Universidade Federal de Pernambuco

Centro de Informática
Graduação em Ciência da Computação

2007.2

Estudo e estado da arte dos provadores automáticos de teoremas

Trabalho de Graduação

Aluno: Everton Guerra Marques – egm2@cin.ufpe.br

Recife, 22 de Janeiro de 2007

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

2007.2

Estudo e estado da arte dos provadores automáticos de teoremas

Trabalho de Graduação

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Departamento de Sistemas de Computação da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profª. Doutora Anjolina Grisi de Oliveira (ago@cin.ufpe.br)

Recife, 22 de janeiro de 2008

Assinaturas

Este trabalho de graduação é resultado dos esforços do aluno Everton Guerra Marques, sob a orientação da professora Anjolina Grisi de Oliveira. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste trabalho de graduação.

_____

Doutora Anjolina Grisi de Oliveira
Orientadora

_____

Everton Guerra Marques
Orientando

Resumo

A lógica possui diversas áreas, dentre elas, a área da teoria da prova que estuda os conceitos de provas formais e os fundamentos relacionados. A teoria da prova pode ser utilizada em diversas aplicações, nas mais diversas áreas, tais como: matemática, filosofia e computação.

Dentre os vários temas relacionados à computação, destaca-se a prova automática de teoremas, objeto de estudo do presente trabalho de graduação. Os provadores são formas automatizadas que tentam provar sistemas em diversas lógicas e utilizando-se de diversos métodos e formas para isso.

Além de apresentar os fundamentos teóricos para a construção de provadores automáticos de teoremas, este trabalho mostra o estado da arte dos provadores, ou seja, estuda alguns dos principais provadores automáticos de teoremas existentes atualmente e qual o estado atual da área.

Palavras-Chave: Lógica, Teoria da Prova, Provadores Automáticos de Teoremas, Estado da Arte

Abstract

The logic has several areas, among them, the area of the theory of proof that studies the concepts of evidence and the formal pleas related. The theory of proof can be used in various applications, in various areas such as: mathematics, philosophy and computation.

Among the various issues related to computation, highlights the proof automatic of theorems, object of study of this work of graduation. The provers are automated forms that trying to prove systems in various logics and using various methods and ways to do it.

In addition to presenting the theoretical foundations for the construction of automatic theorem provers, this work shows the state of the art of provers, that is, studying some of the main automatic theorem provers currently available and what the current status of the area.

Keywords: Logic, Theory of Proof, Automatic Theorem Provers, State of the Art

Agradecimentos

Primeiramente, gostaria de agradecer a Deus que meu deu o Dom da vida, por ter me dado plenas condições de estudar, e pela oportunidade de passar no vestibular e com isso estudar em uma grande e tão bem conceituada universidade como a UFPE. Além disso, Ele esteve junto comigo em toda a caminhada, me apoiando e me dando forças constantes para batalhar e chegar a esse ponto.

Além disso, gostaria de agradecer a Ele por todas as oportunidades que me proporcionou, pelas experiências que eu pude viver, pelas pessoas e profissionais que eu pude conhecer, e principalmente pelo seu Amor eterno.

Gostaria de agradecer a minha família como um todo, com agradecimentos especiais aos meus pais, que me deram uma base educacional sensacional, uma estrutura familiar sólida, valores e princípios que levarei pro resto da minha vida, a noção do que é certo e errado, o apoio e preocupações constantes entre muitas outras coisas que eles me proporcionaram. Também gostaria de agradecer a minha irmã que sempre me apoiou, sempre esteve ao meu lado nos momentos bons e nos momentos difíceis, e que sempre me escutou quando precisei falar.

Gostaria de agradecer também a todos os meus amigos, aos que me apoiaram antes do vestibular, que me deram força para que fosse aprovado em um vestibular tão concorrido quanto o desta universidade. Além disso, gostaria de agradecer aos amigos que fiz na universidade, amigos esses que considero como irmãos, irmãos estes que me ajudaram a chegar até aqui, que me deram forças, que dividiram as alegrias e tristezas comigo, que viraram noites e noites fazendo projetos malucos e que claro, estavam presentes em comemorações e festas das mais diversas e nos mais diversos lugares.

Além disso, gostaria de agradecer aos amigos que tenho de fora da faculdade, que sempre acompanharam a minha luta e sempre me deram palavras de incentivos nos momentos difíceis e sempre se divertiram junto comigo nos momentos felizes.

Também gostaria de agradecer a minha namorada “Cacá”, que esteve junto comigo nos últimos três períodos da universidade, sempre ao meu lado, me dando força e me ajudando sempre, me apoiando constantemente, pessoa que sei que pude e posso contar sempre que precisar.

Agradeço a minha orientadora, a professora Anjolina Grisi de Oliveira, pela oportunidade que me deu ao me orientar, pelo apoio desde o início da minha jornada nessa universidade, onde me aceitou como monitor da disciplina Matemática Discreta e que sempre me apoiou no decorrer do curso. Além disso, tenho que agradecer pela ajuda e orientações que foram de grande valia para o desenvolvimento desse Trabalho de Graduação.

Também não poderia deixar de agradecer ao professor Ruy José Guerra Barreto de Queiroz que me deu um grande apoio, tanto nesse trabalho, quanto na vida acadêmica como

graduando, inclusive me aceitando também como monitor da disciplina Matemática Discreta e, além disso, aceitando me orientar no meu mestrado.

Agradeço, enfim, a todas as pessoas que de alguma forma contribuíram e se envolveram na realização deste trabalho e que participaram da minha vida acadêmica, me dando forças e apoio até hoje.

Índice

Lista de Ilustrações

Lista de Tabelas

Introdução

As pesquisas direcionadas à área de teoria da prova buscam estudar os conceitos de provas formais e os fundamentos relacionados. O desenvolvimento de sistemas dedutivos e de técnicas como a normalização de provas são exemplos de relevantes investigações na área.

Um outro tópico de investigação é a construção de provas formais, que pode de forma grosseira, ser classificada da seguinte maneira [1]:

- **Prova dirigida por humanos**: Neste tipo de prova, ela é feita no estilo de uma prova matemática, geralmente manuscrita, de maneira informal e usando linguagem natural. Geralmente é considerada uma boa prova aquela que é legível e é entendida por outros leitores humanos.

Os estudiosos da área mostram que a ambigüidade inerente na linguagem natural permite que erros sejam dificilmente encontradas em tal tipo de prova. Freqüentemente, erros sutis podem estar presentes em detalhes de baixo nível, tipicamente negligenciados por tais provas. Além disso, o trabalho para produzir este tipo de prova requer um alto nível de sofisticação e especialização matemática.

- Prova automatizada: Na área da computação, há um grande interesse por parte dos estudiosos da área em produzir provas de sistemas por meios automatizados. Neste tipo de prova, o sistema tenta produzir uma prova formal, sendo dada uma descrição do sistema, um conjunto de regras de inferência e um conjunto de axiomas lógicos. Geralmente, este tipo de prova requer orientação sobre quais as propriedades são desejáveis para prosseguir. E, costuma-se chamar esse tipo de sistema de prova automatizada, de provador automático de teoremas.

Desta maneira, o uso de provadores automáticos de teoremas é bastante difundido na área de construção de provas formais, e eles podem ser em: lógica de primeira ordem, lógica clássica proposicional, lógica modal, lógica intuicionista, entre outras, e eles podem utilizar-se de métodos tais como: resolução, tableaux, cálculo de seqüentes, anéis booleanos, dedução natural, etc..

Também podem ser utilizados para verificar se um sistema particular apresenta uma propriedade específica, o que é uma atividade complexa, que envolve o tratamento das especificações, muitos passos de inferências e a gerência de muitas informações.

Com isso, pode-se resumir o conceito de prova automática de teoremas da seguinte maneira: um programa computacional que mostra se a conjectura é uma conseqüência lógica de um conjunto de sentenças (os axiomas e hipóteses). A linguagem utilizada pelos programas de provas automáticas de teoremas deve ser formal de modo a não permitir ambigüidade.

E a verificação de uma sentença produzida por este programa computacional é conhecida como prova. Esta prova descreve uma seqüência de conseqüências lógicas que validam uma conjectura. Os passos seguidos pelo programa acima citado podem ser entendidos e seguidos por outras pessoas ou programas [2].

1.1 Objetivos

Levando em conta o contexto acima citado, o objetivo deste Trabalho de Graduação consiste em estudar os fundamentos teóricos para a construção de provadores automáticos de teoremas, pesquisar sobre os mesmos e analisar alguns e suas aplicações, para então esboçar uma proposta de pesquisa na área.

1.2 Organização do documento

Além do capítulo inicial, este trabalho está organizado em mais quatro capítulos:

Capítulo 2: Contêm um estudo aprofundado dos fundamentos teóricos para a construção de provadores automáticos de teoremas, principalmente o teorema de Herbrand, fundamental para concepção dos provadores automáticos de teoremas.

Capítulo 3: Inclui um estudo aprofundado sobre alguns provadores de lógica de primeira ordem.

Capítulo 4: Apresenta um estudo sobre alguns provadores de outras lógicas.

Capítulo 5: Descreve as conclusões obtidas e as considerações finais, as dificuldades encontradas, e possíveis trabalhos futuros.

Por fim, seguem os apêndices e as referências bibliográficas utilizadas no desenvolvimento deste trabalho.

2. Provadores automáticos de teoremas: fundamentos teóricos

Desde que surgiu o primeiro computador moderno, a tecnologia computacional se desenvolveu rapidamente. Hoje em dia, não se vê computadores sendo utilizados apenas para resolver problemas de difícil computação, como realizar rapidamente uma transformada de Fourier, ou inverter uma matriz de alta dimensão, mas também sendo capazes de realizar tarefas que podem ser chamadas de inteligentes se feitas pelo ser humano. Algumas dessas tarefas podem ser: escrever programas, responder perguntas e provar teoremas [3].

A segunda metade dos anos 60 foi fenomenal no campo da inteligência artificial pelo crescimento do interesse na prova automática de teoremas. O interesse intenso generalizado em provadores automáticos de teoremas não foi causado apenas pela crescente sensibilização de que a habilidade de realizar deduções lógicas é uma parte integral da inteligência humana, mas foi talvez mais um resultado do status das técnicas mecânicas de provas de teoremas desenvolvidas nos anos 60.

A base dos provadores automáticos de teoremas foi desenvolvida por Herbrand em 1930. Seu método era impraticável de se aplicar até a invenção do computador digital. Foi após o marcante artigo de J.A.Robinson em 1965, junto com o desenvolvimento do princípio da resolução, que foram tomadas medidas importantes para se conseguir realísticos provadores de teoremas implementados em computadores. Desde 1965, muitos refinamentos do princípio da resolução foram feitos.

Paralelamente ao progresso nas melhorias das técnicas de provas automáticas de teoremas ocorreu o progresso na aplicação de técnicas de provas de teoremas a vários problemas de inteligência artificial. Elas foram aplicadas inicialmente para responder questões dedutivas e posteriormente em resolver problemas, síntese de programas, análise de programas e muitas outras [3].

Desta forma, segue abaixo um detalhamento maior do teorema de Herbrand, onde Herbrand basicamente afirma que em uma lógica de primeira-ordem formal, se uma fórmula na forma prenex é satisfatível, então ela é satisfatível em um modelo que interpreta símbolos lógicos usando termos da própria lógica.

Por definição, uma fórmula válida é uma fórmula que é verdade sobre todas as interpretações. Herbrand desenvolveu um algoritmo para encontrar uma interpretação que pode falsificar uma dada fórmula. No entanto, se a dada fórmula mantém-se válida, não pode existir nenhuma interpretação e seu algoritmo irá parar depois de um número finito de tentativas. O método de Herbrand é a base para os mais modernos procedimentos de prova automática.

Desta forma, ao invés de provar se uma fórmula é válida, o algoritmo prova que a negação da fórmula é inconsistente. É apenas uma forma de conveniência. Não há perda de generalidade em usar procedimentos de refutação. No entanto, esses procedimentos de refutação são aplicados a uma “forma padrão” de uma fórmula. Essa forma padrão foi

introduzida por Davis e Putnam. O que eles fizeram basicamente foi explorar as seguintes idéias [3]:

1. Uma fórmula de lógica de primeira ordem pode ser transformada na forma normal prenex (ver apêndice 1) onde a matriz não contém nenhum quantificador e o prefixo é uma seqüência de quantificadores.
2. A matriz, desde que não contenha quantificadores, pode ser transformada em uma forma normal conjuntiva (vide apêndice 1).
3. Sem afetar a propriedade de consistência, os quantificadores existenciais ($\exists x$) no prefixo podem ser eliminados usando as funções de Skolem.

Resumindo, dado uma fórmula na forma prenex, o processo de Skolemização resulta da aplicação repetida da seguinte transformação que remove o primeiro dos quantificadores existenciais [4]:

$$\forall x \exists y.R(x,y) \iff \exists f \forall x.R(x,f(x))$$

onde f é um novo símbolo de função.

A seguir são dadas algumas definições para que se tenha um melhor entendimento do trabalho:

***Definição 1**:* Em lógica matemática, um átomo ou fórmula atômica é uma fórmula que não pode ser dividida em sub-fórmulas, ou seja, uma fórmula sem conectivos que representa uma proposição.

***Definição 2**:* Um literal em lógica matemática é um átomo ou a negação de um átomo.

***Definição 3**:* Uma cláusula é uma disjunção de literais.

***Definição 4**:* Uma constante é um valor fixo que pode ou não ser especificado. Durante o decorrer do texto serão utilizadas as letras a, b, c para denotar constantes.

***Definição 5**:* Um termo em um dado sistema lógico é o nome para um objeto do universo de discurso (Indica o conjunto relevante de entidades as quais os quantificadores se referem). Um termo é definido recursivamente como:

- Uma constante é um termo;
- Uma variável é um termo;
- Um símbolo de função aplicada a um termo é um termo.

***Definição 6**:* Predicados são nomes que expressam propriedades ou relações a cercas de objetos de um determinado domínio.

Então, Seja S um dado conjunto de fórmulas bem formadas e suponha que G seja uma conseqüência lógica de S (ou seja, que G segue logicamente de S), e relembrando um pouco mais de lógica, mais especificamente da definição de conseqüência lógica, se I é um modelo de S, I é um modelo de G (em outras palavras, toda interpretação que satisfaz S, satisfaz G também). Nenhum dos modelos de S pode ser um modelo de ¬G e, portanto, nenhum modelo

pode satisfazer S $\wedge$ {¬G}. Desta forma, S $\wedge$ {¬G} deve ser insatisfatível. Portanto, se G é conseqüência lógica de S, S $\wedge$ {¬G} é insatisfatível. Sabe-se que [5]:

***Definição 7**:* Um conjunto de cláusulas é insatisfatível (ou inconsistente) se e somente se for falso sob todas as interpretações, em todos os domínios [5].

Dada a definição acima, pode-se perceber que é praticamente impossível considerar todas as interpretações sobre todos os domínios, e que seria uma boa possibilidade a de se fixar em um domínio especial H, onde S é inconsistente se e somente se S é falso sob todas as interpretações sobre este domínio. Felizmente, existe esse domínio que é chamado de universo de Herbrand de S, definido como:

***Definição 8**:* Seja S um conjunto de cláusulas e H_0 o conjunto de constantes que aparecem em S:

- se nenhuma constante aparece em S, então seja $H_0 = \{a\}$
- para i = 0,1,…., seja

  $$H_{i+1} = H_i \cup T$$

  onde T é o conjunto de todos os termos da forma $f_n(t_1,t_2,\ldots,t_n)$ para todos os símbolos de funções f_n ocorrendo em S, onde t_j, para j=1,…,n são elementos de H_i.

Cada H_i é chamado de conjunto constante de nível i de S e H_∞ (ou simplesmente H) é chamado de *Universo de Herbrand de S*.

Na seqüência, por expressão pode-se entender um termo, um conjunto de termos, um átomo, um conjunto de átomos, um literal, uma cláusula, ou um conjunto de cláusulas. Quando nenhuma variável aparece em uma expressão, ela é chamada de expressão básica (“ground”, do inglês).

Assim, o Universo de Herbrand, coincide com o conjunto de termos básicos que são gerados a partir dos símbolos de constantes e dos símbolos funcionais que aparecem em S. Podem acontecer três situações [5]:

1. Se S contém ambos, constantes e funções (símbolos funcionais), então o Universo de Herbrand será sempre um conjunto infinito enumerável.

2. Se S contém apenas constantes, o Universo de Herbrand será o conjunto finito daquelas constantes.

3. Se S não contém constantes, uma constante, por exemplo *a*, é escolhida arbitrariamente. Dependendo da presença ou ausência de símbolos funcionais em S, o Universo de Herbrand será infinito ou não, respectivamente.

Exemplo 1: Seja S= {¬p(f(X),g(X),h(Y)) $\vee$ q(X),r(Y)}

Desde que não exista constante em S, seja então $H_0 = \{a\}$. Assim

$H_1 = H_0 \cup \{f(a),g(a),h(a)\}$

$H_2 = H_1 \cup \{f(f(a)), f(g(a)), f(h(a)), g(f(a)), g(g(a)), g(h(a)), h(f(a)), h(g(a)), h(h(a))\}$

$H_3 = H_2 \cup \{ f(f(f(a))), f(f(g(a))), f(f(h(a))), f(g(f(a))), f(g(g(a))), f(g(h(a))), f(h(f(a))), f(h(g(a))), f(h(h(a))), g(f(f(a))), g(f(g(a))), g(f(h(a))), g(g(f(a))), g(g(g(a))), g(g(h(a))), g(h(f(a))), g(h(g(a))), g(h(h(a))), h(f(f(a))), h(f(g(a))), h(f(h(a))), h(g(f(a))), h(g(g(a))), h(g(h(a))), h(h(f(a))), h(h(g(a))), h(h(h(a)))\}$

…..

H_∞ = {a, f(a), g(a), h(a), f(f(a)), f(g(a)), f(h(a)), g(f(a)), g(g(a)), g(h(a)), h(f(a)), h(g(a)), h(h(a)), f(f(f(a))), f(f(g(a))), f(f(h(a))), f(g(f(a))), f(g(g(a))), f(g(h(a))), f(h(f(a))), f(h(g(a))), f(h(h(a))), g(f(f(a))), g(f(g(a))), g(f(h(a))), g(g(f(a))), g(g(g(a))), g(g(h(a))), g(h(f(a))), g(h(g(a))), g(h(h(a))), h(f(f(a))), h(f(g(a))), h(f(h(a))), h(g(f(a))), h(g(g(a))), h(g(h(a))), h(h(f(a))), h(h(g(a))), h(h(h(a))), …}

Exemplo 2: Seja S= {p(a,b,f(X),g(Y),c)}

Uma vez que existe três constantes – a, b e c – em S, então:

H_0 = {a, b, c}

$H_1 = H_0 \cup$ {f(a), g(a), f(b), g(b), f(c), g(c)}

$H_2 = H_1 \cup$ {f(f(a)), f(g(a)), f(f(b)), f(g(b)), f(f(c)), f(g(c)), g(f(a)), g(g(a)), g(f(b)), g(g(b)), g(f(c)), g(g(c))}

….

H_∞ = {a, b, c, f(a), g(a), f(b), g(b), f(c), g(c), f(f(a)), …}

Definição 9: Seja S um conjunto de cláusulas e seja H o Universo de Herbrand de S. O conjunto dos átomos básicos da forma $P^n(t_1,t_2,\ldots,t_n)$ para todos os predicados P^n ocorrendo em S, onde t_1, t_2, …, t_n são elementos do Universo de Herbrand de S, é chamado de *Base de Herbrand*, ou conjunto atômico de S e é denotado por B_S [5].

Exemplo 3: Seja S= {p(X) $\vee$ q(X), r(f(X))}

Então, B_S = {p(a), q(a), r(a), p(f(a)), q(f(a)), r(f(a)),…}

Exemplo 4: Seja S= {$\neg$p(X,a,Y) $\vee$ $\neg$p(b,X,Z) $\vee$ q(Y,Z)}

Desde que S não tem símbolo funcional, o Universo de Herbrand de S é $H\infty$ = {a,b}. Então,

B_S = {p(a,a,a), p(a,a,b), p(a,b,a), p(a,b,b), p(b,a,a), p(b,a,b), p(b,b,a), p(b,b,b), q(a,a), q(a,), q(b,a), q(b,b)}

Definição 10: *Uma instância básica de uma cláusula C de um conjunto S de cláusulas* é uma cláusula obtida pela substituição das variáveis em C por elementos do Universo de Herbrand de S [5].

Exemplo 5: Seja S= {p(X),q(f(Y)) $\vee$ r(Y)} , $H\infty$ = {a, f(a),f(f(a)),….}

A tabela 1 mostra algumas das instâncias básicas das duas cláusulas C1 e C2, de S.

C1: p(X)	C2: q(f(Y)) $\vee$ r(Y)
p(a)	q(f(a)) $\vee$ r(a)
p(f(a))	q(f(f(a))) $\vee$ r(f(a))
p(f(f(a)))	q(f(f(f(a)))) $\vee$ r(f(f(a)))
……..	…….

Tabela 1 – Instâncias básicas das cláusulas C1 e C2

***Definição 11*:** Seja S um conjunto de cláusulas. Uma *instanciação básica de S* é o conjunto de todas as instâncias básicas de todas as suas cláusulas e é denotada por G_S.

A seguir será definida uma interpretação especial sobre o Universo de Herbrand de S, chamada de *Interpretação de Herbrand de S*.

Definição 12: Seja [3]:

S – um conjunto de cláusulas

H – o Universo de Herbrand de S

I – uma interpretação de S sobre H

I é uma Interpretação de Herbrand de S se satisfaz às seguintes condições:

1. I mapeia todas as constantes de S nelas próprias

2. Seja f um símbolo funcional n-ário e sejam h_1, h_2, .., h_n elementos de H. Em I, à f é atribuída uma função que mapeia (h_1, h_2, .., h_n) – um elemento de H_n – a $f(h_1,h_2,..,h_n)$ – um elemento de H.

Não existe restrição com relação à atribuição para cada símbolo predicado em S, de maneira que diferentes Interpretações de Herbrand podem existir dependendo de tais diferentes atribuições. Desta forma, uma Interpretação de Herbrand I de um conjunto de cláusulas S é qualquer subconjunto de B_S (a Base de Herbrand).

Uma Interpretação de Herbrand é desta forma uma livre atribuição de valores verdade (verdadeiro ou falso) a todos os átomos da B_S.

Definição 13*:* Seja [5]:

S – um conjunto de cláusulas

H – o Universo de Herbrand de S

I – uma Interpretação de Herbrand de S sobre H

Se S é verdade em I, então I é um *Modelo de Herbrand de S*.

Exemplo 6: Seja $S=\{p(X) \vee \neg q(X), m(Y) \vee \neg g(a), q(Z)\}$ um conjunto de cláusulas.

Desde que S tem B_S = {p(a),q(a),m(a),g(a)}, as possíveis $2^{|B_S|}$ = 16 Interpretações de Herbrand de S são:

∅	{g(a)}	{q(a),m(a)}	{p(a),q(a),g(a)}
{p(a)}	**{p(a),q(a)}**	{q(a),g(a)}	{p(a),m(a),g(a)}
{q(a)}	{p(a),m(a)}	{m(a),g(a)}	{q(a),m(a),g(a)}
{m(a)}	{p(a),g(a)}	**{p(a),q(a),m(a)}**	**{p(a),q(a),m(a),g(a)}**

Tabela 2 – As dezesseis Interpretações de Herbrand de S

As interpretações que são modelos estão em negrito.

Um dos aspectos mais importantes da interpretação de Herbrand é que não é mais necessário considerar o universo de todas possíveis interpretações e usar os objetos de cada universo de toda forma possível, dado que as interpretações de Herbrand representam todas as outras. Todas as interpretações de Herbrand compartilham o mesmo universo – o Universo de Herbrand. Isso significa que é apenas necessário considerar instâncias obtidas substituindo variáveis por elementos do Universo de Herbrand [5].

Ao introduzir o universo de Herbrand, pode-se também considerar árvores semânticas. Conseqüentemente encontrar uma prova de um conjunto de cláusulas é equivalente a gerar uma árvore dessas.

Definição 14 [3]**:** Se A é um átomo, então os dois literais A e ¬A são ditos ser um o complemento do outro, e o conjunto {A, ¬A} é chamando par complementar.

Definição 15 [3]**:** Dado um conjunto S de cláusulas, seja A o conjunto atômico de S. Uma árvore semântica de S é uma árvore T, onde cada ligação está conectada com um conjunto finito de átomos ou negações de átomos de A de tal forma que:

1 – Para cada nó N, existem apenas finitas ligações imediatas $L_1,...L_n$ de N. Seja Q_i a conjunção de todos os literais no conjunto conectado à L_i, i = 1, ..., n. Então, $Q_1 \vee Q_2 \vee ... \vee Q_n$ é uma fórmula proposicional válida.

2 – Para cada nó N, seja I(N) a união de todos os conjuntos conectados às ligações do galho de T para baixo e incluindo o N. Então I(N) não contêm nenhum par complementar.

Definição 16 [3]**:** Seja A = {A_1, A_2, ... , A_k, ...} o conjunto atômico de um conjunto S de cláusulas. Uma árvore semântica de S é dita *completa* se e somente se para cada nó ponta N da árvore semântica, que é, um nó que não possui ligações saindo dele, I(N) contêm ou A_i ou ¬A_i para i = 1, 2, ...

Exemplo 7 [7]**:** Considere S = {P(x), Q(*f*(x))} e o conjunto atômico de S seja {P(a), Q(a), P(*f*(a)), Q(*f*(a)), P(*f*(*f*(a))),...} a árvore semântica de S é a seguinte:

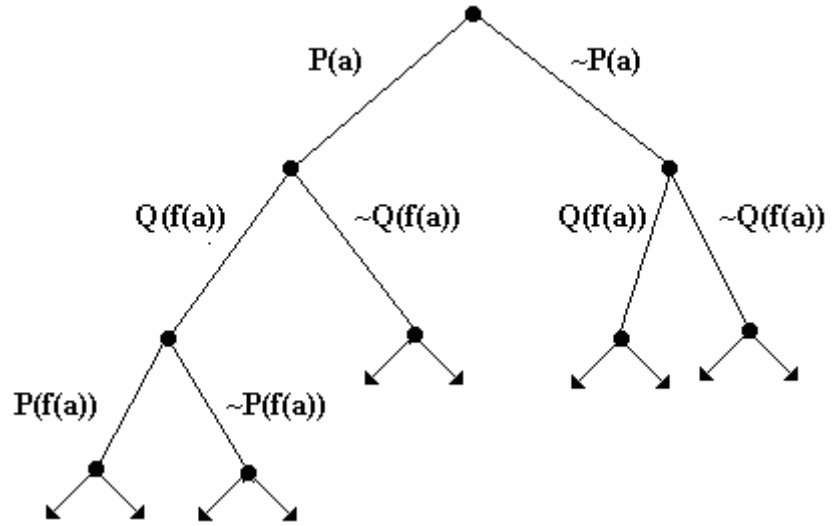

Ilustração 2.1 – Árvore semântica do exemplo 7

Definição 17 [3]**:** Um nó N é um *nó de falha* se I(N) falsificar alguma instância básica de uma cláusula em S, mas I(N') não falsificar nenhuma instância básica de uma cláusula em S para todo nó N´ que é ancestral de N.

Definição 18 [3]**:** Uma árvore semântica T é dita *fechada* se e somente se todo galho de T terminar em um nó de falha.

Definição 19 [3]**:** Um nó N de uma árvore semântica fechada é dito um *nó de inferência* se todos os nós imediatamente descendentes de N são nós de falha.

Exemplo 8 [7]**:** Seja S = {P, Q ∨ R, ¬P ∨ ¬Q, ¬P ∨ ¬R}. O conjunto atômico de S é A = {P, Q, R}. A figura 2.2(a) é uma árvore semântica completa de S, enquanto a figura 2.2(b) é uma arvore semântica fechada de S.

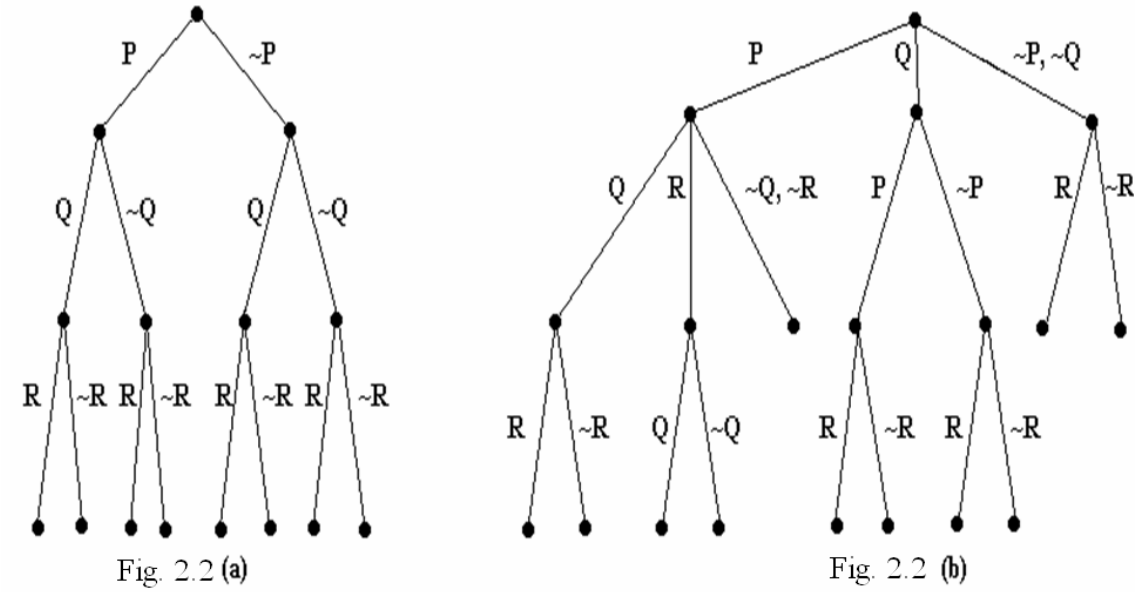

Ilustração 2.2 – Árvore semântica completa de S e Árvore semântica fechada de S do exemplo 8

Desta forma, para testar se um conjunto S de cláusulas é insatisfatível, é preciso apenas considerar interpretações sobre o universo de Herbrand S. Se S é falso sob todas as interpretações sobre o universo de Herbrand de S, então é possível concluir que S é insatisfatível. Desta forma, podemos organizar as possíveis interpretações de duas maneiras:

1 – Um conjunto S de cláusulas é insatisfatível se e somente se, correspondendo a toda árvore semântica completa de S, exista uma árvore semântica finita fechada.

2 – Um conjunto S de cláusulas é insatisfatível se e somente se existe um conjunto finito insatisfatível S' de instâncias de cláusulas básicas de S.

A segunda versão do Teorema de Herbrand sugere um procedimento de refutação. Isto é, dado um conjunto insatisfatível de cláusulas S para provar, se existe um procedimento mecânico que gere sucessivamente conjuntos S_1', S_2', ... , S_n', ... de instâncias básicas de cláusulas em S e pode sucessivamente testar S_1', S_2', ... para a insatisfatibilidade, então, como é garantido pelo teorema de Herbrand, esse procedimento pode detectar um N finito que S_N' é insatisfatível [3].

Com base nas teorias acima, Gilmore foi uma das primeiras pessoas a implementar o procedimento de Herbrand em um computador. Seu programa foi desenvolvido para detectar a inconsistência da fórmula dada. Durante a execução do seu programa, eram gerados conjuntos sucessivos de S_0', S_1',... onde S_i' é o conjunto de todas as instâncias básicas obtidas trocando as variáveis em S pelas constantes do conjunto de constantes nível-i H_i. Como cada S_i' é uma conjunção de cláusulas básicas, pode-se usar qualquer método disponível na lógica

proposicional para checar a sua insatisfatibilidade. Ele utilizou o método da multiplicação (ver apêndice 1) [3].

O programa de Gilmore foi feito para provar fórmulas simples, mas encontrou dificuldades decisivas com a maioria das outras fórmulas da lógica de primeira ordem. Estudos cuidadosos do seu programa revelaram que seu método de testar a inconsistência de uma fórmula proposicional era ineficiente. O método de Gilmore foi melhorado por Davis e Putnam (1960), meses depois de seus resultados serem publicados. No entanto, esses melhoramentos não foram suficientes. Muitas fórmulas válidas da lógica de primeira ordem ainda não puderam ser provadas por computadores em uma quantidade razoável de tempo.

Um grande avanço foi feito por Robson (1965), que introduziu o chamado princípio da resolução. O procedimento de prova por resolução é muito mais eficiente que os procedimentos anteriores. Desde a introdução do princípio da resolução, muitos refinamentos foram sugeridos como tentativas de aumentar a sua eficiência.

3. Estado da arte: Provadores automáticos de teoremas em lógica de primeira ordem

Antes de falar sobre os provadores automáticos de teoremas em lógica de primeira ordem, é necessário falar da “Conference on Automated Deduction (CADE)" que é o principal fórum internacional para a apresentação de pesquisas em todos os aspectos da dedução automática. A primeira conferência realizou-se em 1974 no “Argonne National Laboratory“ próximo de Chicago. Os CADEs anteriores eram geralmente bienais, mas desde 1996 que o CADE passou a ser anual. E desde 2001, o CADE juntou-se com outras conferências para formar a “International Joint Conference on Automated Reasoning (IJCAR)” [6].

Para estimular a pesquisa e desenvolvimento de sistemas na área dos provadores automáticos de teoremas e para expor sistemas de provas automáticas para dentro e fora da comunidade dos provadores automáticos de teoremas, uma competição, a “CADE ATP System Competition (CASC)”, é realizada a cada conferência da CADE. A CASC avalia o desempenho dos sistemas provadores automáticos de teoremas, totalmente automáticos, em lógica de primeira ordem. A avaliação é em termos de:

- O número de problemas resolvidos;
- o número de problemas resolvidos com uma solução de saída;
- e a média de tempo de execução dos problemas resolvidos.

No contexto de:

- Um número limitado de problemas qualificados, escolhidos da “TPTP (“Thousands of Problems for Theorem Provers”, que em português significa: milhares de problemas para provadores de teoremas) Problem Library”, cuja descrição segue abaixo;
- e um determinado tempo limite para cada tentativa de solução.

A TPTP é uma biblioteca de testes para os problemas de provas automáticas de teoremas (cuja sigla em inglês é ATP (Automated Theorem Proving)). O TPTP abastece a comunidade de ATP com [8]:

- Uma biblioteca completa de problemas para testar provadores automáticos que estão disponíveis hoje, a fim de proporcionar um resumo e um simples mecanismo claro de referência;
- uma lista completa de referências e outras informações interessantes para cada problema;
- novas variações generalizadas de problemas cuja apresentação inicial é para um provador automático específico;
- instâncias de tamanhos arbitrários de problemas genéricos (por exemplo, o problema das n-rainhas);
- um utilitário para converter os problemas para formatos de provadores automáticos existentes.

Além disso, a competição CASC possui divisões por áreas. Durante os anos o número de divisões foi mudando e as próprias divisões também. Mas no último CASC, que ocorreu no dia 17 de julho de 2007 em Bremen na Alemanha, a competição se dividiu em seis classes [9]:

- A divisão FOF (“first-order form non-propositional theorems”): axiomática FOF com uma conjectura provável
    - Duas categorias de problemas (para análise da especificação);
    - Duas classes de classificação (para análise da capacidade);
- A divisão CNF (“mixed clause normal form really non-propositional theorems”): conjunto de cláusulas insatisfatíveis
    - Cinco categorias de problemas;
    - Duas classes de classificação;
- A divisão FNT (“first-order form non-propositional non-theorems”): possíveis axiomas FOF(“Problemas formulados na sintaxe completa de lógica de primeira ordem”) com conjecturas que não podem ser provadas
    - Duas categorias de problemas;
    - Duas classes de classificação;
- A divisão SAT (“clause normal form really non-propositional non-theorems”): conjunto de cláusulas satisfatíveis
    - Duas categorias de problemas;
    - Duas classes de classificação;
- A divisão EPR (“effectively propositional clause normal form theorems and non-theorems”): conjunto de cláusulas finitas
    - Duas categorias de problemas;
    - Uma classe de classificação;
- A divisão UEQ (“unit equality clause normal form really non-propositional theorems”): cláusulas de unidade equitativas insatisfatíveis
    - Uma categoria de problema;
    - Uma classe de classificação;

Baseado na competição e suas divisões acima citadas, segue um detalhamento de alguns dos vencedores das categorias e alguns dos melhores provadores automáticos de teoremas em lógica de primeira ordem.

3.1 E-SETHEO

O E - SETHEO é um provador automático de teoremas composicional com estratégia paralela. Combina uma variedade de provadores de teoremas de alto desempenho e procedimentos de decisão especializados em um dos mais poderosos sistemas ATP disponíveis. A idéia central do framework E - SETHEO é deixar diferentes procedimentos de busca de provas competirem por recursos para resolver um determinado problema.

O E - SETHEO utiliza uma série de diferentes mecanismos de inferência para realizar diferentes cálculos e utiliza diferentes heurísticas para isso. Existem dois modos distintos de prova: modo de treinamento e modo de aplicação. No modo aplicação, o provador é totalmente automático. Ele é executado em três diferentes fases:

- Normalização do problema;
- classificação do problema e seleção do cronograma;
- e busca da prova (estratégia paralela).

No modo de treinamento, uma interação mínima do usuário é necessária. O treinamento consiste de uma única etapa complexa (envolvendo normalização e classificação para todos os conjuntos de problema):

- Otimização do cronograma

  A seguir, será explicada cada uma das fases [10]:

  - Normalização do problema - Numa primeira etapa, o formato do problema é reconhecido, e o problema é transformado na forma normal de cláusula. Além disso, este problema é ainda mais simplificado o tanto quanto for possível. O resultado é uma representação do problema da prova em um formato que é (após eventuais transformações triviais) acessível para todas as máquinas de inferência de propósito geral do sistema.
  - Classificação do problema e seleção do cronograma – Na segunda fase, o problema é classificado de acordo com um conjunto de propriedades simples do problema. O sistema verifica essas propriedades e define qual a classe do problema. Para cada uma das classes resultantes, o provador seleciona um cronograma para as diferentes estratégias. Esse cronograma contém uma lista de tuplas (estratégia, limite de tempo), de tal forma que a soma de todos os tempos limites individuais é igual ao limite de tempo total para cada tentativa de prova.
  - Busca da prova – Na fase final, o provador executa cada estratégia com o tempo limite atribuído. Ela pára logo que uma das estratégias puder determinar o status do problema (por encontrar uma prova ou, mostrando que o problema não tem uma prova). Em princípio, as estratégias podem ser executadas em paralelo, quer em um único processador, em um sistema SMP, ou em um conjunto de computadores. No entanto, na experiência dos desenvolvedores, a sobrecarga para a distribuição do problema, e um grande esforço para a manutenção da versão paralela, não valem a pena. Na implementação atual, o E-SETHEO faz a pesquisa da prova seqüencialmente, ou seja, as estratégias são executadas uma após a outra, em uma ordem fixa.
  - Otimização do cronograma – Sabe-se que diferentes domínios de aplicação exigem diferentes estratégias de busca da prova. A fim de adaptar o E - SETHEO a um determinado domínio, todas as estratégias são executadas em uma amostra representativa de problemas. Em seguida, aplica-se uma

combinação de otimização genética e escalonamentos (“hill-climbing”) para encontrar um (aproximadamente) ótimo conjunto de cronogramas para esta amostra. Se a amostra é bem escolhida (ou suficientemente grande), os cronogramas gerados geralmente mostram um excelente desempenho ao longo de todo o domínio.

O sucesso do E - SETHEO pode ser parcialmente explicado pelo conceito de estratégias paralelas e pela fácil adaptação a um determinado domínio exigido. No entanto, uma outra importante razão é o número de excelentes máquinas de inferência utilizadas para as diferentes estratégias. E - SETHEO emprega provadores em três paradigmas muito diferentes. Cada um destes provadores está entre os melhores sistemas de alto desempenho em sua classe. Além disso, ele também usa procedimentos de decisões especiais para problemas proposicionais e finitamente básicos, bem como estratégias de cooperação baseadas no lema de intercâmbio entre os diferentes sistemas [10]:

- SETHEO – SETHEO é um provador de teoremas baseado no modelo do cálculo de eliminação. No E - SETHEO, várias versões do SETHEO com diferentes estratégias de busca e diferentes pré-processadores são utilizados como estratégias distintas.
- E – E é um provador de teoremas baseado em saturação numa visão puramente equacional. Combina superposição com reescrita e um grande número de técnicas de eliminação de redundância. Atualmente, o E é utilizado no modo automático como uma estratégia única do E - SETHEO.
- DCTP – DCTP é o mais recente motor de inferência. Trata - se de um provador de teoremas baseado no cálculo de tableaux desconexos, estendido com regras de igualdade de tratamento. A força especial do DCTP é a capacidade de encontrar modelos para uma classe bastante grande de conjuntos de cláusulas satisfatíveis.
- Eground e SEMPROP – O procedimento de “grouding” chamado Eground e o (estendido) provador proposicional SEMPROP, juntos, formam um procedimento de decisão para a classe de problemas CNF em um universo finito de Herbrand. O Eground transforma tais problemas em (espera-se que em pequenos) problemas proposicionais, utilizando técnicas baseadas em constantes, divisão de cláusulas, e simplificações para lidar com os problemas que ainda são impossíveis de resolver com abordagens simples. SEMPROP é então usado para decidir o problema resultante.
- Estratégias complexas - Além de estratégias baseadas em um único cálculo ou idéia, o E - SETHEO utiliza também estratégias complexas, em que diferentes provadores colaboram entre si para resolver o problema. A maioria delas é escolhida com base em um conjunto de provadores, onde um sistema é usado como um pré-processador para simplificar o problema para o outro

sistema, ou no lema da troca, onde o problema é modificado pela adição de resultados intermediários de outros provadores.

Além disso, o E-SETHEO venceu algumas competições tais como o CASC-17 nas classes MIX e na SEM e ficou em terceiro lugar na classe FOF. Já no CASC-JC, o E-SETHEO venceu nas classes FOF, MIX, classe essa onde dividiu o prêmio com o Vampire, e na classe EPR. Além disso, ficou em terceiro lugar nas classes UEQ e SAT da mesma competição acima citada.

3.2 Vampire

O intuito inicial do Vampire era o de criar um protótipo de provador que pudesse ajudar na pesquisa de algoritmos eficientes e estruturas de dados para resolução e paramodulação (ver apêndice 2). Inicialmente esse protótipo serviu para dois propósitos [11]:

- Com a análise do comportamento do sistema, permitiu aos desenvolvedores identificar gargalos de desempenho, formular problemas de implementação relacionados e verificar quais técnicas poderiam funcionar melhor.
- Verificar a necessidade de otimizações propostas, onde em muitos casos poderiam ser feitas medindo o progresso global em termos do número de problemas que poderiam ser resolvidos com os recursos computacionais disponíveis.

Depois dessa fase, quando o Vampire ganhou maturidade e provou ser eficiente na resolução de problemas não-triviais, os desenvolvedores analisaram e viram que o mesmo poderia ser usado como base para desenvolver um provador automático de teoremas que poderia ser usado em um grande número de áreas de aplicação, em particular, no desenvolvimento formal em hardware e software e na representação de conhecimento.

Após essa fase, como o escopo era grande, os desenvolvedores decidiram refiná-lo para que o Vampire fosse principalmente eficiente em encontrar boas soluções para problemas algoritimicamente básicos de busca de provas, utilizando resolução e paramodulação. E num nível mais baixo de importância, encontrar formas de melhorar o desempenho do sistema.

Com o escopo definido, pode-se falar do projeto Vampire propriamente dito. O projeto foi originado na universidade Uppsala onde o PhD Andrei Voronkov criou de 1993 até 1995 o primeiro protótipo, que basicamente implementava uma resolução binária.

Em 1998, o Dr. Alexandre Riazanov que também fazia parte do projeto nessa época, decidiu que o sistema poderia ser mais robusto e poderia suportar um cálculo mais eficiente de resolução ordenada e superposição. O projeto original foi inspirado na implementação do OTTER (primeiro provador de teoremas de alto desempenho largamente utilizado). A prioridade mais alta foi dada ao desenvolvimento e experimentação com algoritmos eficientes e estrutura de dados para operações básicas de busca de provas baseado em saturação (ver apêndice 2), em particular técnicas de indexação de termos e procedimentos práticos de saturação. A competitividade foi aumentando na mesma medida que o projeto progredia, e

desde o ano de 1999 o projeto Vampire vem participando de todas as competições do CASC [11].

Em 1999, com a total re-implementação do protótipo, surgiu a versão 0.0 que implementava o método de resolução ordenada e de superposição, com algumas técnicas de simplificação básicas. Desde então a arquitetura do sistema ficou estável e o desenvolvimento passou a ser incremental.

A implementação do Vampire 0.0 caracterizava-se pelo uso de várias técnicas de indexação, nem sempre muito eficientes, para implementar todas as operações de busca requeridas em conjunto de termos e cláusulas. Segundo os desenvolvedores, provavelmente a característica mais importante introduzida nessa versão foi a da estratégia de recursos limitados, destinada a tornar mais eficiente a busca de provas na presença de um tempo limite.

No CASC-16 (1999), o Vampire ficou em segundo lugar em uma das mais importantes divisões da época, a divisão CNF (problemas de vários tipos na forma normal de cláusulas). Posteriormente, o vencedor E-SETHEO (acima citado) foi desqualificado por alguns problemas, e desta forma, o prêmio ficou com o Vampire.

A versão do ano 2000, Vampire 1.0, melhorou significativamente as técnicas de indexação, conhecidamente como maior gargalo de desempenho. E, além disso, passou a ter um procedimento alternativo de saturação, o algoritmo DISCOUNT, inspirado no SPASS e E (outros provadores automáticos de teoremas).

A performance do Vampire no CASC-17 não foi muito boa, ficando apenas com o quarto lugar na divisão CNF, embora, combinado com o gerador de cláusulas FLOTTER, venceu a divisão FOF.

Até 2001, os desenvolvedores do Vampire deram pouca atenção à flexibilidade das estratégias de definição. No Vampire 2.0 eles deram mais atenção a essa área e definiram vários esquemas de seleção de literais e com isso garantiram um aumento em termos de números de problemas resolvidos. Junto com uma série de mudanças menos importantes, o Vampire 2.0 passou para o primeiro lugar no CASC-JC nas classes de segurança (apenas a resposta “insatisfatível” e “satisfatível” é requerida) e na classe de provas (a prova de alguma forma é requerida) [11].

Durante o período anterior de 2001, o Vampire só trabalhava com formas normais de cláusulas. Mas após esse período, especificamente no ano acima citado, Andrei Voronkov implementou um componente pré-processador que fazia conversão em cláusulas e algumas transformações simplificadoras adicionais, como expansões de alguns predicados e definições de funções. Com isso, o Vampire tratava-se basicamente de um sistema que consistia de um núcleo baseado em resolução e superposição e do pré-processador. Apesar da pouca idade do pré-processador o sistema ficou na terceira colocação na divisão FOF do CASC-JC.

Em 2002, os esforços dos desenvolvedores se concentraram na melhoria da eficiência em problemas com igualdade, mais especificamente em problemas de igualdade unitária. Desta forma, o Vampire 5.0 caracterizou-se pela implementação altamente melhorada da indexação modulada e pela checagem refinada de ordenação de constantes. Nas demais

versões do Vampire, o pré-processador foi conectado ao núcleo através de alguns códigos desenvolvidos pelos programadores do Vampire [11].

No CASC-18, o Vampire 5.0 ganhou ambas as classes da divisão CNF e da divisão FOX. A partir daí, o Vampire vem ganhando em todos os anos as divisões CNF e FOF. No ano de 2007, a versão do Vampire que venceu a divisão CNF foi a 8.1 e a que venceu a FOF foi a 9.0.

Desde 2002 para cá muitos melhoramentos foram feitos, mas todos mantendo a mesma base. Desta forma, a última versão do Vampire funciona utilizando-se de muitas técnicas eficientes para realizar a maioria das operações nos conjuntos de termos e cláusulas. Além disso, um algoritmo em tempo de execução é utilizado para acelerar algumas operações custosas (outro melhoramento), tal como, controlar a ordenação das constantes.

E apesar do núcleo do Vampire continuar trabalhando apenas com formas normais de cláusulas, o pré-processador aceita o problema em sintaxes específicas de cada área, transforma-o em cláusulas e faz uma série de transformações, onde após isso, passa o resultado para o núcleo. Quando o teorema é provado, o sistema produz uma prova verificável que atende as exigências das classes do CASC que ele disputa [12].

Abaixo, segue uma imagem do Vampire em execução:

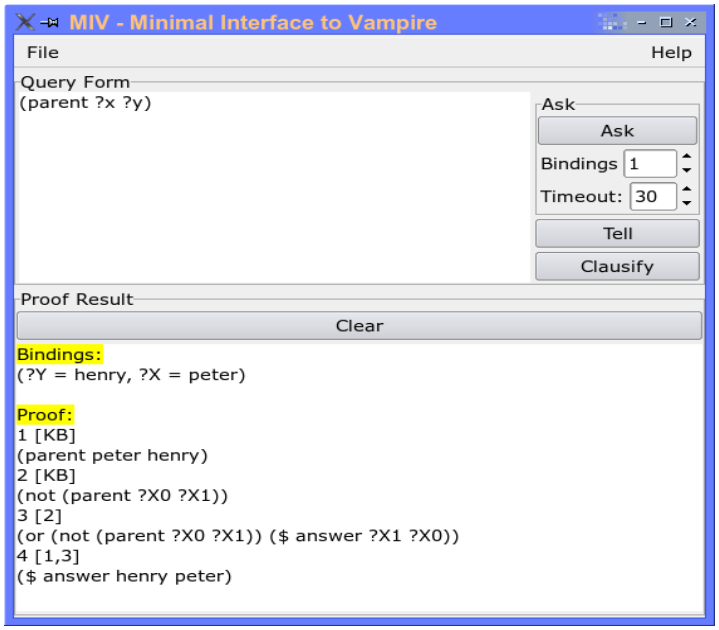

Ilustração 3.1 – Interface criada para a utilização do Vampire

3.3 Paradox

O Paradox é um provador automático de teoremas desenvolvido por Koen Lindström Claessen e Niklas Sörensson na “Chalmers University of Technology”. Ambos utilizaram a

linguagem Haskell no seu desenvolvimento e o Paradox é regido pelos termos da licença GNU (software livre) [13].

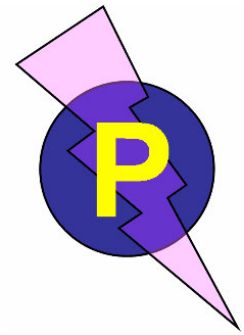

Ilustração 3.2 – Símbolo do Paradox

Existem muitos métodos para encontrar modelos finitos de conjuntos de cláusulas de lógica de primeira ordem (FOL). Os dois estilos mais bem sucedidos de métodos são usualmente chamados: estilo de métodos MACE, chamado dessa forma após a ferramenta MACE de McCune, bem como o estilo de métodos SEM, chamado dessa maneira após a ferramenta SEM de Zhang e Zhang [14].

O estilo de métodos MACE transforma o conjunto de cláusulas FOL e um tamanho de domínio em um conjunto de cláusulas de lógica proposicional através da introdução de variáveis proposicionais representando a função e tabelas de predicado, e consecutivamente achatando e instanciando as cláusulas do conjunto de cláusulas. Um provador de teoremas de lógica proposicional, também chamado solucionador SAT, é então usado para atacar o problema resultante.

O estilo de métodos SEM realiza uma busca diretamente sobre o problema sem convertê-lo em uma simples lógica. Uma busca de “backtracking” básica apoiada por poderosos métodos de propagação, baseados principalmente na exploração da igualdade, são utilizados para a busca de interpretações de funções e tabelas de predicado. Um princípio denominado redução simétrica é usado para evitar buscas por modelos isomórficos repetidamente.

No Paradox, os desenvolvedores implementaram um conjunto de novas técnicas de geração de modelos finitos para lógica de primeira ordem desordenada. Estas técnicas melhoraram consideravelmente a idéia básica por trás do programa MACE. As principais contribuições do Paradox no MACE são [14]:

- O principal problema no estilo de métodos MACE é a instanciação de cláusulas, que é exponencial no número de variáveis de primeira ordem em uma cláusula. No Paradox, foi criada uma nova heurística polinomial para a quebra de cláusulas. Além disso, uma técnica completamente nova de redução de variáveis foi criada, denominada *definições de termos*.
- A busca de um modelo passa por fases consecutivas de tamanhos de domínios crescentes. Nos algoritmos atuais, não existe quase nenhuma ligação entre as buscas por modelos de diferentes tamanhos. O Paradox faz uso de um verificador de

satisfatibilidade incremental, a fim de reutilizar informações sobre buscas falhas de um modelo de tamanho s na busca de um modelo de tamanho s + 1.

- Os buscadores de modelos do estilo SEM fazem uso de técnicas a fim de reduzir as simetrias na busca do problema. O Paradox foi o primeiro a adaptar as mesmas técnicas num framework de estilo MACE. A contribuição do Paradox é que quando se utiliza um resolvedor SAT, o Paradox aplica a redução de simetria estaticamente, isto é, adicionando constantes extras, enquanto os métodos do estilo SEM aplicam-na dinamicamente, ou seja, durante a busca.
- Sabe-se que informações ordenadas podem ajudar na busca por modelos. No entanto, o Paradox trabalha com problemas desordenados, e por isso, necessita criar modelos desordenados. Os desenvolvedores do Paradox implementaram uma espécie de algoritmo de inferência, que automaticamente encontra informações ordenadas apropriadas para problemas desordenados. Este tipo de informação pode ser utilizado de diversas formas para reduzir a complexidade do problema de busca do modelo, enquanto ainda é buscado um modelo desordenado.

Além disso, o mais popular algoritmo básico para resolver SAT é o procedimento DPLL, que é um procedimento de "backtracking" baseado em propagação unitária. Versões mais modernas deste algoritmo normalmente incluem diversas melhorias, tais como heurísticas para seleção de variáveis, "backjumping", aprendizagem de conflitos, etc..

No contexto do Paradox, a aprendizagem de conflitos é o principal interesse. Ela permite que o processo possa aprender a partir de erros anteriores. Concretamente, para cada contradição que ocorre durante a busca, a razão do conflito é analisada, resultando em uma cláusula aprendida que pode evitar situações semelhantes em partes de buscas futuras. Desta forma, um conjunto de cláusulas de conflitos é recolhido durante a busca, representando informações sobre aquela busca.

Como parte do Paradox, os desenvolvedores implementaram uma versão do algoritmo DPLL, estendido com a possibilidade de resolver incrementalmente uma seqüência de problemas. A idéia que o sistema quer se beneficiar é com a semelhança entre a seqüência de instâncias SAT, geradas pelas codificações proposicionais que foram criadas pelo Paradox para cada tamanho de domínio. Isto é feito pela manutenção das cláusulas aprendidas que foram geradas pela busca de uma instância na próxima busca.

Resumindo, o Paradox é um buscador de modelos em domínios finitos de lógica de primeira ordem com igualdade. Além disso, ele é um buscador de modelos do estilo MACE, que basicamente traduz um problema de primeira ordem em uma seqüência de problemas SAT, que são resolvidos por um solucionador SAT [15].

O Paradox venceu a classe SAT dos CASC dos anos de 2003, 2004, 2005 e 2006. No ano de 2007, ele venceu tanto a classe SAT quanto a classe FNT do CASC-21.

3.4 Darwin

Darwin é um provador automático de teoremas para lógica de primeira ordem. Ele aceita problemas formulados nos formatos TPTP ou TME, onde problemas não clausais são convertidos em problemas clausais usando o provador E (provador de teoremas equacional, que já foi citado anteriormente). A igualdade não é construída na versão atualmente implementada do cálculo, é ao invés disso, automaticamente axiomatizada para um determinado problema. O Darwin é um procedimento de decisão para conjuntos de cláusulas livres de função, e é em geral mais rápido e tem um desempenho melhor sobre esses problemas do que abordagens proposicionais [16].

O Darwin é a primeira implementação do cálculo de evolução de modelos. O cálculo de evolução de modelos eleva o procedimento proposicional DPLL para a lógica de primeira ordem. Uma das principais motivações para esta abordagem foi a possibilidade de migrar para o nível de primeira ordem, algumas das técnicas mais eficazes de busca desenvolvidas pela comunidade SAT para o procedimento DPLL [17].

A versão atual do Darwin implementa versões de primeira ordem de regras de inferência de unidade de propagação analogamente a uma forma restrita de unidade de resolução e inclusão por cláusulas unitárias. Para manter a integralidade, o Darwin inclui uma versão de primeira ordem da regra de inferência proposicional.

A busca de prova no Darwin começa com uma interpretação padrão de um dado conjunto de cláusulas, que está expandindo-se no sentido de um modelo ou, até uma refutação ser encontrada. O espaço de busca é explorado por aprofundamento iterativo sobre a profundidade do termo de literais candidatos. Darwin emprega uma estratégia de busca uniforme para todas as classes de problemas.

A estrutura central de dados é o contexto. Um contexto representa uma interpretação como um conjunto de literais de primeira ordem. O contexto é desenvolvido usando instâncias de literais das cláusulas de entrada. A implementação do Darwin destina-se a suportar operações básicas nos contextos de uma forma eficiente. Isso envolve a manipulação de grandes conjuntos de literais candidatos para modificar o contexto atual. Os literais candidatos são computados através de simultâneas unificações entre dadas cláusulas e literais do contexto. Este processo é rápido pelo armazenamento parcial dos unificadores para cada cláusula dada, e fundindo-as em diferentes combinações de literais do contexto, ao invés de refazer os cálculos dos unificadores parciais [16].

Para aumentar a eficiência da filtragem dos candidatos desnecessários que estão junto com os literais do contexto, indexação por árvores discriminatórias ou árvores de substituição são empregadas. A regra de divisão gera pontos de escolha na derivação que são retrocedidas usando a forma de “backjumping”, semelhante à que foi utilizada no DPLL – provador baseado no SAT. Também semelhante aos provadores SAT, lemas podem ser aprendidos com ramos fechados e utilizados para podar o restante do espaço de busca [16].

A Versão 1.3 do Darwin pode ser usada como um buscador de modelos finitos para lógica de primeira ordem com igualdade. A abordagem é semelhante a outros buscadores de

modelos finitos como o Paradox, mas, em vez de transformar um problema em lógica proposicional, ele é convertido em lógica de primeira ordem livre de função [16].

O Darwin 1.3 venceu a classe EPR no CASC - J3, concurso realizado em 2006, e a variante do buscador de modelos finitos do Darwin conseguiu o terceiro lugar na classe SAT, da mesma competição. Além disso, o Darwin 1.3 venceu a classe EPR no CASC-21.

3.5 Waldmeister

O WALDMEISTER é um provador de teoremas para lógica equacional de primeira ordem com cláusulas unitárias como axiomas, baseado no procedimento de complemento de Knuth-Bendix (ver Apêndice 2) . Durante o desenvolvimento do WALDMEISTER o objetivo principal foi ser eficiente em todo o processo. Para isso, os desenvolvedores seguiram um paradigma de abordagem da engenharia, onde analisando provadores típicos baseados em saturação, eles reconheceram uma estrutura lógica de três níveis [18].

Os desenvolvedores analisaram esses três níveis com respeito aos pontos críticos responsáveis pela performance global dos provadores. Quando necessário, as questões de como atacar esses pontos críticos foram respondidas por experimentações extensas e avaliações estatísticas. No contexto do WALDMEISTER, um experimento é uma comparação quantitativa de diferentes implementações da mesma funcionalidade no mesmo sistema. Isso foi possível porque a modularidade foi um dos princípios básicos do sistema.

Desta forma, a base para o desenvolvimento do sistema foi o modelo de três níveis lógicos dos provadores baseados em completude. Quando se realiza procedimentos de completude em provadores, isto conduz a um grande conjunto de algoritmos determinísticos e parametrizados. Dois parâmetros principais são típicos para esse tipo de provador: a redução ordenada e a heurística de busca para guiar a prova. A escolha desses dois parâmetros para uma dada busca forma o nível mais alto do WALDMEISTER [19].

O nível intermediário desse sistema é uma máquina de inferência, que agrega as regras de inferência dos cálculos de prova dentro do loop principal. Esse loop é determinístico para qualquer escolha fixa de redução ordenada e heurística de seleção. Uma grande quantidade de tempo é usada e muitos experimentos são necessários para avaliar agregações diferentes desses parâmetros até que uma boa combinação seja encontrada [19].

O nível mais baixo fornece algoritmos eficientes e sofisticados, e estruturas de dados para a execução das operações básicas que são mais freqüentemente utilizadas, por exemplo, comparação, união, armazenamento e recuperação de dados. Estas operações consomem a maior parte do tempo e da memória [19].

Basicamente o que a arquitetura do WALDMEISTER faz é, ao aplicar o complemento de Knuth-Bendix, ele satura a axiomatização dada até que o objetivo possa ser mostrado por diminuição e/ou reescrita. A saturação é realizada em um loop que trabalha em um conjunto de ocorrências esperadas (pares críticos) e um conjunto de ocorrências aplicáveis. Dentro do loop, o WALDMEISTER basicamente executa os seguintes passos [19]:

1. Seleciona uma equação no conjunto de pares críticos.
2. Simplifica esta equação para uma forma normal (e a descarta, se ela for trivial).
3. Modifica o conjunto de regras atual, de acordo com a equação.
4. Gera todos os novos pares críticos.
5. Adiciona a equação no conjunto de regras.

Como já foi dito, o principal objetivo durante o desenvolvimento do WALDMEISTER era o de construir uma máquina de inferência de alta performance depois de ser coberto por estratégias de seleção sofisticadas. Graças às estruturas de dados e algoritmos utilizados no WALDMEISTER e com o controle flexível do nível intermediário, os desenvolvedores conseguiram seu objetivo.

Seguindo a abordagem de desenvolvimento da engenharia que já foi citada acima, o WALDMEISTER é muito eficiente (no que diz respeito ao tempo e a memória utilizada). O WALDMEISTER não só mostra uma alta taxa de inferência, mas também utiliza economicamente os recursos de memória e, conseqüentemente, evita diminuição de desempenho. Mas como o sistema está em constante construção, ele ainda possui alguns pontos fracos que precisam ser melhorados com o tempo, como por exemplo, melhorar as estratégias de buscas.

O WALDMEISTER foi codificado na linguagem de programação C. Inicialmente ele foi desenvolvido para funcionar sobre o sistema operacional da SUN, mas atualmente existem versões para Solaris e Linux.

O que motivou a construção do WALDMEISTER foi um projeto do professor dos criadores do WALDMEISTER (Buch e Hillenbrand), projeto esse, chamado de DISCOUNT, que foi desenvolvido na “University of Kaiserslautern”. Esse sistema alcançava bons resultados nos campos de heurísticas de busca em geral, aprendizado de estratégias úteis para os domínios dados e prova distribuída de teoremas. Mas a máquina de inferência do DISCOUNT não era a ideal. Por isso, Buch e Hillenbrand desenvolveram o WALDMEISTER, visando aperfeiçoar todos os três níveis acima citados [19].

Com isso, o WALDMEISTER tem se destacado no CASC, onde tem vencido esta competição, em sua classe UEQ, desde o CASC-14, que aconteceu no ano de 1997. O Waldmeister 806 foi o vencedor do último CASC, o CASC-21, que ocorreu no ano de 2007.

4. Estado da arte: Outros tipos de provadores automáticos de teoremas

No capítulo anterior foram mostrados alguns dos principais provadores automáticos de teoremas em lógica de primeira ordem existentes atualmente. Mas além de provadores de lógica em primeira ordem, existe uma grande quantidade de provadores em outras lógicas, e com outros propósitos, que utilizam uma diversidade de métodos para resolver os problemas. Desta forma, abaixo seguem alguns desses provadores.

4.1 Um Provador Automático de Teoremas para a Lógica Modal, Baseado em Anéis Booleanos

Este é um provador automático de teoremas que foi desenvolvido por Fabio Alexandre Campos Tisovec no IME-SP (Instituto Militar de Engenharia) no ano de 2007. Ele foi desenvolvido para a lógica modal (ver apêndice 3) e tem como objetivo principal ser um provador automático de teoremas com um bom grau de eficiência na prova de problemas SAT, que são reconhecidamente NP – difícil (ver apêndice 3).

Além disso, o uso da teoria de anéis booleanos (ver apêndice 3) para apoiar a resolução de problemas de satisfatibilidade não é muito comum. Desta forma, basicamente o que esse provador faz é pegar a expressão trabalhada e subdividi-la em inúmeras mini-expressões, que então são comparadas duas a duas, para que desta maneira possa se verificar a existência de contradições entre elas. Caso seja encontrada alguma contradição, sabe-se que a expressão não é válida, caso contrário ela é aceita.

Caracterizando o contexto do provador em questão, segue uma análise dos principais tipos de provadores existentes e algumas características dos mesmos [20]:

- Verificadores de tautologias - dado um seqüente, este tipo de verificador procura inferir se o seqüente é uma tautologia, ou seja, verificando a existência de valorações que o tornam inválido;
- Verificadores de provas - o sistema lê uma seqüência de fórmulas lógicas e verifica se cada fórmula pode ser derivada da anterior, dado seu conjunto de axiomas;
- Focados em um único teorema - o sistema é implementado para verificar um determinado teorema. Geralmente, trata-se de teoremas difíceis e busca-se por simplificações que tornem a verificação possível.

O provador em questão obedece às características do primeiro tipo acima citado, onde ele recebe uma seqüência de expressões lógicas, e o sistema verifica a validade de cada uma dessas expressões. Caso a expressão que esta sendo verificada seja uma fórmula, o sistema verifica se há uma valoração que a torna verdadeira e mostra a primeira solução encontrada, ou avisa que não há soluções para a dada expressão.

Caso a expressão seja um seqüente, o sistema verifica se o seqüente é uma tautologia ou não. Isto é realizado fazendo uma pequena transformação na expressão e então aplicando o mesmo algoritmo de cálculo de satisfatibilidade do primeiro caso acima citado.

A técnica de desenvolvimento que foi utilizada para a implementação deste provador foi o XP (eXtreme Programming). A linguagem de programação utilizada foi a C++. O sistema apresenta a seguinte estrutura [20]:

- Modelo de representação: este é o módulo do sistema que é responsável por manter uma representação interna dos objetos que estão sendo analisados.
- Analisador léxico: é o responsável pela conversão da entrada do programa (em formato de texto) para objetos do modelo.
- Coordenador: Parte do sistema que integra os outros módulos e é responsável pela saída do programa.
- Calculador de fórmulas: sub-módulo responsável por analisar a validade de cada expressão lógica processada pelo sistema.
- Testes: módulo auxiliar para garantir a corretude dos demais módulos.

Abaixo segue uma descrição mais detalhada dos principais pontos da estrutura acima citados:

- Modelo de representação: O conceito fundamental que determinou como ele seria implementado foi o polimorfismo. Todas as entidades extraídas das expressões lógicas que estão sendo analisadas são armazenadas em classes, que derivam da classe “Formula”. Desta forma, todas as operações nessas instâncias, como por exemplo verificação de igualdade, são tratadas de uma forma padrão e cabe à estrutura de classes decidir qual o método que será chamado.

  Além disso, o polimorfismo permite mudar dinamicamente entre os algoritmos de cálculo lógico. O algoritmo básico organiza as instâncias em uma árvore de sub-fórmulas. O segundo algoritmo desenvolvido mantém uma tabela de fórmulas já existentes, onde as sub-fórmulas são organizadas em um grafo dirigido acíclico.

  Com um mapeamento que indica em que lugar da tabela de fórmulas encontra-se a sub-fórmula desejada, o polimorfismo pode determinar quais métodos serão executados.
- Analisador léxico: A principal classe desse módulo é a classe “Parser”. Ela é responsável por extrair pedaços da expressão lógica a ser analisada e criar as instâncias do modelo de acordo com os pedaços extraídos. É a implementação deste módulo que define qual a linguagem reconhecida pelo programa. Também neste módulo utiliza-se polimorfismo para alternar entre os métodos de cálculo.
- Testes: A estrutura básica desse módulo dita o funcionamento geral dos testes individuais, bem como o de um grupo de testes. Como um grupo de testes também pode ser visto como um teste, é possível inserir grupos de testes em grupos maiores de testes. É desta forma que estão organizados os testes do programa. Cada módulo tem seu próprio conjunto de testes, e todos estes conjuntos estão inseridos em um teste completo.

O programa lê expressões lógicas tanto da linha de comando quanto de arquivos de entrada. Em seguida encontra-se a descrição da linguagem reconhecida nas expressões lógicas:

- Seqüente: premissas |- conclusão;
- Premissas: fórmula | premissas;
- Fórmula: identificador | (fórmula) And conjunção | Not(fórmula) | (fórmula) Or disjunção | (fórmula)->(fórmula) | Box(fórmula);
- Conjunção: (fórmula) | (fórmula) And conjunção;
- Disjunção: (fórmula) | (fórmula) Or disjunção;
- Identificador: seqüência de letras e/ou dígitos (o primeiro caractere deve ser uma letra).

4.2 PLLIC - Provador para as Lógicas Linear, Intuicionista e Clássica

O PLLIC foi desenvolvido no ano de 2006 em um projeto de graduação por alguns alunos da universidade UFMG (Universidade Federal de Minas Gerais), e foi orientado pela professora Elaine Pimentel. As linguagens de programação que foram utilizadas para o desenvolvimento do sistema foram: Java e λ-Prolog.

O provador automático de teoremas PLLIC, analisa quando seqüentes do tipo $\Gamma \vdash L\ \Delta$ são prováveis, onde Γ, Δ são conjuntos de fórmulas e L pode ser uma das seguintes lógicas: linear LL (ver apêndice 3), intuicionista LJ (ver apêndice 3), ou clássica LK (ver apêndice 3). O PLLIC decide sobre a provabilidade do fragmento proposicional das lógicas acima citadas, uma vez que a intenção do sistema é que o programa sempre pare, ou seja, que a pergunta: *o seqüente* $\Gamma \vdash L\ \Delta$ *é provável?* tenha sempre uma resposta: sim ou não. Além disso, se um seqüente é provável, o sistema PLLIC exibe a sua prova em cálculo de seqüentes [21].

As vantagens do PLLIC são [21]:

1. É totalmente automático.
2. A página de acesso é em português, inclusive o tutorial, os exemplos e a fundamentação teórica do sistema.
3. É acessado via web. Desta forma, nenhum conhecimento de linguagens de programação é necessário, funciona em qualquer sistema operacional e não é necessário fazer o "download" do programa: basta ter um "browser" de navegação.
4. É muito fácil de usar. A interface é completamente amigável.

O projeto possui um ponto fraco que é a o resultado de saída, que apresenta uma lista que quando lida da direita para a esquerda, apresenta as regras utilizadas na árvore de prova ordenada.

O objetivo de construir o PLLIC foi o de poder utilizar uma ferramenta de fácil manuseio e que possa ser acessada remotamente para o ensino de lógica em cursos de graduação. O fato de ser automático não atrapalha o processo de aprendizagem, uma vez que o aluno pode sempre comparar a sua prova com a fornecida pelo PLLIC.

Pode-se utilizar PLLIC para especificar pequenos problemas lógicos, como os que usualmente aparecem na literatura. E respondê-los de maneira fácil e rápida.

Abaixo seguem alguns exemplos que podem ser rodados no PLLIC e algumas imagens da execução dos exemplos:

1. Lógica clássica:

- |- A \/ (~A) - Princípio do terceiro excluído.

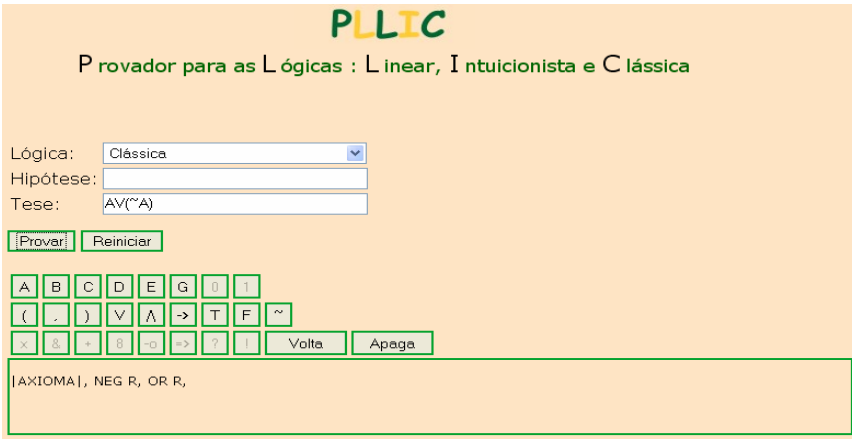

Ilustração 4.1– Exemplo 1 do PLLIC

- ~(~A) |- A - Lei de dupla negação.

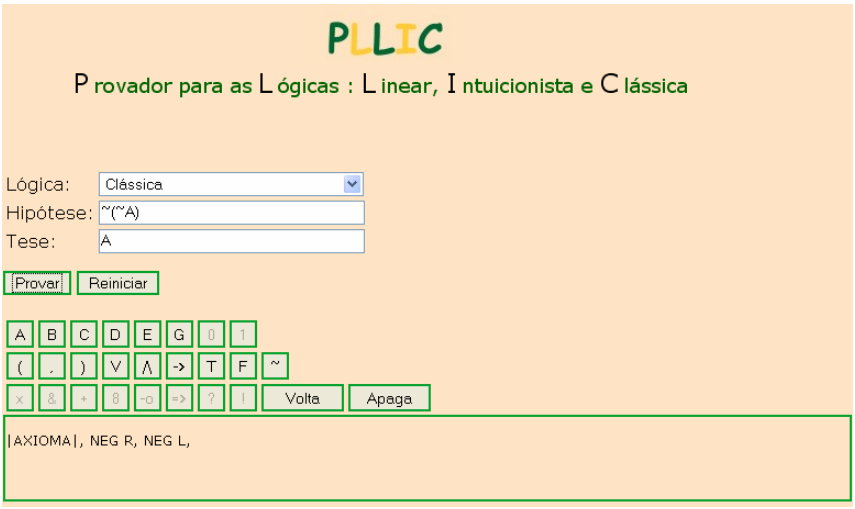

Ilustração 4.2 – Exemplo 2 do PLLIC

- ~((~ A)/\(~B)) |- A\/B

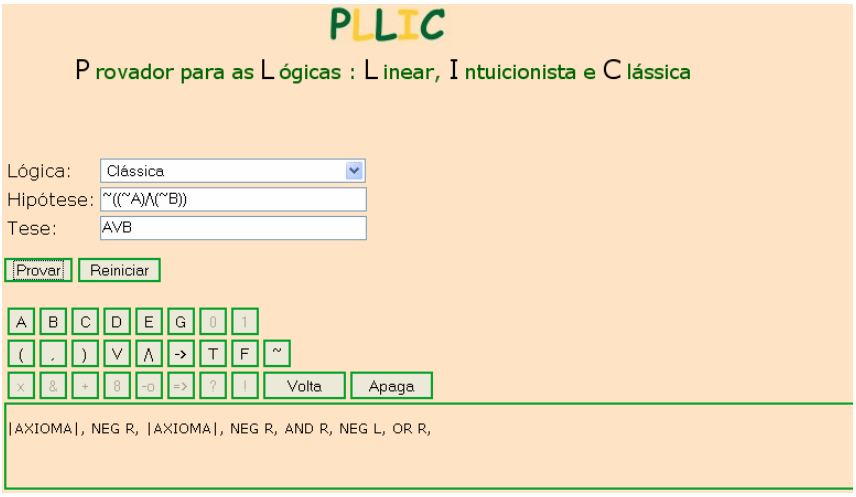

Ilustração 4.3 – Exemplo 3 do PLLIC

2. Lógica Intuicionista:

- A \/ B|- B \/ A - Associatividade da disjunção.

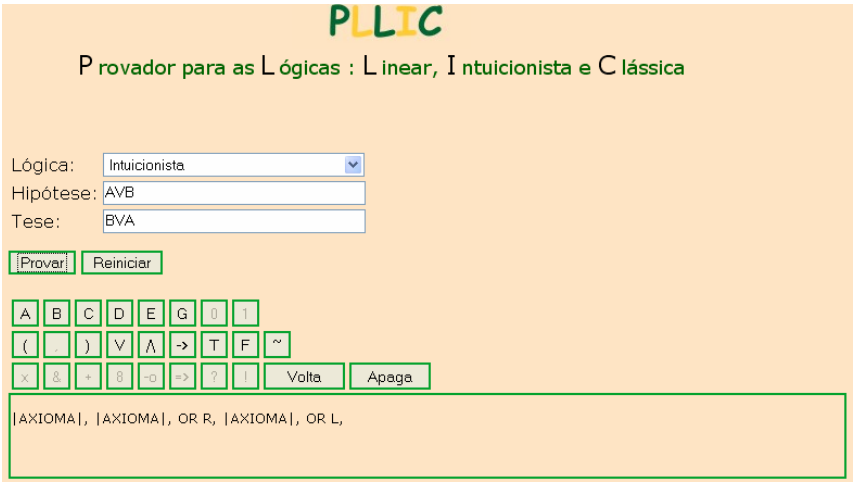

Ilustração 4.4 – Exemplo 4 do PLLIC

- A /\ B|- B /\ A - Associatividade da conjunção.

Ilustração 4.5 – Exemplo 5 do PLLIC

- A |- ~(~A) - Dupla negação.

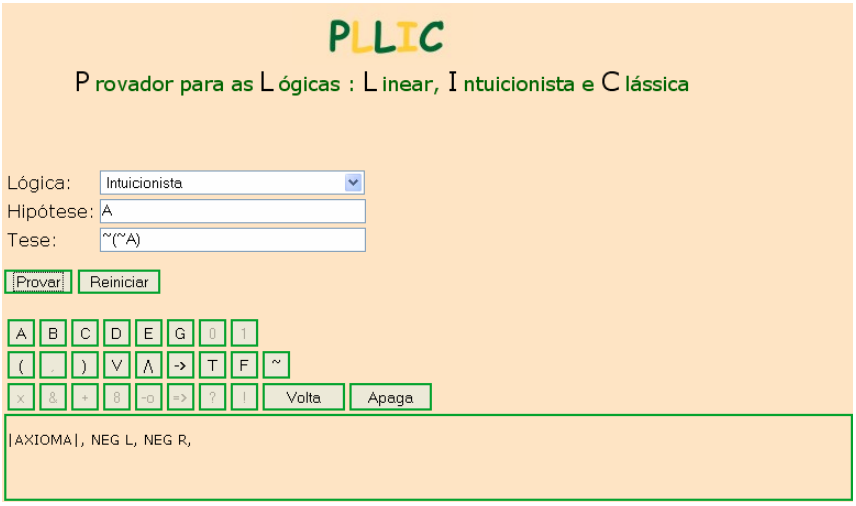

Ilustração 4.6 – Exemplo 6 do PLLIC

3. Lógica Linear:

- !A |- ?A - Exponenciais.

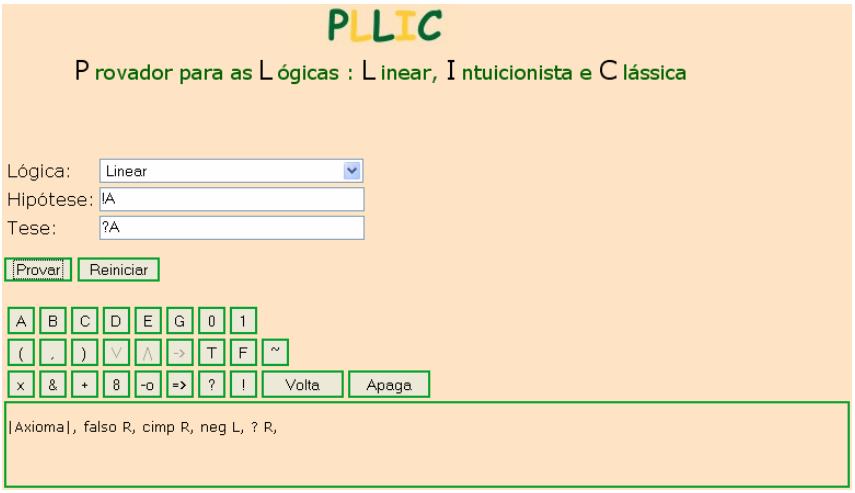

Ilustração 4.7 – Exemplo 7 do PLLIC

- A x B |- B x A - Associatividade da conjunção multiplicativa.

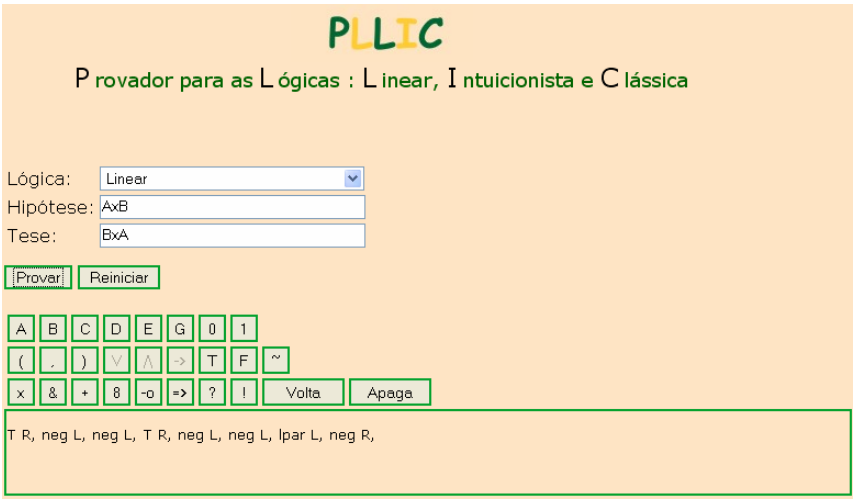

Ilustração 4.8 – Exemplo 8 do PLLIC

- A x B |- A 8 B - Dupla negação.

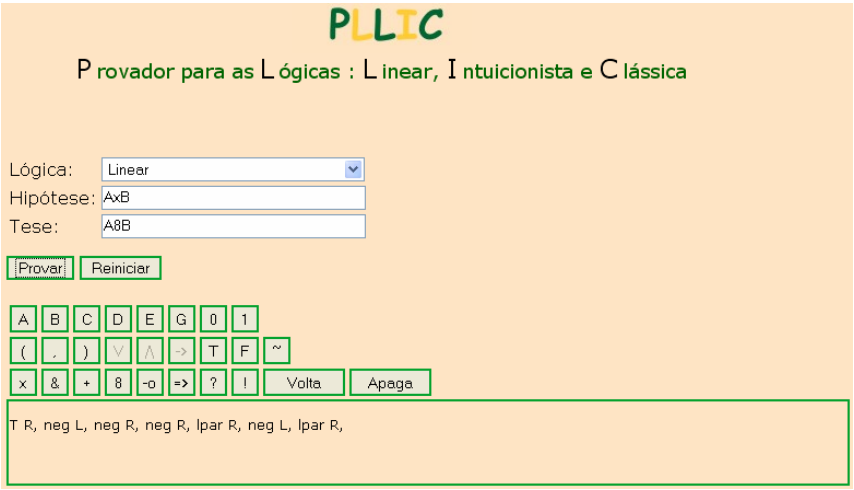

Ilustração 4.9 – Exemplo 9 do PLLIC

4.3 KEMS - Um provador de teoremas multi-estratégia

O KEMS foi desenvolvido na USP como tese de doutorado do aluno Adolfo Gustavo Serra Seco Neto. Ele é um provador de teoremas multi-estratégia baseado no método de tableaux KE (ver apêndice 3). Um provador de teoremas multi-estratégia é um provador de teoremas onde se pode variar as estratégias utilizadas sem modificar a base da implementação. Além de multi-estratégia, o KEMS é capaz de provar teoremas em três sistemas lógicos: lógica clássica proposicional, mbC (ver apêndice 3) e mCi (ver apêndice 3).

Abaixo seguem algumas das contribuições do KEMS [22]:

- Um sistema KE para mbC que é analítico, correto e completo; Um sistema KE para mCi que é correto e completo;
- Têm seis estratégias implementadas para lógica clássica proposicional, duas para mbC e duas para mCi;
- Têm treze ordenadores que são usados em conjunto com as estratégias;
- Implementa regras simplificadoras para lógica clássica proposicional;
- Possui uma interface gráfica que permite a visualização de provas;
- É um software livre e está disponível na Internet;
- Benchmarks obtidos através da comparação das estratégias para lógica clássica proposicional resolvendo várias famílias de problemas;
- Sete famílias de problemas para avaliar provadores de teoremas paraconsistentes;
- Os primeiros benchmarks para as famílias de problemas para avaliar provadores de teoremas paraconsistentes.

Um provador de teoremas multi-estratégia como o KEMS, pode ser utilizado com três objetivos: educacional, exploratório e adaptativo. Com fins educacionais, pode ser utilizado para ilustrar como a escolha de uma estratégia pode afetar o desempenho de um provador de teoremas.

Como uma ferramenta exploratória, um provador de teoremas multi-estratégia pode ser usado para testar novas estratégias e compará-las com outras já existentes.

Por fim, podemos ainda imaginar um provador de teoremas multi-estratégia adaptativo, que modifica a estratégia utilizada de acordo com as características do problema que é submetido ao provador.

A arquitetura do KEMS é apresentada na figura abaixo [22]:

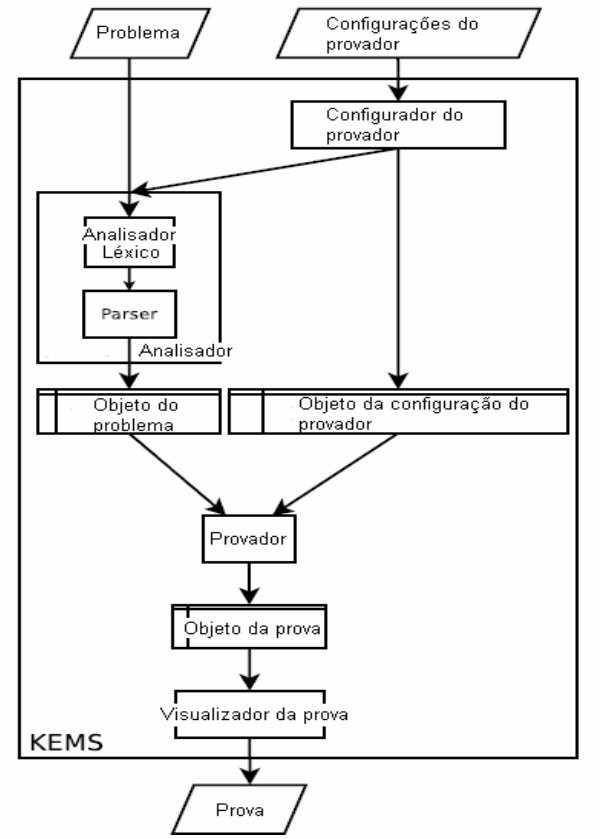

Ilustração 4.10 – Arquiteruta do Kems

O diagrama acima descreve que o usuário apresenta como entrada para o KEMS uma instância de um problema e uma configuração pro provador. A configuração do provador deve conter os valores de quatro parâmetros importantes para o KEMS [22]:

1. O sistema lógico para que seja escolhido o procedimento de busca da prova;
2. O analisador utilizado para a análise léxica para ser dado o “parse” do problema;
3. A estratégia escolhida para a busca da prova para o problema;
4. O ordenador escolhido para ser utilizado na estratégia.

Além disso, a configuração do provador pode ser usada para dar valores para outros cinco parâmetros menos importantes [22]:

1. O número de vezes que o provador deve executar o procedimento de busca da prova no problema dado.
2. O tempo limite para o procedimento de busca da prova (se este prazo for ultrapassado, o provador é interrompido);

3. Uma opção booleana para determinar se o provador deve salvar a fórmula original ou não.
4. Uma opção booleana para determinar se o provador deve descartar galhos fechados ou não;
5. Uma opção booleana para determinar se o provador deve guardar galhos descartados em disco (para ser re-utilizado após o procedimento de prova ter terminado) ou não.

Dadas estas entradas, o sistema retorna uma prova como saída, que contém, entre outras coisas:

- O status do tableau (aberto, ou fechado);
- A árvore tableau de prova, que pode ser parcial, se a opção de descartar os galhos fechados estiver ativa;
- O tamanho do problema;
- O tempo gasto pelo provador enquanto construía uma prova para o problema de entrada;
- O tamanho da prova;
- Um valor de contra - modelo, se o tableau é aberto.

A versão atual do KEMS é implementada com a linguagem de programação Java 1.5 com alguns aspectos escritos e com a linguagem AspectJ 1.5. A linguagem de programação Java foi a escolhida porque é orientada a objetos, e a extensão AspectJ foi escolhida pois suporta o paradigma de desenvolvimento de software orientado a aspectos.

O KEMS foi avaliado com várias instâncias de famílias de problemas e nenhuma configuração do KEMS obteve resultados incorretos.

Para várias destas instâncias, mesmo algumas de tamanho bem pequeno, o procedimento de busca de prova não terminou no tempo limite estabelecido para nenhum par (estratégia, ordenador) utilizado nos testes. Algumas instâncias de outras famílias foram difíceis apenas para alguns dos pares. Os testes foram executados com problemas para os três sistemas lógicos implementados.

Seguem abaixo algumas imagens do KEMS:

Ilustração 4.11 – Tela principal do KEMS

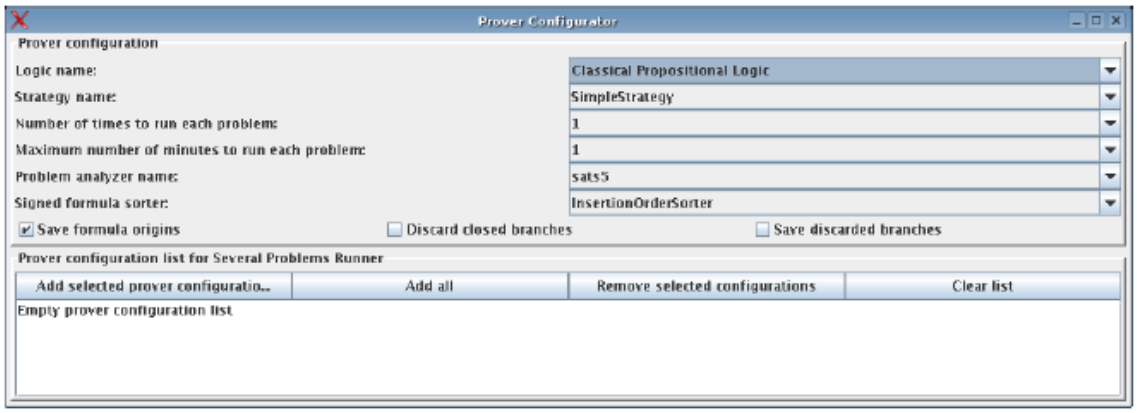

Ilustração 4.12 – Tela de configurações do provador

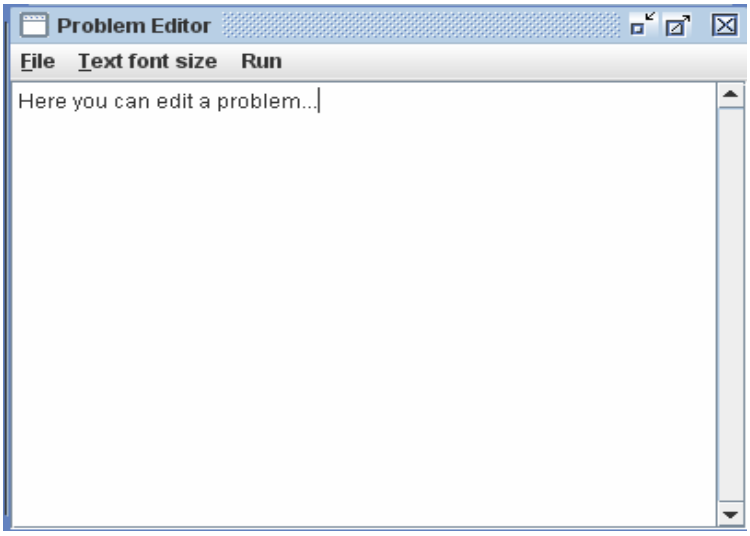

Ilustração 4.13 – Editor do Programa

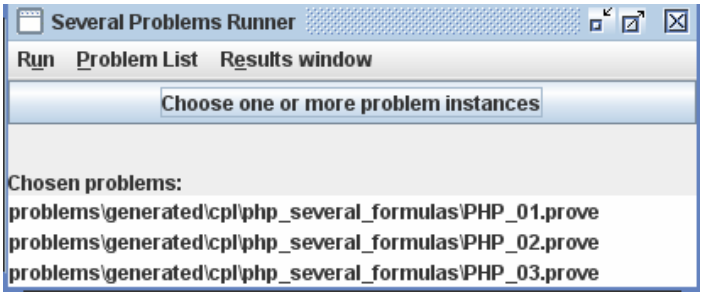

Ilustração 4.14 – Executor de vários problemas

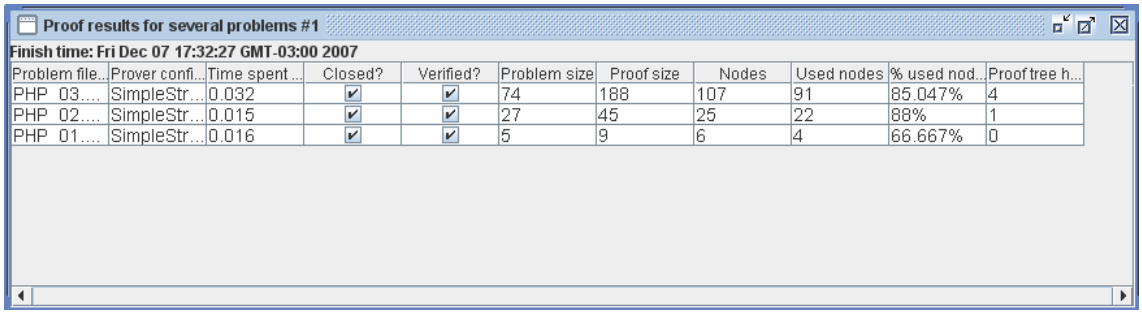

Ilustração 4.15 – Resultados das provas de vários problemas

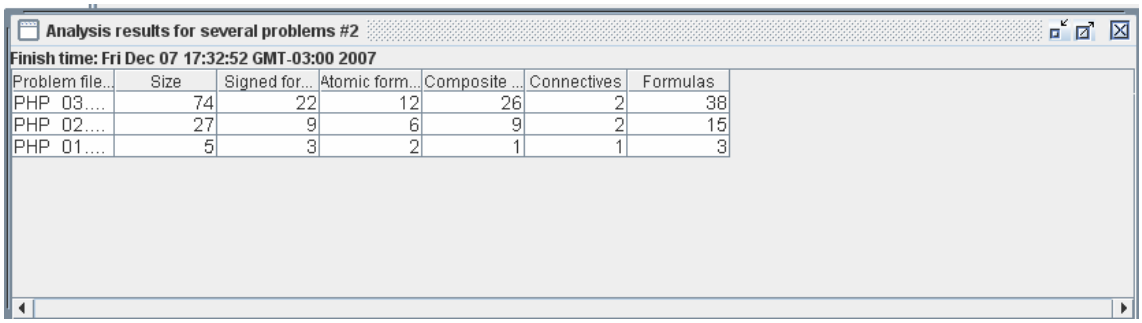

Ilustração 4.16 – Análise dos resultados para muitos problemas

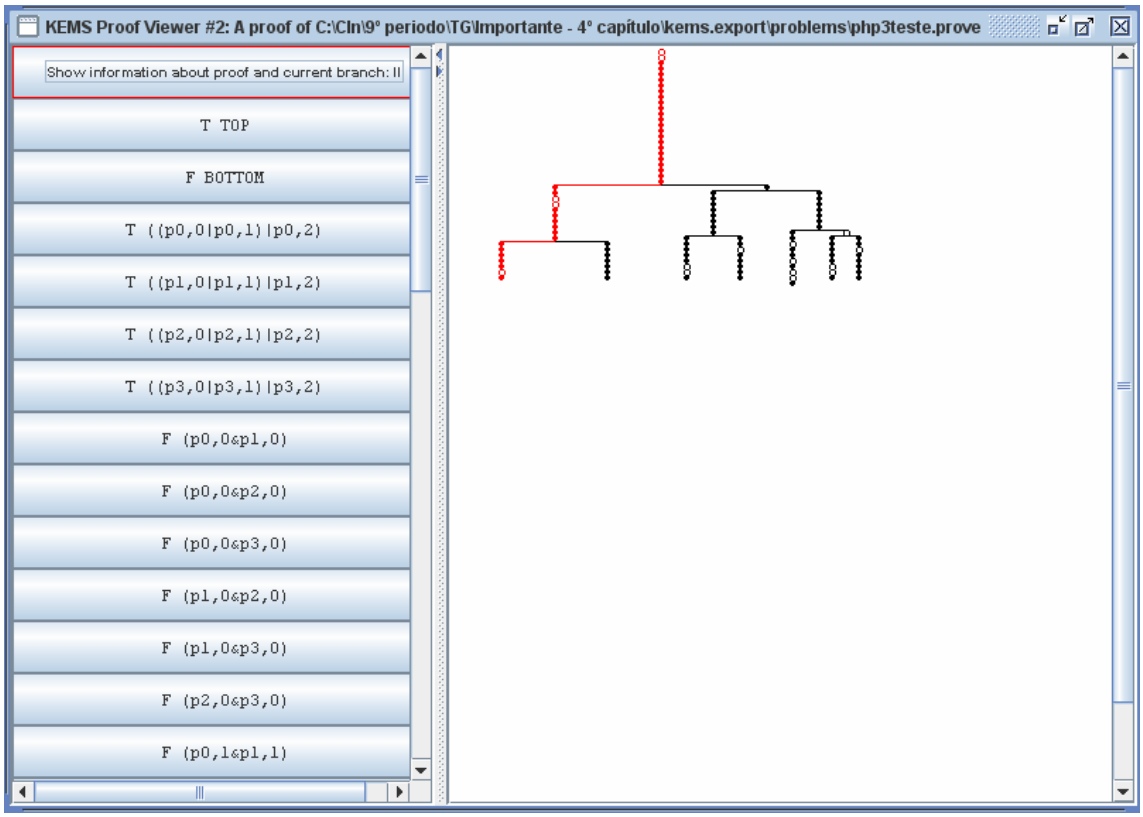

Ilustração 4.17 – A tela de visualização da prova

4.4 Isabelle

Isabelle é um assistente automático de provas genérico. Ele permite que fórmulas matemáticas sejam expressas em linguagem formal e provê ferramentas para provar essas fórmulas em cálculo lógico. Isabelle foi desenvolvido pela “University of Cambridge” (através do PhD em ciência da computação Larry Paulson) e “Technical University of Munich” (através do PhD em ciência da computação Tobias Nipkow) [23].

A principal aplicação é a formalização de provas matemáticas e em particular verificação formal, que inclui provar a correção de hardwares e softwares e provar propriedades de protocolos e linguagens computacionais.

Abaixo seguem algumas das principais características do Isabelle [23]:

- Aplicação pronta para uso em Lógica de alta ordem.
- Ferramentas poderosas de especificação para definições e funções indutivas.
- Automação rica para o raciocínio clássico, lógica equacional e álgebra.
- Excelente apoio notacional.
- Linguagem de prova estruturada, permitindo um texto de prova naturalmente compreensível por nós e pelos computadores.
- Sistema modular para especificações abstratas.
- Numerosas aplicações acessíveis no arquivo de provas formal, resultantes tanto de softwares de engenharia quanto de matemática.
- Boa interface visual para o usuário.
- Ampla documentação, incluindo um tutorial de como usar o sistema.
- Várias interfaces com outros sistemas, incluindo suporte para provadores externos e geração de código para linguagens de programação funcional.
- Framework lógico genérico Isabelle / Pure adequado para uma grande variedade de cálculos formais (por exemplo, conjunto axiomático de teorias) e como uma base para lidar com aplicações com entidades formais.

Comparado com ferramentas semelhantes, a característica que distingue o Isabelle é a sua flexibilidade. A maioria dos assistentes de prova é construída em torno de um único cálculo formal, normalmente de lógica de alta ordem. Isabelle tem capacidade para aceitar uma variedade grande de cálculos formais. A versão distribuída suporta lógica de alta ordem, conjunto axiomático de teorias e vários outros formalismos.

A principal limitação de todos os sistemas de prova é que os teoremas de prova requerem muito esforço de um usuário especialista. Isabelle incorpora algumas ferramentas para melhorar a produtividade do usuário, automatizando algumas partes do processo de prova. Em particular, O raciocinador clássico do Isabelle pode realizar longas cadeias de passos de raciocínio para provar fórmulas. Fatos de aritmética linear são provados automaticamente [23].

Isabelle proporciona excelente suporte notacional: novas notações podem ser introduzidas, utilizando símbolos matemáticos normais. Provas podem ser escritas em uma

notação estruturada com base em estilo de provas tradicionais, ou mais simplesmente como seqüência de comandos.

Ele vem com uma grande biblioteca teórica de matemáticas formalmente verificadas, incluindo a teoria elementar dos números (por exemplo, lei de Gauss da reciprocidade quadrática), de análise (propriedades básicas dos limites, derivadas e integrais), álgebra, e teoria dos conjuntos. Também são fornecidos numerosos exemplos resultantes da investigação em verificações formais [23].

Este provador pode ser visto de duas perspectivas. De um lado, ele pode servir como um framework genérico para prototipação genérica rápida de sistemas dedutivos. Por outro lado, provê um ambiente de prova de teoremas pronto para uso para várias aplicações.

Isabelle foi utilizado para formalizar numerosos teoremas da matemática e da ciência da computação, como o teorema da completude de Gödel, teorema de Gödel sobre a consistência do axioma da escolha, o teorema dos números primos, correção dos protocolos de segurança, e as propriedades das linguagens de programação semântica.

O provador de teoremas Isabelle é um software livre, distribuído sob a licença revisada BSD.

Abaixo seguem algumas imagens do sistema:

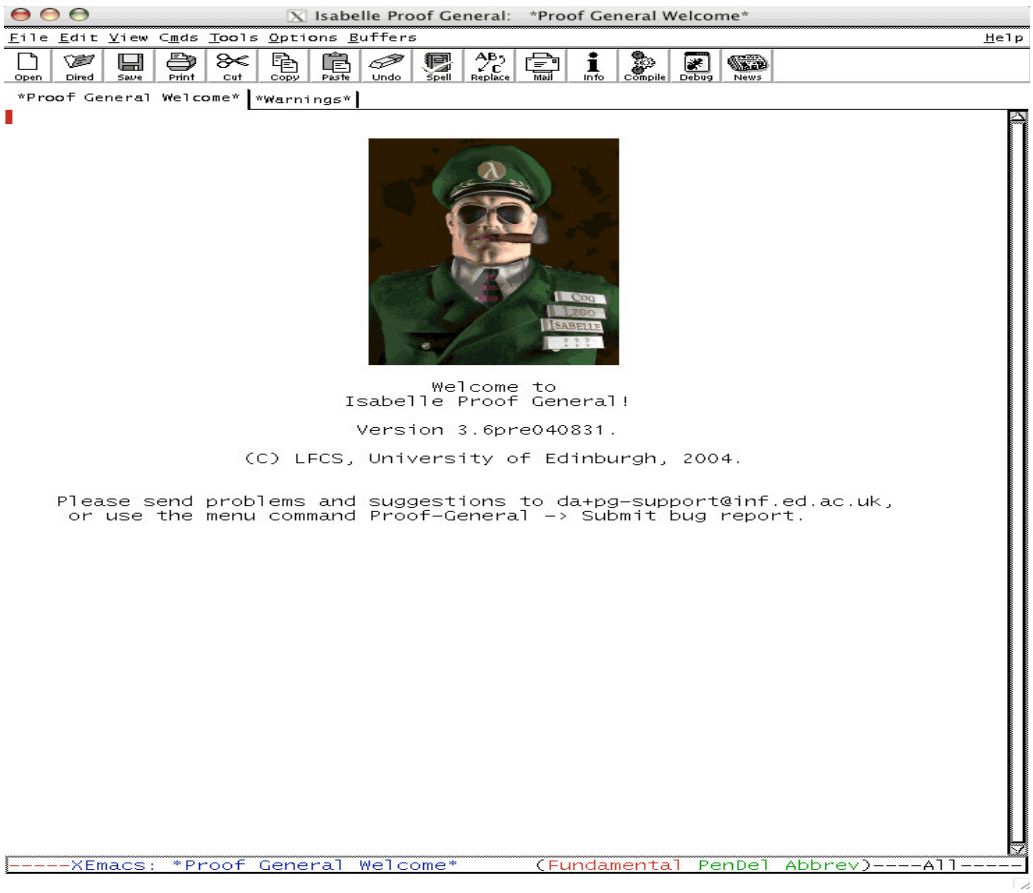

Ilustração 4.18 – Tela inicial do Isabelle

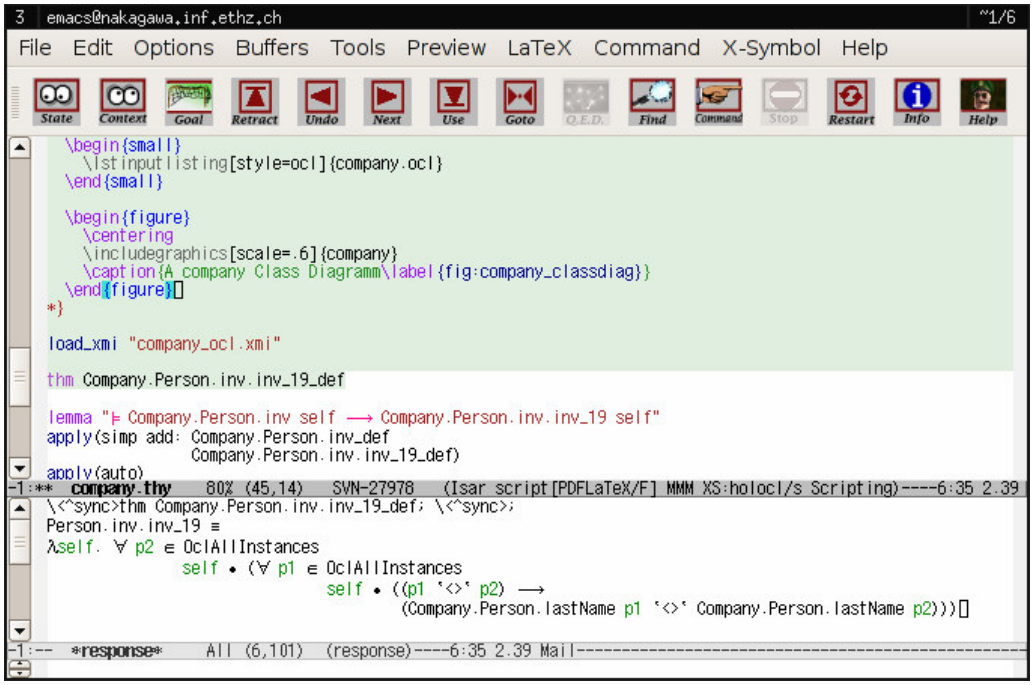

Ilustração 4.19 – Entrada de dados no Isabelle

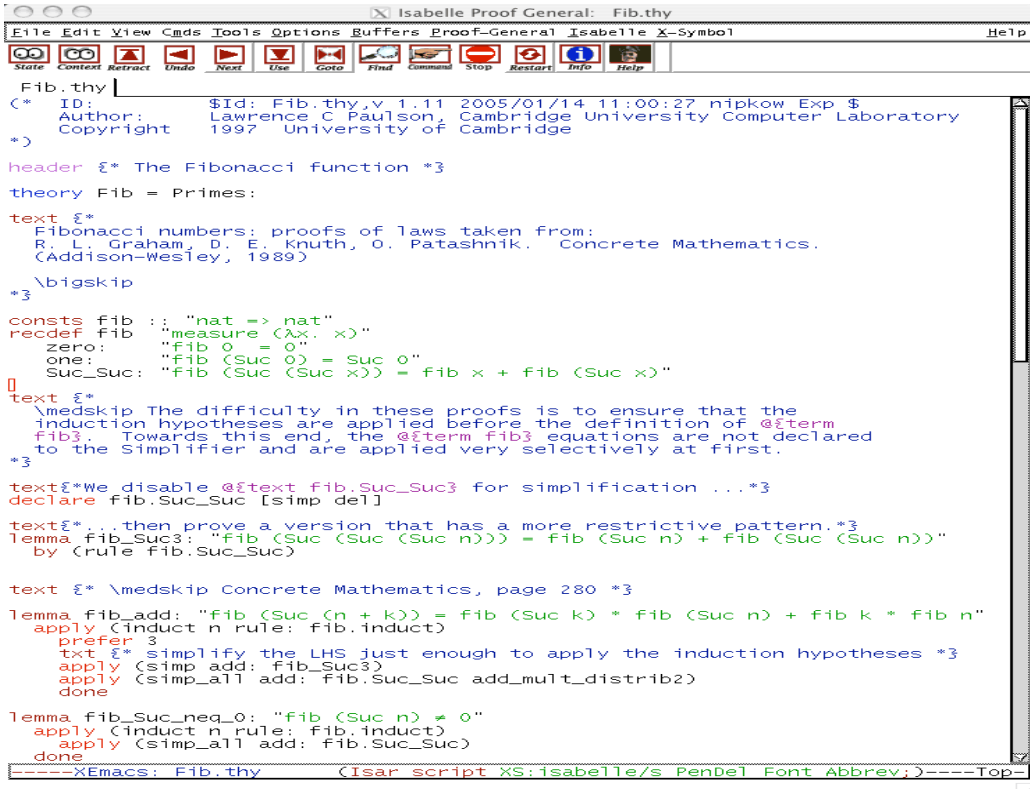

Ilustração 4.20 – Editor de texto do Isabelle

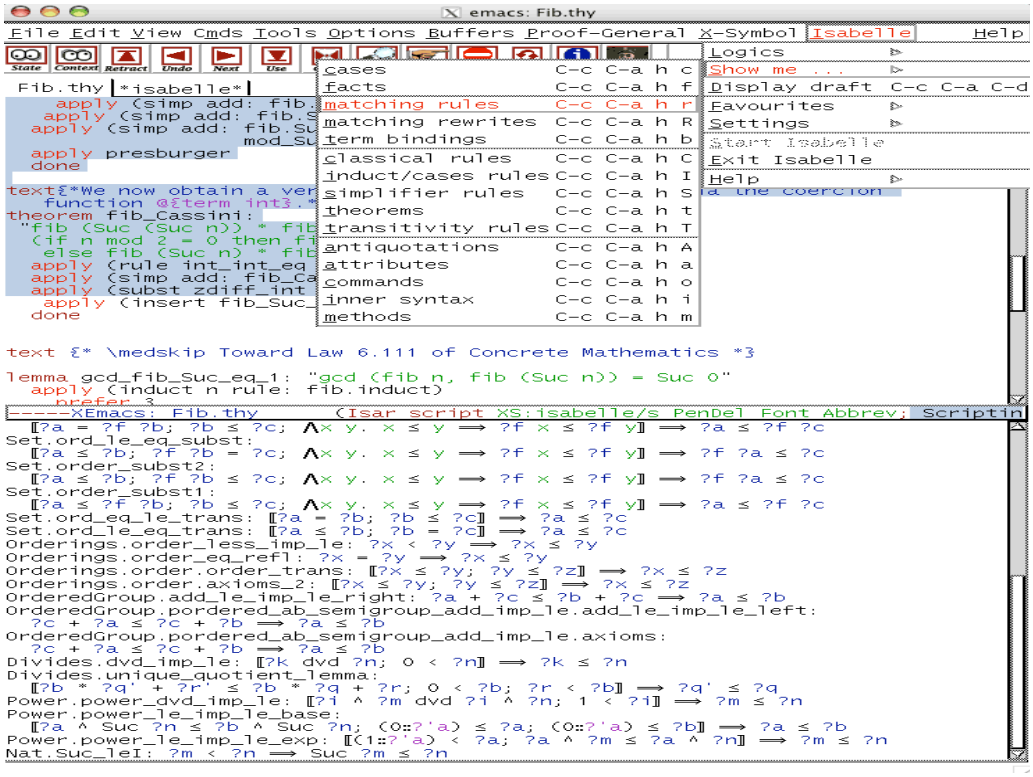

Ilustração 4.21 – Menu e saída de dados no Isabelle

4.5 Ergo

Ergo é um provador automático de teoremas dedicado a verificação de programas. O Ergo é baseado no CC(X), um algoritmo de conclusão de congruência, parametrizado por uma teoria equacional X. Geralmente, CC(X) pode ser instanciado em uma teoria equacional vazia e por aritméticas linear. Ergo contêm tanto um solucionador SAT quanto um mecanismo de instanciação [24].

Abaixo seguem algumas das principais características do Ergo [25]:

- Ergo é um provador de teoremas interativo baseado no cálculo de seqüentes.
- Ergo é implementado em Qu-Prolog – uma extensão de Prolog que provê suporte para objetos variáveis, quantificadores e substituições.
- A versão atual do Ergo vem com cinqüenta teorias predefinidas, aproximadamente 2000 teoremas e muitas táticas predefinidas.
- Contêm uma interface de prova predefinida chamada Gumtree que vem com sua própria linguagem e seu próprio compilador.

A arquitetura do Ergo é altamente modular: cada parte (exceto os “parsers”) do código é descrito por um pequeno conjunto de regras de inferência e é implementada como um módulo (possivelmente parametrizado).

A Ilustração 4.22 descreve a dependência entre os módulos. Cada sintaxe de entrada é manipulada pelo “parser” correspondente. Ambos produzem uma árvore sintática abstrata no mesmo estilo de dados.

Desta forma, existe um único módulo para escrever ambas as sintaxes de entrada. O loop principal consiste em quatro módulos [26]:

- Um eficiente solucionador SAT com “backjumping”, que também monitora os lemas do problema de entrada e aqueles que são gerados durante a execução.
- CC (X), que trata de átomos básicos simulados pelo solucionador SAT. Combina símbolos não interpretados com uma teoria X através de um algoritmo fechado de congruência.
- O módulo de equivalência X em si é implementado pelo módulo parametrizado UF (X), com base em técnicas de busca.
- Um módulo de correspondência constrói instâncias dos lemas contidos no módulo do solucionador SAT e classes de equivalência de CC (X).

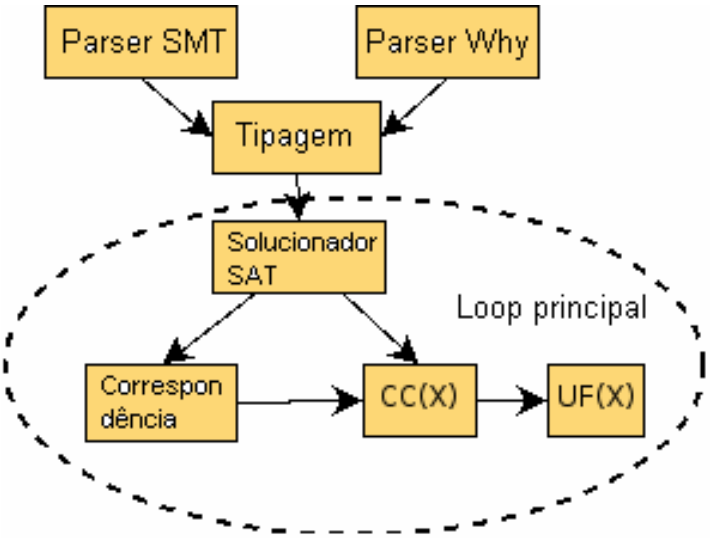

Ilustração 4.22 – Arquitetura do Ergo

Desta forma, pode-se dizer que o Ergo é um novo provador de teoremas para lógica polimórfica de primeira ordem com teorias embutidas. O desenvolvimento começou em janeiro de 2006 na Universidade de Paris e as atuais experiências são muito promissoras com respeito à velocidade e ao número de objetivos automaticamente resolvidos.

Além disso, o Ergo é um software livre que obedece aos termos da licença C-CeCILL.

4.6 aRa

O aRa é um provador automático de teoremas para vários tipos de relações algébricas. Baseia-se no Cálculo de Redução de Predicados de Gordeev para n-variáveis lógicas (RPCn), que permite provas finitas variáveis de primeira ordem [27].

Empregando resultados do Tarski / Givant e Maddux, os desenvolvedores puderam provar a validade de teorias sobre relações de álgebra simples semi-associativas, álgebras relativas e representações de álgebras relacionais [27].

O ARa, que foi implementado em Haskell, oferece diferentes estratégias de redução para RPCn, e um conjunto de simplificações que preservam a provabilidade das n-variáveis.

As relações são indispensáveis em muitas áreas da informática, como a teoria dos grafos, bancos de dados relacionais, a lógica para programação, e semântica de programas de computador, dentre muitos outros. Então, álgebras relacionais - que são extensões da álgebra booleana - formam a base de muitas investigações teóricas. O aRa segue as linhas do Tarski, Maddux e Gordeev por converter equações de várias teorias algébricas relacionais em sentenças de lógica de primeira-ordem com varáveis finitas. Desta forma, o aRa poder aplicar o cálculo de Gordeev RPCn à fórmula transformada.

Ao usar o aRa, muitos teoremas pequenos e médios em diversas álgebras relacionais podem ser provados. Fórmulas que não contenham predicados unitários podem ser trabalhadas eficientemente, outras fórmulas sofrem pelo fato de nem o cálculo RPCn, nem o provador aRa oferecem um tratamento especial de igualdade [27].

Comparado com outras implementações, as diferenças mais notáveis são [27]:

- O procedimento automático de prova utilizando estratégias de redução;
- e a tradução para o cálculo RPCn.

O aRa também pode ser usado para gerar provas em lógica de primeira-ordem e lógicas de variáveis restritas.

5. Conclusão

Neste Trabalho de Graduação, pude estudar o principal fundamento matemático que baseia todos os provadores automáticos de teoremas, o teorema de Herbrand. Além disso, pudemos observar o estado da arte dos provadores e estudar alguns deles, analisando suas características, entendendo como eles funcionam e quais são as suas finalidades.

5.1 Considerações Finais

Como pôde ser visto nesse trabalho, o teorema de Herbrand e toda a teoria que o envolve, foram os responsáveis pela base dos provadores automáticos de teoremas. Graças a essa teoria, Gilmore pôde desenvolver um dos primeiros provadores e com isso deu início aos trabalhos nesta área. Com mudanças e aprimoramentos nos desenvolvimentos foi possível chegar ao estado atual dos provadores.

Também foi apresentada nesse trabalho uma introdução ao CASC, a maior competição que existe atualmente na área de prova automática de teoremas. Com base nessa competição, os campeões atuais das principais classes desta competição e outros que se destacaram na mesma foram apresentados.

Com isso, foi possível analisar mais profundamente as principais características dos mesmos: como eles foram implementados; em que lógica trabalham; quais técnicas utilizam; possíveis falhas desses provadores; e algumas imagens dos mesmos funcionando foram mostradas.

Entretanto esses provadores que competem no CASC se restringem apenas à lógica de primeira ordem. Então, neste trabalho foram apresentados outros provadores, das mais diversas áreas e lógicas, descrevendo as mesmas características e informações que foram citadas acima.

Com isso, o leitor deste trabalho pôde conhecer mais sobre a teoria que envolve os provadores automáticos de teoremas e vislumbrar qual o estado atual dos mesmos, podendo utilizar algum(ns) deste(s) para possíveis trabalhos, aulas, estudos, etc., já que esses provadores podem ser utilizados para as mais diversas finalidades e nas mais diversas áreas de atuação.

5.2 Dificuldades Encontradas

Seguem algumas das dificuldades encontradas durante o desenvolvimento deste trabalho:

- Dificuldade para escolher o escopo do trabalho, já que a área de prova automática de teoremas é bem ampla.
- Dificuldade para encontrar bibliografia, pois apesar de existir muitos provadores, referências bibliográficas não são tão fáceis de encontrar.

- Dificuldade para conseguir o funcionamento de todos os provadores que foram mostrados nesse trabalho, já que a maioria foi desenvolvido em C ou C++ e não vinham com os executáveis. Ou então eram feitos em Haskell ou em outras linguagens de programação. Além disso, a maioria deles só podia ser compilada no sistema operacional Linux.

5.3 Trabalhos Futuros

Como foi visto anteriormente, existem provadores automáticos de teoremas de vários tipos: web, em java, em Haskell, etc. e em várias lógicas: de primeira-ordem, proposicional, linear, intuicionista, clássica, modal, etc. que se utilizam dos mais variados métodos: tableux, cálculo de seqüentes, anéis booleanos, dedução natural, etc..

Além disso, esses provadores podem ser utilizados nas mais variadas áreas: matemática, lógica, álgebra, saúde, entre outras.

Desta forma, um trabalho que pode ser realizado futuramente seria a implementação de um provador automático de teoremas se utilizando das teorias aqui aprendidas. Alternativamente, pode-se desenvolver um provador automático de teoremas que utilize um método novo, baseando-se nas pesquisas realizadas para a elaboração deste documento.

Desta maneira, tudo que foi aprendido com a teoria que embasou os provadores de teoremas e com a análise de alguns dos principais provadores existentes atualmente, poderá ser o ponto de partida de um projeto mais amplo, como por exemplo uma dissertação de mestrado.

Um objetivo a ser alcançado a longo prazo incluiria o desenvolvimento de um provador automático de teoremas para competir em uma das classes do CASC, que foram apresentadas no capítulo 3, e com isso testar os conhecimentos aprendidos e obter reconhecimento da comunidade acadêmica.

Já uma perspectiva mais a curto e médio prazo seria fazer um estudo comparativo mais profundo com alguns dos provadores acima citados, como por exemplo, uma análise comparativa entre os provadores que foram campeões em suas respectivas classes no CASC.

Apêndice 1

Forma normal prenex e Forma normal conjuntiva

No Cálculo de Predicados existe uma forma normal chamada de forma normal prenex - FNP. O fato de uma fórmula estar na forma prenex simplifica procedimentos de manipulação da fórmula. A FNP é importante na medida em que sua obtenção é um passo necessário para a obtenção da forma normal conjuntiva - FNC, que é utilizada pelo método da resolução [28].

Desta forma pode-se dizer que uma dada fórmula G está na FNP se e somente se ela está na seguinte forma:

$$(Q_1X_1)(Q_2X_2)\ldots(Q_nX_n)\ (M)$$

onde cada (Q_iX_i), i = 1,…,n é $(\forall X_i)$ ou $(\exists X_i)$, e M é uma fórmula livre de quantificadores, chamada de matriz. A parte $(Q_1X_1)(Q_2X_2)\ldots(Q_nX_n)$ é chamada de prefixo da fórmula. Quando todos os Q_i são quantificadores universais, a fórmula em questão é chamada de universalmente fechada.

Além disso, diz-se que uma fórmula G está na FNC se e somente se estiver na FNP e sua matriz for uma conjunção de disjunções de fórmulas atômicas, negadas ou não.

Para pôr uma fórmula G na forma normal prenex, deve-se mover todos os quantificadores à esquerda dos conectivos lógicos $\neg$, $\wedge$, v, $\rightarrow$. Se a fórmula contiver algum conectivo $\leftrightarrow$, então antes de realizar este processo, devemos eliminar este conectivo da fórmula usando a redução [29]:

$$\alpha \leftrightarrow \beta \ \approx (\alpha \rightarrow \beta) \ \wedge \ (\beta \rightarrow \alpha)$$

Para mover cada um dos quantificadores à esquerda de cada tipo destes conectivos devemos usar uma das reduções a seguir [29]:

1. $\neg\exists x.\alpha \twoheadrightarrow \forall x.\neg\alpha$
2. $\neg\forall x.\alpha \twoheadrightarrow \exists x.\neg\alpha$
3. $\forall x.\alpha \wedge \beta \twoheadrightarrow \forall y.(\alpha[y/x] \wedge \beta)$
4. $\exists x.\alpha \wedge \beta \twoheadrightarrow \exists y.(\alpha[y/x] \wedge \beta)$
5. $\forall x.\alpha \vee \beta \twoheadrightarrow \forall y.(\alpha[y/x] \vee \beta)$
6. $\exists x.\alpha \vee \beta \twoheadrightarrow \exists y.(\alpha[y/x] \vee \beta)$
7. $\beta \wedge \forall x.\alpha \twoheadrightarrow \forall y.(\beta \wedge \alpha[y/x])$
8. $\beta \wedge \exists x.\alpha \twoheadrightarrow \exists y.(\beta \wedge \alpha[y/x])$
9. $\beta \vee \forall x.\alpha \twoheadrightarrow \forall y.(\beta \vee \alpha[y/x])$
10. $\beta \vee \exists x.\alpha \twoheadrightarrow \exists y.(\beta \vee \alpha[y/x])$
11. $\forall x.\alpha \rightarrow \beta \twoheadrightarrow \exists y.(\alpha[y/x] \rightarrow \beta)$
12. $\exists x.\alpha \rightarrow \beta \twoheadrightarrow \forall y.(\alpha[y/x] \rightarrow \beta)$
13. $\alpha \rightarrow \forall x.\beta \twoheadrightarrow \forall y.(\alpha \rightarrow \beta[y/x])$
14. $\alpha \rightarrow \exists x.\beta \twoheadrightarrow \exists y.(\alpha \rightarrow \beta[y/x])$

Ilustração A.1 – Reduções de quantificadores

Segue abaixo alguns exemplos de fórmulas na FNP:

- $(\forall X)(\forall Y)\ (p(X,Y) \wedge q(Y))$
- $(\forall X)(\forall Y)\ (\neg p(X,Y) \rightarrow q(Y))$
- $(\forall X)(\forall Y)(\exists Z)\ (q(X,Y) \rightarrow r(Z))$

Portanto, visto a maneira de como se chegar a uma fórmula na forma FNP, facilmente pode-se chegar a uma fórmula na forma FNC, que nada mais é do que uma fórmula na FNP e sua matriz é uma conjunção de cláusulas. Sendo uma forma normal, a FNC é útil em provas automáticas de teoremas. Segue um exemplo de uma fórmula na FNC:

- $(\forall X)(\exists Y)\ ((\neg p(X,Y) \vee q(X)) \wedge (\neg p(X,Y) \vee q(Y)))$.

A fórmula que segue é uma gramática formal para FNC [30]:

1. <or> → ∨
2. <e> → ∧
3. <não> → ¬
4. <conjunção> → <disjunção>
5. <conjunção> → <conjunção> <ou> <disjunção>
6. <disjunção> → <literal>
7. <disjunção> → (<disjunção> <e> <literal>)
8. <literal> → <termo>
9. <literal> → <não><termo>

onde <termo> é qualquer variável.

A FNC pode ser levada adiante de modo a produzir a forma normal clausal de uma fórmula lógica, a qual é usada para se efetuar resolução de primeira ordem.

Método da multiplicação

O método da multiplicação consiste de, para cada S_i' (conjunto de instâncias básicas de cláusulas no conjunto S que se quer provar) produzido pelo procedimento mecânico, S_i' é multiplicado por sua forma normal conjuntiva [3].

Cada disjunção na forma normal disjuntiva que contêm um par complementar é removida. Quando algum S_i' for vazio, então Si' é insatisfatível e a prova é encontrada.

Exemplo:

S= {P(X), ~P(a)}

H_0 = {a}

S_0' = P(a) ∧ ~P(a) = □

desta forma, é provado que S é insatisfatível.

Mas o método da multiplicação é ineficiente. Pode-se ver facilmente que, por exemplo, em um pequeno conjunto de dez cláusulas básicas de dois literais, existem 210 conjunções. Por isso, o método da multiplicação não é o melhor método para ser usado nas provas automáticas de teoremas.

Apêndice 2

Prova de teoremas por resolução e paramodulação

Para falar do Vampire, é necessário falar antes de prova de teoremas por resolução e paramodulação.

Definição 20 [11]: Prova refutacional – O Vampire tem o interesse de resolver mecanicamente problemas do tipo: dado uma formula objetivo X e uma teoria de primeira-ordem representado por um conjunto finito de axiomas ϕ1, ... , ϕn, mostre que um objetivo é implicado pela teoria. Semanticamente isso significa que em qualquer modelo de ϕ1, ... , ϕn, isto é, uma interpretação que faz todas essas fórmulas verdadeiras, a meta X também é verdade. Uma prova por refutação resolve esse tipo de problema mostrando que o conjunto ϕ1, ... , ϕn, ¬X é insatisfatível, isto é, não tem modelo.

Definição 21 [11]: Resolução – O Vampire é baseado no cálculo de resolução, introduzido por J.A. Robinson. As duas regras do cálculo de resolução são as seguintes:

1. $\underline{C \vee A \quad D \vee \neg B}$ (Resolução Binária)

   $(C \vee D)\,\theta$

   onde θ é o unificador mais geral dos átomos A e B.

2. $\underline{C \vee L_1 \vee ... \vee L_n}$ (Fatoração)

   $(C \vee L_1)\,\theta$

   onde θ é o unificador mais geral de todos os literais L_i.

O cálculo da resolução é refutacionalmente completo, isto é, suas regras são suficientes para derivar a cláusula vazia de qualquer conjunto de cláusulas insatisfatível.

Definição 22 [11]: Paramodulação – Formulações de vários problemas de prova interessantes em lógica de primeira ordem envolvem igualdade. Genericamente, é possível resolver esses problemas por resolução considerando-se igualdade como um predicado binário ordinário e axiomas necessários para isso.

Na prática, isto não funciona porque os axiomas de igualdade criam um aumento da supressão de muitas inferências. Para resolver esse problema, foi criada a seguinte regra de paramodulação:

$\underline{C \vee s \approx t \; D[u]}$ (Paramodulação)

$(C \vee D[t])\,\theta$

onde θ é o unificador mais geral dos termos s e u.

É possível evitar totalmente a aplicação da regra da resolução aos literais de igualdade e o uso de axiomas de igualdade, melhorando-se o cálculo da resolução com paramodulação e com a regra de resolução reflexiva abaixo [11]:

$\underline{C \vee s \neq t}$ (resolução reflexiva)

$C\,\theta$

onde θ é o unificador mais geral dos termos s e t.

Método da saturação

Se o interesse é achar uma refutação de um conjunto de cláusulas em cálculos baseados em resolução e paramodulação, pode-se fazer isso saturando o conjunto com todas as inferências possíveis no cálculo escolhido [11].

Em cada passo do processo de saturação, é escolhida uma inferência no cálculo, que pode ser construída por algumas cláusulas do conjunto corrente. O novo conjunto é obtido pela adição da cláusula resultante da inferência no conjunto corrente.

Ao encontrar uma cláusula vazia em algum dos passos, isto indica que o conjunto de cláusulas inicial é insatisfatível.

Além disso, se o conjunto inicial é insatisfatível, é garantido que será encontrada a refutação eventualmente, provando que o cálculo é refutacionalmente completo e o método usado para organizar a seleção de inferências em cada passo é justo, isto é, qualquer inferência disponível é eventualmente desenvolvida.

Complemento de Knuth-Bendix

Knuth e Bendix introduziram um algoritmo de complemento que tenta derivar um conjunto de regras convergentes de um dado conjunto de equações. Com a extensão sugerida pelos dois para um algoritmo de complemento, esse método passou a ser muito importante para provadores de teoremas em teorias equacionais.

As nove regras de inferências formam uma variante do complemento que é conveniente para o processamento das equações. O sistema de inferência trabalha com triplas (E, R, G) de conjuntos de pares de termos (equações, regras e objetivos). O símbolo $\succ$ denota uma ordem de redução arbitrária que é total em termos básicos equivalentes e o símbolo $\rhd$ que representa a ordem de especialização (s $\rhd$ t se e somente se algum sub-termo de s é uma instância de t, mas não vice-versa). Abaixo seguem as nove regras [31]:

Orientação

$$\frac{E \cup \{s \doteq t\}, R, G}{E, R \cup \{s \to t\}, G} \quad \text{se} \quad s \succ t$$

Geração

$$\frac{E, R, G}{E \cup \{s = t\}, R, G} \quad \text{se} \quad s = t \in CP(R, E)$$

Simplificando uma equação

$$\frac{E \cup \{s \doteq t\}, R, G}{E \cup \{u \doteq t\}, R, G} \quad \text{se} \quad s \to_R u \text{ ou } s \to_{E_\succ, l=r} u \text{ com } s \rhd l$$

Deletando uma equação

$$\frac{E \cup \{s = s\}, R, G}{E, R, G}$$

Agrupando uma equação

$$\frac{E \cup \{s \doteq t, u[\sigma(s)] \doteq u[\sigma(t)]\}, R, G}{E \cup \{s \doteq t\}, R, G}$$

Simplificando o lado direito de uma regra

$$\frac{E, R \cup \{s \to t\}, G}{E, R \cup \{s \to u\}, G} \text{ se } t \to_{R \cup E_{\succ}} u$$

Simplificando o lado esquerdo de uma regra

$$\frac{E, R \cup \{s \to t\}, G}{E \cup \{u = t\}, R, G} \text{ se } s \to_{R \cup E_{\succ}, l=r} u \text{ with } s \rhd l$$

Simplificando um objetivo

$$\frac{E, R, G \cup \{s \doteq t\}}{E, R, G \cup \{u \doteq t\}} \text{ se } s \to_{R \cup E_{\succ}} u$$

Sucesso

$$\frac{E, R, G \cup \{s = t\}}{SUCCESS} \text{ se } s \equiv t$$

Ilustração A.2 – As nove regras de inferências do Complemento de Knuth-Bendix

Apêndice 3

Uma introdução à lógica modal

A lógica modal faz parte da pesquisa atual em diversas áreas da ciência da computação. Encontram-se algumas aplicações na área de inteligência artificial, representação de conhecimento e dedução automática, especificação formal de sistemas, engenharia de software e lingüística computacional.

A lógica modal pode ser encarada como uma extensão da lógica proposicional. Grande parte das lógicas modais tiveram origem em uma lógica "fraca", conhecida como lógica K, que leva este nome em homenagem a Saul Kripke por sua contribuição. Um modelo de Kripke é uma tripla m = <Wm,Rm,hm> tal que [32]:

- Wm é um conjunto não vazio dos mundos possíveis de m;
- Rm C Wm x Wm representa a relação de acessibilidade de m;
- hm : v → ρ(Wm) é uma função que estabelece um valor de verdade arbitrário para cada fórmula atômica da linguagem e um valor para cada fórmula molecular em vista dos valores das fórmulas atômicas.

A lógica modal é bastante utilizada na análise semântica, visto que as representações dos conectivos modais permitem expressar advérbios, dentre os quais a lógica clássica não pode representar.

Uma compreensão da lógica modal é particularmente valiosa na análise formal de argumento filosófico onde expressões da família modal são comuns e confusas. Trata-se da lógica do "é necessário que" (representado por "[]") e do é "possível que" (representado por "<>"). Portanto, não considera apenas a veracidade e a falsidade das proposições como se apresentam, mas como seriam se fossem diferentes [20].

Como um operador pode ser derivado do outro, pode-se manter uma representação de apenas um deles e fazer uma transformação na expressão trabalhada sempre que se encontra o outro. Há algumas variações de lógica modal, dependendo de quais axiomas são incluídos no conjunto de axiomas básicos (da lógica proposicional) [20].

Há outros operadores lógicos que podem ser derivados dos já definidos (os quatro da lógica proposicional, mas os dois acima citados). Por exemplo, o 'ou-exclusivo'. Apesar de não ter uma notação padrão, é comum representá-lo por $\oplus$. A regra do ou-exclusivo é se duas fórmulas f1 e f2 são ambas verdadeiras ou ambas falsas, f1$\oplus$f2 é falso. Caso contrário é verdadeiro [20].

Esta lógica permite analisar não só o que dizem as coisas no mundo, mas o que diriam em um mundo alternativo; não factual, mas possível. Isto é, se interessa pelas verdades e falsidades que são geradas por asserções neste mundo real e em outros possíveis mundos, visto que se chama de mundo possível uma situação contra-factual que não aconteceu, mas poderia ter acontecido.

Neste sentido, uma proposição será necessária em um mundo se ela é verdadeira em todos os possíveis mundos relacionados com este, e possível em um mundo se essa é verdadeira em pelo menos um daqueles mundos relacionados a este.

Princípios básicos de anéis booleanos

Anéis são entidades matemáticas definidas por uma tripla <C, +, . >, onde C é o conjunto de valores existentes no anel, '+' e '.' são operadores binários, definidos por funções [20]:

- '+' : C × C → C;
- '.' : C × C → C.

Sejam x, y, z Є C. Os anéis respeitam as seguintes propriedades [20]:

1. x.0 = 0
2. x.1 = x
3. x + 0 = x
4. x.y = y.x
5. (x.y).z = x.(y.z)
6. x + y = y + x
7. (x + y) + z = x + (y + z)
8. x.(y + z) = x.y + x.z

Anéis booleanos integram um conjunto restrito de anéis, onde C = {0, 1}. Assim sendo, pode-se provar as seguintes propriedades sobre anéis booleanos (além das demais propriedades apresentadas) [20]:

1. x.x = x
2. x + x = 0

Desta forma, a operação '.' é análoga à conjunção e a operação '+' é análoga ao ou-exclusivo, o que permite a tradução de expressões lógicas em expressões aritméticas e, portanto, o uso de anéis booleanos para o cálculo de satisfatibilidade de seqüentes [20].

NP-completude, consumo exponencial de tempo e de memória

Uma das áreas da ciência da computação é o estudo de complexidade de algoritmos. Um dos resultados mais importantes para esta área foi provado por Alan Turing, relacionado ao que se conhece por problema da parada. Em resumo, o que foi provado é que há problemas que podem ser descritos inteiramente de forma matemática, mas que nenhum computador é capaz de resolvê-los [20].

Dentre os problemas resolvíveis, estuda-se a complexidade computacional de seus algoritmos de resolução. A classe P é o conjunto dos problemas para os quais existem algoritmos que os resolvam, polinomiais no tamanho da entrada [20].

Um problema D é dito de decisão se sua resposta é r, r Є {SIM, NÃO}. Seja D um problema e A um algoritmo que resolve D.

Uma justificativa polinomial j é um elemento de tamanho polinomial na entrada de A, capaz de justificar a resposta SIM, quando A gerar tal resposta. Caso a verificação de j possa ser feita em tempo polinomial no tamanho da entrada, então se diz que D pertence à classe NP.

Podem-se fazer algumas observações quanto às implicações desta definição [20]:

- Não se exige uma solução polinomial para os problemas da classe NP;
- P C NP, pois se D Є P, então existe um algoritmo polinomial que apresenta a solução de D, e que pode ser utilizado como algoritmo de verificação para uma justificativa. Logo, D Є NP;
- P é a subclasse de "menor dificuldade" de NP. A subclasse de "maior dificuldade" de NP é a classe dos problemas NP-completos. Intuitivamente, se uma solução polinomial for encontrada para um problema NP-completo, então todo problema de NP também admite solução polinomial;
- Não se sabe se P = NP. Não foi encontrado até o momento nenhum algoritmo polinomial para nenhum dos problemas NP-completos conhecidos, mas também não foi provado que não existe tal algoritmo.

Este estudo é importante ao trabalho realizado, pois foi demonstrado que o SAT Є NP e atualmente só se conhece algoritmos que consomem tempo exponencial no tamanho da entrada para resolvê-lo. Vários destes algoritmos são polinomiais para um subconjunto de instâncias do SAT, mas todos apresentam algum contra-exemplo para o qual rodam em tempo exponencial. Além disso, alguns deles apresentam uma característica indesejável: consomem espaço de memória exponencial no tamanho da entrada [20].

Método de tableaus KE

A maior parte dos provadores automáticos de teoremas hoje em dia são baseados ou no princípio da resolução ou no procedimento de Davis-Logemann-Loveland.

Porém, outros métodos também podem ser utilizados. Os métodos baseados em tableaus são particularmente interessantes para a prova automática de teoremas por existirem em diferentes variedades e para várias lógicas. Além disso, estes métodos não exigem a conversão dos problemas para a forma clausal.

Um dentre os muitos métodos lógicos que podem ser utilizados para resolver o problema da satisfatibilidade é o sistema KE. É um método de tableaux originalmente desenvolvido para lógica clássica por Marco Mondadori e Marcello D'Agostin, mas que foi estendido para outros sistemas lógicos. O sistema KE foi apresentado como uma melhoria, no aspecto computacional, em relação ao sistema de tableaux analíticos [22].

Apesar de parecido com o sistema de tableaux analíticos, o sistema KE é um sistema refutacional que não é afetado pelas anomalias dos sistemas livres de corte.

A principal motivação para o desenvolvimento desse método foi obter um método em série com princípios clássicos. A segunda motivação foi desenvolver um sistema computacionalmente mais eficiente [22].

Lógicas Linear (LL), Clássica(LK) e Intuicionista (LJ)

Lógica Linear: A lógica linear, introduzida por Girard em 1987, se tornou bastante popular na comunidade de ciência da computação. A grande novidade é a existência de novos conectivos que formam um novo sistema lógico com várias características interessantes, como por exemplo, a possibilidade de se interpretar um seqüente como um estado de um sistema, assim como o tratamento de uma fórmula como um recurso [33].

Um outro aspecto considerado na literatura como bastante inovador é a noção de *redes-de-prova* (do inglês, *proof-nets*). A idéia é trabalhar com uma representação "grafo-teórica" para as deduções na lógica linear. Girard define *estruturas-de-prova*, *i.e.* grafos de prova, chamados por ele de "a dedução natural do cálculo de seqüentes linear", utilizando a noção de *links*, que são relações entre ocorrências de fórmulas. Aquelas *estruturas-de-prova* que representam provas logicamente corretas são chamadas de *redes-de-prova*. Um dos aspectos talvez mais interessantes da teoria de redes-de-prova é a possibilidade de caracterizar dentro da classe de estruturas-de-prova a subclasse de *redes-de-prova* através de critérios puramente "*geométricos/algébricos*", *i.e,.* baseados apenas na estrutura dos grafos de prova [33].

Lógica Clássica: Considerada como o núcleo da lógica dedutiva. É o que é chamado hoje de cálculo de predicados de 1ª ordem com ou sem igualdade e de alguns de seus subsistemas.

Dois princípios (entre muitos outros) regem a Lógica Clássica: da *não-contradição* e do *terceiro excluído.* A lei da não-contradição diz que nenhuma afirmação pode ser verdadeira e falsa ao mesmo tempo e a lei do terceiro excluído diz que uma afirmação deve ser ou verdadeira ou falsa. Combinadas, estas duas leis requerem dois valores de verdade que são mutuamente exclusivos. Uma afirmação pode ser falsa ou verdadeira, mas, de modo algum, pode ser verdadeira e falsa ao mesmo tempo [35].

A lógica clássica abrange a lógica proposicional e a lógica de predicados. Na lógica clássica, a lógica proposicional está integrada na lógica de predicados [34].

Lógica Intuicionista: Na lógica clássica as afirmativas são ou falsas ou verdadeiras, uma vez que vale o princípio do terceiro excluído.

A lógica intuicionista abandona a idéia de verdade absoluta, e as afirmativas são consideradas válidas se e somente se existe uma prova construtiva da mesma. Ou seja, o princípio do terceiro excluído não é mais uma tautologia uma vez que não existe um método construtivo único que prove qualquer proposição ou a sua negação.

A implementação da lógica intuicionista é um pouco mais trabalhosa do que a da lógica clássica por causa de algumas regras, principalmente a regra de implicação à esquerda.

Lógicas mbC e mCi

A versão atual do KEMS implementa estratégias para três sistemas lógicos: lógica clássica proposicional, mbC e mCi. mbC e mCi são lógicas para-consistentes. As lógicas para-consistentes podem ser usadas para representar teorias inconsistentes, porém não triviais [22].

Essas duas lógicas são de uma classe especial de lógicas para-consistentes, as Lógicas de Inconsistência Formal uma família de lógicas para-consistentes que internalizam as noções de consistência e inconsistência no nível de linguagem-objeto.

Esta família de lógicas tem algumas propriedades interessantes em sua teoria da prova e foi utilizada em algumas aplicações em ciência da computação, como na integração de informação inconsistente em bases de dados.

A lógica mCi é uma extensão da lógica mbC. E ambas as lógicas possuem dois operadores diferentes dos convencionais: consistência (○) e inconsistência (●).

Referências

[1] Wikipedia – Métodos formais. Disponível em: < http://pt.wikipedia.org/wiki/M%C3%A9todos_formais#Provadores_de_teoremas>. Acesso em: 04 out. 07.

[2] ISOTANI, Seiji. **Desenvolvimento de ferramentas no iGeom: utilizando a geometria dinâmica no ensino presencial e a distância**. Publicado em Abril de 2005. Dissertação de Mestrado - IME-SP, São Paulo. Disponível em: < http://www.ei.sanken.osaka-u.ac.jp/~isotani/mestrado/dissertacao.pdf >. Acesso em: 30 nov. 07

[3] CHANG, chin-liang; LEE, richard char-tung, **Simbolic Logic and Mechanical Theorem Proving**, Academic Press, 1973.

[4] Lógica de Primeira Ordem. Notas de aula da disciplina de Lógica Computacional. Disponível em: <http://twiki.di.uminho.pt/twiki/pub/Education/LC/MaterialApoio/LogPO.pdf>. Acesso em: 05 dez. 07

[5] NICOLETTI, Maria do Carmo. **Provando que um Conjunto de Cláusulas é insatisfatível**. Notas de aula do curso de introdução à lógica da Universidade Federal de São Carlos. Disponível em: <http://www.dc.ufscar.br/~carmo/notas_curso/herbrand.doc>. Acesso em: 30 nov. 07

[6] Conference on Automated Deduction (CADE). Disponível em: < http://www.cadeconference.org/>. Acesso em: 30 set. 07.

[7] Lógica de Predicados. Transparências da Aula da disciplina de Lógica para Computação do CIn - UFPE. Disponível em: <http://www.cin.ufpe.br/~if673/1-2006/Log14.ppt>. Acesso em: 05 dez. 07

[8] The TPTP Problem Library for Automated Theorem Proving. Disponível em: <http://www.cs.miami.edu/~tptp/>. Acesso em: 07 dez. 07

[9] The CADE ATP System Competition. Disponível em: <http://www.cs.miami.edu/~tptp/CASC/21/>. Acesso em: 07 dez. 07

[10] E-SETHEO. Disponível em: <http://www4.informatik.tu-muenchen.de/~schulz/WORK/e-setheo.html>. Acesso em: 07 dez. 07

[11] Riazanov, Alexandre. **Implementing an Efficient Theorem Prover**. 2003. Tese de doutorado – Department of computer science, Faculty of Science and Engineering, University of Manchester, Manchester. Disponível em: < http://www.freewebs.com/riazanov/Riazanov_PhD_thesis.pdf/>. Acesso em: 08 dez. 07.

[12] Wikipedia – Vampire Theorem Prover. Disponível em: < http://en.wikipedia.org/wiki/Vampire_theorem_prover/>. Acesso em: 08 dez. 07.

[13] Wikipedia – Paradox (Theorem Prover). Disponível em: < http://en.wikipedia.org/wiki/Paradox_theorem_prover/>. Acesso em: 08 dez. 07.

[14] Claessen, Koen; Sorensson, Niklas. **New Thecniques that Improve MACE-style Finite Model Finding**. Artigo sobre o Paradox. Disponível em: < http://www.cs.chalmers.se/~koen/pubs/entry-model-paradox.html/>. Acesso em: 08 dez. 07.

[15] Site oficial do Paradox. Disponível em: < http://www.cs.chalmers.se/~koen/folkung/>. Acesso em: 08 dez. 07.

[16] Site do Darwin. Disponível em: < http://combination.cs.uiowa.edu/Darwin/>. Acesso em: 09 dez. 07.

[17] BAUMGARTNER, Peter; FUCHS, Alexander; TINELLI, Cesare. **Darwin: A Theorem Prover for the Model Evolution Calculus**. Artigo sobre o Darwin. Disponível em: < http://combination.cs.uiowa.edu/Darwin/papers/darwin-ESFOR.pdf>. Acesso em: 09 dez. 07.

[18] HILLENBRAND, Thomas; JAEGER, Andreas; LOCHNER, Bernd. **System Description: Waldmeister – Improvements in Perfomance and Ease of use**. Artigo sobre o Waldmeister. Disponível em: < http://citeseer.ist.psu.edu/hillenbrand99system.html>. Acesso em: 12 dez. 07.

[19] BUCH, Arnim; HILLENBRAND, Thomas; VOGT, Roland; LOCHNER, Bernd. **WALDMEISTER – High-Performance Equational Deduction**. Artigo sobre o Waldmeister. Disponível em: < http://citeseer.ist.psu.edu/555570.html >. Acesso em: 12 dez. 07.

[20] TISOVEC, F.A.C., **Um Provador Automático de Teoremas para a Lógica**

Modal, Baseado em Anéis Booleanos. Janeiro de 2007. Trabalho de Graduação do Instituto de Matemática e Estatística – IME, São Paulo. Disponível em: < http://www.linux.ime.usp.br/~cef/mac499-06/monografias/tisovec/ >. Acesso em: 03 out. 07.

[21] PIMENTEL, G. E. **PLLIC – Provador para as Lógicas Linear, Intuicionista e Clássica** . Relatório de apresentação do provador automático de teoremas PLLIC. Disponível em: <http://www.pllic.org/>. Acesso em: 03 out. 07.

[22] NETO, A. G. S. S. **Um Provador de Teoremas Multi-Estratégia**. Março de 2007. Tese de doutorado do Instituto de Matemática e Estatística – IME, São Paulo. Disponível em: < http://www.teses.usp.br/>. Acesso em: 03 out. 07.

[23] Site oficial do Isabelle. Disponível em: < http://isabelle.in.tum.de/ >. Acesso em: 15 dez. 07.

[24] Site oficial do Ergo. Disponível em: < http://ergo.lri.fr >. Acesso em: 16 dez. 07.

[25] ROBISON, Peter. **An introduction to the Ergo Theorem Prover**. Artigo introdutório sobre o Ergo. Disponível em: < http://www.itee.uq.edu.au/~pjr/HomePages/ErgoFiles/notes.pdf >. Acesso em: 17 dez. 07

[26] CONCHON, Sylvain; CONTEJEAN, Evelyne; KANIG, Johannes. **Ergo: a theorem prover for polymorphic first-order logic modulo theories**. Artigo sobre o Ergo. Disponível em: < http://ergo.lri.fr/papers/ergo.ps >. Acesso em: 17 dez. 07.

[27] SINZ, Carsten. **System Description: aRa – An automatic Theorem Prover for Relation Algebras**. Artigo sobre o aRa. Disponível em: < http://www.carstensinz.de/papers/CADE-2000.pdf >. Acesso em: 18 dez. 07.

[28] NICOLETTI, Maria do Carmo. Forma Normal Prenex. Notas de aula do curso de introdução à lógica da Universidade Federal de São Carlos. Disponível em: <http://www.dc.ufscar.br/~carmo/notas_curso/prenex.doc >. Acesso em: 20 dez. 07

[29] Wikipedia – Forma Normal Prenex. Disponível em: < http://pt.wikipedia.org/wiki/Forma_normal_prenex >. Acesso em: 20 dez. 07.

[30] Wikipedia – Forma Normal Conjuntiva. Disponível em: <

http://pt.wikipedia.org/wiki/Forma_normal_conjuntiva >. Acesso em: 20 dez. 07.

[31] BUCH, Arnim; HILLENBRAND, Thomas. WALDMEISTER: **Development of a High Performance Completion-based Theorem Prover**. Artigo sobre o Waldmeister. Disponível em: < http://citeseer.ist.psu.edu/156638.html >. Acesso em: 20 dez. 07.

[32] Wikipedia – Lógica Modal. Disponível em: < http://pt.wikipedia.org/wiki/L%C3%B3gica_modal >. Acesso em: 20 dez. 07.

[33] Lógica Linear – Teoria da Prova. Site introdutório de Teoria da prova. Disponível em: < http://www.cin.ufpe.br/~teoria/provas/ >. Acesso em: 20 dez. 07.

[34] POLÓNIO, Artur. Lógica Aristotélica ou Lógica Clássica? Texto sobre lógica clássica. Disponível em: < http://ocanto.esenviseu.net/apoio/logica2.htm >. Acesso em: 20 dez. 07.

[35] Wikipedia – Lógica Aristotélica. Disponível em: < http://pt.wikipedia.org/wiki/L%C3%B3gica_aristot%C3%A9lica >. Acesso em: 20 dez. 07.