deste capítulo). A seguir os resultados obtidos são apresentados:

Primeira Fórmula:

AnotadaSIX.valido "$a : t$ && $(a : t => b : t)$ && $(b : t => c : t)$ && $(c : t => d : t)$ && $(d : t => e : t)$ && $(e : t => f : t)$ && $(f : t => g : t)$ && $(g : t => h : t)$ && $(h : t => i : t)$ && $(i : t => j : t)$ && $(j : t => k : t)$ && $(k : t => l : t)$ && $(l : t => m : t)$ && $(m : t => n : t)$ && $(n : t => o : t)$ && $(o : t => p : t)$ && $(p : t => q : t)$ && $(q : t => r : t)$ && $(r : t => s : t)$ && $(s : t => t : t)$ && $(t : t => u : t)$ && $(u : t => v : t)$ && $(u : t => v : t)$ && $(v : t => x : t)$ && $(x : t => y : t)$ => $y : t$";

- Na terceira versão da implementação o interpretador SML abortou. Desta forma não conseguiu provar a fórmula acima.
- Na quarta versão da implementação levou em média 0.12 segundos para provar a validade da fórmula.

Segunda fórmula:

AnotadaSIX.valido "$a : t$ && $(a : t => b : t => c : t => d : t => e : t => f : t => g : t => h : t => i : t => j : t => k : t => l : t => m : t => n : t => o : t => p : t)$ => $a : t$";

- Na terceira versão da implementação levou em média 9.48 segundos para provar a validade da fórmula.
- Na quarta versão da implementação levou em média 0.11 segundos para provar a validade da fórmula.

Terceira fórmula:

AnotadaSIX.valido "$a : t$ && $(a : t => b : t => c : t => d : t => e : t => f : t => g : t => h : t => i : t => j : t => k : t => l : t => m : t => n : t => o : t => p : t => q : t)$ => $a : t$";

Note que, embora a fórmula acima possua somente um literal a mais que a segunda fórmula, o resultado do teste realizado na terceira versão da implementação foi muito diferente. Isto indica a explosão combinatória que a adição de um único literal causou.

- Na terceira versão da implementação o interpretador SML abortou. Desta forma não conseguiu provar a fórmula acima.

- Na quarta versão da implementação levou em média 0.11 segundos para provar a validade da fórmula.

Quarta fórmula

$AnotadaSIX.valido"a : t \;\&\&\; (a : t => b : t) \;\&\&\; (b : t => c : t) \;\&\&\; (c : t => d : t) \;\&\&\; (d : t => e : t) \;\&\&\; (e : t => f : t) \;\&\&\; (f : t => g : t) \;\&\&\; (g : t => h : t) \;\&\&\; (h : t => i : t) \;\&\&\; (i : t => j : t) \;\&\&\; (j : t => k : t) \;\&\&\; (k : t => l : t) \;\&\&\; (l : t => m : t) \;\&\&\; (m : t => n : t) \;\&\&\; (n : t => o : t) \;\&\&\; (o : t => p : t) \;\&\&\; (p : t => q : t) \;\&\&\; (q : t => r : t) \;\&\&\; (r : t => s : t) \;\&\&\; (s : t => t : t) \;\&\&\; (t : t => u : t) \;\&\&\; (u : t => v : t) \;\&\&\; (u : t => v : t) \;\&\&\; (v : t => x : t) \;\&\&\; (x : t => y : t) \;\&\&\; (y : t => z : t) \;\&\&\; (z : t => a1 : t) \;\&\&\; (a1 : t => b1 : t) \;\&\&\; (b1 : t => c1 : t) \;\&\&\; (c1 : t => d1 : t) \;\&\&\; (d1 : t => e1 : t) \;\&\&\; (e1 : t => f1 : t) \;\&\&\; (f1 : t => g1 : t) \;\&\&\; (g1 : t => h1 : t) \;\&\&\; (h1 : t => i1 : t) \;\&\&\; (i1 : t => j1 : t) \;\&\&\; (j1 : t => k1 : t) \;\&\&\; (k1 : t => l1 : t) \;\&\&\; (l1 : t => m1 : t) \;\&\&\; (m1 : t => n1 : t) \;\&\&\; (n1 : t => o1 : t) \;\&\&\; (o1 : t => p1 : t) \;\&\&\; (p1 : t => q1 : t) \;\&\&\; (q1 : t => r1 : t) \;\&\&\; (r1 : t => s1 : t) \;\&\&\; (s1 : t => t1 : t) \;\&\&\; (t1 : t => u1 : t) \;\&\&\; (u1 : t => v1 : t) \;\&\&\; (v1 : t => x1 : t) \;\&\&\; (x1 : t => y1 : t \;||\; z1 : t \;||\; y2 : t \;||\; z2 : qt) \;\&\&\; (y1 : qt => y1 : f) \;\&\&\; (z1 : qt => z1 : qf) \;\&\&\; (y2 : t => y2 : qf) \;=>\; y1 : 1 \;||\; z1 : 1 \;||\; y2 : 1 \;||\; z2 : qt";$

- Na terceira versão da implementação o interpretador SML abortou. Desta forma não conseguiu provar a fórmula acima.

- Na quarta versão da implementação levou em média 0.16 segundos para provar a validade da fórmula.

De acordo com a comparação acima ficou claro que a busca tendo por base as conexões é mais eficiente que a busca tendo por base os caminhos. Isto, embora seja um teste empírico, reforça as suposições apresentadas no capítulo 5.

CAPÍTULO 8

CONCLUSÃO

Este trabalho apresenta o uso de lógica matemática (raciocínio lógico, definições, teoremas, etc...), teoria de compiladores (parser, tokens, AST, etc..) e programação funcional para implementar o método de Bibel para provas de fórmulas da lógica paraconsistente.

O método apresentado inicialmente em [25] foi implementado. O procedimento de extensão tornou o método mais eficiente. Os testes práticos, embora empíricos, reafirmaram que a busca tendo por base conexões é mais eficiente que a busca tendo por base os caminhos. No senso do método das conexões de Bibel, foi visto que o tableau também é orientado a conexões. Foram vistas fortes semelhanças entre o método tableau e o método de Bibel.

Além do proposto, o método de Bibel para a lógica paraconsistente anotada proposicional foi adequado e estendido através das definições de literal complementar e decomposição, dentre outros detalhes. Na implementação, foram definidas as estruturas e operações de ordem e decomposição, possibilitando assim que novos reticulados possam ser adicionados ao programa facilmente.

Como aplicações do trabalho realizado, o exemplo de aplicação apresentado no capítulo anterior fornece uma visão geral de como o provador pode ser utilizada como módulo raciocinador. É obvio que somente este exemplo não prevê todas as aplicações, porém, intuitivamente mostra que a lógica anotada é importante na construção de módulos raciocinadores de agentes inteligentes.

Um artigo [28] sobre parte do trabalho realizado nesta dissertação será publicado.

Este trabalho pode ser estendido de muitas formas. A seguir são apresentadas algumas sugestões de pesquisa.

8.1 Sugestões de Pesquisa

- A lógica de primeira ordem tem um poder de expressão maior que a lógica proposicional. É interessante estender este trabalho para uma lógica anotada de primeira ordem;

- Construção de agentes inteligentes baseados neste método de prova;

- Estender este trabalho para outras lógicas paraconsistentes para verificar quais são suas implicações e aplicações. Como por exemplo, a lógica anotada indutiva [16];

- Adequar e implementar as reduções [6, 7] de matrizes para lógica anotada;

- Parece interessante que os agentes inteligentes possam fazer raciocínios através de diversas lógicas. Ampliar este trabalho, para que possam ser provados teoremas de outras lógicas não-clássicas, além da lógica paraconsistente;

- Geralmente agentes inteligentes podem explicar *como* chegaram a conclusões. Estender este trabalho para que método apresentado aqui mostre de modo natural a prova de um teorema, ou seja, explique como se chegou a uma conclusão;

- O problema da explosão combinatória em provadores de teoremas é um dos principais problemas da inteligência artificial bem como na Ciência da Computação. Muitas esforços vem sendo realizados com o intuito de amenizar este problema. Um matróide [31] é uma estrutura matemática abstrata que captura propriedades combinatoriais de matrizes. Desta forma, uma possível pesquisa poderia ser em apresentar a teoria e implementação de um provador de teoremas que faz uso de matróides para tornar o método mais eficiente;

BIBLIOGRAFIA

[1] A.V. Aho, R. Sethi, e J.D. Ullman. *Compilers Principles, Techniques and Tools.* Addison Wesley, UK, 1986.

[2] J.C.B. Almeida e J.S. Pinto. *Programação Funcional com Tipos em SML.* Universidade de Minho, Portugal, 1995.

[3] O. Arieli. *Multiple-valued Logic for Reasoning with Uncertainty.* Tese de Doutorado, Tel-Aviv. University, Israel, 1999.

[4] Evert W. Beth. Semantic entailment and formal derivability. *Mededlingen der Koninklijke Nederlandse Akademic van Kaufmann*, 18(13):309–342, 1955.

[5] W. Bibel. *Automated Theorem Proving.* Braumschweig, Wiesbaden, 1982.

[6] W. Bibel. *Deduction: Automated Logic.* Academic Press, 1993.

[7] W. Bibel. *Methods of Inferencing: Course of Automated Deduction.* Department of Computer Sci., Darmstadt Institute of Tecnology, Germany, 1997.

[8] H.A. Blair e V.S. Subrahmanian. Paraconsistent logic programming. *Proc. 4th Conference on Fondations of Software Technology and Theoretical Computer Sci., Lecture Notes on Computer Sci.*, 287:340–360, 1987.

[9] H.A. Blair e V.S. Subrahmanian. Paraconsistent fondations for logic programming. *Journal of Non-Classical Logic*, 5(2):45–73, 1988.

[10] A. Buchsbaun e T. Pequeno. A reasonig method for a paraconsistent logic. 2000.

[11] P.R. Cohen e E.A. Fergenbaun. *The Handbook of Artificial Inteligence III.* Addison Wesley, Department of Computer Sci. Stanford University, 1994.

[12] Constable. Construtive mathematics who think. *Logique et Analyse*, 115:361–371, 1986.

[13] P. Mc. Corduck. *Machines who Think.* Freeman, San Francisco, 1979.

[14] N.C.A. da Costa, J.M. Abe, e V.S. Subrahmanian. *Remarks on Annotated Logic.* Intituto de Estudos Avançados. Série Lógica e Teoria da Ciência, Universidade de São Paulo, 1991.

[15] N.C.A. da Costa, L.J. Henschen, J.J. Lu, e V.S. Subrahmanian. Automatic theorem proving in paraconsistent logics: Fondations and implementation. *Proc. 10th Intl. Conf. on Automated Deduction, Lecture Notes in Computer Sci. Springer Verlag,* 1990.

[16] N.C.A. da Costa e D. Krause. An inductive annotated logic. *Paraconsistency: the logical way to inconsistent, Proc. do II World Congress on Paraconsistency,* 2000.

[17] N.C.A. da Costa, V.S. Subrahmanian, e C. Vago. The paraconsistent logics pt. *Zeitschrift fur Mathematische Logik und Grundlagen,* 1991.

[18] Newton C.A. da Costa. On the theory of inconsistent formal systems. *Notre Dame Journal of Formal Logic,* 15:497–510, 1974.

[19] Newton C.A. da Costa. On paraconsistent set theory. *Logique et Analyse,* 115:361–371, 1986.

[20] Newton C.A. da Costa. *Sistemas Formais Inconsistentes.* Editora da UFPR, Curitiba - Paraná - Brasil, 1993.

[21] Newton C.A. da Costa. *O Conhecimento Científico.* Discurso Editorial, 1999.

[22] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, New York, 1990.

[23] J. Grant e V.S. Subrahmanian. Reasoning about inconsistent knowledge bases. *IEEE Transactions on Knowledge and Data Engineering,* 7:177–189, 1994.

[24] K.J.J. Hintika. Form and content in quantification theory. *Acta Philosophica Fennica,* 8:7–55, 1955.

[25] C.A.A. Kaestner e Décio Krause. Matrix proof method in annotated paraconsistent logic. *Anais do Primer Congreso Argentino de Ciencias de la Computacíon, Un. Nacional del Sur, Bahia Blanca, Argentina*, páginas 416–425, 1996.

[26] M. Kifer e E.L. Lozinskii. A logic for reasoning with inconsistency. *4th Symposium on Logic in Computer Sci. Asilomar, CA*, páginas 253–262, 1989.

[27] M. Kifer e V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Jornal of Logic Programming*, 12(4):335–368, 1992.

[28] D. Krause, E.F. Nobre, e M.A. Musicante. Bibel´s matrix connection method in paraconsistent logic: General concepts and implementation. *IEEE Computer Society Press (to be published)*, 2001.

[29] C. Kreitz e J. Otten. Connection-based theorem proving in classical and non-classical logics. *Jornal of Universal Computer Sci.*, 5(3):88–112, 1999.

[30] R. Milner, M. Tofte, e R. Harper. *The Definition of Standard ML.* MIT Press, 1997.

[31] Kazuo Murota. *Matrices and Matroids for Systems Analysis (Algotithms and Combinatorics, 20.* Springer-Verlag, 2000.

[32] E.J. Neuhold e M. Paul, editors. *Formal Description of Programming Concepts*, (IFIP) (S)tate-of-the(A)rt Report. IFIP, Springer-Verlag, 1991.

[33] P. Norvig e S. Russell. *Artificial Intelligence. A Modern Approach.* Pretice Hall, Saddle River, New Jersey, USA, first edition, 1994.

[34] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Jornal ACM*, 12:23–41, 1965.

[35] R.M. Smullyan. *First-Order Logic.* Springer-Verlag, New York, 1968.

[36] V.S. Subrahamanian. On the semantics of quantitative logic programs. páginas 173–182, 1987.

[37] V.S. Subrahmanian. Paraconsistent disjunctive deductive databases. *Theoretical Computer Sci.*, 93:115–141, 1992.

[38] M. Tofte. Four lectures on standard ml. march de 1989.

[39] André Vellino. *The Complexity of Automated Reasonig.* Tese de Doutorado, University of Toronto, 1989.

[40] Ake Wikstron. *Funcional Programing Using Standard ML.* Prentice Hall Internacional, UK, first edition, 1987.

APÊNDICE A

LISTAGEM COMPLETA DA IMPLEMENTAÇÃO

A.1 Quarta versão da Implementação

A listagem a seguir é a versão mais atual da implementação. Nesta versão foi adicionado o procedimento de extensão visto no capítulo 5. Mara maiores detalhes veja o capítulo 7.

```
(* Uma implementação do método das conecções de Bibel para a lógica *)
(* proposicional anotada com reticulados FOUR e SIX*)

(* Linguagem : Standard ML *)
(* Autor     : Emerson Faria Nobre - maio/2001 *)

(* a signature a seguir declara a interface do método de Bibel *)

signature BIBEL =
sig
   val valido: string -> bool
end; (* BIBEL *)

(* a signature a seguir declara a interface da estrutura de reticulados *)

signature ESTRUTURA_RETICULADO =
sig
   val estrutura_ordem: (string * string) list
   val estrutura_decomposicao: (string * string list) list
end; (*TIPO_RETICULADO*)
```

```
(* a signature a seguir declara a interface das operações de reticulados *)
(* discretos *)

signature OPERACOES_RETICULADO =
sig
   val maior_igual: (''a * ''a) list -> ''a * ''a -> bool
   val decomposicao: (''a * 'b list) list -> 'c * ''a * string ->
                                                ('c * 'b * string) list
end; (*OPERACOES_RETICULADO*)


(* A structure a seguir declara o reticulado FOUR     *)

(*                          1 Inconsistente (supremo)  *)
(*                         / \                         *)
(*                        /   \                        *)
(*                 Falso f     t Verdadeiro            *)
(*                        \   /                        *)
(*                         \ /                         *)
(*                          0 Indeterminado (ínfimo)   *)

structure Reticulado_FOUR: ESTRUTURA_RETICULADO =
struct
   val estrutura_ordem = [("1","t"),("1","f"),("t","0"),("f","0")];
   val estrutura_decomposicao = [("1",["t","f"])];
end; (*Reticulado_FOUR*)

(* A structure a seguir declara o reticulado SIX *)
```

```
(*                          1 Inconsistente (supremo)  *)
(*                         / \                         *)
(*                        /   \                        *)
(*                 Falso f     t Verdadeiro            *)
(*                       |     |                       *)
(*          Quase Falso qf     qt Quase verdadeiro     *)
(*                        \   /                        *)
(*                         \ /                         *)
(*                          0 Indeterminado (ínfimo)   *)

structure Reticulado_SIX: ESTRUTURA_RETICULADO =
struct
   val estrutura_ordem =
[("1","t"),("1","f"),("t","qt"),("f","qf"),("qt","0"),("qf","0")];
   val estrutura_decomposicao = [("1",["qt","qf"])];
end; (*Reticulado_SIX*)


(* A structure a seguir declara as operações de reticulados discretos *)

structure Reticulado_Discreto: OPERACOES_RETICULADO =
struct
   fun maior_igual ((a,b)::xs) (x,y) = if (x = y)
               then true
               else if (a,b) = (x,y)
                    then true
                    else if (a = x) andalso (maior_igual ((a,b)::xs) (b,y))
                         then true
```

```
                        else maior_igual xs (x,y)
   |   maior_igual _ _ = false;

   fun decomposicao2 nil (_,_,_)     = nil
   |   decomposicao2 (x::xs) (a,b,c) = (a,x,c) :: decomposicao2 xs (a,b,c);

   fun decomposicao _ (_,_,"~")          = nil
   |   decomposicao nil (_,_,_)          = nil
   |   decomposicao ((x,y)::xs) (a,b,c) = if (x = b)
               then decomposicao2 y (a,b,c)
               else decomposicao xs (a,b,c);
end; (*Reticulado_Discreto*)

functor Bibel(Estrutura:ESTRUTURA_RETICULADO)
            (Operacoes:OPERACOES_RETICULADO):BIBEL =
struct

(* ------------------------------------------------------------------- *)
(* Definição do Tipo FORMULA                                           *)
(* O tipo FORMULA é utilizado para representar uma fórmula proposicional   *)
(* como uma árvore de sintaxe abstrata(AST)                            *)
(* ------------------------------------------------------------------- *)

datatype FORMULA = E of FORMULA*FORMULA          (* conjuncao    &&      *)
|                  Ou of FORMULA*FORMULA        (* disjuncao    ||      *)
|                  Impl of FORMULA*FORMULA      (* implicacao   =>      *)
|                  Nao of FORMULA               (* negacao      ~       *)
|                  Prop of string*string;       (* proposicao           *)
                                                 (* (Proposicao, Anotação) *)
```

```
(* ------------------------------------------------------------------- *)
(* As funções definidas a seguir são utilizadas para identificar os Tokens *)
(* da string de entrada                                                *)
(* obs: São funções auxiliares, utilizadas nas funções que constroem a AST *)
(* ------------------------------------------------------------------- *)

(* A função "digito" retorna true se o caracter passado como parâmetro    *)
(* é um número entre 0 e 9                                             *)

fun digito c = c >= #"0" andalso c <= #"9";

(* A função "minusculo" retorna true se o caracter passado como parâmetro  *)
(* é uma letra minúscula                                               *)

fun minusculo c = c >= #"a" andalso c <= #"z";


(* A função "maiusculo" retorna true se o caracter passado como parâmetro  *)
(* é uma letra maiúscula                                               *)

fun maiusculo c = c >= #"A" andalso c <= #"Z";


(* A função "letra" retorna true se o caracter passado como parâmetro      *)
(* é uma letra minúscula ou maiúscula                                  *)

fun letra c =  minusculo c orelse maiusculo c;
```

```
(* A função "igual" retorna true se os dois parâmetros são iguais          *)

fun igual x y = x = y;

(* A função "leAnot" é uma função auxiliar da função "leprop" que le a      *)
(* anotação de um literal. *)

fun leAnot (id, nil) = (id , nil)
|   leAnot (id, c::cs) =
  if letra c orelse digito c
  then leAnot(id ^ (str c), cs)
  else (id, c::cs);

(* A função "leprop" recebe como parâmetro uma string, e se o inicio da     *)
(* string for um literal, então a função retorna o token deste              *)
(* literal.                                                                 *)
(* obs: Um literal é formado por uma sequência de caracteres (proposição)   *)
(* seguido de dois pontos (:) seguido de uma sequencia de                   *)
(* caracteres (anotação)                                                    *)

fun leProp (Prop(pro,anot), nil) = (Prop(pro,anot), nil)
|   leProp (Prop(pro,anot), c::cs) =
  if letra c orelse digito c
  then leProp(Prop(pro ^ (str c), anot), cs)
  else if c = #":"
       then let val (anotacao, ds) = leAnot(anot,cs)
            in leProp(Prop(pro,anotacao), ds)
            end
  else (Prop(pro,anot), c::cs);
```

```
exception sintaxerror;

(* A função "remove_cabeca" remove o elemento que é a cabeça de uma lista *)

fun remove_cabeca p (x::xs) = if p x then xs else raise sintaxerror;

(* ------------------------------------------------------------------------ *)
(* As Funções definidas a seguir constroem uma Árvore de Sintaxe Abstrata  *)
(* a partir de uma string  que representa uma fórmula.                      *)
(* ------------------------------------------------------------------------ *)

fun expressao cs =  expressao'(termo cs)
and expressao' (l, nil) = (l, nil)
|   expressao' (l, c::nil) = (l, c::nil)
|   expressao' (l,b::c::cs) =
      if "=>" = ((str b) ^ (str c))
      then let val (r, cs) = termo cs
                     in expressao' (Impl(l,r),cs)
           end
      else (l,b::c::cs)
and termo cs = termo'(fator cs)
and termo' (l, nil) = (l, nil)
|   termo' (l,c::nil) = (l,c::nil)
|   termo' (l,b::c::cs) =
      if "&&" = ((str b) ^ (str c))
      then let val (r, cs) = fator cs
           in termo' (E(l,r),cs) end
      else if "||" = ((str b) ^ (str c))
```

```
            then let val (r, cs) = fator cs
                 in termo' (Ou(l,r),cs) end
            else (l,b::c::cs)
and fator nil = raise sintaxerror
|   fator (c::cs) =
    if c = #"("
    then let val (t1, cs) = expressao cs
         in (t1, remove_cabeca (igual #")") cs)
         end
    else if c = #"~"
         then let val (t1, cs) = fator cs
              in (Nao(t1),cs)
              end
         else leProp(Prop("","),c::cs);

(* A função "primeiro" retorna o primeiro elemento de uma tupla  *)

fun primeiro (x, _) = x;

(* A função "remove_espacos", remove os espaços em branco de uma *)
(* lista de caracteres                                           *)

fun remove_espacos nil = nil
|   remove_espacos (#" "::cs) = remove_espacos cs
|   remove_espacos (c::cs) = c :: remove_espacos cs;

(* A função "ast" é uma função composta que recebe uma string    *)
(* como parâmetro e retorna uma                                  *)
(* FORMULA (árvore de sintaxe abstrata)                          *)
```

```
val ast = primeiro o expressao o remove_espacos o explode;

(* ------------------------------------------------------------------- *)
(* As funcões definidas a seguir, transformam uma FORMULA proposicional  *)
(* em sua forma normal disjuntiva                                        *)
(* ------------------------------------------------------------------- *)

(*  Implicacao                                  *)
(*  (A Impl B)   <=>   (Nao(A) ou B)            *)

fun implica (Impl(a, b)) = Ou(Nao(implica a), implica b)
|   implica (Ou(a, b))   = Ou(implica a, implica b)
|   implica (E(a, b))    = E(implica a, implica b)
|   implica (Nao(a))     = Nao(implica a)
|   implica (x)          = x;


(*  Leis de De Morgan                           *)
(*  Nao(A Ou B)   <=>   (Nao(A) E Nao(B))       *)
(*  Nao(A E B)    <=>   (Nao(A) Ou Nao(B))      *)

fun deMorgan (Nao(Nao(a)))      = deMorgan a      (* dupla negacao*)
|   deMorgan (Nao(Ou(a,b)))     = E(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Nao(E(a,b)))      = Ou(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Ou(a,b))          = Ou(deMorgan a, deMorgan b)
|   deMorgan (E(a,b))           = E(deMorgan a, deMorgan b)
|   deMorgan (Nao(a))           = Nao(deMorgan a)
|   deMorgan (x) = x;
```

```
(*  Distributividade                              *)
(*  (A E (B ou C)   <=>   (A E B) ou (A E C)   *)

fun distrib1 (E(a,Ou(b,c)))  = Ou(E(distrib1 a,distrib1 b),
                                  E(distrib1 a,distrib1 c))
|   distrib1 (E(Ou(a,b),c))  = Ou(E(distrib1 a,distrib1 c),
                                  E(distrib1 b,distrib1 c))
|   distrib1 (Ou(a,b))       = Ou(distrib1 a,distrib1 b)
|   distrib1 (E(a,b))        = E(distrib1 a,distrib1 b)
|   distrib1 (x)             = x;

fun distrib x =
      let val f = distrib1 x
        in if (x = f) then x else distrib f
      end;

(* A seguinte função é uma função composta que transforma uma formula *)
(* proposicional para a forma normal disjuntiva                       *)

val normal_disj = distrib o deMorgan o implica;

(* Função "pretty print" para imprimir uma FORMULA *)

fun pprint (Prop(x,y))    = x ^ ":" ^ y
|   pprint (E(l,r))     = "(" ^ pprint l ^ "&&" ^ pprint r ^ ")"
|   pprint (Ou(l,r))    = "(" ^ pprint l ^ "||" ^ pprint r ^ ")"
|   pprint (Impl(l,r)) = "(" ^ pprint l ^ "=>" ^ pprint r ^ ")"
```

```
|   pprint (Nao(r))     = "~" ^ pprint r;

(* ------------------------------------------------------------------- *)
(* As funções a seguir transformam uma FORMULA, na forma normal disjuntiva *)
(* em uma matriz de Bibel positiva.                                    *)
(* ------------------------------------------------------------------- *)

fun matriz_coluna (E(E(a,b),Prop(c1,c2)))       = (c1,c2,""):: 
                                                  (matriz_coluna (E(a,b)))
|   matriz_coluna (E(E(a,b),Nao(Prop(c1,c2)))) = (c1,c2,"~"):: 
                                                  (matriz_coluna (E(a,b)))
|   matriz_coluna (E(Prop(a1,a2),E(b,c)))       = (a1,a2,""):: 
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(Nao(Prop(a1,a2)),E(b,c))) = (a1,a2,"~"):: 
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(E(a,b),E(c,d)))             = (matriz_coluna (E(a,b))) @
                                                  (matriz_coluna (E(c,d)))
|   matriz_coluna (E(Prop(a1,a2),Prop(b1,b2))) = (b1,b2,"")::(a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Prop(b1,b2))) = (b1,b2,""):: 
                                                  (a1,a2,"~")::nil
|   matriz_coluna (E(Prop(a1,a2),Nao(Prop(b1,b2)))) = (b1,b2,"~"):: 
                                                  (a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = (b1,b2,"~"):: 
                                                  (a1,a2,"~")::nil
|   matriz_coluna (Prop(a1,a2))                 = (a1,a2,"")::nil
|   matriz_coluna (Nao(Prop(a1,a2)))             = (a1,a2,"~")::nil;

fun matriz (Ou(Ou(a,b),Prop(c1,c2)))      = ((c1,c2,"")::nil):: 
                                            matriz (Ou(a,b))
```

```
|  matriz (Ou(Prop(a1,a2),Ou(b,c)))      = ((a1,a2,""):::nil)::
                                            matriz (Ou(b,c))
|  matriz (Ou(Ou(a,b),Nao(Prop(c1,c2))))= ((c1,c2,"~"):::nil)::
                                            matriz (Ou(a,b))
|  matriz (Ou(Nao(Prop(a1,a2)),Ou(b,c)))= ((a1,a2,"~"):::nil)::
                                            matriz (Ou(b,c))
|  matriz (Ou(Ou(a,b),Ou(c,d)))          = (matriz (Ou(a,b))) @
                                            (matriz (Ou(c,d)))
|  matriz (Ou(Ou(a,b),E(c,d)))           = (matriz_coluna (E(c,d))):::
                                            matriz (Ou(a,b))
|  matriz (Ou(E(a,b),Ou(c,d)))           = (matriz_coluna (E(a,b))):::
                                            matriz (Ou(c,d))
|  matriz (Ou(E(a,b),Prop(c1,c2)))       = ((c1,c2,""):::nil)::
                                            (matriz_coluna (E(a,b))):::nil
|  matriz (Ou(E(a,b),Nao(Prop(c1,c2)))) = ((c1,c2,"~"):::nil)::
                                            (matriz_coluna (E(a,b))):::nil
|  matriz (Ou(Prop(a1,a2),E(b,c)))       = (matriz_coluna (E(b,c))):::
                                            ((a1,a2,""):::nil):::nil
|  matriz (Ou(Nao(Prop(a1,a2)),E(b,c))) = (matriz_coluna (E(b,c))):::
                                            ((a1,a2,"~"):::nil):::nil
|  matriz (Ou(E(a,b),E(c,d)))            = (matriz_coluna (E(c,d))):::
                                            (matriz_coluna (E(a,b))):::nil
|  matriz (Ou(Prop(a1,a2),Prop(b1,b2))) = ((b1,b2,""):::nil)::
                                            ((a1,a2,""):::nil):::nil
|  matriz (Ou(Nao(Prop(a1,a2)),Prop(b1,b2))) = ((b1,b2,""):::nil)::
                                            ((a1,a2,"~"):::nil):::nil
|  matriz (Ou(Prop(a1,a2),Nao(Prop(b1,b2)))) = ((b1,b2,"~"):::nil)::
                                            ((a1,a2,""):::nil):::nil
|  matriz (Ou(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = ((b1,b2,"~"):::nil)::
```

```
                                        ((a1,a2,"~")::nil)::nil
|   matriz a                          = (matriz_coluna a)::nil;

(* ------------------------------------------------------------------ *)
(* As funções definidas a seguir implementam o procedimento de extensão *)
(* ------------------------------------------------------------------ *)

(* A função "decompoe2" faz a decomposição dos literais de uma coluna *)

fun decompoe2 nil           = nil
|   decompoe2 ((a,b,c)::cs) = let val x = Operacoes.decomposicao
                                    Estrutura.estrutura_decomposicao (a,b,c)
                              in
                              if (x = nil)
                                 then [(a,b,c)] @ decompoe2 cs
                                 else x @ decompoe2 cs
                             end;

(* A função "decompoe" passa cada uma das colunas da matriz para a   *)
(* função "decompoe2"                                                *)

fun decompoe nil       = nil
|   decompoe (xs::xss) = decompoe2 xs :: decompoe xss;

(* a função "segundo" retorna o segundo elemento de um par *)

fun segundo (_,x) = x;

(*   A função membro retorna true se o primeiro parâmetro é membro  *)
```

```
(* da lista passada como segundo parâmetro                          *)

fun membro (a,nil)    = false
|   membro (a,x::xs) = a = x orelse membro (a,xs);

(*  A função remove_repetido1, remove os elementos repetidos de uma lista *)

fun remove_repetido1 nil = nil
|   remove_repetido1 (x::xs) = if membro(x,xs) then remove_repetido1 xs
                               else x:: remove_repetido1 xs;

(* A função remove_repetido, remove os elementos repetidos de todas as *)
(* sublistas de uma lista de listas                                    *)

fun remove_repetido nil = nil
|   remove_repetido [[]] = [[]]
|   remove_repetido ((x::xs)::yss) = remove_repetido1 (x::xs)::
                                     remove_repetido yss;

(* A função "coluna_positiva" retorna true se todos os literais da *)
(* coluna passada como parâmetro não forem negados                 *)

fun coluna_positiva nil = false
|   coluna_positiva ((_,_,"~")::nil) = false
|   coluna_positiva ((_,_,"")::nil) = true
|   coluna_positiva ((_,_,"~")::xs) = false
|   coluna_positiva ((_,_,"")::xs) = coluna_positiva xs;

(* A função "acha_coluna_positiva" recebe uma matriz como parâmetro e *)
```

```
(* retorna uma coluna que possui somente literais não negados        *)

fun acha_coluna_positiva  nil = nil
|   acha_coluna_positiva  (xs::nil) =
                    if (coluna_positiva xs)
                    then xs
                    else nil
|   acha_coluna_positiva (xs::xss) =
                    if (coluna_positiva xs)
                    then xs
                    else acha_coluna_positiva xss;

(* A função complementar recebe dois literais como parâmetro e retorna *)
(* verdadeiro se eles são complementares                               *)

fun complementar (a,b,c) (d,e,f) =
                    if (a=d) andalso (f="") andalso
                      (c="~") andalso (Operacoes.maior_igual
                       Estrutura.estrutura_ordem(b,e))
                    then true
                    else if (a=d) andalso (f="~") andalso
                           (c="") andalso (Operacoes.maior_igual
                            Estrutura.estrutura_ordem(e,b))
                          then true
                          else false;

(* A função "menos_conjunto" remove um elemento de uma lista *)

fun menos_conjunto _ nil = nil
```

```
|   menos_conjunto b (x::xs) =
                    if b = x
                    then xs
                    else x:: menos_conjunto b xs;

(* A função "menos_complementar_conjunto" recebe como parâmetro *)
(* um literal e uma coluna e retorna a coluna sem o elemento    *)
(* complementar do primeiro parâmetro                           *)

fun menos_complementar_conjunto _  nil = nil
|   menos_complementar_conjunto b (x::xs) =
                    if (complementar b  x)
                    then xs
                    else x:: menos_complementar_conjunto b xs;

(* A função "membro_complementar" recebe como parâmetro um literal *)
(* e uma coluna e retorna true se existe um literal complementar   *)
(* na coluna                                                       *)

fun membro_complementar _ nil = false
|   membro_complementar b (x::xs) = if (complementar b  x) then true
                    else membro_complementar b xs;

(* A função "acha_coluna_complementar" recebe como parâmetro um     *)
(* literal e uma matriz e retorna a seguinte tripla:                *)
(* Primeiro elemento da tripla é a matriz completa                  *)
(* Segundo elemento da tripla é a matriz sem a coluna complementar  *)
(* Terceiro elemento da tripla é true se existe coluna complementar *)
```

```
fun acha_coluna_complementar _ nil = (nil,nil,false)
|   acha_coluna_complementar b ((x::xs)::xss) =
                    if (membro_complementar b (x::xs))
                    then ((x::xs),
                       menos_complementar_conjunto b (x::xs),true)
                    else acha_coluna_complementar b xss;

(* A função "expande" implementa o procedimento de extensão *)

fun expande bs ((x::xs)::xss) =
         if (membro_complementar x bs)
         then if xs = nil
           then true
           else expande bs (xs::xss)
         else let
             val (col, col_menos, status) =
                  acha_coluna_complementar x xss
         in
             if status then
               let val status_subnivel =
                     if col_menos = nil then true
                     else expande (x::bs)
                       (col_menos::(menos_conjunto col xss))
               in if status_subnivel then
                     if xs = nil then true
                     else expande bs (xs::xss)
                  else false
               end
             else false
```

```
        end;

(* a função "matriz_infimo" retorna verdadeiro se a matriz pasada *)
(* como parâmetro possui um único literal anotado pelo elemento   *)
(* infimo do reticulado                                           *)

fun matriz_infimo (((_,b,c)::nil)::nil) = if (b = "0") andalso (c = "")
                                          then true
                                          else false
|   matriz_infimo _                      = false;

(* a função "spanning" retorna verdadeiro se a matriz é válida *)

fun spanning xss = if (matriz_infimo xss)
                   then true
                   else let
                      val matriz = acha_coluna_positiva xss
                   in if (matriz <> nil)
                      then expande nil (matriz::(menos_conjunto matriz xss))
                      else false
                   end;


(* A função valido é a função principal deste programa, retorna *)
(* true se a formula é válida. Esta função aplica, direta ou    *)
(* indiretamente todas as outras funções desta implementação    *)

val valido = spanning o remove_repetido o decompoe o matriz o
                                                    norma_disj o ast;
```

```
end; (* Bibel *)

(* A estrututa "AnotadaFOUR" associa a estrutura Bibel com a estrututa *)
(* do reticulado FOUR. Esta é a estrutura que é utilizada para provar  *)
(* teoremas da lógica anotada para o reticulado FOUR                   *)
(* Exemplo: Digite a seguinte linha no prompt do interpretador SML     *)
(*          AnotadaFOUR.valido "A:t && (A:t => A:f) => A:1";           *)
(* retorna verdaderiro pois a fórmula é válida                         *)

structure AnotadaFOUR = Bibel (Reticulado_FOUR) (Reticulado_Discreto);

(* A estrututa "AnotadaSIX" associa a estrutura Bibel com a estrututa  *)
(* do reticulado SIX. Esta é a estrutura que é utilizada para provar   *)
(* teoremas da lógica anotada para o reticulado SIX                    *)
(* Exemplo: Digite a seguinte linha no prompt do interpretador SML     *)
(*          AnotadaFOUR.valido "A:qt && (A:qt=> A:f) => A:1";          *)
(* retorna verdaderiro pois a fórmula é válida                         *)

structure AnotadaSIX = Bibel (Reticulado_SIX) (Reticulado_Discreto);
```

A.2 Terceira versão da Implementação

Na terceira versão foram adicionadas as estruturas e operações de reticulados, permitindo assim que novos reticulados possam ser adicionados de forma simples, sem afetar a estrutura do programa.

```
(* Uma implementação do método das conecções de Bibel para a lógica *)
(* proposicional anotada com reticulados FOUR e SIX                  *)

(* Linguagem : Standard ML *)
```

```
(* Autor      : Emerson Faria Nobre - fevereiro/2001 *)


(* a signature a seguir declara a interface do método de Bibel *)

signature BIBEL =
sig
   val valido: string -> bool

end; (* BIBEL *)

(* a signature a seguir declara a interface da estrutura de reticulados *)

signature ESTRUTURA_RETICULADO =
sig
   val estrutura_ordem: (string * string) list
   val estrutura_decomposicao: (string * string list) list
end; (*TIPO_RETICULADO*)

(* a signature a seguir declara a interface das operações de reticulados *)
(* discretos *)

signature OPERACOES_RETICULADO =
sig
   val maior_igual: (''a * ''a) list -> ''a * ''a -> bool
   val decomposicao: (''a * 'b list) list -> 'c * ''a * string ->
                                          ('c * 'b * string) list
end; (*OPERACOES_RETICULADO*)
```

```
(* A structure a seguir declara o reticulado FOUR     *)

(*                        1 Inconsistente (supremo)  *)
(*                       / \                         *)
(*                      /   \                        *)
(*               Falso f     t Verdadeiro            *)
(*                      \   /                        *)
(*                       \ /                         *)
(*                        0 Indeterminado (infimo)   *)

structure Reticulado_FOUR: ESTRUTURA_RETICULADO =
struct
   val estrutura_ordem = [("1","t"),("1","f"),("t","0"),("f","0")];
   val estrutura_decomposicao = [("1",["t","f"])];
end; (*Reticulado_FOUR*)

(* A structure a seguir declara o reticulado SIX *)

(*                        1 Inconsistente (supremo)  *)
(*                       / \                         *)
(*                      /   \                        *)
(*               Falso f     t Verdadeiro            *)
(*                     |     |                       *)
(*        Quase Falso qf     qt Quase verdadeiro     *)
(*                      \   /                        *)
(*                       \ /                         *)
(*                        0 Indeterminado (infimo)   *)

structure Reticulado_SIX: ESTRUTURA_RETICULADO =
```

```
struct
   val estrutura_ordem =
[("1","t"),("1","f"),("t","qt"),("f","qf"),("qt","0"),("qf","0")];
   val estrutura_decomposicao = [("1",["qt","qf"])];
end; (*Reticulado_SIX*)

(* A structure a seguir declara as operações de reticulados discretos *)

structure Reticulado_Discreto: OPERACOES_RETICULADO =
struct
   fun maior_igual ((a,b)::xs) (x,y) = if (x = y)
               then true
               else if (a,b) = (x,y)
                    then true
                    else if (a = x) andalso (maior_igual ((a,b)::xs) (b,y))
                         then true
                         else maior_igual xs (x,y)
   |   maior_igual _ _ = false;

   fun decomposicao2 nil (_,_,_)     = nil
   |   decomposicao2 (x::xs) (a,b,c) = (a,x,c) :: decomposicao2 xs (a,b,c);

   fun decomposicao _ (_,_,"~")         = nil
   |   decomposicao nil (_,_,_)         = nil
   |   decomposicao ((x,y)::xs) (a,b,c) = if (x = b)
               then decomposicao2 y (a,b,c)
               else decomposicao xs (a,b,c);
end; (*Reticulado_Discreto*)
```

```
functor Bibel(Estrutura:ESTRUTURA_RETICULADO)
(Operacoes:OPERACOES_RETICULADO):BIBEL =
struct

(* ------------------------------------------------------------------------ *)
(* Definicao do Tipo FORMULA                                                *)
(* O tipo FORMULA é utilizado para representar uma formula proposicional    *)
(* como uma árvore de sintaxe abstrata(AST)                                 *)
(* ------------------------------------------------------------------------ *)

datatype FORMULA = E of FORMULA*FORMULA        (* conjuncao     &&        *)
|                  Ou of FORMULA*FORMULA      (* disjuncao     ||        *)
|                  Impl of FORMULA*FORMULA    (* implicacao    =>        *)
|                  Nao of FORMULA             (* negacao       ~         *)
|                  Prop of string*string;     (* proposicao              *)
                                               (* (Proposicao, Anotação) *)

(* A atribuição a seguir declara o reticulado FOUR *)

(*                       1 Inconsistente (supremo)  *)
(*                      / \                         *)
(*                     /   \                        *)
(*               Falso f     t Verdadeiro           *)
(*                     \   /                        *)
(*                      \ /                         *)
(*                       0 Indeterminado (infimo)   *)

(* ------------------------------------------------------------------------ *)
(* As funções definidas a seguir são utilizadas para identificar os Tokens *)
```

```
(* da string de entrada                                                    *)
(* obs: São funções auxiliares, utilizadas nas funções que constroem a AST *)
(* ------------------------------------------------------------------------ *)

(* A função "digito" retorna true se o caracter passado como parâmetro *)
(* é um número entre 0 e 9*)

fun digito c = c >= #"0" andalso c <= #"9";

(* A função "minusculo" retorna true se o caracter passado como parâmetro *)
(* é uma letra minúscula                                                   *)

fun minusculo c = c >= #"a" andalso c <= #"z";

(* A função "maiusculo" retorna true se o caracter passado como parâmetro *)
(* é uma letra maiúscula                                                   *)

fun maiusculo c = c >= #"A" andalso c <= #"Z";

(* A função "letra" retorna true se o caracter passado como parâmetro *)
(* é uma letra minúscula ou maiúscula                                      *)

fun letra c =  minusculo c orelse maiusculo c;

(* A função "igual" retorna true se os dois parâmetros são iguais *)

fun igual x y = x = y;

(* A função "leAnot" é uma função auxiliar da função "leprop" que le a *)
```

```
(* anotação de um literal.                                          *)

fun leAnot (id, nil) = (id , nil)
|   leAnot (id, c::cs) =
  if letra c orelse digito c
  then leAnot(id ^ (str c), cs)
  else (id, c::cs);

(* A função "leprop" recebe como parâmetro uma string, e se o inicio da  *)
(* string for um literal, então a função retorna o token deste           *)
(* literal.                                                              *)
(* obs: Um literal é formado por uma sequência de caracteres (proposição)*)
(* seguido de dois pontos (:) seguido de uma sequencia de                *)
(* caracteres (anotação)                                                 *)

fun leProp (Prop(pro,anot), nil) = (Prop(pro,anot), nil)
|   leProp (Prop(pro,anot), c::cs) =
  if letra c orelse digito c
  then leProp(Prop(pro ^ (str c), anot), cs)
  else if c = #":"
       then let val (anotacao, ds) = leAnot(anot,cs)
            in leProp(Prop(pro,anotacao), ds)
            end
  else (Prop(pro,anot), c::cs);

exception sintaxerror;

(* A função "remove_cabeca" remove o elemento que é a cabeça de uma lista *)
```

```
fun remove_cabeca p (x::xs) = if p x then xs else raise sintaxerror;

(* ------------------------------------------------------------------- *)
(* As Funções definidas a seguir constroem uma Árvore de Sintaxe Abstrata  *)
(* a partir de uma string                                              *)
(* ------------------------------------------------------------------- *)

fun expressao cs =  expressao'(termo cs)
and expressao' (l, nil) = (l, nil)
|   expressao' (l, c::nil) = (l, c::nil)
|   expressao' (l,b::c::cs) =
      if "=>" = ((str b) ^ (str c))
      then let val (r, cs) = termo cs
                     in expressao' (Impl(l,r),cs)
           end
      else (l,b::c::cs)
and termo cs = termo'(fator cs)
and termo' (l, nil) = (l, nil)
|   termo' (l,c::nil) = (l,c::nil)
|   termo' (l,b::c::cs) =
      if "&&" = ((str b) ^ (str c))
      then let val (r, cs) = fator cs
           in termo' (E(l,r),cs) end
      else if "||" = ((str b) ^ (str c))
           then let val (r, cs) = fator cs
                in termo' (Ou(l,r),cs) end
           else (l,b::c::cs)
and fator nil = raise sintaxerror
|   fator (c::cs) =
```

```
    if c = #"("
    then let val (t1, cs) = expressao cs
         in (t1, remove_cabeca (igual #")") cs)
         end
    else if c = #"~"
         then let val (t1, cs) = fator cs
              in (Nao(t1),cs)
              end
         else leProp(Prop("",""),c::cs);

(* A função "primeiro" retorna o primeiro elemento de uma tupla *)

fun primeiro (x, _) = x;

(* A função "remove_espacos", remove os espaços em branco de uma *)
(* lista de caracteres                                           *)

fun remove_espacos nil = nil
|   remove_espacos (#" "::cs) = remove_espacos cs
|   remove_espacos (c::cs) = c :: remove_espacos cs;

(* A função "ast" recebe uma string como parâmetro e retorna uma *)
(* FORMULA(árvore de sintaxe abstrata)                           *)

val ast = primeiro o expressao o remove_espacos o explode;

(* ------------------------------------------------------------- *)
(* As funcões definidas a seguir, transformam uma FORMULA proposicional *)
(* em sua forma normal disjuntiva                                *)
```

```
(* ------------------------------------------------------------------ *)

(*  Implicacao   *)
(*  (A Impl B)   <=>   (Nao(A) ou B)  *)

fun implica (Impl(a, b)) = Ou(Nao(implica a), implica b)
|   implica (Ou(a, b))   = Ou(implica a, implica b)
|   implica (E(a, b))    = E(implica a, implica b)
|   implica (Nao(a))     = Nao(implica a)
|   implica (x)          = x;

(*  Leis de De Morgan                    *)
(*  Nao(A Ou B)   <=>   (Nao(A) E Nao(B))    *)
(*  Nao(A E B)    <=>   (Nao(A) Ou Nao(B))    *)

fun deMorgan (Nao(Nao(a)))      = deMorgan a                (* dupla negacao*)
|   deMorgan (Nao(Ou(a,b)))     = E(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Nao(E(a,b)))      = Ou(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Ou(a,b))          = Ou(deMorgan a, deMorgan b)
|   deMorgan (E(a,b))           = E(deMorgan a, deMorgan b)
|   deMorgan (Nao(a))           = Nao(deMorgan a)
|   deMorgan (x) = x;

(*  Distributividade                              *)
(*  (A E (B ou C)   <=>   (A E B) ou (A E C)      *)

fun distrib1 (E(a,Ou(b,c)))  = Ou(E(distrib1 a,distrib1 b),
                                  E(distrib1 a,distrib1 c))
|   distrib1 (E(Ou(a,b),c))  = Ou(E(distrib1 a,distrib1 c),
```

```
                                    E(distrib1 b,distrib1 c))
|   distrib1 (Ou(a,b))        = Ou(distrib1 a,distrib1 b)
|   distrib1 (E(a,b))         = E(distrib1 a,distrib1 b)
|   distrib1 (x)              = x;

fun distrib x =
       let val f = distrib1 x
         in if (x = f) then x else distrib f
       end;

(* Transforma uma formula proposicional para a forma normal disjuntiva *)

val normal_disj = distrib o deMorgan o implica;

(* Imprime uma FORMULA *)

fun pprint (Prop(x,y))     = x ^ ":" ^ y
|   pprint (E(l,r))     = "(" ^ pprint l ^ "&&" ^ pprint r ^ ")"
|   pprint (Ou(l,r))    = "(" ^ pprint l ^ "||" ^ pprint r ^ ")"
|   pprint (Impl(l,r)) = "(" ^ pprint l ^ "=>" ^ pprint r ^ ")"
|   pprint (Nao(r))     = "~" ^ pprint r;

(* ------------------------------------------------------------------------ *)
(* As funções a seguir transformam uma FORMULA, na forma normal disjuntiva *)
(* em uma matriz de Bibel positiva.                                         *)
(* ------------------------------------------------------------------------*)

fun matriz_coluna (E(E(a,b),Prop(c1,c2)))      = (c1,c2,""):::
                                                  (matriz_coluna (E(a,b)))
```

```
|   matriz_coluna (E(E(a,b),Nao(Prop(c1,c2)))) = (c1,c2,"~")::
                                                  (matriz_coluna (E(a,b)))
|   matriz_coluna (E(Prop(a1,a2),E(b,c)))      = (a1,a2,"")::
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(Nao(Prop(a1,a2)),E(b,c))) = (a1,a2,"~")::
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(E(a,b),E(c,d)))           = (matriz_coluna (E(a,b))) @
                                                  (matriz_coluna (E(c,d)))
|   matriz_coluna (E(Prop(a1,a2),Prop(b1,b2))) = (b1,b2,"")::(a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Prop(b1,b2))) = (b1,b2,"")::
                                                  (a1,a2,"~")::nil
|   matriz_coluna (E(Prop(a1,a2),Nao(Prop(b1,b2)))) = (b1,b2,"~")::
                                                  (a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = (b1,b2,"~")::
                                                  (a1,a2,"~")::nil
|   matriz_coluna (Prop(a1,a2))                = (a1,a2,"")::nil
|   matriz_coluna (Nao(Prop(a1,a2)))           = (a1,a2,"~")::nil;

fun matriz (Ou(Ou(a,b),Prop(c1,c2)))     = ((c1,c2,"")::nil)::
                                           matriz (Ou(a,b))
|   matriz (Ou(Prop(a1,a2),Ou(b,c)))     = ((a1,a2,"")::nil)::
                                           matriz (Ou(b,c))
|   matriz (Ou(Ou(a,b),Nao(Prop(c1,c2))))= ((c1,c2,"~")::nil)::
                                           matriz (Ou(a,b))
|   matriz (Ou(Nao(Prop(a1,a2)),Ou(b,c)))= ((a1,a2,"~")::nil)::
                                           matriz (Ou(b,c))
|   matriz (Ou(Ou(a,b),Ou(c,d)))         = (matriz (Ou(a,b))) @
                                           (matriz (Ou(c,d)))
|   matriz (Ou(Ou(a,b),E(c,d)))          = (matriz_coluna (E(c,d)))::
```

```
                                          matriz (Ou(a,b))
|   matriz (Ou(E(a,b),Ou(c,d)))          = (matriz_coluna (E(a,b)))::
                                          matriz (Ou(c,d))
|   matriz (Ou(E(a,b),Prop(c1,c2)))      = ((c1,c2,"")::nil)::
                                          (matriz_coluna (E(a,b)))::nil
|   matriz (Ou(E(a,b),Nao(Prop(c1,c2)))) = ((c1,c2,"~")::nil)::
                                          (matriz_coluna (E(a,b)))::nil
|   matriz (Ou(Prop(a1,a2),E(b,c)))      = (matriz_coluna (E(b,c)))::
                                          ((a1,a2,"")::nil)::nil
|   matriz (Ou(Nao(Prop(a1,a2)),E(b,c))) = (matriz_coluna (E(b,c)))::
                                          ((a1,a2,"~")::nil)::nil
|   matriz (Ou(E(a,b),E(c,d)))           = (matriz_coluna (E(c,d)))::
                                          (matriz_coluna (E(a,b)))::nil
|   matriz (Ou(Prop(a1,a2),Prop(b1,b2))) = ((b1,b2,"")::nil)::
                                          ((a1,a2,"")::nil)::nil
|   matriz (Ou(Nao(Prop(a1,a2)),Prop(b1,b2))) = ((b1,b2,"")::nil)::
                                          ((a1,a2,"~")::nil)::nil
|   matriz (Ou(Prop(a1,a2),Nao(Prop(b1,b2)))) = ((b1,b2,"~")::nil)::
                                          ((a1,a2,"")::nil)::nil
|   matriz (Ou(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = ((b1,b2,"~")::nil)::
                                          ((a1,a2,"~")::nil)::nil
|   matriz a                             = (matriz_coluna a)::nil;

(* ------------------------------------------------------------------- *)
(* As funções definidas a seguir transformam uma matriz positiva de Bibel *)
(* em uma matriz com todos os "PATHS"                                   *)
(* ------------------------------------------------------------------- *)

fun path1 ((_::nil)::yss) (zs::zss) = zs:: path1 yss zss
```

```
|   path1 ((_::xs)::yss) _ = xs :: yss
|   path1 _ _ = nil;

(* A função path2 recebe uma lista de listas e retorna uma lista formada
   pelos elementos da cabeça de cada sublista *)

fun path2 (nil) = nil
|   path2 ((x::_)::yss) = x:: path2 yss;

fun path3 (_) (_) 0 = nil
|   path3 (xss) (yss) n =
      let val aux = path1 xss yss
          in path2 aux ::  path3 aux yss (n-1)
      end;

(* A função qtde_paths calcula a quantidade de paths de uma lista de listas*)

fun qtde_paths nil = 1
|   qtde_paths (xs::xss) = length (xs) * qtde_paths (xss);

(* Recebe com parâmetro uma lista de listas(Matriz) e retorna uma lista de*)
(* listas(Matriz) com todos os paths                                      *)

fun paths (xss) =
      let val qtde = (qtde_paths xss) - 1
        in path2 xss :: path3 xss xss qtde
      end;

(* ------------------------------------------------------------------ *)
```

```
(* As funções definidas a seguir tem o objetivo de verificar se todos *)
(* os paths são complementares                                        *)
(* ------------------------------------------------------------------ *)

(* A função "decompoe2" faz a decomposição dos literais de uma coluna *)

fun decompoe2 nil           = nil
|   decompoe2 ((a,b,c)::cs) = let val x = Operacoes.decomposicao
                                 Estrutura.estrutura_decomposicao (a,b,c)
                             in
                             if (x = nil)
                                then [(a,b,c)] @ decompoe2 cs
                                else x @ decompoe2 cs
                          end;

(* A função "decompoe" passa cada uma das colunas da matriz para a   *)
(* função "decompoe2"                                                *)

fun decompoe nil       = nil
|   decompoe (xs::xss) = decompoe2 xs :: decompoe xss;

(* a função "segundo" retorna o segundo elemento de um par *)

fun segundo (_,x) = x;

(* A função "membro" retorna true se o primeiro parâmetro é membro da *)
(* lista passada como segundo parâmetro                               *)

fun membro (a,nil)   = false
```

```
|   membro (a,x::xs) = a = x orelse membro (a,xs);

(* A função "complementar1" retorna true se existem elementos
   complementares *)

fun complementar1 ((a,b,c)::nil)         = false
|   complementar1 ((a,b,c)::(d,e,f)::ds) = if (a = d) andalso (f = "")
                                            andalso (c = "~")
                                            andalso (Operacoes.maior_igual
                                           Estrutura.estrutura_ordem (b,e))
                                           then true
                                           else if (a = d) andalso (f = "~")
                                            andalso (c = "")
                                 andalso (Operacoes.maior_igual
                                         Estrutura.estrutura_ordem (e,b))
                                 then true
                                 else complementar1 ((a,b,c)::ds);


fun complementar ((a,b,c)::nil) = false
|   complementar ((a,b,c)::xs)  = if (complementar1 ((a,b,c)::xs))
                                  then true
                                  else complementar xs;

fun tudo_complementar nil           = true
|   tudo_complementar [[]]          = false
|   tudo_complementar ((x::xs)::yss) = if complementar (x::xs) then
                                        tudo_complementar yss
                                      else false;
```

```
fun matriz_infimo (((_,b,c)::nil)::nil) = if (b = "0") andalso (c = "")
                                          then true
                                          else false
|   matriz_infimo _                              = false;


fun spanning x = if (tudo_complementar x)
                 then true
                 else if (matriz_infimo x)
                      then true
                      else false;

(*  A função remove_repetido1, remove os elementos repetidos de uma lista*)

fun remove_repetido1 nil = nil
|   remove_repetido1 (x::xs) = if membro(x,xs) then remove_repetido1 xs
                               else x:: remove_repetido1 xs;


(* A função remove_repetido, remove os elementos repetidos de todas as *)
(* sublistas de uma lista de listas *)

fun remove_repetido nil = nil
|   remove_repetido [[]] = [[]]
|   remove_repetido ((x::xs)::yss) = remove_repetido1 (x::xs)::
                                     remove_repetido yss;

(* A função remove_tautologia, remove as sublistas de uma lista que *)
```

```
(* possuem elementos complementares *)

fun remove_tautologia nil = nil
|   remove_tautologia ((x::xs)::yss) = if complementar (x::xs) then
                                          remove_tautologia yss
                                       else (x::xs):: remove_tautologia yss;

(* A função valido é a função principal deste programa, retorna *)
(* true se a formula é válida. Esta função aplica, direta ou    *)
(* indiretamente todas as outras funções desta implementação    *)

val valido = spanning o remove_tautologia o remove_repetido o decompoe
                                       o matriz o normal_disj o ast;
end; (* Bibel *)

(* A estrututa "AnotadaFOUR" associa a estrutura Bibel com a estrututa *)
(* do reticulado FOUR. Esta é a estrutura que é utilizada para provar  *)
(* teoremas da lógica anotada para o reticulado FOUR                   *)
(* Exemplo: Digite a seguinte linha no prompt do interpretador SML     *)
(*          AnotadaFOUR.valido "A:t && (A:t => A:f) => A:1";           *)
(* retorna verdaderiro pois a fórmula é válida                         *)

structure AnotadaFOUR = Bibel (Reticulado_FOUR) (Reticulado_Discreto);

(* A estrututa "AnotadaSIX" associa a estrutura Bibel com a estrututa  *)
(* do reticulado SIX. Esta é a estrutura que é utilizada para provar   *)
(* teoremas da lógica anotada para o reticulado SIX                    *)
(* Exemplo: Digite a seguinte linha no prompt do interpretador SML     *)
(*          AnotadaFOUR.valido "A:qt && (A:qt=> A:f) => A:1";          *)
```

```
(* retorna verdaderiro pois a fórmula é válida                    *)

structure AnotadaSIX = Bibel (Reticulado_SIX) (Reticulado_Discreto);
```

A.3 Segunda versão da Implementação

Na segunda versão o método passou a tratar a lógica paraconsistente anotada com o retículado FOUR.

A segunda versão da implementação prova teoremas de uma lógica paraconsistente anotada proposicional com o reticulado FOUR através do método das conexões de bibel.

```
(* Uma implementação do método das conecções de Bibel para a lógica *)
(* proposicional anotada com o reticulado FOUR *)

(* Linguagem : Standard ML *)
(* Autor     : Emerson Faria Nobre - novembro/2000 *)

signature BIBEL =
sig
   val valido: string -> bool
end; (* BIBEL *)

structure Bibel =
struct

(* ------------------------------------------------------------------ *)
(* Definicao do Tipo FORMULA                                          *)
(* O tipo FORMULA é utilizado para representar uma formula proposicional  *)
(* como uma árvore de sintaxe abstrata(AST)                           *)
(* ------------------------------------------------------------------*)
```

```
datatype FORMULA = E of FORMULA*FORMULA          (* conjuncao    &&        *)
|                  Ou of FORMULA*FORMULA         (* disjuncao    ||        *)
|                  Impl of FORMULA*FORMULA       (* implicacao   =>        *)
|                  Nao of FORMULA                (* negacao      ~         *)
|                  Prop of string*string;        (* proposicao             *)
                                                 (* (Proposicao, Anotação) *)

(* A atribuição a seguir declara o reticulado FOUR *)

(*                        1 Inconsistente (supremo)  *)
(*                       / \                         *)
(*                      /   \                        *)
(*                Falso t     f Verdadeiro           *)
(*                      \   /                        *)
(*                       \ /                         *)
(*                        0 Indeterminado (infimo)   *)

val reticulado_four = [("1","t"),("1","f"),("t","0"),("f","0")];

(* ------------------------------------------------------------------ *)
(* A função definida a seguir retorna TRUE se o parametro x for maior ou  *)
(* igual ao parametro y. (obs: a relação de maior igual é referente ao    *)
(* reticulado.                                                            *)
(* ------------------------------------------------------------------ *)

fun maior_igual ((a,b)::xs) (x,y) = if (x = y)
                then true
                else if (a,b) = (x,y)
```

```
                    then true
                    else if (a = x) andalso (maior_igual ((a,b)::xs) (b,y))
                         then true
                         else maior_igual xs (x,y)
|   maior_igual _ _ = false;

(* ------------------------------------------------------------------------ *)
(* As funções definidas a seguir são utilizadas para identificar os Tokens *)
(* da string de entrada                                                     *)
(* obs: São funções auxiliares, utilizadas nas funções que constroem a AST *)
(* ------------------------------------------------------------------------ *)

(* A função "digito" retorna true se o caracter passado como parâmetro *)
(* é um número entre 0 e 9*)

fun digito c = c >= #"0" andalso c <= #"9";


(* A função "minusculo" retorna true se o caracter passado como parâmetro *)
(* é uma letra minúscula *)

fun minusculo c = c >= #"a" andalso c <= #"z";

(* A função "maiusculo" retorna true se o caracter passado como parâmetro *)
(* é uma letra maiúscula *)

fun maiusculo c = c >= #"A" andalso c <= #"Z";

(* A função "letra" retorna true se o caracter passado como parâmetro *)
```

```
(* é uma letra minúscula ou maiúscula *)

fun letra c =  minusculo c orelse maiusculo c;

(* A função "igual" retorna true se os dois parâmetros são iguais *)

fun igual x y = x = y;

(* A função "leAnot" é uma função auxiliar da função "leprop" que le a  *)
(* anotação de um literal. *)

fun leAnot (id, nil) = (id , nil)
|   leAnot (id, c::cs) =
  if letra c orelse digito c
  then leAnot(id ^ (str c), cs)
  else (id, c::cs);

(* A função "leprop" recebe como parâmetro uma string, e se o inicio da    *)
(* string for um literal, então a função retorna o token deste             *)
(* literal.                                                                *)
(* obs: Um literal é formado por uma sequência de caracteres (proposição)  *)
(* seguido de dois pontos (:) seguido de uma sequencia de                  *)
(* caracteres (anotação)                                                   *)

fun leProp (Prop(pro,anot), nil) = (Prop(pro,anot), nil)
|   leProp (Prop(pro,anot), c::cs) =
  if letra c orelse digito c
  then leProp(Prop(pro ^ (str c), anot), cs)
  else if c = #":"
```

```
        then let val (anotacao, ds) = leAnot(anot,cs)
             in leProp(Prop(pro,anotacao), ds)
             end
   else (Prop(pro,anot), c::cs);

exception sintaxerror;

fun remove_cabeca p (x::xs) = if p x then xs else raise sintaxerror;

(* ------------------------------------------------------------------------*)
(* As Funções definidas a seguir constroem uma Árvore de Sintaxe Abstrata *)
(* a partir de uma string                                                  *)
(* ------------------------------------------------------------------------*)

fun expressao cs =  expressao'(termo cs)
and expressao' (l, nil) = (l, nil)
|   expressao' (l, c::nil) = (l, c::nil)
|   expressao' (l,b::c::cs) =
      if "=>" = ((str b) ^ (str c))
      then let val (r, cs) = termo cs
                     in expressao' (Impl(l,r),cs)
           end
      else (l,b::c::cs)
and termo cs = termo'(fator cs)
and termo' (l, nil) = (l, nil)
|   termo' (l,c::nil) = (l,c::nil)
|   termo' (l,b::c::cs) =
      if "&&" = ((str b) ^ (str c))
      then let val (r, cs) = fator cs
```

```
            in termo' (E(l,r),cs) end
       else if "||" = ((str b) ^ (str c))
            then let val (r, cs) = fator cs
                 in termo' (Ou(l,r),cs) end
            else (l,b::c::cs)
and fator nil = raise sintaxerror
|   fator (c::cs) =
    if c = #"("
    then let val (t1, cs) = expressao cs
         in (t1, remove_cabeca (igual #")") cs)
         end
    else if c = #"~"
         then let val (t1, cs) = fator cs
              in (Nao(t1),cs)
              end
         else leProp(Prop("",""),c::cs);


(* A função "primeiro" retorna o primeiro elemento de uma tupla *)

fun primeiro (x, _) = x;

(* A função "remove_espacos", remove os espaços em branco de uma *)
(* lista de caracteres *)

fun remove_espacos nil = nil
|   remove_espacos (#" "::cs) = remove_espacos cs
|   remove_espacos (c::cs) = c :: remove_espacos cs;
```

```
(* A função "ast" recebe uma string como parâmetro e retorna uma *)
(* FORMULA(árvore de sintaxe abstrata) *)

val ast = primeiro o expressao o remove_espacos o explode;

(* ------------------------------------------------------------------*)
(* As funcões definidas a seguir, transformam uma FORMULA proposicional *)
(* em sua forma normal disjuntiva                                     *)
(* ------------------------------------------------------------------ *)

(*  Implicacao   *)
(*  (A Impl B)   <=>   (Nao(A) ou B)  *)

fun implica (Impl(a, b)) = Ou(Nao(implica a), implica b)
|   implica (Ou(a, b))   = Ou(implica a, implica b)
|   implica (E(a, b))    = E(implica a, implica b)
|   implica (Nao(a))     = Nao(implica a)
|   implica (x)          = x;

(*  Leis de De Morgan                  *)
(*  Nao(A Ou B)   <=>   (Nao(A) E Nao(B))    *)
(*  Nao(A E B)    <=>   (Nao(A) Ou Nao(B))    *)

fun deMorgan (Nao(Nao(a)))      = deMorgan a                (* dupla negacao*)
|   deMorgan (Nao(Ou(a,b)))     = E(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Nao(E(a,b)))      = Ou(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Ou(a,b))          = Ou(deMorgan a, deMorgan b)
|   deMorgan (E(a,b))           = E(deMorgan a, deMorgan b)
|   deMorgan (Nao(a))           = Nao(deMorgan a)
```

```
|   deMorgan (x) = x;

(*  Distributividade                                    *)
(*  (A E (B ou C)   <=>   (A E B) ou (A E C)        *)

fun distrib1 (E(a,Ou(b,c)))  = Ou(E(distrib1 a,distrib1 b),
                                  E(distrib1 a,distrib1 c))
|   distrib1 (E(Ou(a,b),c))  = Ou(E(distrib1 a,distrib1 c),
                                  E(distrib1 b,distrib1 c))
|   distrib1 (Ou(a,b))       = Ou(distrib1 a,distrib1 b)
|   distrib1 (E(a,b))        = E(distrib1 a,distrib1 b)
|   distrib1 (x)             = x;

fun distrib x =
      let val f = distrib1 x
        in if (x = f) then x else distrib f
      end;

(* Transforma uma formula proposicional para a forma normal disjuntiva *)

val normal_disj = distrib o deMorgan o implica;

(* Imprime uma FORMULA *)

fun pprint (Prop(x,y))    = x ^ ":" ^ y
|   pprint (E(l,r))    = "(" ^ pprint l ^ "&&" ^ pprint r ^ ")"
|   pprint (Ou(l,r))   = "(" ^ pprint l ^ "||" ^ pprint r ^ ")"
|   pprint (Impl(l,r)) = "(" ^ pprint l ^ "=>" ^ pprint r ^ ")"
|   pprint (Nao(r))    = "~" ^ pprint r;
```

```
(* ------------------------------------------------------------------------ *)
(* As funções a seguir transformam uma FORMULA, na forma normal disjuntiva *)
(* em uma matriz de Bibel positiva.                                         *)
(* ------------------------------------------------------------------------*)

fun matriz_coluna (E(E(a,b),Prop(c1,c2)))       = (c1,c2,"")::
                                                  (matriz_coluna (E(a,b)))
|   matriz_coluna (E(E(a,b),Nao(Prop(c1,c2)))) = (c1,c2,"~")::
                                                  (matriz_coluna (E(a,b)))
|   matriz_coluna (E(Prop(a1,a2),E(b,c)))       = (a1,a2,"")::
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(Nao(Prop(a1,a2)),E(b,c))) = (a1,a2,"~")::
                                                  (matriz_coluna (E(b,c)))
|   matriz_coluna (E(E(a,b),E(c,d)))            = (matriz_coluna (E(a,b))) @
                                                  (matriz_coluna (E(c,d)))
|   matriz_coluna (E(Prop(a1,a2),Prop(b1,b2))) = (b1,b2,"")::(a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Prop(b1,b2))) = (b1,b2,"")::
                                                  (a1,a2,"~")::nil
|   matriz_coluna (E(Prop(a1,a2),Nao(Prop(b1,b2)))) = (b1,b2,"~")::
                                                  (a1,a2,"")::nil
|   matriz_coluna (E(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = (b1,b2,"~")::
                                                  (a1,a2,"~")::nil
|   matriz_coluna (Prop(a1,a2))                 = (a1,a2,"")::nil
|   matriz_coluna (Nao(Prop(a1,a2)))            = (a1,a2,"~")::nil;

fun matriz (Ou(Ou(a,b),Prop(c1,c2)))      = ((c1,c2,"")::nil)::
                                            matriz (Ou(a,b))
|   matriz (Ou(Prop(a1,a2),Ou(b,c)))      = ((a1,a2,"")::nil)::
```

```
                                          matriz (Ou(b,c))
|  matriz (Ou(Ou(a,b),Nao(Prop(c1,c2))))= ((c1,c2,"~")::nil)::
                                          matriz (Ou(a,b))
|  matriz (Ou(Nao(Prop(a1,a2)),Ou(b,c)))= ((a1,a2,"~")::nil)::
                                          matriz (Ou(b,c))
|  matriz (Ou(Ou(a,b),Ou(c,d)))          = (matriz (Ou(a,b))) @
                                          (matriz (Ou(c,d)))
|  matriz (Ou(Ou(a,b),E(c,d)))           = (matriz_coluna (E(c,d)))::
                                          matriz (Ou(a,b))
|  matriz (Ou(E(a,b),Ou(c,d)))           = (matriz_coluna (E(a,b)))::
                                          matriz (Ou(c,d))
|  matriz (Ou(E(a,b),Prop(c1,c2)))       = ((c1,c2,"")::nil)::
                                          (matriz_coluna (E(a,b)))::nil
|  matriz (Ou(E(a,b),Nao(Prop(c1,c2)))) = ((c1,c2,"~")::nil)::
                                          (matriz_coluna (E(a,b)))::nil
|  matriz (Ou(Prop(a1,a2),E(b,c)))       = (matriz_coluna (E(b,c)))::
                                          ((a1,a2,"")::nil)::nil
|  matriz (Ou(Nao(Prop(a1,a2)),E(b,c))) = (matriz_coluna (E(b,c)))::
                                          ((a1,a2,"~")::nil)::nil
|  matriz (Ou(E(a,b),E(c,d)))            = (matriz_coluna (E(c,d)))::
                                          (matriz_coluna (E(a,b)))::nil
|  matriz (Ou(Prop(a1,a2),Prop(b1,b2))) = ((b1,b2,"")::nil)::
                                          ((a1,a2,"")::nil)::nil
|  matriz (Ou(Nao(Prop(a1,a2)),Prop(b1,b2))) = ((b1,b2,"")::nil)::
                                          ((a1,a2,"~")::nil)::nil
|  matriz (Ou(Prop(a1,a2),Nao(Prop(b1,b2)))) = ((b1,b2,"~")::nil)::
                                          ((a1,a2,"")::nil)::nil
|  matriz (Ou(Nao(Prop(a1,a2)),Nao(Prop(b1,b2)))) = ((b1,b2,"~")::nil)::
                                          ((a1,a2,"~")::nil)::nil
```

```
|   matriz a                              = (matriz_coluna a)::nil;

(* ------------------------------------------------------------------- *)
(* As funções definidas a seguir transformam uma matriz positiva de Bibel *)
(* em uma matriz com todos os "PATHS"                                  *)
(* ------------------------------------------------------------------- *)

fun path1 ((_::nil)::yss) (zs::zss) = zs:: path1 yss zss
|   path1 ((_::xs)::yss) _ = xs :: yss
|   path1 _ _ = nil;

(* A função path2 recebe uma lista de listas e retorna uma lista formada
   pelos elementos da cabeça de cada sublista *)

fun path2 (nil) = nil
|   path2 ((x::_)::yss) = x:: path2 yss;

fun path3 (_) (_) 0 = nil
|   path3 (xss) (yss) n =
       let val aux = path1 xss yss
           in path2 aux ::  path3 aux yss (n-1)
       end;

(* A função qtde_paths calcula a quantidade de paths de uma lista de listas*)

fun qtde_paths nil = 1
|   qtde_paths (xs::xss) = length (xs) * qtde_paths (xss);

(* Recebe com parâmetro uma lista de listas(Matriz) e retorna uma lista de*)
```

```
(* listas(Matriz) com todos os paths *)

fun paths (xss) =
       let val qtde = (qtde_paths xss) - 1
         in path2 xss :: path3 xss xss qtde
       end;

(* ------------------------------------------------------------------ *)
(* As funções definidas a seguir tem o objetivo de verificar se todos  *)
(* os paths são complementares                                         *)
(* ------------------------------------------------------------------ *)

fun separa1 nil = nil
|   separa1 ((a,b,c)::cs) = if (c = "") andalso (b = "1")
                            then (a,"t",""):::(a,"f",""):: separa1 cs
                            else (a,b,c):: separa1 cs;

fun separa nil = nil
|   separa (xs::xss) = separa1 xs :: separa xss;

fun segundo (_,x) = x;

(* A função membro retorna true se o primeiro parâmetro é membro da *)
(* lista passada como segundo parâmetro                              *)

fun membro (a,nil) = false
|   membro (a,x::xs) = a = x orelse membro (a,xs);

fun complementar1 ((a,b,c)::nil)          = false
```

```
|   complementar1 ((a,b,c)::(d,e,f)::ds) = if (a = d) andalso (f = "")
                                            andalso (c = "~")
                                            andalso (maior_igual
                                                    reticulado_four (b,e))
                                           then true
                                           else if (a = d) andalso (f = "~")
                                             andalso (c = "")
                                             andalso (maior_igual
                                                    reticulado_four (e,b))
                                          then true
                                          else complementar1 ((a,b,c)::ds);

fun complementar ((a,b,c)::nil) = false
|   complementar ((a,b,c)::xs)  = if (complementar1 ((a,b,c)::xs))
                                  then true
                                  else complementar xs;

fun tudo_complementar nil  = true
|   tudo_complementar [[]] = false
|   tudo_complementar ((x::xs)::yss) = if complementar (x::xs) then
                                         tudo_complementar yss
                                       else false;

fun matriz_infimo (((_,b,c)::nil)::nil) = if (b = "0") andalso (c = "")
                                          then true
                                          else false
|   matriz_infimo _              = false;

fun spanning x = if (tudo_complementar x)
```

```
                then true
                else if (matriz_infimo x)
                     then true
                     else false;


(*  A função remove_repetido1, remove os elementos repetidos de uma lista *)

fun remove_repetido1 nil = nil
|   remove_repetido1 (x::xs) = if membro(x,xs) then remove_repetido1 xs
                               else x:: remove_repetido1 xs;

(* A função remove_repetido, remove os elementos repetidos de todas as *)
(* sublistas de uma lista de listas *)

fun remove_repetido nil = nil
|   remove_repetido [[]] = [[]]
|   remove_repetido ((x::xs)::yss) = remove_repetido1 (x::xs)::
                                     remove_repetido yss;

(* A função remove_tautologia, remove as sublistas de uma lista que *)
(* possuem elementos complementares *)

fun remove_tautologia nil = nil
|   remove_tautologia ((x::xs)::yss) = if complementar (x::xs) then
                                          remove_tautologia yss
                                       else (x::xs):: remove_tautologia yss;

(* A função valido é a função principal deste programa, retorna true se a *)
```

```
(* formula é valida *)

val valido = spanning o paths o remove_tautologia o remove_repetido o
                                    separa o matriz o normal_disj o ast;
end; (* Bibel *)
```

A.4 Primeira versão da Implementação

A primeira versão da implementação prova teoremas da lógica clássica através do método das conexões de Bibel.

```
(* Implementação do método das conexões de bibel para a lógica *)
(* proposicional clássica. *)

(* Linguagem : Standard ML *)
(* Autor     : Emerson Faria Nobre - Junho/2000 *)

(* ------------------------------------------------------------------ *)
(* Definicao do Tipo FORMULA *)
(* O tipo FORMULA é utilizado para representar uma formula proposicional *)
(* como uma árvore de sintaxe abstrata(AST) *)
(* ------------------------------------------------------------------ *)

datatype FORMULA = E of FORMULA*FORMULA        (* conjuncao    &&        *)
|                  Ou of FORMULA*FORMULA       (* disjuncao    ||        *)
|                  Impl of FORMULA*FORMULA     (* implicacao   =>        *)
|                  Nao of FORMULA              (* negacao      ~         *)
|                  Prop of string;             (* proposicao   A..z|0..9 *)
```

```
(* ------------------------------------------------------------------------*)
(* As funções definidas a seguir são utilizadas para identificar os Tokens*)
(* da string de entrada *)
(* obs: São funções auxiliares, utilizadas nas funções que constroem a AST*)
(* ------------------------------------------------------------------------*)

(* A função "digito" retorna true se o caracter passado como parâmetro *)
(* é um número entre 0 e 9*)

fun digito c = c >= #"0" andalso c <= #"9";

(* A função "minusculo" retorna true se o caracter passado como parâmetro *)
(* é uma letra minúscula *)

fun minusculo c = c >= #"a" andalso c <= #"z";

(* A função "maiusculo" retorna true se o caracter passado como parâmetro *)
(* é uma letra maiúscula *)

fun maiusculo c = c >= #"A" andalso c <= #"Z";

(* A função "letra" retorna true se o caracter passado como parâmetro *)
(* é uma letra minúscula ou maiúscula *)

fun letra c =  minusculo c orelse maiusculo c;

(* A função "igual" retorna true se os dois parâmetros são iguais *)

fun igual x y = x = y;
```

```
(* A função "leprop" recebe como parâmetro uma string, e se o inicio da *)
(*string for uma proposicao,então a função retorna o token desta proposição*)
(*obs: Uma proposição é formada por letras minúsculas, maiúsculas e números*)

fun leProp (id, nil) = (Prop id, nil)
|   leProp (id, c::cs) =
        if letra c orelse digito c
        then leProp(id ^ (str c), cs)
        else (Prop id, c::cs);

exception sintaxerror;

fun remove_cabeca p (x::xs) = if p x then xs else raise sintaxerror;

(* ------------------------------------------------------------------------ *)
(* As Funções definidas a seguir constroem uma Árvore de Sintaxe Abstrata *)
(* a partir de uma string *)
(* ------------------------------------------------------------------------ *)

fun expressao cs =  expressao'(termo cs)
and expressao' (l, nil) = (l, nil)
|   expressao' (l, c::nil) = (l, c::nil)
|   expressao' (l,b::c::cs) =
      if "=>" = ((str b) ^ (str c))
      then let val (r, cs) = termo cs
           in expressao' (Impl(l,r),cs)
           end
      else (l,b::c::cs)
```

```
and termo cs = termo'(fator cs)
and termo' (l, nil) = (l, nil)
|   termo' (l,c::nil) = (l,c::nil)
|   termo' (l,b::c::cs) =
      if "&&" = ((str b) ^ (str c))
      then let val (r, cs) = fator cs
           in termo' (E(l,r),cs) end
      else if "||" = ((str b) ^ (str c))
           then let val (r, cs) = fator cs
                in termo' (Ou(l,r),cs) end
           else (l,b::c::cs)
and fator nil = raise sintaxerror
|   fator (c::cs) =
    if c = #"("
    then let val (t1, cs) = expressao cs
         in (t1, remove_cabeca (igual #")") cs)
         end
    else if c = #"~"
         then let val (t1, cs) = fator cs
              in (Nao(t1),cs)
              end
         else leProp(str c,cs);


(* A função "primeiro" retorna o primeiro elemento de uma tupla *)

fun primeiro (x, _) = x;

(* A função "remove_espacos", remove os espaços em branco de uma *)
```

```
(* lista de caracteres *)

fun remove_espacos nil = nil
|   remove_espacos (#" "::cs) = remove_espacos cs
|   remove_espacos (c::cs) = c :: remove_espacos cs;

(* A função "ast" recebe uma string como parâmetro e retorna uma *)
(* FORMULA(árvore de sintaxe abstrata) *)

val ast = primeiro o expressao o remove_espacos o explode;

(* ------------------------------------------------------------ *)
(* As funcões definidas a seguir, transformam uma FORMULA       *)
(* proposicional em sua forma normal disjuntiva                 *)
(* ------------------------------------------------------------ *)

(*  Implicacao   *)
(*  (A Impl B)   <=>   (Nao(A) ou B)  *)

fun implica (Impl(a, b)) = Ou(Nao(implica a), implica b)
|   implica (Ou(a, b))   = Ou(implica a, implica b)
|   implica (E(a, b))    = E(implica a, implica b)
|   implica (Nao(a))     = Nao(implica a)
|   implica (x)          = x;

(*  Leis de De Morgan                    *)
(*  Nao(A Ou B)   <=>   (Nao(A) E Nao(B))    *)
(*  Nao(A E B)    <=>   (Nao(A) Ou Nao(B))    *)
```

```
fun deMorgan (Nao(Nao(a)))      = deMorgan a                (* dupla negacao*)
|   deMorgan (Nao(Ou(a,b)))     = E(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Nao(E(a,b)))      = Ou(deMorgan (Nao(a)),deMorgan (Nao(b)))
|   deMorgan (Ou(a,b))          = Ou(deMorgan a, deMorgan b)
|   deMorgan (E(a,b))           = E(deMorgan a, deMorgan b)
|   deMorgan (Nao(a))           = Nao(deMorgan a)
|   deMorgan (x) = x;

(*  Distributividade                                    *)
(*  (A E (B ou C)   <=>   (A E B) ou (A E C)            *)

fun distrib1 (E(a,Ou(b,c))) = Ou(E(distrib1 a,distrib1 b),
                                 E(distrib1 a,distrib1 c))
|   distrib1 (E(Ou(a,b),c)) = Ou(E(distrib1 a,distrib1 c),
                                 E(distrib1 b,distrib1 c))
|   distrib1 (Ou(a,b))       = Ou(distrib1 a,distrib1 b)
|   distrib1 (E(a,b))        = E(distrib1 a,distrib1 b)
|   distrib1 (x)             = x;

fun distrib x =
       let val f = distrib1 x
         in if (x = f) then x else distrib f
       end;

(* Transforma uma formula proposicional para a forma normal disjuntiva *)

val normal_disj = distrib o deMorgan o implica;

(* Imprime uma FORMULA *)
```

```
fun pprint (Prop i)     = i
|   pprint (E(l,r))     = "(" ^ pprint l ^ "&&" ^ pprint r ^ ")"
|   pprint (Ou(l,r))    = "(" ^ pprint l ^ "||" ^ pprint r ^ ")"
|   pprint (Impl(l,r)) = "(" ^ pprint l ^ "=>" ^ pprint r ^ ")"
|   pprint (Nao(r))     = "~" ^ pprint r;

(* ------------------------------------------------------------------------*)
(* As funções a seguir transformam uma FORMULA, na forma normal disjuntiva,*)
(* em uma matriz de Bibel positiva                                         *)
(* ------------------------------------------------------------------------*)

fun matriz_coluna (E(E(a,b),Prop c))       = c::(matriz_coluna (E(a,b)))
|   matriz_coluna (E(E(a,b),Nao(Prop c))) = "~" ^ c::(matriz_coluna (E(a,b)))
|   matriz_coluna (E(Prop a,E(b,c)))       = a::(matriz_coluna (E(b,c)))
|   matriz_coluna (E(Nao(Prop a),E(b,c))) = "~" ^ a::(matriz_coluna (E(b,c)))
|   matriz_coluna (E(E(a,b),E(c,d)))       = (matriz_coluna (E(a,b))) @
                                             (matriz_coluna (E(c,d)))
|   matriz_coluna (E(Prop a,Prop b))       = b::a::nil
|   matriz_coluna (E(Nao(Prop a),Prop b)) = b::"~" ^ a::nil
|   matriz_coluna (E(Prop a,Nao(Prop b))) = "~" ^ b::a::nil
|   matriz_coluna (E(Nao(Prop a),Nao(Prop b))) = "~" ^ b::"~" ^ a::nil
|   matriz_coluna (Prop a)                 = a::nil
|   matriz_coluna (Nao(Prop a))            = "~" ^ a::nil;

fun matriz (Ou(Ou(a,b),Prop c))           = (c::nil)::matriz (Ou(a,b))
|   matriz (Ou(Prop a,Ou(b,c)))           = (a::nil)::matriz (Ou(b,c))
|   matriz (Ou(Ou(a,b),Nao(Prop c)))      = ("~" ^ c::nil):: matriz (Ou(a,b))
|   matriz (Ou(Nao(Prop a),Ou(b,c)))      = ("~" ^ a::nil):: matriz (Ou(b,c))
```

```
|  matriz (Ou(Ou(a,b),Ou(c,d)))          = (matriz (Ou(a,b))) @
                                            (matriz (Ou(c,d)))
|  matriz (Ou(Ou(a,b),E(c,d)))           = (matriz_coluna (E(c,d))):: 
                                             matriz (Ou(a,b))
|  matriz (Ou(E(a,b),Ou(c,d)))           = (matriz_coluna (E(a,b))):: 
                                             matriz (Ou(c,d))
|  matriz (Ou(E(a,b),Prop c))            = (c::nil)::
                                            (matriz_coluna (E(a,b)))::nil
|  matriz (Ou(E(a,b),Nao(Prop c)))       = ("~" ^ c::nil)::
                                            (matriz_coluna (E(a,b)))::nil
|  matriz (Ou(Prop a,E(b,c)))            = (matriz_coluna (E(b,c))):: 
                                            (a::nil)::nil
|  matriz (Ou(Nao(Prop a),E(b,c)))       = (matriz_coluna (E(b,c))):: 
                                            ("~" ^ a::nil)::nil
|  matriz (Ou(E(a,b),E(c,d)))            = (matriz_coluna (E(c,d))):: 
                                            (matriz_coluna (E(a,b)))::nil
|  matriz (Ou(Prop a,Prop b))            = (b::nil)::(a::nil)::nil
|  matriz (Ou(Nao(Prop a),Prop b))       = (b::nil)::("~" ^ a::nil)::nil
|  matriz (Ou(Prop a,Nao(Prop b)))       = ("~" ^ b::nil)::(a::nil)::nil
|  matriz (Ou(Nao(Prop a),Nao(Prop b))) = ("~" ^ b::nil):: 
                                            ("~" ^ a::nil)::nil
|  matriz a                              = (matriz_coluna a)::nil;

(* ------------------------------------------------------------------------*)
(* As funções definidas a seguir transformam uma matriz positiva de Bibel*)
(* em uma matriz com todos os PATHS                                       *)
(* ------------------------------------------------------------------------*)
```

```
fun path1 ((_::nil)::yss) (zs::zss) = zs:: path1 yss zss
|   path1 ((_::xs)::yss) _ = xs :: yss
|   path1 _ _ = nil;

(* A função path2 recebe uma lista de listas e retorna uma lista formada *)
(*   pelos elementos da cabeça de cada sublista *)

fun path2 (nil) = nil
|   path2 ((x::_)::yss) = x:: path2 yss;

fun path3 (_) (_) 0 = nil
|   path3 (xss) (yss) n =
       let val aux = path1 xss yss
           in path2 aux ::  path3 aux yss (n-1)
       end;

(* A função qtde_paths calcula a quantidade de paths de uma lista de listas*)

fun qtde_paths nil = 1
|   qtde_paths (xs::xss) = length (xs) * qtde_paths (xss);

(* Recebe com parâmetro uma lista de listas(Matriz) e retorna uma lista de*)
(* listas(Matriz) com todos os paths *)

fun paths (xss) =
       let val qtde = (qtde_paths xss) - 1
         in path2 xss :: path3 xss xss qtde
       end;
```

```
(* ------------------------------------------------------------------- *)
(* As funções definidas a seguir tem o objetivo de verificar se todos os  *)
(* paths são complementares                                            *)
(* ------------------------------------------------------------------- *)

(*   A função membro retorna true se o primeiro parâmetro é membro da lista*)
(* passada como segundo parâmetro  *)

fun membro (a,nil) = false
|   membro (a,x::xs) = a = x orelse membro (a,xs);

(*  A função remove_repetido1, remove os elementos repetidos de uma lista  *)

fun remove_repetido1 nil = nil
|   remove_repetido1 (x::xs) = if membro(x,xs) then remove_repetido1 xs
                               else x:: remove_repetido1 xs;

(* A função remove_repetido, remove os elementos repetidos de todas as *)
(*   sublistas de uma lista de listas                                   *)

fun remove_repetido nil = nil
|   remove_repetido [[]] = [[]]
|   remove_repetido ((x::xs)::yss) = remove_repetido1 (x::xs)::
                                     remove_repetido yss;

(* A função negativa retorna true se o primeiro caracter de uma string *)
(* é o til*)

fun negativa x = hd(explode x) = #"~";
```

```
(* A função inverte_prop inclui como primeiro caracter de uma string o Til,*)
(* ou remove caso este já seja o primeiro caracter, remove                 *)

fun inverte_prop x = if (negativa x) then implode(tl(explode x))
                     else implode(#"~"::(explode x));

(* A função complementar retorna true se existem dois elementos *)
(* (do tipo string em uma) lista que tem como única diferença   *)
(* que um começa com til e outro não. *)
(* Exemplo:  ["~A","B","A"]. Retorna true devido aos elementos A e ~A. *)

fun complementar nil = false
|   complementar(x::xs) = if membro(inverte_prop x,xs) then true
                          else complementar xs;

(* A função remove_tautologia, remove as sublistas de uma lista que *)
(* possuem elementos complementares *)

fun remove_tautologia nil = nil
|   remove_tautologia ((x::xs)::yss) = if complementar (x::xs) then
                                              remove_tautologia yss
                                       else (x::xs):: remove_tautologia yss;

(* A função tudo_complementar, retorna true se todas sublistas de uma lista*)
(* são complementares *)

fun tudo_complementar nil  = true
|   tudo_complementar [[]] = false
```

```
|   tudo_complementar ((x::xs)::yss) = if complementar (x::xs) then
                                            tudo_complementar yss
                                       else false;

(* A função valido é a função principal deste programa, retorna true se a *)
(* formula é válida de acordo com a definição da "Verdade de Tarski"      *)

val valido = tudo_complementar o paths o remove_tautologia o
                          remove_repetido o matriz o normal_disj o ast;
```