TESE DE DOUTORADO EM COTUTELA PELAS UNIVERSIDADES:

UNIVERSIDADE ESTADUAL DE CAMPINAS - UNICAMP (CAMPINAS, S.P., BRASIL)
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL
E "UNIVERSITE PAUL SABATIER" - UPS (TOULOUSE, FRANÇA)
INSTITUTE DE RECHERCHE EN INFORMATIQUE DE TOULOUSE (IRIT)

por

Rita Maria da Silva Julia

UM SISTEMA HÍBRIDO PARA O PROCESSAMENTO DE LINGUAGEM NATURAL E PARA A RECUPERAÇÃO DA INFORMAÇÃO

ORIENTAÇÃO:

Dr. Márcio Luiz Andrade Netto (UNICAMP)
Directeur de Recherche Mario Borillo (UPS)

CO-ORIENTAÇÃO:

phD. Antônio Eduardo Costa Pereira (Univ. Federal de Uberlândia)

Dissertação apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas (Campinas, Brasil), como parte dos requisitos exigidos para a obtenção do Título de Doutorado em Engenharia Elétrica, e à "Université Paul Sabatier" (Toulouse, França), para a obtenção do título "Docteur en Science de l'Université Paul Sabatier de Toulouse".

NOVEMBRO DE 1995

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

J942s

Julia, Rita Maria da Silva
Um sistema híbrido para o processamento de linguagem natural e para a recuperação da informação / Rita Maria da Silva Julia.--Campinas, SP: [s.n.], 1995.

Orientadores: Márcio Luiz Andrade Netto ; Mario Borillo; Antônio Eduardo Costa Pereira.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica; Universite Paul Sabatier (Toulouse, França).

1. Processamento de linguagem natural (Computação). 2. Inteligência artificial. 3. Teoria do conhecimento. 4. Lógica. 5. Sintaxe. 6. Recuperação da informação. I. Andrade Netto, Márcio Luiz. II.Borillo, Mario. III. Pereira, Antônio Eduardo Costa. IV. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica. V. Universite Paul Sabatier. VI. Título.

MINHA DECLARAÇÃO DE AMOR

Já nem sei mais precisar há quanto tempo você me apresentou aos bonecos de papel da minha vida. Mas, certamente, uma existência inteira se passou deste então. Lembro-me que nos assentávamos no chão ou à mesa e que você punha entre nós alguns bonecos de barriga engraçada. Você os havia feito coloridos, moldados pela genética da tesoura. O nome de cada um deles estava no formato da barriga. Formatos bizarros: A, 1, C, B, 2, O ...

Lenta e variadamente, você combinava os bonecos: punha o B com o A, o 2 com o 1, o A com o C ... E neste balé de figurantes tão díspares, ora barrigudos, ora esqueléticos, ora cheio de curvas mirabolantes, ora retos, neste balé de bonecos, você construía um universo incompreensivelmente harmonioso. Pouco a pouco, percebi que tudo o que estava compreendido no mundo e tudo aquilo que não cabia nele mas que era exato nos moldes dos meus sonhos, tudo, absolutamente tudo poderia ser descrito ali, naquele emaranhado em que A, C, 3, Z e tantos outros barrigudos se combinavam em paz.

E foi assim, brincando, e foi assim, observando suas mãos serenas combinarem os bonecos da minha vida, foi assim que descobri, muito mais tarde, que amor e partida, que eternos retornos, que cicuta e Sócrates, que crime e castigo, que tudo gira, existe e toma forma no aconchego e no sopro das palavras.

E, se de palavras também somos feitos, onde, em que ponto enigmático da evolução apareceram as palavras, aparecemos nós? Quando tentamos responder isto formulando teorias malucas sobre as quais nós mesmos somos cépticos, recaímos em paradoxos e em nem sei em quantas outras perguntas sem resposta. E as palavras riem de nossa inaptidão para domá-las, para retê-las nas mãos e compreendê-las. As palavras riem das fitas de Turing que tentam processá-las e de mim, pretensiosa, que um dia julguei conhecê-las. Hoje eu sei que nunca as palavras tiveram tanto sentido como quando, sentados no chão, você escrevia com barrigas de bonecos as histórias mais belas e mais autênticas, como jamais eu veria iguais. Nasceu ali, naquele espaço de sala, a única filosofia que realmente conta, porque, desinteressante que fosse aos olhos do mundo, enchia meus olhos de criança a filosofia em cores dos meus bonecos que dançavam.

Um dia saímos juntos, mãos dadas, e parei teimosa diante de um muro branco de uma esquina de Goiás. Foi então que, olhando para qualquer coisa que estava escrita no muro, eu lhe fiz a pergunta que só se sabe fazer quando se tem cinco anos:

- Por que é que o A é tão mais bonito do que o B?

Surpreso ou despreparado, você não soube responder-me. Mas a pergunta o marcou profundamente, a ponto de você me contar e de me recontar este momento nos anos que tivemos a sorte de ter pela frente.

Hoje, tempos depois, procuro nas palavras a magia e a festa escondidas. Mas, não o vendo mais manipulando meus bonecos, elas me parecem estéreis, um tanto inúteis. Inúteis e injustas, pois, se um dia você me deu as palavras, agora é vã minha tentativa de convencê-las a me darem você. Contudo, eu as amo ainda, eu as amo através de você, porque você só plantou amores eternos em tudo o que fez ou tocou.

Hoje percebo que meus bonecos de letras me levaram a cantos distantes: levaram-me a Dostoevski, à engenharia, a Branca de Neve, a Platão, a Chapeuzinho Vermelho, a Machado. Meus bonecos de letra me levaram também a teses malucas e a bancas de tese com cara de Lobo-Mau, de Bicho-Papão. E você, por trás disto tudo, sempre dizendo:

- Voe, Bebê, é exatamente esta a sua rota!

Bebê Johnson... reconfortante demais este seu jeito de me chamar, aos 5 ou aos 35 anos, para você, eternamente, Bebê.

Pois é, papai, você e a mamãe construíram para filhos e netos um mundo sólida e ternamente estruturado. Mais do que nunca sabemos que a firmeza deste mundo não provém de engenharia e nem de ferros, mas do amor do qual ele foi feito. E você que me ensinou tanto, você, extraordinário professor de matemática e de vida, você, que sempre me disse para voar, você, para onde é que voa agora? Amante do sol, que sol é este que o atrai neste momento? De fé inabalável, o que você tem aprendido a mais sobre o Deus da nossa vida inteira?

Nunca me esquecerei da beleza do que Platão conseguiu produzir com os bonecos de letra ao descrever o resultado do julgamento de Sócrates. Condenado injustamente à morte, Sócrates mantinha-se inexplicavelmente sereno. Perguntado sobre as razões que o mantinham calmo, ele respondeu:

- Se a morte for o fim de tudo, o sono eterno, então eu ganharei hoje a mais longa e a mais serena das noites, pois, não existindo mais nada além da vida, eu terei ganhado uma noite sem pesadelos. Se, por outro lado, a morte for continuação, certamente eu vivi de forma suficientemente digna para, a partir dela, dirigir-me para o encontro com os grandes homens, os mestres da humanidade que se encontram além desta vida.

Sabendo disto, você voou, papai, mas, certamente, voou para o encontro com o Deus da nossa vida e com aqueles que também foram grandes homens. Sei que não devo interromper sua viagem (voe, papai, certamente é esta a sua rota!), mas, qualquer dia desses, será que não daria para você passar por aqui de novo, sentar-se comigo no chão, e me dar a reposta para a única pergunta que me interessa, talvez por ter sido a questão primeira a desencadear todas as outras da minha existência:

- Por que é, papai, que o A é tão mais bonito do que o B?

Com a minha mais profunda capacidade de amar,

Sua filha Bebê Johnson (de 35 anos).

As almas gêmeas certamente se aninham e dançam no espaço abençoado onde razão e sentimento encontram o seu ponto de harmonioso equilíbrio.
A você, Stéphane Julia, ofereço todas as danças da minha vida, toda minha vida, a totalidade do que eu sou.

Sua mulher.

Quando Kaváfis compara o itinerário de um homem em busca de seu objetivo à viagem de Ulisses em busca de Ítaca, ele insiste que o homem deve caminhar de tal modo que o brilho do que ele adquire no caminho supere sempre o fulgor do seu destino final. Sinto esta tese como minha Ítaca particular. Prestes a chegar à minha Ítaca, não me frustro por constatar que ela é muito menos bela do que eu havia planejado no início do percurso. Talvez isso se deva à beleza e à riqueza do que tive no meu caminho durante o tempo em que a busquei com ansiedade. Aos que enfeitaram e enriqueceram este caminho, dedico minha Ítaca particular:

A meus pais Abedenago e Rita Garcia.
A meu marido Stéphane Julia e a nosso bebê.
A meus irmãos Mári, Rosângela, Rosângela Maria, Abedenago Filho e Cecília.
A meus sobrinhos Letícia, Danilo, Lara Cristina e César.
Ao amigo Antônio Costa, no sentido mais profundo e sincero que a amizade pode representar.

AGRADECIMENTO

Não teria sido possível chegar à minha Ítaca particular sem ter contado com apoios diversos. Serei sempre reconhecida por todos eles. Meus agradecimentos profundos:

A Deus, pelo direito humano ao raciocínio, à sensibilidade e à manifestação de ambos.

A Antônio Eduardo Costa Pereira (UFU), pela competência do apoio técnico.

A Márcio Luiz de Andrade Netto (UNICAMP), pela orientação.

A Mario Borillo (UPS), pela orientação.

A Stéphane Julia (UPS), pelas sugestões valiosas a nível da apresentação do texto em francês da tese e da defesa da mesma.

A Simone Julia (UPS), pelo seu apoio fundamental no processo administrativo de co-tutela de tese.

A Daniel Nordemann (INPE), pelas correções do texto em francês.

A Daniel Kayser (Université de Paris-Nord).

A Mario Jino (UNICAMP).

A Edson Françoso (UNICAMP).

A Rafael Mendes (UNICAMP).

A CAPES.

Ao CNPq.

A José Borges (Lingüística- Universidade Federal de Curitiba).

A Wagner do Amaral, Secundino Soares Filho, Cristina, Mazé, Heloísa (UNICAMP).

A meu irmão, Abedenago Filho, pela ingrata tarefa de atuar como meu procurador.

A meus pais, a minhas irmãs, a meus sobrinhos, a meus cunhados.

A João Bosco, José Carlos de Oliveira, Sílvia Martins, Valder Steffen, Gilberto Carrijo, Marcos Barros, Anilton Silva, Sílvio Bacalá, Graça e Nora (UFU).

Aos amigos e colegas: Myriam e Armando, Jussara e Gonzaga, Maristela e Jean-Paul, Ely, Ricardo e Luciana, Valéria, Olga, Ronaldo e Célia, Gina Maira, Iva e Maurício, Gugu, Ricardo Gudwin, Mohammed, Christine, Myriam Bras, Yannick Toussaint, Pierre, Michel, Bruno, Anne Condamines, Omar, Elias.

RESUMO

Nós apresentamos um sistema que analisa sintática e semanticamente um conjunto de asserções, que introduz as asserções analisadas em uma base de conhecimentos e que recupera informações a partir desta base.

As asserções correspondem a exigências que compõem um conjunto de especificações de programas. Elas correspondem a um tipo particular de frases em linguagem natural que se referem ao contexto das ciências espaciais.

As asserções são estocadas na base de conhecimentos como fórmulas do Cálculo dos Predicados cujas variáveis são anotadas por conceitos da Lógica Terminológica.

As perguntas propostas ao sistema também precisam ser analisadas sintática e semanticamente de tal maneira a apresentarem a mesma forma correspondente às asserções estocadas na base.

O analisador sintático e semântico implementado é capaz de gerar automaticamente algumas regras semânticas.

Para a recuperação da informação, nós usamos um provador de teoremas híbrido do Cálculo dos Predicados que responde perguntas efetuando uma avaliação parcial delas a partir da base de conhecimentos. O provador de teoremas utiliza a semântica da Lógica Terminológica para guiar seu mecanismo de inferência.

Os recursos da subsunção da Lógica Terminológica são utilizados para simplificar a base de conhecimentos e o traço de prova.

RESUME

Nous proposons dans ce mémoire la mise en oeuvre d'un système hybride de liaison entre les ressources du Calcul des Prédicats et celles de la Logique Terminologique. Ce système est capable de:

- produire une représentation formelle d'exigences exprimées en Langage Naturel dans le cadre de spécifications de logiciel.
- introduire dans une base de connaissances la représentation formelle obtenue pour chaque exigence en prenant soin de détecter les possibles redondances et contradictions.
- répondre à des questions posées au système, par l'exécution d'un mécanisme d'inférence permettant la récupération de l'information stockée dans la base de connaissances.

Les exigences appartiennent à un sous-ensemble restreint du Langage Naturel qui se situe dans le contexte du domaine spatial.

La représentation formelle d'une exigence est obtenue par une analyse syntaxique et sémantique. Elle correspond à une formule du Calcul des Prédicats dont les variables sont annotées par des expressions de la Logique Terminologique qui les particularisent.

L'analyseur syntaxico-sémantique implémenté est un système formel construit selon la théorie du structuralisme. Ce système formel définit une Grammaire Applicative dont le mécanisme d'application est guidé par une méthode heuristique de l'Intelligence Artificielle.

Les réponses du système correspondent à l'évaluation partielle des questions par rapport à la base de connaissances. La récupération de l'information est effectuée par un démonstrateur de théorème basé sur la technique de la résolution linéaire. Ce démonstrateur utilise la sémantique de la Logique Terminologique pour guider son mécanisme d'inférence.

Les principales contributions de notre travail sont les suivantes:

- La mise en oeuvre d'un analyseur syntaxico-sémantique qui engendre automatiquement des règles sémantiques, ce qui dispense le linguiste de la tâche de les définir.
- L'utilisation d'une méthode heuristique de l'Intelligence Artificielle pour guider le processus d'analyse.
- L'utilisation d'une unification sémantique pour lier les méthodes d'inférence du Calcul des Prédicats et de la Logique Terminologique.
- L'utilisation de la subsomption pour simplifier la base de connaissances et le processus de récupération de l'information.

ABSTRACT

We present a system to analyse a set of assertions, to introduce the analysed assertions into a knowledge base and to retrieve information from it. These assertions are requirements specified by the system engineer. They correspond to a particular type of sentences in Natural Language referring to Space Science context. The assertions are stored in the Knowledge Base as formulae of Predicate Calculus whose variables are annotated by concepts of Terminological Logic. The queries posed to the system must also be analysed in such a way as to get a form similar to that presented by the stored assertions. For information retrieval, we use a hybrid Theorem Prover of Predicate Calculus that answers questions by partially evaluating the query from the knowledge base. The Theorem Prover utilizes the semantics of Terminological Logic to guide its inference engine.

UM SISTEMA HÍBRIDO PARA O PROCESSAMENTO DE LINGUAGEM NATURAL E PARA A RECUPERAÇÃO DA INFORMAÇÃO

ÍNDICE

0. INTRODUÇÃO

I) Descrição Geral do Projeto

Esta tese tem sua origem no projeto LESD (Linguistic Engineering for Software Development) financiado pelo "Ministère de la Recherche et de l'Espace" (França) e pelo programa de ciências congnitivas do CNRS (França). O trabalho aqui descrito foi desenvolvido durante atividades de doutorado misto ("co-tutela") realizadas na UNICAMP e no laboratório IRIT ("Institute de Recherche en Informatique de Toulouse"), na "Université Paul Sabatier" (Toulouse, França).

O LESD visa a construir um sistema capaz de:

- produzir uma representação formal para exigências expressas em linguagem natural usadas como elementos de especificação de programas. Tal representação formal corresponde a sentenças do Cálculo de Predicados e é obtida através de análise sintática e semântica das exigências em linguagem natural.

- armazenar as exigências analisadas em uma base de conhecimentos.

- recuperar informações a partir da base de conhecimentos, isto é, responder perguntas sobre as exigências.

Tanto as exigências quanto as perguntas devem ser expressas numa forma restrita da linguagem natural relacionada às ciências espaciais.

As exigências são asserções especificadas pelos engenheiros de sistema. Tais asserções devem corresponder a frases objetivas e simples, de modo a atender ao padrão de qualidade de especificações de programa expresso no guia "IEEE Guide to Software Requirements Specifications" [39, 40].

Neste trabalho não serão focalizados detalhes ligados a estas normas de especificação, pois ele se inicia a partir das especificações em linguagem natural já preparadas pelos engenheiros do sistema durante o ciclo de vida da Engenharia de

Programas. Aos leitores interessados em maiores detalhes sobre tais especificações, aconselha-se a leitura de [40, 73].

II) Generalidades sobre Processamento da Linguagem Natural

O campo de lingüística computacional [2, 31, 38] é uma área multidisciplinar oriunda de pesquisas em Inteligência Artificial (IA) [5]. Este campo surgiu através de incentivos provindos de duas necessidades:

- Necessidade tecnológica de construir sistemas computacionais inteligentes (interfaces em linguagem natural para bases de dados, sistemas de tradução automática por máquinas, sistemas capazes de compreender a fala, sistemas de instrução através de computadores etc).

- Necessidade da lingüística ou ciência cognitiva de melhor compreender como os seres humanos se comunicam através da linguagem natural.

Os lingüistas teóricos e os psicolingüistas estão envolvidos nas mesmas metas da lingüística computacional. Contudo, eles adotam técnicas de ação distintas. Os lingüistas teóricos se interessam basicamente em produzir uma descrição estrutural da linguagem natural sem considerar detalhes de como as sentenças reais são processadas (processo de análise) nem de como as sentenças reais são geradas a partir das descrições estruturais. Eles se preocupam principalmente em caracterizar o princípio organizador geral sobre o qual se sedimentam todas as linguagens humanas, sem grandes preocupações com linguagens particulares. Os psicolingüistas, como os lingüistas computacionais, interessam-se tanto pelas representações das estruturas lingüísticas quanto pelo modo através do qual as pessoas realmente produzem e compreendem a linguagem natural.

Conhecimento e Linguagem

Para participar de uma conversa, uma pessoa tem que dispor de certos conhecimentos fundamentais. O mesmo deve ocorrer com qualquer sistema que pretenda lidar com a linguagem natural. Tais conhecimentos têm sido definidos como:

- Conhecimento fonético e fonológico: refere-se a como as palavras são concebidas como som (compreensão auditiva).
- Conhecimento morfológico: refere-se a como as palavras são construídas a partir de raízes e de unidades de significado chamadas morfemas (ex: construção de *amoroso* a partir da raiz *amor* e do morfema *oso*).
- Conhecimento sintático: refere-se a como as palavras devem-se organizar de modo a formar uma sentença aceita pela linguagem.
- Conhecimento semântico: refere-se ao significado das palavras e a como estes significados se combinam para formar o significado da sentença.
- Conhecimento pragmático: refere-se a como as sentenças são usadas em diferentes contextos e como o contexto interfere na interpretação da sentença.
- Conhecimento do mundo: refere-se ao conhecimento do mundo que os usuários da linguagem devem ter a fim de manterem uma conversação.

A seguir se mostram alguns exemplos que ajudam a distinguir as definições de sintaxe, semântica e pragmática. Considere cada uma das frases abaixo como sendo a frase inicial do livro de Processamento de Linguagem Natural de Allen [2]:

a) Este livro descreve as técnicas básicas que são usadas na construção de modelos computacionais de compreensão da linguagem natural.

Certamente, esta frase atende às expectativas de correção sintática, semântica e pragmática e é apropriada como frase introdutória do referido livro. Já o mesmo não ocorre com as três frases seguintes, que violam pelo menos uma destas definições:

b) Sapos verdes têm narizes enormes.

Esta frase satisfaz os critérios sintáticos e semânticos mas fica completamente fora de contexto como introdução do livro. Logo, ela é pragmaticamente incorreta.

c) Idéias verdes têm narizes enormes.

Esta frase é pragmatica e semanticamente incorreta, pois, além de ficar fora de contexto, apresenta um significado mal formado. Ela só atende aos critérios sintáticos.

d) Enormes têm verdes idéias narizes.

Mesmo que esta frase tenha as mesmas palavras da frase do item c, ela não é inteligível. Ela é tão deformada em sua estrutura que não podemos apontar o que está errado com ela. Ela é mal formada a nível sintático, semântico e pragmático.

Eventualmente, uma frase pode ser sintaticamente mal formada mas ser pragmaticamente e até mesmo semanticamente bem formada. Por exemplo, se uma pessoa pergunta a outra: *Aonde você vai?* e a outra responde: *Eu vou compras*, sua resposta é inteligível, ainda que sintaticamente mal formada.

Nem sempre é possível fixar um limite preciso entre as definições de sintática, semântica e pragmática. Porém, as técnicas computacionais tentam dividir o Processamento da Linguagem Natural em 3 fases:

- A fase da análise sintática e morfológica que verifica a correção estrutural da sentença de acordo com a gramática da linguagem;

- A fase da interpretação semântica [4] que mapeia a descrição estrutural da frase em uma forma lógica que represente o significado da frase independentente do contexto. Esta fase usa conhecimentos semânticos e algumas partes do conhecimento pragmático.

- A fase da interpretação contextual que mapeia as formas sintáticas e lógicas numa representação final dos efeitos causados por se entender a sentença. Esta representação final é expressa numa linguagem apta a aceitar um componente de inferência que "raciocine" e recupere informações nesta linguagem. Nesta tese, a representação final corresponde à base de conhecimentos e o componente de inferência corresponde a um provador de teoremas automático.

III) Generalidades Sobre Engenharia de Programas

A Engenharia de Programas (EP) nasceu da vontade de racionalizar o desenvolvimento de sistemas informáticos e de programas. Sem dúvida, a crescente complexidade dos sistemas informáticos, bem como a rápida evolução material a eles associada justificam e exigem avanços progressivos no campo da EP [60].

A experiência tem mostrado que quanto mais cedo um erro for detectado na seqüência de planejamento e de implementação de um projeto informático, menor custo (ou perda) ele representará. Além disso, quanto mais se investir na qualidade do planejamento de um projeto, menos passível de erros ele será. Daí a grande importância da EP no contexto computacional vigente.

O ciclo de vida da EP se desenrola nas seguintes fases:

a) *Fase do Planejamento*: antes de se desenvolver um sistema, é conveniente que se faça um bom planejamento do que será feito. O planejamento se divide em duas etapas:

a.1) Etapa da análise e engenharia do sistema: nela se estabelecem as exigências relativas a todos os elementos do sistema e distingue-se o subconjunto destas exigências que deve referir-se ao programa (lembremos que o programa é sempre uma parte de um grande sistema). Esta distinção das exigências é fundamental quando o programa deve ser interfaceado com outros elementos tais como o "hardware", as pessoas e as bases de dados.

a.2) Etapa da análise das exigências do programa: nesta etapa intensifica-se ainda mais o esforço para se evidenciar o conjunto das exigências que se referem ao programa.

Nela o programa é definido em detalhe, com todas as suas características funcionais. Nela se definem também as interfaces do sistema. Esta segunda etapa é conduzida juntando-se os esforços da pessoa (ou equipe) que desenvolverá o programa e da pessoa (ou organização) que o solicitou (cliente).

Como resultado desta segunda etapa, produz-se o documento chamado *Especificação das Exigências do Programa.*

b) *Fase do Desenvolvimento*: nesta fase o programa é implementado. Ela se divide em três etapas:

b.1) Projeto: focaliza-se nos seguintes atributos do programa: estrutura de dados, arquitetura do programa, detalhes procedimentais e caracterização de interfaces. Durante o projeto, traduzem-se as exigências em uma representação do programa que atenda a certos critérios de qualidade. Tal representação antecede o início da codificação. Tanto quanto as exigências, o projeto também é documentado e se torna parte da configuração do programa.

b.2) Codificação: é a tradução do projeto numa forma legível pelo computador.

b.3) Teste: nesta etapa se verifica a correção da lógica do código do programa, certificando-se de que as entradas definidas produzirão resultados reais que estejam de acordo com os resultados especificados.

c) *Fase da Manutenção*: é a fase onde se efetuam modificações no programa entregue ao cliente. A necessidade de modificação do programa na manutenção pode ser decorrente da detecção de erros ou então de alguma alteração no ambiente externo (por exemplo, sistema operacional ou dispositivos periféricos).

Pressman [60] avalia que o custo da correção de um erro cometido na fase de especificação é 1,5 a 6 vezes mais elevado quando ele é detectado na fase de desenvolvimento e 60 a 100 vezes maior quando ele é detectado durante a manutenção.

Nesta tese nós lidamos apenas com as especificações produzidas na segunda etapa da primeira fase. Conforme veremos, propomos uma técnica para tratar lingüística e computacionalmente as especificações de programa (exigências) expressas num domínio particular da linguagem natural ligado à ciência espacial.

IV) Principais Enfoques da Tese

Dentre as questões e os problemas ligados à análise e ao armazenamento das exigências, bem como à recuperação de informação [14, 15, 16], destacam-se:

- *Questão da análise das exigências*

  Implementou-se um analisador sintático e semântico *inteligente* das exigências. O analisador efetua as análises sintática e semântica das exigências simultaneamente, tal como proposto na Gramática de Montague. Contudo, distintamente de Montague, no analisador implementado não é necessário que se defina uma abstração lambda (regra semântica) correspondente a cada categoria sintática da gramática, pois o analisador gera automaticamente algumas delas.

  O processo de análise sintática consiste em se combinar categoria sintática com categoria sintática até que se produza uma sentença. O processo de análise semântica consiste em se combinar abstração lambda com abstração lambda até que se produza uma fórmula cujo significado seja verdadeiro ou falso. Este último processo é guiado por um método heurístico da Inteligência Artificial.

- *Expressividade da resposta*

Este problema se relaciona com a pragmática. A resposta deve satisfazer às expectativas do usuário. Um tipo de fracasso de expressividaade é quando a resposta não é informativa o bastante. Por exemplo, suponha-se que a base de conhecimentos armazene a seguinte exigência:

The ioi-gs shall have the capacity to control the computers on board the space vehicle,

Suponha-se também que o usuário pergunte ao sistema:

Which equipments does the ioi-gs control?

Caso se use um provador de teoremas do Cálculo de Predicados para responder essa pergunta e o provador produza uma variável X como resposta, o usuário não ficará satisfeito com tal resposta. O usuário também ficará insatisfeito se o sistema produzir como resposta uma lista enorme e não simplificada de todas as propriedades que X deve satisfazer. A resposta ideal seria:

The on board computers.

Dentre tudo que for obtido na longa cadeia de prova, somente a informação de que X é um computador a bordo do veículo espacial interessa ao usuário.

- *Subsunção de conceitos*

Um meio de obter expressividade consiste em focalizar a informação provida ao usuário num conjunto de conceitos que descreva objetos e propriedades de objetos. Se um determinado literal pertence ao conjunto, este literal pode fazer parte do traço de prova. No capítulo 5 nos endereçaremos ao problema de composição do traço de prova. Por enquanto, adianta-se apenas que foram usados recursos da Lógica Terminológica [8, 23, 43, 50, 51, 72] como ferramenta simplificadora do resultado dessa composição. Por exemplo, se o provador de teoremas descobre que a variável X da resposta é *humano & homem & ¬mulher* (o símbolo ¬ representa a negação lógica), basta que ele responda que X é *homem*, uma vez que, sendo X *humano*, então *homem* é equivalente a *¬mulher*. Além disso, o conceito *humano* subsume o conceito *homem*. Logo, a intersecção deles corresponde simplesmente ao conceito *homem*. Posteriormente se apresentará essa idéia de subsunção com mais detalhes.

- Inserção na base de conhecimentos

Usar-se-ão noções da Lógica Terminológica, tais como *classificação* e *subsunção*, como elementos de simplificação da base de conhecimentos.

- *Composição da resposta através de Aprendizado Baseado em Explicação ("Explanation Based Learning") ou de Avaliação Parcial* [19, 26, 27, 36]

A resposta dada ao usuário é uma avaliação parcial [26] da pergunta com relação à base de conhecimentos. Um sistema de representação do conhecimento é formado por um conjunto Fs de fórmulas e por um conjunto de regras de inferência. Há dois subconjuntos recursivos de Fs, D e P, tal que Fs = D $\cup$ P. Todas as fórmulas pertencentes a P são axiomas. Todas as fórmulas pertencentes a D são dados. O conjunto de regras de inferência é chamado de semântica do sistema de representação do conhecimento. Dentre as regras de inferência, há uma que especifica como e quando as outras regras de inferência serão usadas. Esta regra especial é denominada regra computacional.

Suponha-se que D e P sejam descritos por um grupo de regras sintáticas (isto é sempre possível porque D e P são conjuntos recursivos). Sejam Dp e Pp os conjuntos de regras de produção que descrevem D e P, respectivamente. Uma fórmula tem componentes somente em D se ela é descrita pelo conjunto Dp - (Dp $\cap$ Pp). Uma fórmula tem componentes somente em P se ela é descrita pelo conjunto Pp - (Dp $\cap$ Pp).

Considere-se o sistema de representação de conhecimento krs(D, P, semântica), onde *semântica* corresponde às *regras de inferência* do sistema. Se uma inferência *f* tem componentes somente em P, ela é chamada de *parcial*. Chama-se inferência a qualquer dedução feita a partir do dado de entrada denominado *pergunta*. O conjunto escolhido para expressar a resposta é reconhecido por um subconjunto de Pp - (Dp $\cap$ Pp). Isso significa que a resposta é uma avaliação parcial. O subconjunto de Pp - (Dp $\cap$ Pp) foi escolhido de tal maneira a definir uma linguagem terminológica.

Isso significa que o usuário poderá usar os métodos da Lógica Terminológica (subsunção e classificação) para analisar as respostas.

No capítulo 1 deste trabalho discutem-se generalidades sobre especificação de programa, tais como critérios de qualidade. No capítulo 2 são apresentadas teorias necessárias à compreensão da análise das exigências. No capítulo 3 é mostrada a análise das exigências e a inserção das exigências analisadas na base de conhecimentos. No capítulo 4 apresentam-se teorias úteis à compreensão do processo de recuperação de informação. Finalmente, no capítulo 5, discute-se o processo de recuperação de informação.

V) Contribuições

Acreditamos que nossas principais contribuições consistem em:

- Implementar um analisador sintático e semântico que automaticamente gera regras semânticas, dispensando o lingüista da tarefa de defini-las.

- Usar um método heurístico de Inteligência Artificial para guiar o processo de análise.

- Usar unificação semântica para ligar os métodos de inferência do Cálculo de Predicados e da Lógica Terminológica.

- Usar a subsunção para simplificar a base de conhecimentos e o traço de prova.

1. Especificações de Programas

Introdução

Conforme já foi dito na introdução deste trabalho, a meta deste projeto é analisar, armazenar e recuperar informações a partir de asserções (exigências) de especificação de programas feitas em linguagem natural pelos engenheiros do sistema. Assim sendo, lidamos com especificações já prontas e partimos do princípio de que elas tenham sido adequadamente elaboradas de acordo com as normas vigentes. Portanto, faremos apenas uma discussão superficial sobre especificações. Maiores detalhes podem ser vistos em [20, 40, 73].

1.1. Critérios de Qualidade das Especificações de Programa

A norma IEEE 830 [40] cita como qualidades essenciais de uma especificação de programa as seguintes:

- *a não ambigüidade*

  Uma especificação é ambígüa quando contém uma exigência que é ambígüa, ou seja, uma exigência que pode ser interpretada de mais de uma maneira.

- *a rastreabilidade*

  Qualidade que permite que os elementos aos quais uma exigência se refere possam ser rastreados nas outras exigências e nas diversas fases de desenvolvimento do sistema. No caso do sistema implementado nesta tese, a rastreabilidade deve estar presente em suas fases componentes, do seguinte modo:

(a) A rastreabilidade deve permitir o reconhecimento de elementos idênticos dentro de uma mesma exigência.

(b) Durante a análise sintática e semântica de uma exigência, ela vai sofrendo transformações até chegar a uma forma adequada ao armazenamento na base de conhecimentos (no nosso caso, esta forma é uma fórmula bem formada (fbf) do Cálculo de Predicados). Assim sendo, a rastreabilidade deve permitir o reconhecimento de elementos idênticos em qualquer uma das formas em que ela se converte durante a análise até chegar à sua representação computacional definitiva.

(c) Durante a fase da inserção da exigência analisada na base de conhecimentos, a rastreabilidade permite que se estabeleçam vínculos entre uma exigência e outra através do reconhecimento dos elementos que elas mantêm em comum. Conforme será visto posteriormente, tal reconhecimento permitirá simplificações oportunas na base de conhecimentos no momento em que se inserir nela uma nova exigência;

(d) Durante a fase de recuperação da informação a rastreabilidade permite que se estabeleçam vínculos entre os elementos da pergunta e os elementos da base de conhecimentos.

- *a coerência*

Uma especificação é coerente quando ela não contém exigências que se contradigam mutuamente.

- *a completude*

Uma especificação é completa quando ela agrupa todas as exigências relativas à funcionalidade, ao desempenho, às restrições sobre a estrutura do sistema, aos atributos e às interfaces externas. Quando o programa pode ser especificado em função de suas entradas, as especificações devem prever a resposta que o programa deve fornecer em função de todas as classes de entrada possíveis, sejam elas

corretas ou não. Com isto, facilita-se a verificação do comportamento do programa durante a fase de testes.

- *a verificabilidade*

Uma especificação é verificável quando cada uma de suas exigências o é. Uma exigência é verificável quando existe um procedimento racional, humano ou automático, capaz de mostrar que o programa satisfaz a exigência. Se uma exigência é não verificável, este fato deve ser mencionado explicitamente. Como exemplo de exigências não verificáveis, citemos a seguinte:

O sistema deve ter uma boa interface gráfica.

Neste caso, o adjetivo *boa* não pode ser definido por critérios objetivos. Como, então, garantir que uma *parte do código* cumpre tal exigência?

- *a modificabilidade*

Refere-se à qualidade de se conseguir facilmente atualizar uma exigência da especificação sem que isto cause danos aos critérios de completude e coerência da especificação.

- *utilizabilidade das especificações nas fases operacionais e de manutenção*

Durante a operação ou durante a manutenção do programa, pode-se detectar a necessidade de se alterar profundamente algum aspecto da especificação original; neste caso, é importante que o pessoal da manutenção (que geralmente não participou da fase da especificação) tenha facilidade em lidar com as especificações no sentido de modificá-las.

Neste trabalho, nós nos concentramos principalmente nos critérios de rastreabilidade do sistema.

1.2. Tipos de Especificação

Há um grande debate entre correntes que defendem especificações formais ou informais de sistemas [73]. Em termos gerais, as especificações formais variam entre:

- especificações matemáticas: são aquelas para as quais a avaliação de alguns de seus critérios de qualidade se faz através de verificação automática (demonstradores de teoremas). Cite-se como exemplo a linguagem algébrica Z [73]).

- especificações operacionais: permitem uma representação computacional para as exigências e a rápida concepção de um protótipo.

As especificações informais são aquelas efetuadas em linguagem natural, conforme brevemente descrito a seguir.

Especificação em Linguagem Natural

O trabalho de especificação exige uma boa experiência por parte de seu executor. Apesar da existência de diversas linguagens formais de especificação, freqüentemente é extremamente difícil de se começar um trabalho de concepção sem que se recorra primeiramente às facilidades de expressão da linguagem natural. Uma especificação de programa em linguagem natural corresponde a um conjunto de exigências. Cada exigência representa uma necessidade do programa. A vantagem de se usar a linguagem natural como linguagem de especificação consiste no fato de ela permitir um diálogo direto (diferentemente das linguagens formais) entre clientes, projetistas e usuários do projeto. Posteriormente, as especificações efetuadas em linguagem natural podem ser processadas de modo a se transformarem em especificações mais formais, tal como se fez no decorrer do projeto que originou esta tese.

A vantagem da linguagem natural de representar uma ferramenta direta, informal e simples na elaboração de especificações, apresenta uma contrapartida lamentável: sua informalidade pode tornar as especificações mais susceptíveis a determinadas irregularidades indesejáveis. Tais irregularidades provêm de dois fatores: primeiramente, os engenheiros que fazem as especificações, apesar da competência técnica, nem sempre são sensíveis às questões lingüísticas. O segundo fator refere-se a irregularidades ligadas à própria linguagem natural como elemento de especificação. Abaixo citamos alguns exemplos de irregularidades que devem ser evitadas em especificações em linguagem natural:

- *o barulho*

  Consiste na introdução de informações desnecessárias na especificação.

- *o silêncio*

  Consiste na omissão de informações necessárias à especificação. Normalmente isso ocorre com informações que são extremamente óbvias para o especialista que elabora as especificações. Em virtude dessa obviedade, ele julga desnecessária a introdução dessas informações na especificação, quando na realidade elas são imprescindíveis às pessoas não tão familiarizadas com o domínio em questão.

- *a contradição*

  Consiste na introdução de informações antagônicas na especificação;

- *a sobre-especificação*

  Consiste na antecipação exagerada de detalhes já no nível da especificação, sem que ainda se tenha bastante conhecimento sobre eles (detalhes cuja necessidade só ficará clara posteriormente, por exemplo, no momento do desenvolvimento do programa).

- *a ambigüidade*

Uma especificação é ambígüa quando ela dá margem a mais de uma interpretação;

- *a falta de realismo*

Consiste na especificação de necessidades que na realidade são impraticáveis.

Obviamente, também as especificações formais estão sujeitas a algumas dessas irregularidades.

O uso da linguagem natural como ferramenta de especificação é condicionado por um conjunto de restrições que visam a limitar irregularidades diversas a que ela está sujeita (polissemia, paráfrases, ambigüidades etc). Tais restrições compreendem os três tipos de recomendação citados a seguir:

- elaboração de frases sintaticamente simples, isto é, deve-se limitar o uso de conectores, o uso de partículas de negação, a extensão dos grupos nominais etc.

- uso de um vocabulário restrito que se limite aos termos técnicos do domínio e a palavras e expressões que constituam um conjunto simplificado da linguagem natural (devem-se evitar sinônimos, paráfrases etc).

- evitar descrições nebulosas e dúbias.

A seguir são mostrados alguns exemplos de exigências relativas ao programa *ioi-gs* que deve controlar um veículo espacial [73].

- The ioi-gs shall provide space vehicle monitoring and control.
- The ioi-gs shall have the capability to control the automatic systems of the flight configuration.

- The ioi-gs shall have the capability to obtain and analyse memory dumps from any computer on-board the space-vehicle.

Cada exigência consiste de uma frase simples, objetiva e independente. O fato de uma exigência ser estruturalmente independente da outra permite que elas sejam analisadas sintática e semanticamente de forma individual. Mas o fato de elas se referirem com freqüência a mesmos objetos ou a objetos que se relacionam cria a necessidade de se construir uma base de conhecimentos onde haja uma associação apropriada entre as exigências. Os elementos que guiam esta associação são justamente os objetos idênticos ou relacionados a que as exigências se referem.

Nós desenvolvemos nossos trabalhos no contexto lingüístico das ciências espaciais. Este domínio é particularmente apropriado ao desenvolvimento de projetos em informática lingüística, uma vez que a complexidade dos sistemas envolvidos torna praticamente inevitável o uso da linguagem natural nas fases iniciais dos projetos. Convém salientar também um outro fator importante: normalmente tais projetos envolvem parceiros industriais em mais de uma nação (como o LESD, que envolve alguns países da Europa). Isso traz um problema adicional: nem todos os participantes do projeto têm como língua materna aquela na qual serão elaboradas as especificações iniciais do projeto (que, no caso do LESD, é o inglês). Conseqüentemente, eles apresentam diferentes níveis de domínio da língua de especificação.

Apesar de trabalharmos neste projeto com um corpo de especificações referentes ao contexto espacial, podemos aplicar as técnicas aqui implementadas a qualquer contexto cujo corpo de especificações apresente características estruturais semelhantes àquelas sobre as quais trabalhamos (ex: corpo de especificações de receituários médicos etc).

2. Tópicos Teóricos Importantes Para a Análise Sintática e Semântica das Exigências

Introdução

As análises sintática e semântica das exigências são conduzidas no sentido de transformá-las em fórmulas bem formadas (fbfs) híbridas do Cálculo de Predicados, isto é, fbfs que armazenem conhecimento terminológico e assertivo. A análise é feita através de uma gramática aplicativa cujos operadores correspondem a combinadores (abstrações lambda com algumas características particulares, conforme discutido neste capítulo).

Durante a análise das exigências, os critérios de subsunção da Lógica Terminológica são utilizados como elementos de simplificação da base de conhecimentos.

Este capítulo introduz discussões teóricas fundamentais à compreensão dos processos de análise.

2.1. Considerações Sobre o Cálculo Lambda

Nessa abordagem considerar-se-á que o leitor esteja familiarizado com o Cálculo Lambda. Conseqüentemente, far-se-á apenas uma breve recapitulação sobre abstrações lambda e sobre seu mecanismo de aplicação (redução beta) [41]. Neste sistema utilizam-se os recursos do Cálculo Lambda durante a análise sintática/semântica das exigências, uma vez que este cálculo representa uma linguagem simples e expressiva. Recorde-se que as expressões lambda são definidas como:

<exp> ::= <constante>

| <variável>	Nomes das variáveis
| <exp> <exp>	Aplicações
| λ <variável> . <exp>	Abstrações lambda (onde <*exp*> representa o corpo da abstração lambda)

Note-se que as abstrações lambda correspondem a um tipo particular de expressão lambda que representa uma função. O resultado de se aplicar uma abstração lambda a um argumento corresponde a uma instanciação do corpo dessa abstração em que as ocorrências livres do parâmetro formal desse corpo são substituídas por uma cópia do argumento. O processo seguido para se chegar a tal instanciação é conhecido como β-redução.

Ex: a β-redução de *(λx. + x 1) 4* produz *+ 4 1* .

O Cálculo Lambda é um sistema aplicativo cuja operação de aplicação é a β-redução.

Compilação das abstrações lambda:

O processo de instanciação do corpo de uma abstração lambda pode atingir graus indesejáveis de ineficiência em virtude de procedimentos tais como [57]:

(i) a cada nó do corpo o processo tem que efetuar uma análise de caso no rótulo desse nó;

(i i) em cada nó de variável e em cada nó lambda, o processo de instanciação tem que testar se o nó corresponde ao parâmetro formal.

Para fins de eficiência, pode-se substituir esse processo de instanciação por um processo de compilação que associa a cada corpo lambda uma seqüência fixa de instruções que construirão a instância desse corpo. Desse modo, o ato de instanciar um corpo lambda corresponde a obedecer a seqüência de instruções a qual ele é associado. Essa seqüência de instruções pode ser obtida previamente por um compilador e contém informações sobre a forma do corpo e sobre o local onde ocorrem os parâmetros formais. Conforme cita Peyton Jones [57], infelizmente nem todas abstrações lambda podem ser compiladas desse jeito. Por exemplo, é impossível compilar a abstração lambda *λx.(λy. - y x)* para uma única seqüência de instruções, pois a cada argumento

arg a que ela seja aplicada, será produzida uma nova seqüência do tipo *(λy. - y arg)*. Assim, caso se aplique a abstração *λx.(λy. - y x)* a *3* produzir-se-á *(λy. - y 3)*; caso ela seja aplicada a *5* produzir-se-á *(λy. - y 5)*, e asssim por diante. Note-se que a origem desse problema provém de um único fato: a ocorrência livre da variável *x* no corpo da abstração *(λy. - y x)*. Tal problema é resolvido com a transformação das abstrações lambda em supercombinadores, a serem discutidos logo a seguir.

2.2. Supercombinadores

Um supercombinador $S de aridade n é uma expressão lambda da forma $\lambda x_1. \lambda x_2 ... \lambda x_n.E$, onde E não é uma abstração lambda, de tal forma que:

(i) $S não tenha variáveis livres;
(i i) qualquer abstração lambda em E seja um supercombinador;
(i i i) $n \geq 0$ (ou seja, um supercombinador pode não apresentar λ algum).

Exemplos de supercombinadores:

4
(+ 2 4)
λx. + x 1
λf. f (λx. + x x)

Não são supercombinadores:
λ x . y
λ f. f (λx . f x 2) (a abstração interna *λx* não é um supercombinador, pois *f* ocorre livre nela)

Transformação de variáveis livres em parâmetros extras:

Pode-se desaparecer com as variáveis livres com a introdução de parâmetros extras.

Exemplo: transforma-se a abstração:

$$(\lambda y.\ +\ y\ x)$$

em sua equivalente:

$$(\lambda x.\ \lambda y.\ +\ y\ x)\ x$$

ou, ainda, por outra equivalente:

$$(\lambda w.\ \lambda y.\ +\ y\ w)\ x$$

Observe-se que *(λw. λy. + y w)* é um supercombinador.

Algoritmo de compilação de abstrações lambda em supercombinadores:

ATÉ não mais existirem abstrações lambda

(1) Escolha qualquer abstração lambda que não tenha nenhuma abstração lambda mais interna em seu corpo.

(2) Transforme todas suas variáveis livres em parâmetros extras.

(3) Dê um nome arbitrário à abstração lambda transformada (ex: $X1)

(4) Substitua a ocorrência da abstração lambda transformada pelo seu nome aplicado às variáveis livres.

(5) Compile a abstração lambda e associe o nome com o código compilado.

END

A seguir descreve-se a compilação da abstração lambda $\lambda x.\ (\lambda y.\ +\ y\ x)\ x)\ 4$ para um supercombinador, seguindo-se os passos previstos pelo algorítimo acima:

(1) Escolhe-se a abstração $\lambda y.\ +\ y\ x$

(2) A transformação da variável livre x em parâmetro extra produz:

$$(\lambda w.\ \lambda y.\ +\ y\ w)\ x$$

(3) Atribua-se à abstração $\lambda w.\ \lambda y.\ +\ y\ w$ o nome $\$Y$ (logo, $\$Y\ w\ y\ =\ +\ y\ w$).

(4) A substituição pelo nome e a aplicação às variáveis livres produz:

(λx. $Y x x) 4 (equivalente à expressão original)

(5) Repetem-se passos (1), (2), (3), (4), (5) para a expressão $(\lambda x.\ \$Y\ x\ x)\ 4$

(1) não existe abstração mais interna
(2) nenhuma variável livre
(3) (λx. $Y x x) 4 = $ X (logo, $X x = $Y x x)
(4) $X 4 (equivalente à expressão original)
(5) não há mais abstrações lambda em $X 4

Logo, a abstração lambda $(\lambda x.\ (\lambda y.\ +y\ x)\ x)\ 4$ pode ser compilada para a forma do supercombinador $\$X\ 4$. Realmente,

$X 4

→ SY 4 4
→ + 4 4
→ 8

produz o mesmo resultado que:

$(\lambda x.\ (\lambda y.\ +y\ x)\ x)\ 4$

$\rightarrow$ $(\lambda y.\ +y\ 4)\ 4$

$\rightarrow$ $+\ 4\ 4$

$\rightarrow$ 8

Chamam-se *combinadores* às abstrações lambda que não contêm nenhuma ocorrência de variável livre [57]. Logo, percebe-se que *combinadores* são as abstrações lambda que satisfazem o requisito *(i)* dos supercombinadores e que alguns *combinadores* podem ser *supercombinadores*, desde que satisfaçam também *(i i)* e *(i i i)*. Contudo, para fins de compatibilidade com outros trabalhos, a partir daqui usar-se-á o termo único e geral *combinadores*.

A seguir será abordada a teoria de combinadores de uma forma mais próxima ao que foi implementado no presente processo de análise.

Intuitivamente pode-se definir um combinador como sendo um operador cujo comportamento é bem definido independentemente de qualquer interpretação externa (sua representação dispensa variáveis).

Os combinadores são, portanto, adequados à realização de uma semântica íntrinseca independente mas não oposta a uma semântica extrínseca como a proposta por Tarsky. Portanto, o comportamento descrito numa abstração lambda pode ser representado por meio deles, como aliás acabamos de constatar. Além disso, eles podem combinar-se para definirem um operador mais complexo. A atuação de um combinador se faz de acordo com o previsto em sua regra de β-redução. Pode-se, por exemplo, imaginar-se um operador de composição B que permita a combinação de dois operadores f e g para produzir um operador complexo h (onde $h \equiv B\ f\ g$), independentemente de qualquer interpretação que seja atribuída aos operadores f e g. A regra de β-redução de B (semântica) é:

$$B\ X\ Y\ Z \geq X\ (Y\ Z),$$

onde X, Y, Z são expressões combinadoras quaisquer. O símbolo ≥ indica que a avaliação interna (independe de interpretações) da expressão à sua esquerda produz a expressão à sua direita. Em outras palavras, independentemente da interpretação dada a X, Y, Z, o combinador B, quando aplicado a eles, produz um novo combinador complexo que corresponde a X aplicado ao resultado da aplicação de Y a Z. Por outro lado, uma avaliação externa (para qual adotaremos o símbolo ♠) representa uma avaliação associada a uma interpretação. Como exemplo, cite-se:

$$(+\ 3\ 4) \spadesuit 7,$$

que indica que a avaliação da expressão (+ 3 4) interpretada no domínio dos números inteiros produz o inteiro *7*.

Para ilustrar uma avaliação interna envolvendo o combinador B, considere-se *d* a função de diferenciação e *f* a função descrita abaixo:

$$f \equiv \lambda x.\ 3x^3$$

Logo,

$$B\ d\ d\ f \geq (B\ d\ d)\ (\lambda x\ .\ 3x^3) \geq d\ (d\ (\lambda x.\ 3x^3))$$

A partir daqui, a avaliação da expressão já é externa, conforme mostrado abaixo:

$$d\ (d\ (\lambda x.\ 3x^3)) \spadesuit d\ (\lambda x.\ 9x^2) \spadesuit \lambda x.\ 18x$$

2.3. Sistemas Híbridos de Representação do Conhecimento e Lógica Terminológica

Introduzimos aqui apenas as noções sobre representação híbrida do conhecimento e sobre Lógica Terminológica que se fazem imprescindíveis à

compreensão do sistema. Maiores detalhes podem ser consultados em textos específicos do assunto [50, 51, 53]. Antes de tratar dos referidos tópicos, discutamos algumas definições fundamentais , a saber:

- **Conceitos**

Podem ser definidos pela notação Backus-Naur:

<conceito> ::= <conceito-atômico> |
<conceito-atômico> & <conceito> % representa um conceito não atômico;

<conceito-atômico> ::= <identificador>

Um conceito pode designar um indivíduo, uma classe de indivíduos ou uma propriedade de indivíduos ou de classes de indivíduos.

Pode-se estipular uma relação entre alguns conceitos em função de sua especificidade. Por exemplo, o conceito *humano & homem* define uma classe mais específica que o conceito *humano*. Neste caso, diz-se que o conceito *humano* subsume o conceito *humano & homem*. Posteriormente veremos com mais detalhes a definição de *subsunção*.

Classificação dos conceitos: os conceitos dividem-se em dois grupos: conceitos genéricos e conceitos individuais.

a) ***Conceitos genéricos***: denotam classes de indivíduos. De acordo com sua descrição dividem-se em:

a.1) *Conceitos definidos*: são aqueles cujos significados são completamente determinados pelas suas descrições.

a.2) *Conceitos primitivos*: são aqueles cujos significados são apenas parcialmente determinados pelas suas descrições (não podem ser exaustivamente descritos), tal como o conceito *espécies_de_animais*. Num sistema de representação os conceitos primitivos correspondem a conceitos básicos não definíveis mas reconhecíveis pelo seu primitivismo.

b) Conceitos individuais: são usados para representar indivíduos do domínio. Eles não podem ser usados para descrever outros conceitos. Sua existência é apontada somente em eventuais descrições do mundo. Os conceitos podem organizar-se segundo uma taxonomia baseada em sua maior ou menor *especificidade*. Vejamos um exemplo onde:

- os conceitos primitivos são indicados pelo símbolo * ;

- o símbolo $\leq$. introduz o conceito primitivo atômico localizado a sua esquerda através do conceito localizado a sua direita (o conceito da direita descreve o conceito primitivo);

- o símbolo = . introduz um conceito definido atômico localizado à sua esquerda através do conceito localizado à sua direita.

Exemplo de introdução de conceitos:

humano* $\leq$ .animal*
animal_fêmea* $\leq$. animal*
mulher = . humano* & animal_fêmea*
Maria é um indivíduo pertencente ao conceito *mulher*.

As descrições acima intoduzem, por exemplo, o conceito primitivo *animal* como sendo menos específico que o conceito primitivo *humano*. Note-se que os conceitos primitivos não ficam completamente definidos em suas descrições, tal como o conceito *humano** , em que sua descrição indica apenas que ele é uma subclasse de

animal sem, contudo, oferecer um conjunto de informações que caracterizem claramente sua classe. Já a classe *mulher* fica precisamente definida como a classe formada pela intersecção das classes *humano* e *animal-fêmea*. O conceito individual *maria* indica um elemento do mundo (domínio) que pertence à classe *mulher*. *Maria*, como conceito individual, não pode ser usado para descrever outros conceitos, isso é, não pode ser usado à direita do símbolo $\leq$. nem do símbolo $=$. .

Valor semântico de algumas expressões de conceitos: o valor semântico de uma expressão E qualquer é representado por $\lceil E \rceil$. Abaixo definimos o valor semântico de alguns tipos de conceitos [74]:

Conceito	Tipo	Valor Semântico
c	atômico	λX [c X]
c1 & c2	não atômico	λX [$\lceil$ c1 $\rceil$ X] & [$\lceil$c2$\rceil$ X]

Note-se que que a notação *[exp1 exp2]* indica aplicação da expressão *exp1* à expressão *exp2*.

Diz-se que um determinado indivíduo *j* do domínio é uma instância do conceito *c* quando *j* satisfaz o valor semântico de *c*. Neste caso, diz-se que a a aplicação *[c j]* produz *verdadeiro* como resposta. Exemplo: o indivíduo *maria* é instância do conceito *mulher*.

- **"Roles"**

São relações entre os indivíduos das classes denotadas pelos conceitos e outros indivíduos do mundo. Podemos definir sua sintaxe através da representação Backus-Naur abaixo:

<role> ::= <role-atômico> |
<role-atômico> & <role> % representa um "role" não atômico;
<role-atômico> ::= <identificador>

Um conceito também pode ser definido a partir de restrições impostas sobre os "roles", conforme indica a nova expressão do tipo *r(c)* que introduziremos agora. Tal expressão representa o conceito *qualquer que seja o segundo argumento do "role" r, ele deve ser uma instância do conceito c*. Por exemplo, a expressão *membro-time(homem)* representa o conceito *todos os membros do time são homens*, onde *membro-time* representa o "role" *r* e *homem* representa o conceito *c*. Chamemos este novo conceito de *cc*. Consequentemente, se a aplicação *[cc vasco]* é avaliada como sendo verdadeira, então *vasco* é uma instância do conceito *cc*, isso é, todos os membros de *vasco* são *homens*. Acrescentando esta nova definição de conceito, expandimos as tabelas onde definimos conceitos e valores semânticos de conceitos para:

- **Expansão da definição de conceitos:**

<conceito> ::= <conceito-atômico> |
<conceito-atômico> & <conceito> | % representa um conceito não atômico;
<role>(<conceito>)

<conceito-atômico> ::= <identificador>

- **Expansão da tabela de valores semânticos:**

Conceito	Tipo	Valor Semântico
c	atômico	$\lambda X\ [\lceil c \rceil X]$
c1 & c2	não atômico	$\lambda X\ [\lceil c1 \rceil X] \& [\lceil c2 \rceil X]$
r(c)	não atômico	$\lambda X \forall Y\ [\lceil r \rceil X Y] \rightarrow [\lceil c \rceil Y]$

2.3.1. Formalismo Híbrido de Representação do Conhecimento:

É um formalismo que, tal como KL-ONE [64], divide a representação do conhecimento de um sistema em no mínimo dois grupos: Terminológico e Assertivo [50, 66, 72].

- **Representação do Conhecimento Terminológico:** assim como as redes semânticas e os quadros (em inglês, "frames"), a representação do conhecimento terminológico é uma representação centrada no objeto. A parte do conhecimento representada pela terminologia corresponde ao conhecimento do significado dos conceitos e "roles" (conhecimento intensional). Um formalismo terminológico deve fornecer meios de se descreverem novos "roles" e conceitos no sistema. Abaixo mostramos um formalismo terminológico bem simples:

```
<terminologia>          ::= {<introdução_termo> | <restrição>}
<introdução_termo>      ::= <introdução_conceito> |
                            <introdução_role>
<introdução_conceito>   ::= <conceito_atômico> = . <conceito> |
                            <conceito_atômico> ≤ .. <conceito>
<introdução_role>       ::= <role_atômico> = . <role> |
                            <role_atômico> ≤ .. <role>
<conceito>              ::= <conceito-atômico> |
                            <conceito-atômico> & <conceito> |     % representa
                                                                    conceito não
                                                                    atômico;
                            <role>(<conceito>)
<conceito-atômico>      ::= <identificador>
<role>                  ::= <role_atômico>
<role_atômico>          ::= <identificador>
<restrição>             ::= (disjunto <conceito_atômico> <conceito_atômico>)
```

Além de introduzir conceitos primitivos, o símbolo $\leq$ .. também pode ser usado para introduzir "roles" primitivos no sistema ("roles" cujas descrições não definem com precisão a relação que exprimem).

Uma restrição é usada para indicar que dois conceitos atômicos são disjuntos, isso é, que qualquer indivíduo que seja instância de um deles não pode ser instância do outro. Exemplo:

<retrição> ::= (disjunto <homem> <mulher>).

Neste sistema não nos preocuparemos com a descrição de conceitos e "roles" primitivos. Nós partimos do princípio de que eles são termos básicos reconhecíveis pelo sistema.

Exemplo de terminologia:

Considerem-se *animal*, *carne* e *vegetal* como conceitos primitivos *e alimenta-se_de* como "role" primitivo. Uma terminologia pode ser descrita como:

herbívoro ::= animal & alimenta-se_de(vegetal)
carnívoro ::= animal & alimenta-se_de(carne)

- **Representação do Conhecimento Assertivo:**

O conhecimento assertivo refere-se ao conhecimento do mundo, associando conceitos e "roles" com indivíduos do mundo (conhecimento extensional). Num sistema de representação híbrido, a informação sobre o mundo (assertivo) é fornecida através de termos (conceitos e "roles") descritos na terminologia. Abaixo fornecemos um formalismo assertivo de representação:

<descrição_do_mundo> ::= (<descrição_de_objeto> |
<descrição_de_relação>)
<descrição_de_objeto> ::= (<conceito_atômico><objeto>)
<descrição_de_relação> ::= (<role_atômico> <objeto> <objeto>)

A seguir mostramos um exemplo de descrição de um mundo a que chamaremos W1:

(herbívoro objeto1)

(carnívoro objeto2)

Definição de modelo de uma descrição de mundo: seja D, o mundo, um conjunto qualquer. Sejam No, Nc, Nr conjuntos de objetos, de conceitos atômicos e de "roles" atômicos, respectivamente. Seja I, chamada função de interpretação, uma função:

$$I: \begin{array}{l} No \rightarrow D \\ Nc \rightarrow 2^{D} \\ NR \rightarrow 2^{DxD} \end{array}$$

sendo injetiva em No. Um par μ = <D,I>, chamado interpretação, satisfaz uma descrição δ, com notação $|=_{\mu} \delta$, sob as seguintes condições:

$|=_{\mu} (c\ o)$ sse $I(o) \in I(c)$

$|=_{\mu} (r\ o\ p)$ sse $<I(o), I(p)> \in I(r)$

Uma interpretação é um modelo de uma descrição do mundo W, com notação $|=_{\mu} W$, sse ela satisfaz a todas as descrições em W [50, 51].

Se considerarmos que *No = {objeto1, objeto2, objeto3, objeto4}*, *Nc = {vegetal, carne, animal}*, *NR = {alimenta-se-de}* e *D = {leão, pernalonga, cenoura, filé}*, pode-se citar a interpretação I que se segue como um modelo do mundo W1 descrito há pouco:

{<objeto1, pernalonga>, <objeto2, leão>, <objeto3, cenoura>, <objeto4, filé>}

{<vegetal, <<leão, 0>, <pernalonga, 0>, <cenoura, 1>, <filé, 0>>>
<carne, , <<leão, 0>, <pernalonga, 0>, <cenoura, 0>, <filé, 1>>>
<animal, , <<leão, 1>, <pernalonga, 1>, <cenoura,0>, <filé, 0>>>}

I:

{<alimenta-se-de , < <<leão, leão>, 0>, <<leão, pernalonga>, 0>,
<<leão, cenoura>,0>, <<leão, filé>, 1>,
<<pernalonga, pernalonga>,0>, <<pernalonga, leão>, 0>,
<<pernalonga, cenoura>, 1>, <<pernalonga, filé>, 0>,
<<filé, filé>, 0>, <<filé, leão>, 0>,
<<filé, pernalonga>, 0>, <<filé, cenoura>, 0> > }

2.3.2. Lógica Terminológica

A Lógica Terminológica [23, 43, 50, 51], é uma lógica de primeira ordem que lida com "roles" (relações binárias) e conceitos (relações unárias) segundo métodos de inferência baseados na subsunção. Tentemos definir um pouco mais formalmente a *subsunção*. Chama-se EXTENSÃO de um conceito c , com notação c^e , ao conjunto :

$\{X \mid \lceil c \rceil (X)\}$

Note-se que tal conjunto é formado pelos indivíduos do domínio que são instância do conceito c. Diz-se que um conceito c subsume um conceito d se $c^e \supseteq d^e$. Exemplo:

c1 = humano
c2 = humano & homem
c3 = humano & homem & brasileiro

Neste caso, *c2* subsume *c3* e *c1* subsume *c2* e *c3*.

A Lógica Terminológica se apresenta como um formalismo útil ao nosso problema em decorrência de três de seus elementos que nos servirão como ferramentas poderosas, a saber:

- usaremos os conceitos como elementos capazes de caracterizar as variáveis de nosso sistema (por exemplo, diremos que uma certa variável deve ser instância de um certo conceito, em outras palavras, deve satisfazê-lo);

- usaremos os "roles" como elementos capazes de relacionar duas variáveis do sistema;

- usaremos a *subsunção* como otimizador dos processos de construção da base de conhecimentos e de prova. Durante o precesso de construção incremental da base de conhecimentos, a *subsunção* será usada como critério de detecção de redundância, de contradições etc. Além disso, ela será responsável pela semântica adotada durante a unificação efetuada pelo provador.

Conforme veremos mais tarde, a representação final das exigências a ser inserida na base de conhecimentos (produzida pela análise sintática e semântica das mesmas) corresponde a uma fórmula bem formada do Cálculo de Predicados em que os termos da Lógica Terminólogica- relações unárias (conceitos) e binárias ("roles")- ficam em posição específica e bem definida na fórmula. Há duas vantagens principais na adoção deste tipo de representação: a primeira é que o provador saberá onde buscar na fórmula os elementos aos quais aplicará o processo de unificação semântica guiada pela subsunção. A segunda é conseguir uma boa modularidade na representação, conforme discutiremos a seguir [52].

Pode-se escrever um predicado n-ário como uma conjunção de predicados binários. Para tanto, postula-se a existência de um evento particular expresso no predicado e de um conjunto contendo os eventos dessa mesma natureza. Além disso, para cada argumento do predicado original inventa-se um novo predicado binário que relaciona o valor do argumento ao evento postulado. Exemplificando, o predicado ternário:

entregou(joão, maria, livro),

que representa a frase:

João entregou um livro a Maria,

pode ser convertido para:

$(\exists X)$[el(X, eventos-entregar) $\wedge$ entregador(X, joão) $\wedge$ receptor(X, maria) $\wedge$
objeto(X, livro)],

onde o predicado *el* representa a relação *pertence*, *eventos-entregar* corresponde ao conjunto postulado e o quantificador existencial introduz o evento postulado. "Skolemizando" esse resultado, obtemos a fórmula final FF:

FF: el(g1, eventos-entregar) $\wedge$ entregador(g1, joão) $\wedge$ receptor(g1, maria) $\wedge$
objeto(g1, livro),

que representa a conversão do predicado ternário *entregar* para uma forma de predicados binários. Uma vantagem dessa conversão é a modularidade. Realmente, analisemos o caso em que precisemos incluir ao sistema de representação do exemplo anterior o momento em que o evento *entregou* aconteceu. Sem a conversão, precisaríamos de introduzir um quarto argumento ao predicado *entregou*, bem como de introduzir grandes mudanças ao sitema de controle e às regras de produção que fazem referência a tal predicado. Com a conversão, a inclusão do momento do evento se torna uma mera questão de se adicionar a FF uma nova conjunção com o predicado binário *momento (g1, _)*.

Nas próximas seções veremos que a representação da exigência na base de conhecimentos é uma representação híbrida a cuja porção terminológica se aplicará a subsunção, tanto como critério de construção e simplificação da base, como como critério semântico da unificação [61].

2.4. Gramática

É definida como uma quádrupla G = (V, T, P, S), onde:

- V: conjunto finito não vazio do vocabulário total da linguagem (terminais mais não terminais).
- T: alfabeto terminal da linguagem (subconjunto não vazio de V).
- P: conjunto finito de regras de produção do tipo $a \rightarrow b$, onde a pertence ao vocabulário não terminal e b ao vocabulário terminal ou não.
- S: pertence ao conjunto dos não terminais da linguagem (S é chamado símbolo inicial da gramática).

Diz-se que uma frase não tem sentido ou significado quando apresenta terminais não pertencentes a T ou uma ordem não prevista pelas regras de produção.

Chama-se *linguagem* ao conjunto de fórmulas bem formadas da gramática.

Exemplo de gramática:

T: {josé, maria, ama}

V: {josé, maria, ama, frase, nome-próprio, verbo, sintagma-verbal}

S: frase

P: {frase → nome-próprio, sintagma-verbal,

nome-próprio → josé,

nome-próprio → maria,

sintagma-verbal → verbo, nome-próprio,

verbo → ama}

O estudo das relações sentido/significado entre palavras da frase e entre as frases entre si é chamado de *semântica*.

O estudo do uso da linguagem dentro de um contexto (visando a descobrir a intenção contida nele) é denominado pragmática.

2.4.1. Gramática Transformacional

Foi concebida por Chomsky. Ela propõe dois níveis de estrutura para tratamento da linguagem: a estrutura superficial (referente à sintaxe) e a estrutura profunda (referente à semântica). Sentenças do Inglês tais como *The cook is baking* e *The cake is baking* têm mesma estrutura superficial mas diferentes estruturas profundas. O inverso ocorre com as sentenças do Português: *João escalou a montanha* e *A montanha foi escalada por João*, que têm a mesma estrutura profunda mas diferentes estruturas superficiais.

As gramáticas transformacionais, apesar de serem criticadas por não efetuarem simultaneamente as análises sintática e semântica, o que aumentaria sua eficiência, são bastante usadas.

2.4.2. Gramáticas de Caso

Casos são especificações que ajudam o usuário da linguagem a identificar a função que um determinado elemento exerce em um sentença [28] . Pode-se citar como exemplo de casos as declinações do Latim e os pronomes pessoais do caso reto (definindo sujeito) e do caso oblíqüo (definindo complemento verbal).

Sentenças de mesmo conteúdo tais como *José ama Maria* e *Maria é amada por José*, apesar de apresentarem diferentes estruturas sintáticas, encaixar-se-ão dentro de um caso único previsto para o verbo *amar*, a saber:

(ama (agente ...)
(paciente ...)),

onde *agente José* e *paciente Maria* são casos semânticos que relacionam solidamente o verbo com os demais argumentos da frase.

As gramáticas de caso são extensões das gramáticas transformacionais.

Sentenças do Inglês como *The cake is baking* e *The cook is baking* são distintas, pois *cake* e *cook* têm casos diferentes. Como casos diferentes não podem ser

misturados, sentenças sem sentido, tal como *The cake and the cook are baking*, são rejeitadas pela gramática.

A gramática de casos elimina ambigüidades estipulando uma ordem para os casos (por exemplo, estipula que o sujeito é o caso de mais alto nível).

2.5. Sistemas Aplicativos

Os sistemas aplicativos [21] são sistemas dotados de:

a) Um conjunto (eventualmente vazio) de objetos atômicos (constantes ou variáveis) e de operadores atômicos (constantes ou variáveis).

b) De uma operação de aplicação considerada como operação primitiva que permite construir expressões derivadas (objetos ou operadores derivados) a partir de átomos (objetos e operadores). A noção de tipos é introduzida para que se concebam diferentes formas de expressões. Logo, cada tipo representará uma classe de expressões evidenciadas por apresentarem determinadas características em comum. Por exemplo, no domínio de certas linguagens de programação, podemos considerar a classe de inteiros naturais, a classe dos reais, a classe dos "booleanos", a classe das cadeias de caracteres etc, cada uma delas com um tipo particular que a represente. No domínio da análise das linguagens naturais dotadas de *casos* podemos distinguir a classe do acusativo, do genitivo, do ablativo etc. A partir dessas classes de expressões chamadas de *tipos elementares*, podemos construir, através de operaçõe explícitas, todos os tipos derivados. Chamamos *Sistema Aplicativo Tipado* a um sistema aplicativo a cujas expressões associamos tipos. Restrições impostas sobre os tipos podem ser usadas para guiar aplicações de expressões lambda (reduções beta), impedindo as indesejadas e propiciando as adequadas.

O operando de um operador pode ser simples ou estruturado. Caso aceitemos operandos estruturados, devemos adotar, além de uma operação primitiva de aplicação, outras primitivas que nos permitam a construção de objetos. Como exemplo, podemos adotar a operação primitiva *Produto Cartesiano Finito* para

construir t-uplas de objetos tais como <a,b>, <a1, a2, ..., an>. Com isso, podemos trabalhar com operadores n-ários.

Se S é um conjunto de classes de expressões, definimos recursivamente o conjunto Tip[S] como sendo:

- As classes de S são tipos de Tip[S];

- As expressões da forma $t_1 * ... * t_n$ representam a tupla $<a_1, ..., a_n>$, onde cada a_j (com j variando de 1 a n) tem o tipo t_j. Tais expressões representam o chamado *tipo cartesiano*. Um tipo cartesiano pertence a *Tip[S]* quando cada um de seus tipos t_j também pertence a *tip[S]*. O símbolo * representa um operador de concatenação.

- *(F t_1 t_2)* é um tipo de *Tip[S]* chamado tipo funcional, onde F representa um operador formador de tipos, t_1 representa o tipo do argumento ao qual F deve ser aplicado e t_2 representa o tipo da expressão resultante da aplicação. Exemplo: se temos que r representa a classe dos reais, a soma de dois reais produzindo um real representa-se por: *(+ (r r) r)*.

De acordo com a definição dada acima, se considerarmos, por exemplo, o conjunto de classes $S = \{n, p\}$, temos como amostras de tipos de *Tip[S]:*

n

p

(F n n)

(F (F n n) (F n n))

(F n * n * n p) (neste caso F representa uma função que ao ser aplicada deve receber três argumentos do tipo n, produzindo um resultado do tipo p).

(F (F n n) * (F n n) (F n n))

Para efeito de simplificação, desconsiderar-se-ão os parênteses sempre que isso não causar ambiguidade. Por exemplo, escrever-se-á *(F n n)* como *F n n*.

Chama-se ***sistema aplicativo de tipos generalizados*** (com produtos cartesianos finitos) a um conjunto S de classes associado às 3 regras de geração de *Tip[S]* especificadas acima.

Um ***sistema de categorias de base S*** corresponde a um sistema aplicativo de tipos generalizados ampliado por dois esquemas de simplificação, um a direita (*Sd*) e outro a esquerda (*Se*), a saber:

Sd: (F1 t_1 t_2) * $t_1 = t_2$ para todo t_1, t_2 em *Tip[S]*

Se: t_1 * (F1 t_1 t_2) = t_2 para todo t_1, t_2 em *Tip[S]*

A regra *Sd* afirma que a concatenação de um tipo funcional (F1 t_1 t_2) com o tipo t_1 , onde t_1 vem à direita do tipo funcional, produz o tipo funcional simplificado t_2 (neste caso a simplificação é obtida por concatenação sintática).

A regra *Se* afirma que a concatenação de um tipo funcional *(F1* t_1 t_2 *)* com o tipo t_1, onde t_1 vem à esquerda do tipo funcional, produz o tipo funcional simplificado t_2 .

Observa-se que as regras de simplificação estipulam a posição sintática que operador deve ocupar com relação ao operando. Por exemplo, se *g* representa um tipo *grupo nominal* de uma gramática do Português, tanto o artigo indefinido *um* quanto o adjetivo *belo* podem ser definidos pelo tipo *(Op_esq g g)*, isso é, ambos representam um operador *Op_esq* que ao ser aplicado a um argumento de tipo *g* que o suceda em posição, produz uma expressão do tipo *g*. Assim, o tipo da frase *um belo homem* pode ser obtido através da concatenação:

(Op_esq g g) * (Op_esq g g) * g,

que obviamente produz o tipo *g* correspondente ao grupo nominal *um belo homem*.

Por comodidade, a partir daqui utilizaremos como símbolos de operação à esquerda ou à direita os mesmos utilizados por Desclés [21], isso é, " \ " representa uma aplicação em que o operando se encontra à esquerda do operador e " / "representa uma aplicação onde o operando se encontra à direita do operador. Logo:

a / b * b $\rightarrow$ a (entenda-se a seta como *produz*)

b * a \ b $\rightarrow$ a

2.6. Gramática de Categorias

Uma gramática de categorias é definida por:

- Um sistema de categorias de base S.

- Um conjunto (geralmente finito) de elementos chamados de *unidades*. Aos elementos que têm tipo em S chamamos *unidades de base* e aos que têm tipo da forma (t_1 / t_2) ou $(t_1 \setminus t_2)$ chamamos *unidades funcionais*.

- Uma lei de justaposição de unidades.

- Um processo de atribuição de tipos à direita (Ad) ou à esquerdq (Ae) que de forma simplificada podem ser representados, respectivamente, por:

Ad: $(t_2 / t_1):X * t_1:Y \rightarrow t_2: XY.$

Deve-se ler esse processo como: uma expressão X do tipo (t_2 / t_1) aplicada a uma expressão Y do tipo t_1 localizada à sua direita produz uma expressão XY do tipo t_2.

Ae: $t_1:Y * (t_2 \setminus t_1):X \rightarrow t_2: XY.$

Analogamente, tal expressão pode ser lida como: uma expressão X do tipo

$(t_2 \setminus t_1)$ aplicada a uma expressão Y do tipo t_1 localizada à sua esquerda produz uma expressão XY do tipo t_2 .

Uma linguagem de categorias de tipo t gerada por uma gramática de categorias G é constituída de todas as expressões que têm o mesmo tipo t.

Uma gramática de categorias tem como propósitos:

- Descrever todas as categorias sob forma de tipos de base e de tipos funcionais com produtos cartesianos finitos.

- Prover um dicionário onde cada entrada é uma expressão a qual é atribuída no mínimo uma categoria.

- Verificar a *boa conexão* de expressões de um dado tipo efetuando simplificações segundo os esquemas *Ae* e *Ad*.

- Gerar expressões bem formadas efetuando aplicações apropriadas entre operadores e operandos de tipos compatíveis.

O uso de gramáticas de categorias para descrever a sintaxe de línguas naturais teve suas origens nos trabalhos de Y. Bar-Hillel e se prolonga até hoje (tendo também passado por Montague). Estes trabalhos visam a descrição e a verificação de uma boa conexão sintática entre as frases da linguagem natural. Podemos ilustrar com o exemplo de análise de uma das frases de nosso domínio de trabalho (exigências):
Seja o dicionário:

unidade de base	*tipo sintático*	*categoria*
ioi-gs	n	nome comum
vehicle	n	nome comum
the	n / n	artigo
controls	((f \ n) / n)	verbo

onde o tipo *n* representa um grupo nominal e o tipo *f* representa uma frase bem formada. A seqüência da análise para a exigência *The ioi-gs controls the space-vehicle* é:

The	ioi-gs	controls	the	véhicule
n / n	n	((f \ n) / n)	n / n	n
n	((f \ n) / n)	n / n	n	
n	((f \ n) / n)	n		
n	f \ n			
f				

2.7. Gramática Aplicativa

Uma gramática aplicativa é uma extensão das gramáticas de categorias. Tal extensão inclui um operador prefixo O que gera recursivamente o sistema de tipos sintáticos. Numa gramática aplicativa consideramos três tipos de expressões: termos, frases e operadores. Os termos são os elementos aos quais podemos associar referências. Dizemos que eles têm tipo *t*. As frases representam fórmulas bem formadas as quais atribuímos o tipo *f*. Os operadores são usados na produção de novas expressões. Eles são do tipo Oxy, onde *x* é o tipo do operando e *y* é o tipo da nova expressão produzida. Dessa forma, um nome próprio, por exemplo, tem tipo *t* e um verbo transitivo tem tipo OtOtf (ao aplicarmos um verbo transitivo de tipo OtOtf a um termo de tipo *t* produzimos uma expressão de tipo Otf que, aplicada a um termo, produz uma frase). Exemplificando, na frase *João ama Maria*, *João* e *Maria* são do tipo *t* e *ama* é do tipo OtOtf. A aplicação de *ama* a *Maria* produz a expressão *ama Maria*, de tipo Otf, que aplicada a *João* produz a frase *João ama Maria*.

S representa o conjunto de classes da gramática e *Typ(S)* representa o conjunto básico de tipos da gramática. No nosso exemplo anterior temos que S = {termos, frases}. Logo, *Typ(S)* = *{t, f}*. Consideramos como tipos da gramática aplicativa:

- Os tipos de Typ(S).
- Os tipos Oxy, onde *x* e *y* são tipos da gramática.

2.8. Princípio Aplicativo e Expressões Combinadoras:

Distintamente da Lógica Clássica, onde um predicado n-ário é aplicado aos seus n argumentos de uma só vez, constata-se que na linguagem natural há diferenças na força da ligação entre um predicado e seus diversos argumentos. Por exemplo, observa-se que um predicado binário mantém uma relação mais estreita com seu segundo argumento (objeto direto) que com seu primeiro argumento (sujeito). Além disso, a intensidade desses inter-relacionamentos muda de frase para frase. Desclés [21] cita como exemplos as frases:

a) Pegar um rato com uma ratoeira
b) pegar um resfriado

Na primeira frase o objeto *rato* é completamente incorporado ao predicado *pegar*, chegando mesmo a perder sua autonomia; já na segunda frase, o objeto *resfriado* é praticamente autônomo com relação ao predicado *pegar*. Fenômenos como esses requerem um sistema de aplicação diferente da Lógica Clássica para o tratamento da linguagem natural; nesse novo sistema a aplicação de um predicado a seus argumentos não é feita necessariamente em uma só etapa, conforme se observou na análise aplicativa da frase *João ama Maria* mostrada há pouco.

2.8.1. Princípio Aplicativo de Schonfinkel

O princípio aplicativo é a base da lógica introduzida por M.Schonfinkel (1924) e da lógica combinatória de Curry (1958), sendo que ambas propõem uma aplicação não simultânea de um predicado a seus argumentos. O princípio de Schonfinkel é:

Um operador n-ário f_n pode ser representado por um operador unário Curry(f_n) equivalente que, ao ser aplicado a um operando, produz a representação de um operador (n - 1)-ário.

Por exemplo, o operador Curry que representa um predicado binário é:

$$Curry(f_2) = \lambda y.\ (\lambda x.\ f_2(x, y))$$

Realmente, o predicado binário *é maior que* pode ser representado pelo seguinte opreador P1' de Curry:

$$P_1' \equiv \lambda y.\ (\lambda x.\ (x \text{ é maior que } y)).$$

P_1' é um operador unário. Quando aplicado, por exemplo, a *3*, P_1' produz o seguinte predicado unário P_1'':

$$P_1'' \equiv \lambda x.\ (x \text{ é maior que } y).$$

Quando P_1'' é aplicado, por exemplo a *5*, produz como resultado a proposição *5 é maior que 3*, cujo valor é *verdadeiro*.

A gramática aplicativa utiliza os operadores de Curry.

2.8.2. Expressões Combinadoras

São sistemas aplicativos que atuam segundo o princípio aplicativo e que podem ser gerados pelas regras:

(1) Os átomos (operandos e operadores absolutos, sejam variáveis ou constantes) são expressões combinadoras.

(2) Se X e Y são expressões combinadoras, então (X Y) é uma expressão combinadora em que X é um operador que se aplica ao operando Y.

Para simplificar os parênteses, admita-se a associatividade à esquerda no processo de redução. Assim, uma aplicação X Y_1 Y_2 Y_3... Y_n indica que o resultado da aplicação do operador X a Y_1 deve ser aplicado a Y_2, que o resultado desta última aplicação deve ser aplicado a Y_3 e assim sucessivamente até a aplicação final a Y_n.

Expressões Combinadoras Tipadas:

São expressões combinadoras que seguem o princípio aplicativo e cujas aplicações são guiadas por um grupo de tipos formados segundo as seguintes regras (a partir de um conjunto S de classes):

(3) As classes de S são tipos.

(4) Se *x* e *y* são tipos, então Oxy é um tipo.

Uma expressão de tipo Oxy admite como operando apenas expressões do tipo *x*, sendo que o resultado dessa aplicação é do tipo *y*. Claramente a introdução de tipos cria um mecanismo de controle para as aplicações, guiando-as de uma maneira mais objetiva. Representa-se isso pela regra:

(5) Se Oxy:X e x:Y, então y:XY,

conforme será mostrado a seguir. Além disso, os combinadores podem-se combinar entre si para definir um novo operador mais complexo. A atuação de um combinador se faz de acordo com o previsto em sua regra de β-redução.

Um sistema genótipo é um sistema aplicativo definido pelas 5 regras descritas acima. Para se tratar a linguagem natural, torna-se conveniente a uilização de um sistema genótipo particular denominado linguagem genótipo. Neste sistema os operadores são chamados de *combinadores* e permitem a criação de operadores e predicados mais complexos. No capítulo 3 veremos que a análise sintática e semântica implementada neste projeto se serve dos combinadores.

2.9. Processo de "Skolemização" de Fórmulas Bem Formadas

É o processo que converte fórmulas bem formadas do Cálculo de Predicados de Primeira Ordem em sua forma Skolem padrão. No próximo capítulo será visto que

a análise sintática e semântica das exigências é processada de modo a produzir uma representação "skolemizada" das exigências.

Antes de se definir forma Skolem de uma fórmula, precisa-se conhecer a definição de *forma conjuntiva normal* e de *forma prenex* [4, 12].

Diz-se que uma fórmula proposicional α qualquer está na forma conjuntiva normal (FCN) se, e somente se, α é uma conjunção $\beta_1 \wedge ... \wedge \beta_n$, $(n \geq 1)$, tal que, para cada i $(1 \leq i \leq n)$, β_i é uma disjunção de literais, ou um literal.

As seguintes fórmulas são exemplos de FCN:

$(\neg p \vee r) \wedge (q \vee r)$

$p \wedge (q \vee \neg r)$

$p \wedge r$

Para se transformar um fórmula em FNC usam-se as equivalências lógicas do Cálculo das Proposições. As principais equivalências lógicas são:

a) $\alpha \Leftrightarrow \neg \neg \alpha$

b) $\alpha \wedge \beta \Leftrightarrow \beta \wedge \alpha$

c) $\alpha \vee \beta \Leftrightarrow \beta \vee \alpha$

d) $(\alpha \leftrightarrow \beta) \Leftrightarrow (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$

e) $\alpha \rightarrow \beta \Leftrightarrow \neg \alpha \vee \beta$

f) $\neg(\alpha \wedge \beta) \Leftrightarrow \neg\alpha \vee \neg\beta$

g) $\neg(\alpha \vee \beta) \Leftrightarrow \neg\alpha \wedge \neg\beta$

h) $\alpha \vee (\beta \wedge \gamma) \Leftrightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

i) $\alpha \wedge (\beta \vee \gamma) \Leftrightarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$

j) $(\alpha \rightarrow \beta \wedge \gamma) \Leftrightarrow (\alpha \rightarrow \beta) \wedge (\alpha \rightarrow \gamma)$

k) $(\alpha \vee \beta \rightarrow \gamma) \Leftrightarrow (\alpha \rightarrow \gamma) \wedge (\beta \rightarrow \gamma)$

l) $\alpha \rightarrow (\beta \rightarrow \gamma) \Leftrightarrow (\alpha \wedge \beta) \rightarrow \gamma$

m) $(\alpha \rightarrow \beta) \rightarrow \gamma\ \alpha \Leftrightarrow (\alpha \vee \gamma) \vee (\beta \rightarrow \gamma)$

n) $(\alpha \wedge \neg\beta) \rightarrow \gamma \Leftrightarrow \alpha \rightarrow (\beta \vee \gamma)$

Por exemplo, a FNC da fórmula $\alpha \leftrightarrow \beta$ pode ser encontrada pela aplicação das equivalências lógicas *d* e *e*, o que produz como resultado a FNC:

$$(\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$$

Uma fórmula do Cálculo de Predicados α está na *forma normal conjuntiva prenex* se, e somente se, $x^1,\ldots, x^n$ são variáveis distintas e α é da forma $Q^1(x^1),\ldots, Q^n(x^n)M$, onde cada Q^i representa um quantificador e M está na forma normal conjuntiva (em M não ocorrem quantificadores).

Para se transformar uma fórmula α do Cálculo de Predicados em sua forma normal conjuntiva prenex, devem-se seguir os seguintes passos:

- Usando as equivalências do Cálculo Proposicional, transformar *a*, caso possível, em sua forma normal conjuntiva.
- Caso seja necessário, renomeie as variáveis.
- Use as equivalências:

$$\neg\forall x\, \beta(x) \Leftrightarrow \exists x\, \neg\beta(x)$$
$$\neg\exists x\, \beta(x) \Leftrightarrow \forall x\, \neg\beta(x)$$

- Usando as equivalências lógicas:

a) $Qx\beta(x) \vee \gamma \Leftrightarrow Q(x)\, [\beta(x) \vee \gamma]$ x não ocorre em γ

b) $Qx\beta(x) \wedge \gamma \Leftrightarrow Q(x)\, [\beta(x) \wedge \gamma]$ x não ocorre em γ

c) $\forall x\beta(x) \wedge \forall(x)\, \gamma(x) \Leftrightarrow \forall(x)\, [\beta(x) \wedge \gamma(x)]$

d) $\exists x\beta(x) \vee \exists(x)\, \gamma(x) \Leftrightarrow \exists(x)\, [\beta(x) \vee \gamma(x)]$

e) $Q^1x\beta(x) \vee Q^2x\gamma(x) \Leftrightarrow Q^1(x)\, Q^2y\, [\beta(x) \vee \gamma(y)]$

f) $Q^1x\beta(x) \wedge Q^2x\gamma(x) \Leftrightarrow Q^1(x)\, Q^2y\, [\beta(x) \wedge \gamma(y)]$

mova os quantificadores para a esquerda.

- Repita a operação até que α seja $Q^1 x^1 \ldots Q^n x^n \beta$ e β esteja na forma normal conjuntiva e em β não ocorram quantificadores.

Seguindo-se este algoritmo, chega-se que a forma normal conjuntiva prenex da fórmula:

$$\forall x \, \forall y \, [\exists z(P(x,z) \wedge P(y, z)) \rightarrow \exists u \, R(x, y, u)]$$

é a fórmula:

$$\forall x \, \forall y \, \forall z \, \exists u \, (\neg P(x,z) \vee \neg P(y, z) \vee R(x, y, u))$$

A forma Skolem padrão conjuntiva de uma fórmula da Lógica de Primeira Ordem é uma fórmula que apresenta as seguintes propriedades:

- Ela se encontra na forma prenex normal conjuntiva.
- A matriz da fórmula está na forma conjuntiva normal.
- Sem que se prejudique a propriedade da inconsistência, os quantificadores existenciais são eliminados do prefixo através das funções Skolem, conforme descrito a seguir:

Seja F uma fórmula normal prenex do tipo $(Q_1 x_1) \ldots (Q_n x_n) M$, onde M é uma forma normal conjuntiva e $Q_1, \ldots, Q_n$ são os quantificadores de F com suas respectivas variáveis $x_1, \ldots, x_n$. Seja Q_r um quantificador existencial [22] no prefixo $(Q_1 x) \ldots (Q_n x)$, onde $1 \leq r \leq n$. Se nenhum quantificador universal antecede Q_r no prefixo de quantificadores, escolhe-se uma constante *c* distinta de qualquer constante de M, substitui-se todas as ocorrências de x_r em M por *c* e elimina-se $Q_r x_r$ do prefixo. Se $Q_{s1}, \ldots, Q_{sm}$ são quantificadores universais que antecedem Q_r no prefixo de quantificadores, ou seja, se $1 \leq s1 < s2 \ldots < sm < r$, escolhe-se uma função m-ária *f*, diferente de todas as funções de M, substituem-se todas as ocorrências de x_r em M por $f(x_{s1}, x_{s2}, \ldots, x_{sm})$ e elimina-se $(Q_r x_r)$ do prefixo. Eliminados todos os quantificadores

existenciais do prefixo, tem-se, finalmente, a forma Skolem padrão de F. Como exemplo, a "skolemização" da fórmula:

$$\forall x \, \forall y \, \forall z \, \exists u \, (\neg P(x,z) \vee \neg P(y, z) \vee R(x, y, u))$$

produz a fórmula:

$$\forall x \, \forall y \, \forall z(\neg P(x,z) \vee \neg P(y, z) \vee R(x, y, f(x, y, z))).$$

No próximo capítulo será feita uma adaptação da teoria discutida até aqui à análise sintática e semântica das exigências efetuada pelo sistema implementado neste projeto.

3. ANÁLISE SINTÁTICA E SEMÂNTICA

Introdução

O analisador implementado no sistema [16] corresponde a um sistema formal construído de acordo com a teoria do estruturalismo [10, 58]. Este sistema formal define uma gramática aplicativa. As regras de gramática são definidas em termos dos casos do Latim (entretanto, poderia ter sido usado qualquer outro sistema de casos padrão, tal como, por exemplo, os casos do grego [21]). Visando a executar as análises sintática e semântica simultaneamente, o sistema foi implementado de tal maneira que o lingüista que o utilize tenha que definir uma abstração lambda tipada correspondente a cada categoria de caso e a cada categoria de verbo. As demais abstrações lambda, correspondentes às demais categorias, serão automaticamente geradas pelo sistema durante a análise.

Tal como na Gramática de Montague, a análise é executada combinando-se categoria com categoria até que uma sentença seja produzida; paralelamente, combina-se abstração lambda com abstração lambda até que uma fórmula híbrida (contendo uma parte terminológica e uma parte assercional) do Cálculo de Predicados, cujo significado é verdadeiro ou falso, seja produzida. Estas combinações são efetuadas em segundo os critérios de coesão e coerência [7, 35, 37, 56, 65] e são guiadas por um método heurístico da Inteligência Artificial, através dos tipos das abstrações lambda.

Acreditamos que nossas contribuições nesta secção de análise das exigências concentram-se em:

- Implementar um analisador sintático e semântico inteligente que automaticamente gera um conjunto de regras semânticas, dispensando o lingüista de ter de defini-las.

- Usar um método heurístico da Inteligência Artificial para guiar e otimizar o processo de análise.

A seguir apresentamos algumas teorias sobre as quais se fundamenta o analisador implementado e, posteriormente, o próprio analisador.

3.1. Fundamentos Teóricos do Analisador Implementado

3.1.1. Estruturalismo

O estruturalismo é uma teoria adotada por grandes pesquisadores. Dentre eles, destacam-se o grupo de matemáticos franceses denominado Bourbaki [10] e o psicólogo suíço Jean Piaget [58]. A seguir serão expostas resumidamente as idéias de Piaget [58] sobre o estruturalismo.

Uma estrutura é um sistema de transformações que comporta leis enquanto sistema (por oposição às propriedades dos elementos) e que se conserva ou se enriquece pelo próprio jogo de suas transformações, sem que estas conduzam para fora de suas fronteiras ou façam apelo a elementos exteriores. Em resumo, uma estrutura compreende os critérios de totalidade, de transformações e de auto-regulação a serem vistos a seguir:

- Critério da totalidade:

Uma estrutura é, por certo, forma de elementos subordinados às leis que caracterizam o sistema como tal. Essas leis, ditas de composição, não se reduzem a associações cumulativas, mas conferem ao todo propriedades de conjuntos distintas daquelas pertencentes aos elementos.

Exemplo:

Os números inteiros não existem isoladamente e não foram descobertos em uma ordem qualquer para em seguida serem reunidos em um todo. Na verdade, os números inteiros não se manifestam senão em função da própria seqüência dos números, que por sua vez apresenta propriedades estruturais de *grupos*, *corpos*, *anéis* etc bem distintas das que pertencem a cada número, tais como: ser par ou ímpar, ser primo ou divisível por $n > 1$ etc.

- Critério das transformações:

Se a característica das totalidades estruturadas é depender de suas leis de composição, elas são, portanto, estruturantes por natureza. Essa bipolaridade das propriedades de serem sempre e simultaneamente estruturantes e estruturadas é que explica o sucesso da concepção estruturalista. Ora, uma atividade estruturante tem, necessariamente, de representar um sistema de transformações. Todas as estruturas conhecidas, partindo dos *grupos* matemáticos mais elementares até aqueles que regulam, por exemplo, os parentescos, são sistemas de transformações que podem ser *atemporais* ou *temporais*. Por exemplo, a estrutura dos inteiros é *atemporal*, pois *1* + *1* produz imediatamente *2*, e *3* sucede *2* sem intervalo de duração. Já a estrutura de parentesco é *temporal*, pois o ato de se casar e de produzir descendentes depende do tempo. Se não houvesse transformações, as estruturas condundir-se-iam com formas estáticas quaisquer.

- Critério da Auto-regulação

É a terceira característica fundamental das estruturas. Através dela as estruturas se auto-regulam visando a sua auto-conservação. Isso corresponde a uma propriedade de fechamento, ou seja, as transformações inerentes a uma estrutura não conduzem para fora de suas fronteiras e não geram senão elementos que pertençam a ela, respeitando e conservando as suas leis. Assim é que, adicionando-se um número inteiro o outro ou subtraindo-se um número inteiro de outro, obtêm-se sempre números inteiros que obedecem às leis do grupo ativo de tais números.

Por razões históricas e lógicas, torna-se inconveniente dedicar-se a uma exposição crítica do estruturalismo sem começar pelo exame das estruturas matemáticas. Uma vez aceita a definição de estrutura, parece incontestável o fato de que a mais antiga estrutura, conhecida e estudada como tal, seja a de *grupos*, descoberta por Galois. A estrutura dos grupos verdadeiramente encantou os matemáticos do século XIX. Um grupo é um conjunto de elementos (exemplo: conjunto dos números inteiros positivos e negativos) reunidos por uma operação de

composição (exemplo: adição) que, aplicada aos elementos do conjunto, gera um outro elemento do conjunto. O grupo conta também com um elemento neutro (que no exemplo analisado corresponde ao *zero*) que, composto com um outro elemento do conjunto, não o modifica. O grupo conta também com uma operação inversa (no caso particular do exemplo, a subtração) que, composta com a operação direta, fornece o elemento neutro (+ n - n = - n + n = 0). Finalmente, as composições do grupo são associativas ([n + m] + l = n + [m + l]).

Este sistema foi implementado de acordo com as idéias de Bourbaki [10] segundo as quais as estruturas em geral são geradas por um conjunto de estruturas-mãe [3], tais como:

- Estrutura-mãe da Álgebra:

É responsável pela introdução de novos elementos na estrutura pela composição de elementos já existentes.

- Estrutura-mãe da Ordem:

É responsável pelo controle da ordem das composições executadas pela estrutura-mãe da álgebra.

O analisador sintático e semântico implementado baseia-se nas estruturas-mãe da álgebra (composições) e da ordem (seqüência das composições), conforme será visto na secção 3.3.

3.1.2. Coesão e Coerência

Coesão e coerência são critérios que asseguram uma boa formação sintática e semântica das sentenças de uma linguagem.

Resumidamente, coesão corresponde a critérios que precisam ser seguidos durante a composição das categorias sintáticas, de tal maneira a se produzir uma sentença bem formada como resultado. Normalmente, nas línguas modernas os critérios de coesão são estabelecidos pela ordem na qual as palavras precisam organizar-se nas sentenças em função de sua categoria sintática. Esta ordem é fixada pela gramática.

Coerência corresponde a critérios que precisam ser seguidos durante a composição das abstrações-lambda que representam o significado de cada categoria da sentença, de tal forma que no final das composições se obtenha uma fórmula bem formada cujo significado seja verdadeiro ou falso.

Maiores detalhes sobre tais critérios podem ser vistos em [7, 37, 62, 63].

3.1.3. Sistemas Formais

Definição: S é um sistema formal se S possui:

- Um alfabeto Ss numerável que pode ser finito ou infinito.
- Um subconjunto recursivo Fs do conjunto de todas as sequências finitas de Ss. Fs é denominado conjunto das fórmulas bem formadas de Ss.
- Um subconjunto recursivo Cs de Fs denominado axiomas de S.
- Um conjunto Rs de predicados decidíveis definidos sobre Fs denominado Regras de Inferência.

Chama-se dedução a partir das hipóteses {h1, h2, ..., hn} a qualquer seqüência finita de fórmulas *f1, f2, ..., fm*, tal que, para qualquer *j* pertencente a *{1, 2, ..., m}*, ocorre uma das seguintes possibilidades:

- fj é um axioma
- fj é uma das hipóteses
- fj pode ser obtida por aplicação das regras de inferência a fórmulas que a antecedem na seqüência.

3.1.4. A Concepção Semântica da Verdade Segundo Tarski

Em sua tentativa de definir uma semântica apropriada para o conceito de verdade, Tarski [71] tem como objetivo aproximar-se o máximo possível da definição descrita na metafísica de Aristóteles, mesmo reconhecendo a impossibilidade de obter êxito absoluto com esta aproximação. Segundo Aristóteles:

"Decidir que é o que não é, ou que não é o que é, é falso, ao passo que decidir o que é que é, ou o que não é que não é, é verdadeiro".

A seguir resume-se o trabalho de Tarski no qual ele propõe uma maneira de se conceber semanticamente a noção de verdade.

Baseado na concepção clássica de verdade, por exemplo, pode-se dizer que a oração *a neve é branca* é verdadeira se a neve é branca e que é falsa se a neve não é branca. Conseqüentemente, a definição de verdade procurada deve implicar na seguinte equivalência:

A oração "a neve é branca" é verdadeira se e somente se a neve é branca.

Note-se que a oração *"a neve é branca"* aparece entre aspas no primeiro membro da equivalência e sem aspas no segundo membro. Isto porque, no segundo membro, tem-se a própria oração, ao passo que, no primeiro membro, tem-se uma expressão que corresponde ao nome desta oração. Por que batizar (dar um nome a) uma oração? Porque, gramaticalmente, na nossa língua uma expressão da forma *X é verdadeira* não se converterá em uma oração significativa se substituirmos X por uma oração ou por qualquer outra coisa que não seja um nome, já que o sujeito de uma oração só pode ser um nome ou uma expressão que funcione como nome. Além disso, as convenções fundamentais que regulam o uso de qualquer linguagem requerem que toda vez que nos pronunciamos acerca de um objeto, é o nome do objeto que empregamos e não o próprio objeto.

Considere-se uma oração arbitrária que substituiremos pela letra *P*. Formemos o nome desta oração e o substituamos por outra letra, por exemplo, *X*. A verdade individual desta oração pode ser definida pela equivalência (T) abaixo:

X é verdadeira se e somente se P.

Observe-se que tal definição corresponde apenas a uma definição parcial de verdade. Uma definição geral de verdade deve ser, em certo sentido, uma conjunção lógica de todas estas definições parciais.

Conforme temos visto, a definição procurada de verdade não se refere às orações em si, mas aos objetos dos quais elas falam e aos estados que elas descrevem. Por esta razão, propõe-se definir a verdade como um conceito semântico cuja definição é feita através de um outro conceito semântico: a satisfactibilidade, a ser discutido logo mais. Mas isto tem de ser feito com cautela, uma vez que os conceitos semânticos facilmente conduzem a paradoxos. Daí a importância de se especificar adequadamente uma linguagem na qual se definirá o conceito de verdade. Para especificar com precisão a estrutura de uma linguagem devemos, por exemplo, caracterizar inequivocamente a classe das palavras ou expressões a serem consideradas como significativas. Em particular, devem-se indicar as palavras a serem usadas sem serem definidas (termos indefinidos ou primitivos). Devem-se também propor as regras de definição para introduzir termos definidos ou novos. Além disso, devem-se estabelecer critérios para distinguir, dentre a classe das expressões, aquelas a serem chamadas de orações (ou sentenças). Devem-se também formular as condições em que se pode afirmar uma oração da linguagem. Em particular, devem-se indicar todos os axiomas (ou operações primitivas), isto é, orações que tenhamos decidido afirmar sem prova. Finalmente, devem-se fornecer as regras de inferência através das quais podem-se deduzir novas orações afirmadas a partir de outras orações afirmadas previamente. Os axiomas e as orações dedutíveis a partir dos axiomas e das regras de inferência são chamados de teoremas (ou orações comprováveis).

Nas chamadas linguagens formalizadas a especificação de suas estruturas se refere exclusivamente à forma de suas expressões. Em tais linguagens, os teoremas são

as únicas orações que podem ser afirmadas. Incluem-se dentre as linguagens formalizadas a matemática e a física, por exemplo.

No contexto das ciências empíricas, torna-se muito útil a criação de linguagens alternativas que tenham uma estrutura exatamente especificada mas que não sejam formalizadas, onde a afirmabilidade ("assertability") de suas orações não seja condicionada somente pelas suas formas, podendo mesmo depender de fatores de índole não lingüística.

A definição de verdade adquire um significado preciso em linguagens cuja estrutura seja especificada com exatidão. Para as demais linguagens (como a linguagem natural) tal definição tende a se tornar vaga e só pode ser tratada de maneira aproximada. A grosso modo, tal aproximação consiste em substituir a linguagem em questão (por exemplo, a linguagem natural) por outra cuja estrutura exatamente especificada e que seja o mais próximo possível da linguagem que ela substitui. É exatamente isso que se faz quando se implementa os processadores da linguagem natural, isso é, processa-se, na realidade, um subconjunto bem definido dela.

Já dissemos que uma linguagem deve ser definida de modo a não acarretar os paradoxos. Como surgem os paradoxos e como evitá-los?

Considere-se a seguinte oração:

Esta oração não é verdadeira,

cuja verdade pode ser definida pela equivalência abaixo:

s é verdadeira se e somente se esta oração não é verdadeira,

onde *s* representa o nome da oração. Mas, pelo significado do símbolo *s* (que batiza a oração), estabelece-se, empiricamente, que *s* é idêntico à expressão *esta sentença,* uma vez que esta última também batiza a oração. Daí pode-se produzir:

"s" é verdadeira se e somente se s não é verdadeira,

que é obviamente um paradoxo. Analisando-se a origem deste paradoxo, constata-se que ele surge do fato de a oração apresentar as seguintes características: ela contém expressões que correspondem ao nome da própria oração e ela contém também termos semânticos (tal como *verdadeiro*) referindo-se a ela mesma. Chama-se *linguagem semanticamente fechada* àquela que aceita orações com estas características. Logo, a fim de se evitarem contradições (paradoxos) na definição do conceito semântico de verdade, não se deve usar uma linguagem semanticamente fechada para defini-lo.

Assim sendo, se quisermos falar sobre uma linguagem L qualquer não o devemos fazer através da própria linguagem, mas através de uma outra linguagem L' que tenha sua parte lógica mais rica que L (neste caso, L' é chamada de metalinguagem e L de linguagem objeto). Para que se tenha uma idéia do que representa mais ou menos riqueza lógica, se nos limitarmos à teoria lógica dos tipos, para que a metalinguagem seja mais rica que a linguagem objeto, é preciso que ela contenha variáveis de um tipo lógico superior ao das variáveis da linguagem objeto. Logo, os tipos das linguagens tipadas servem como ferramenta para evitar a geração dos paradoxos (a teoria dos tipos foi inicialmente proposta por B. Russel [32] como maneira de resolver o paradoxo que ele detectou na Lógica de Frege).

Analisemos agora a proposta de Tarski para se construir uma definição apropriada de verdade de uma oração. Todas as formulações propostas até aqui para definir tal conceito não se referem à oração em si mesma, mas aos objetos dos quais elas falam e aos estados que ela descreve.

A definição de verdade pode ser obtida através de uma outra noção semântica: a da *satisfação*. A satisfação é uma relação entre objetos quaisquer e as funções proposicionais (*funções proposicionais* é uma locução introduzida por Russel e Whitehead no "Principia Mathematica" - segunda edição- 1925; contudo, a idéia desta locução remonta a Frege por volta de 1879, e se refere a um enunciado ϕx que se torna uma proposição quando a variável livre x recebe uma significação determinada), tal como a função proposicional *x é branca*. Uma oração ou sentença pode ser definida como uma função proposicional que não contenha variáveis livres. Certos objetos satisfazem uma função proposicional caso ela se converta em uma oração verdadeira quando da substituição de suas variáveis livres por estes objetos (por exemplo, o objeto *neve* satisfaz a função proposicional *x é branca*). A definição de satisfação é

recursiva: caso conheçamos os objetos que satisfazem as funções simples que compõem uma função composta, anunciamos as condições nas quais tais objetos satisfazem as funções compostas. Exemplo: certos números satisfazerm a disjunção lógica: $(x > y) \vee (x = y)$ se satisfazem pelo menos uma das funções $x > y$ ou $x = y$. Logo, para uma oração só há duas possibilidades: ou ela é satisfeita por todos os objetos ou por nenhum. Logo, uma oração é verdadeira se é satisfeita por todos os objetos e falsa em caso contrário.

A partir da secção 3.2 veremos que as exigências de nosso sistema são transformadas durante a análise de tal modo a se converterem em uma forma apropriada a esta definição de verdade de Tarski.

3.1.5. Histórico Resumido de Questões Fundamentais Abordadas Por Grandes Nomes Ligados à Lógica, à Filosofia e à Linguagem Natural

Sem dúvida a questão da formalização de teorias sempre atraiu o interesse dos estudiosos dos mais diversos campos de atuação: Aristóteles, Frege, Russel, Tarski, Wittgenstein, Qüine, Montague, Chomsky - difícil listar todos os nomes de grandes pesquisadores que investiram e investem ainda seus esforços na tentativa de formalizar as linguagens.

Nesta secção tentamos focalizar questões importantes abordadas por alguns destes estudiosos que são de extrema importância no tratamento de linguagens. Tais questões estão direta ou indiretamente ligadas à interpretação de sentenças e, obviamente, ao contexto desta tese.

Frege defendia que as imprecisões da linguagem natural a tornam incompatível com a correção das provas que é exigida numa ciência dedutiva [32]. Daí seu esforço no sentido de formalizar o que até então era proposto e discutido sem muita precisão. Deve-se a Frege (1879) a primeira apresentação axiomática da Lógica das Proposições, seguida da formalização proposta por B.Russel e Whitehead (1910-1913).

Certamente é inestimável e inédita a contribuição de Frege no domínio da formalização da Lógica, levantando questões cruciais e propondo soluções de

indiscutível genialidade. Contudo, coube a Bertrand Russel a formalização mais completa da Lógica tal como a conhecemos hoje [32].

É de Frege o seguinte argumento associando valor verdade de uma sentença de uma linguagem formal aos elementos aos quais ela se refere [17]:

"Qualquer par de sentenças tem a mesma referência se ambas as sentenças tiverem o mesmo valor verdade".

A partir deste argumento surgiu a idéia de se associar significado com referência, tal como proposto por Tarski para as linguagens formais (secção 3.1.4), onde uma sentença significa aquilo a que ela se refere. Nas linguagens formais funciona bem a idéia de associar significado de expressões aos elementos aos quais elas se referem. Por exemplo, na matemática é bem coerente de se afirmar que as expressões *2 + 5, 9 - 2* e *21 / 3* têm o mesmo significado (são equivalentes) por apresentarem a mesma referência *7*.

Alguns filósofos da linguagem propõem que se adote esta mesma técnica de associar significado à referência na linguagem natural [17]. Contudo, eles são conscientes dos problemas associados a esta adoção, uma vez que na linguagem natural ela não é assim tão evidente. Por exemplo, as expressões *o presidente do Brasil em 1995, o marido de Ruth Cardoso* e *o vencedor das eleições presidenciais no Brasil em 1994* referem-se todas ao mesmo elemento *Fernando Henrique Cardoso*, sem serem, contudo, equivalentes. Como resolver isso na linguagem natural? Foi Montague (1930 - 1970) quem primeiro propôs uma tratamento formal para este problema através de sua teoria de mundos possíveis, a ser vista na próxima secção.

Como podemos ver, diversas correntes se formam em torno da discussão sobre a maneira de extrair o que uma sentença quer dizer. As teorias correntes definem *intensão* de uma frase declarativa como sendo suas condições de verdade (na língua portuguesa, a palavra *intensão* perdeu esta conotação original e passou a significar veemência, intensidade) e *extensão* como sendo seu valor verdade.

Wittgenstein em seu Tratactus [76] afirma que:

"A expressão do acordo e do desacordo com as possibilidades de verdade das proposições elementares exprime as condições de verdade de uma proposição. A proposição é a expressão de suas condições de verdade".

Nesta linha de raciocínio, podem-se conhecer as condições de verdade de uma frase (intensão) sem que se conheça seu valor verdade (extensão ou referência). Por exemplo, podem-se conhecer as condições de verdade da frase *O rei Artur existiu* sem se conhecer nem seu valor verdade, nem a maneira de verificá-lo ou mesmo de confirmá-lo: o rei Artur existiu se um ser humano portador deste nome e detentor deste trabalho viveu no lugar e na época mencionada nas citações existentes [32].

As correntes contemporâneas que se dedicam ao estudo do significado dividem-se em dois grupos: o grupo dos seguidores da semântica formal e o grupo dos seguidores da intenção comunicativa [69]. O grupo da semântica formal considera que uma sentença diz exatamente o que ela expressa, ou seja, o que está previsto no seu significado convencional. Já o grupo da intenção comunicativa considera que uma sentença diz o que o falante tem a intenção de dizer, e isso pode diferir de seu significado convencional. Por exemplo, quando um rapaz que está só se aproxima de uma moça sentada sozinha a uma mesa de bar e lhe diz:

Estou me sentindo muito solitário,

caso esta moça interprete tudo segundo a linha da semântica formal, ela considerará, para desgosto do pobre rapaz, que ele apenas está dizendo que está se sentindo só, ao passo que se ela fizer esta mesma interpretação nos moldes da intenção comunicativa, ela pode entender, com esta simples frase, que o rapaz está esperando ansioso um convite para se sentar com ela à mesa.

Incluem-se no grupo da semântica formal nomes como Chomsky, Frege, Montague e o Wittgenstein da primeira fase. Como integrantes do grupo da intenção comunicativa, citem-se Grice, Austin e o Wittgenstein da segunda fase.

Como o domínio da linguagem natural com que lidamos neste trabalho é bem objetivo, restrito e específico, adotamos a linha da semântica formal e a estratégia de

Tarski de associar significado a valores verdade (referência) durante a análise das exigências.

3.1.6. Montague

Richard Montague (1930-1970) foi o lingüista norte-americano que elaborou o projeto lingüístico conhecido como Gramática de Montague, que é um modelo para a descrição da linguagem natural, particularmente da semântica [48]. A seguir, resumem-se os principais aspectos da Gramática de Montague.

Base Teórica da Gramática de Montague

- Para Montague não existe uma diferença qualitativa entre a linguagem natural e as linguagens formais construídas pela Lógica e que, portanto, a linguagem natural pode ser estudada matematicamente. No modelo de Montague, uma Gramática Universal é uma estrutura formal abstrata capaz de descrever todas as linguagens. Obviamente esta idéia vai contra a maioria das teorias lingüísticas correntes.

- Montague pressupõe também que a semântica é o ponto de partida para a descrição da linguagem natural, ou seja, é a semântica que dá os limites de uma gramática. Na gramática de Montague, o significado é uma relação entre a linguagem e o mundo, ou seja, a linguagem é usada para falar sobretudo de coisas externas a ela mesma, sejam estas coisas do mundo real ou de mundos fictícios. Nota-se aqui um distanciamento entre a teoria de Montague e as teorias lingüísiticas contemporâneas, pois estas últimas vêem o significado principalmente como uma relação intralingüística ou, então, como uma relação entre a linguagem e a mente (por exemplo, a gramática gerativa).

- Montague considera que a ponte entre a linguagem e o mundo é feita através do conceito de verdade: o significado de uma sentença é idêntico às suas condições de verdade (concepções já defendidas por Wittgenstein [76] e o filósofo Davidson

[17]). Intuitivamente, compreender o significado de uma sentença é ser capaz de imaginar como teria de ser o mundo para que ela seja verdadeira.

Vantagens de Uma Semântica de Valor de Verdade

As vantagens de se associar significado a valor verdade, tal como adotado por Montague são:

- O rompimento da circularidade a que nos leva uma definição intra-lingüística do significado.
- A Possibilidade de definir não circularmente, a partir do conceito de verdade, os conceitos de sinonímia, ambigüidade, implicação e verdade lógica.
- A possibilidade de uma construção recursiva do significado.

Uma Semântica de Valor de Verdade Aplicada Às Línguas Naturais

Para que se associe significado a valor verdade é necessário que se tenha uma descrição do mundo, uma descrição da linguagem e, finalmente, uma descrição de como o mundo e a linguagem se relacionam. Montague [11, 24, 29, 30, 48] concretiza esta necessidade adotando três ferramentas:

a) *O Princípio da Composicionalidade*: este princípio é atribuído a Frege (1893). Para que se consiga construir uma semântica de valor verdade em qualquer linguagem, necessita-se de um instrumento capaz de construir composicionalmente o significado das sentenças a partir do significado de seus constituintes. Este instrumento é o Princípio da Composicionalidade, pois ele força a construção paralela da sintática e da semântica. Cada constituinte da sentença deve contribuir para a construção em paralelo da sintática e da semântica, tal como ocorre no exemplo de frase abaixo extraído de [48]:

Pedro corre

Nesta frase, o sintagma nominal *Pedro* compõe-se com o sintagma verbal *corre* para formar uma sentença. Semanticamente, teríamos que uma entidade do mundo, o indivíduo ao qual se refere o nome Pedro, e um conjunto de entidades, no caso, o conjunto de entidades que correm, compõem-se para afirmar um estado de coisas no qual o indivíduo Pedro pertence ao conjunto das entidades que correm; este estado de coisas é verdadeiro ou falso. Como se vê, o significado de uma sentença afirma composicionalmente suas condições de verdade.

b) *A Semântica do Modelo Teórico*: uma Semântica de Modelo Teórico é uma ferramenta para a descrição formal do mundo e de sua relação com a linguagem. Esta descrição é feita através da Teoria dos Conjuntos. Retornando ao exemplo anterior, em *Pedro corre*, o nome *Pedro* poderia significar diretamente o indivíduo que se chama *Pedro* e *correr* poderia significar o conjunto das entidades que correm. Neste caso, o significado da sentença *Pedro corre* seria a afirmação de que o indivíduo *Pedro* pertence ao conjunto das entidades que correm. Esta maneira de descrever significados, considerando as sentenças e seus constituintes como significando coisas do mundo, é chamada de semântica referencial ou extensional.

c) *A Semântica dos Mundos Possíveis*: o argumento mais comum contra a Semântica do Valor Verdade é o fato de não usarmos a linguagem apenas para nos referir ao mundo dito real. Através da linguagem, referimo-nos a mundos que poderiam ter sido, que poderão ser e a mundos que nunca serão. Esta questão é resolvida na Gramática de Montague pela adoção de uma Semântica de Mundos Possíveis - uma Semântica de Modelo Teórico que constrói modelos do mundo nos quais se interpreta uma linguagem. Desta forma, teríamos todos os mundos dos quais fala a linguagem, como o mundo descrito em Branca de Neve e os Sete Anões, o mundo tal como seria se não tivesse havido as grandes guerras mundiais, o mundo real tal como ele é etc.

d) *A Lógica Intensional*: obviamente, nem sempre a relação entre a linguagem e o(s) mundo(s) é tão bem comportada quanto na relação estabelecida, por exemplo, entre a sentença *Pedro corre* e o estado de coisas por ela descrito. Vimos uma amostra de mal comportamento na secção 3.1.5 onde se tenta analisar o significado da expressão

"o atual presidente do Brasil". Como tratar o significado desta expressão extensionalmente se, em cada momento e em cada mundo possível, ela se refere a um objeto diferente? Montague resolve isso com a seguinte técnica: o significado de uma palavra, de um sintagma ou de uma sentença passa a ser associado à intensão deste elemento. Intensão é definida como uma função (no sentido matemático do termo) de mundos possíveis na referência ou extensão de um item lingüístico. Desta forma, a intensão associa, em cada mundo possível, cada item lingüístico à sua respectiva extensão (referência) naquele mundo. Por exemplo, a intensão do item lingüístico *o atual presidente do Brasil* pode ser representada por uma função:

{(W1, Itamar Franco), (W2, Fernando Henrique Cardoso), (W3, Leonel Brizola), ...},

e a intensão do item lingüístico *o marido de Ruth Cardoso* pode ser representada por uma função:

{(W4, Joaquim Cardoso), (W2, Fernando Henrique Cardoso), ...}

onde W1 corresponde ao mundo real em 1994, W2 corresponde ao mundo real em 1995, W3 corresponde a um mundo fictício em que Leonel Brizola seja presidente do Brasil em 1995 e W4 correspode a um mundo fictício em que Ruth Cardoso seja casada com Joaquim Cardoso.

Para Montague duas sentenças são equivalentes quando têm as mesmas intensões (mesmas referências nos mesmos mundos e tempos possíveis). Segundo esta concepção de equivalência, as expressões *o atual presidente do Brasil* e *o marido de Ruth Cardoso* deixam de ser equivalentes, o que resolve o problema abordado na secção 3.1.5.

3.2. EVOLUÇÃO DO ANALISADOR SINTÁTICO SEMÂNTICO

Com a finalidade de simplificar a exposição do analisador implementado, começaremos por descrever resumidamente o resultado de análise de frases efetuada

pelo conhecido analisador da língua inglesa chamado ALVEY [33]. A partir daí descreveremos o que desejamos que seja feito pelo nosso analisador.

Nas frases da linguagem natural os quantificadores de Hilbert [1, 34, 44, 78] correspondem a referências de Hilbert da linguagem natural. Uma referência de Hilbert da linguagem natural é introduzida por um artigo definido [75] que se refere a um objeto conhecido no contexto analisado. A título de exemplo, considere-se a exigência E1 abaixo:

E1: ***The*** ioi-gs shall control ***the*** computers on-board ***the*** space-vehicle .

Os três artigos definidos "***the***" da exigência E1 se referem, respectivamente, aos objetos "***ioi-gs***", "***computers***"e "***space-vehicle***", que são conceitos primitivos no contexto das exigências. Consequentemente, este três artigos introduzem referências de Hilbert.

O analisador Alvey efetua a análise de uma frase qualquer traduzindo-a de modo a produzir predicados ternários que representem os quantificadores de Hilbert da frase analisada.

O primeiro argumento do predicado ternário no qual se traduz um quantificador de Hilbert corresponde ao nome da variável que representa o objeto ao qual o quantificador se refere; o segundo argumento corresponde a restrições a serem impostas a esta variável (que são as próprias características do objeto que ela representa) e, finalmente, o terceiro argumento corresponde ao evento no qual o objeto está envolvido.

O Alvey produz como resultado da análise da exigência E1, a fórmula F1 abaixo:

```
F1: τ(x1, ioi-gs(x1),
      τ(x2, τ(x3, space-vehicle(x3),
         computer(x2) & on-board(x2, x3))
      control(x1, x2) )
     ),
```

onde τ é o símbolo adotado para representar os predicados ternários correspondentes aos quantificadores de Hilbert.

No capítulo 5 será estudado que o sistema de recuperação de informação é implementado de tal modo que a resposta a uma pergunta posta ao sistema corresponda a uma avaliação parcial da pergunta com relação ao segundo argumento dos quantificadores (na introdução desta tese antecipou-se uma breve apresentação da noção de avaliação parcial a ser discutida com mais detalhes nos capítulos 4 e 5). Conseqüentemente, tanto a resposta quanto os segundos argumentos dos quantificadores produzidos pela análise devem pertencer a um subconjunto de {Dp - (Dp $\cap$ Pp)}. Este subconjunto é escolhido de maneira tal que ele possa ser facilmente expresso como fórmulas da Lógica Terminológica. Tanto o provador de teoremas que responderá perguntas quanto o usuário do sistema serão beneficiados com esta escolha, pois ambos poderão usar os recursos da subsunção e da classificação [49, 54, 59] da Lógica Terminológica para facilitar suas tarefas. Isso significa que o provador pode usar esses recursos no processo de prova e que o usuário do sistema pode usá-los para raciocinar com as respostas fornecidas pelo provador.

Tal processo de recuperação de informação requer que tanto as exigências quanto as perguntas sejam analisadas e postas numa forma cujos segundos argumentos dos quantificadores sejam expressões da Lógica Terminológica. Logo, estes segundos argumentos têm que ser expressões livres de quantificadores. Para isso, a análise é conduzida no sentido de produzir uma fórmula que seja próxima de uma "skolemização" de F1, pois em F1 encontram-se quantificadores aninhados nos segundos argumentos (tal como no segundo argumento do quantificador que introduz a variável *x2*). Poder-se-ia imaginar, por exemplo, um processo de skolemização executado em duas etapas:

- Na primeira etapa, as expressões correspondentes aos segundos argumentos dos quantificadores são transformadas em expressões da Lógica Terminológica (expressões livres de quantificadores compostas por conceitos e "roles"), tal como a expresão *F2* abaixo:

F2: τ(x1, ioi-gs(x1), τ(x2, computer(x2), τ(x3, space-vehicle(x3), on-board(x2,x3) & control(x1, x2))))

- Na segunda etapa da "skolemização", o segundo argumento dos quantificadores (expressões da Lógica Terminológica) são escritos como anotações que restringem as variáveis introduzidas nos primeiros argumentos dos quantificadores, conforme pode-se ver na seguinte expressão *F3*:

F3: control(x1/τ(ioi-gs), x3/τ(computer & part-of(space-vehicle))).

A semântica das expressões da Lógica Terminológica usadas como anotação é a mesma proposta por Vilain [21, artigo IEEE], ou seja:

Expression	***Semantics***
exp	$\lceil exp \rceil$
c1 & c2	$\lambda X(\lceil c1 \rceil X) \& (\lceil c2 \rceil X)$
r(c)	$\lambda X \forall Y(\lceil r \rceil X Y) \rightarrow (\lceil c \rceil Y))$
τ(c)	$\lambda X \lVert \{X \mid (\lceil c \rceil X))\} \rVert = 1$
Y/c	$\{Y \mid (\lambda X (\lceil c \rceil X) Y)\}$

Esta tabela complementa a tabela de valores semânticos apresentada no capítulo anterior.

Para simplificar as notações, a partir de agora não se introduzirá o símbolo τ nas expressões das exigências. Recorde-se que τ foi introduzido para permitir a "skolemização" dos quantificadores hilbertianos e dos demais quantificadores relacionados à cardinalidade dos conjuntos (tais como *at least*, *at most*, *the majority* etc). Nebel [51] mostra como tratar os quantificadores *at least* (*no mínimo*) e *at most* (*no máximo*).

Mesmo que os operadores *at least* e *at most* não estejam incluídos na presente implementação por não fazerem parte do corpo das exigências com as quais se lida aqui, não haveria maiores dificuldade em fazê-lo seguindo-se a proposta de Nebel.

3.3. O Analisador

Conforme se antecipou na introdução deste capítulo, as regras da gramática implementada neste sistema são baseadas nos casos do Latim. Mas não há razão especial alguma para esta escolha, em outras palavras, poder-se-ia ter usado qualquer outro sistema de casos padrão (por exemplo, de uma outra língua indo-européia [21]). O que se deseja mostrar de contribuição nesta parte do trabalho é o processo de análise propriamente dito, independentemente da natureza dos elementos (ou das categorias sintáticas, conforme será discutido logo a seguir) que compõem as regras da gramática. Por esta razão, será feito a seguir apenas um rápido comentário sobre os casos de Latim. No próximo ítem, onde se discutirão as abstrações lambda do analisador, será feita uma exposição das propriedades lingüísticas dos casos de Latim que nos permitirão produzir as abstrações lambda do sistema.

Sabe-se que a maior parte das línguas modernas (dentre elas o Inglês, no qual estão formuladas as exigências) praticamente perdeu os casos existentes em línguas clássicas tais como o Latim e o Grego. Sabe-se que, em geral, estes casos marcam a função sintática de cada componente lingüístico de uma frase, independentemente da posição que ele ocupe nela. Por exemplo, o caso nominativo, que geralmente indica o sujeito da ação, fica bem determinado em Latim em função da terminação do elemento (palavra) que o representa. Essa terminação particular permite que a função de sujeito desse elemento fique bem determinada independentemente da posição que ele ocupe na frase. Nas línguas modernas em geral, a eliminação quase total dos casos teve que ser compensada de alguma forma, a fim de que a função sintática de cada termo ficasse bem definida. Nelas, os casos ficam "indiretamente" indicados pelas posições (ordens) sobretudo rígidas que os elementos ocupam nas frases. Assim sendo, por exemplo, o sujeito (nominativo) normalmente ocupa a primeira posição da frase e o objeto direto (acusativo) normalmente vem após o verbo transitivo. Mesmo que essa ordem na linguagem natural não seja intransigentemente rígida, existe um grau de liberdade limitado no qual se podem fazer inversões sem se comprometer o sentido da frase.

A partir dessas observações, introduz-se a seguir uma descrição geral do analisador implementado.

A parte sintática de uma gramática gerativa é expressa através de regras, tais como as seguintes regras de gramática propostas por Chomsky e Panini:

ato → verbo transitivo, acusativo.

ato → verbo intransitivo.

sentença → nominativo, ato.

Montague [11] identifica os elementos das regras gramaticais (tais como *ato*, *verbo transitivo*, *acusativo*, *verbo intransitivo*, *sentença* e *nominativo* nas regras acima) como sendo *categorias sintáticas*. Em sua gramática ele propõe que se associe cada categoria sintática [68] a uma abstração lambda que lhe corresponda em significado. Dessa forma, as análises sintática e semântica podem ser executadas ao mesmo tempo, efetuando-se composições (combinações) entre uma categoria sintática e outra (análise sintática) em paralelo com as composições entre as abstrações lambda que lhes são associadas (análise semântica). Tais combinações devem prosseguir até que uma fbf cujos significado seja verdadeiro ou falso seja produzida. Conseqüentemente, na gramática de Montague deve-se definir tantas abstrações lambda quantas sejam as categorias sintáticas.

Implementa-se neste tese uma gramática aplicativa na qual o lingüista precisa definir apenas as abstrações lambda correspondentes às categorias sintáticas dos casos e dos verbos (*categorias fundamentais*). As abstrações lambda correspondentes às demais categorias, chamadas *abstrações não fundamentais*, serão automaticamente geradas pelo computador durante a análise. Por exemplo, para as regras definidas há pouco, o lingüista precisaria definir apenas as abstrações lambda correspondentes às categorias sintáticas do verbo transitivo, do verbo intransitivo, do acusativo e do nominativo (abstrações fundamentais). As abstrações correspondentes a *sentença* e a *ato* (não fundamentais) serão automaticamente geradas. O analisador corresponde a uma estrutura, tal como definem Bourbaki e Piaget (veja secção 3.3.1), representada por um sistema formal cujos axiomas são abstrações lambda tipadas de casos e cujas regras de inferência são regras β de redução (também representadas por abstrações lambda).

Conforme já se disse, se f_1, f_2, ..., f_d são fbfs do sistema, define-se dedução como uma seqüência (f_1, f_2, ..., f_d), onde f_1 é um axioma, onde f_d é o que se deseja demonstrar (no caso, f_d representa o resultado da análise, cujo significado pode ser verdadeiro ou falso) e f_i ($1 \leq i \leq d$) é um axioma ou então uma fbf obtida pela aplicação de uma regra de inferência (regra β) a fórmulas que antecedem f_i na seqüência. A ordem na qual as aplicações são executadas durante a análise é determinada pelo computador através de um método heurístico da Inteligência Artificial. Este método heurístico é guiado pelo tipo das abstrações lambda dos casos.

Durante a análise, o computador combina categorias sintáticas entre si (análise sintática) e também combina as respectivas abstrações lambda correspondentes a estas categorias sintáticas (análise semântica). Obviamente, tais combinações devem seguir os critérios de coesão e coerência que garantem a correção sintática e semântica da sentença analisada.

O analisador é baseado num algorítmo de busca no qual as categorias não fundamentais são gradualmente geradas pelo percurso de um caminho numa árvore de busca. A meta da análise é descobrir um caminho na árvore cuja folha seja uma fórmula bem formada cujo significado seja verdadeiro ou falso. Cada nó da árvore é estabelecido por um conjunto de abstrações lambda. Um novo nó aparece quando este conjunto é modificado pela aplicação de uma regra β a este conjunto.

Já foi dito que as estruturas são geradas por um conjunto de umas poucas estruturas-mãe, tais como estrutura-mãe da álgebra, estrutura-mãe da ordem e estrutura-mãe da topologia (veja secção 3.1.1). O analisador foi implementado baseado em duas estruturas-mãe: a estrutura-mãe da álgebra que indica como combinar as abstrações lambda e a estrutura-mãe da ordem que estabelece em que ordem elas devem ser combinadas. A estrutura-mãe da álgebra é introduzida pelas regras β de composição que combinam as abstrações lambda. A estrutura-mãe da ordem é introduzida pela heurística dos tipos das abstrações lambda e corresponde a caminhos na árvore de busca que são percorridos através de busca heurística. A estrutura-mãe da ordem produz uma seqüência dedutiva, em outras palavras, ela impõe uma relação de ordem que estabelece a seqüência de aplicações. Tal relação de ordem é estabelecida pelos tipos das abstrações lambda, de tal modo a gerar tipos de significados mais complexos a partir de tipos de significados menos complexos.

3.3.1. Definição das expressões do analisador

O conjunto de expressões é definido na seguinte notação de Backus-Naur:

<expressão> ::= <termo> | <abstração> | <aplicação> | <descrição>

<termo> ::= <constante> | <variável>

<abstração> ::= <lambda1> | <lambda2>

<lambda1> ::= l(<termo>, <expressão>) |
l(<termo>, <expressão>)
s(<termo>, <expressão>)

<lambda2> ::= l(<tipo-guia>, <abstração>, <expressão>)

<tipo-guia> ::= nom | acc | dat | gen | ablat | vi | vt % representando nominativo,
% acusativo, dativo, genitivo, ablativo, verbo intransitivo
% e verbo transitivo, respectivamente.

<descrição> ::= <evento> & < conjunção>

<evento> ::= [verbo <lista-de-termos>]

<lista-de-termos> ::= arg_1, arg_2, ..., arg_n,, onde arg_1, arg_2, ..., arg_n são termos

<terminologia> ::= [<termo><lista-de-termos>]
<terminologia> ::= [<caso-semântico> <lista-de-termos>]

<caso-semântico> ::= ag | pac | instrum % *ag*, *pac, instrum* representam *agente*,

paciente e *instrumental*, respectivamente

<conjunção> ::= <terminologia> |
[<terminologia>] & [<terminologia>]

<aplicação> ::= <terminologia> |
[lambda1, <lista-de-termos>] |
[lambda2, abstração]

As expressões *abstração* representam as categorias da gramática e as expressões *tipo-guia* representam os tipos a serem usados para guiar o processo de dedução.

No próximo item expõem-se com mais detalhes as características das expressões fundamentais da análise: as abstrações lambda introduzidas em *<abstração>*.

3.3.1.1. As Abstrações Lambda do Sistema

Já foi dito que os tipos são usados para guiar o processo dedutivo. Os tipos que somente podem ser aplicados a termos são chamados *tipos primários*. Se *t1* e *t2* são tipos, então *Ot1t2* é chamado *tipo composto*. Tal tipo representa uma expressão que deve ser aplicada a um argumento do tipo *t1*, produzindo como resultado uma expressão do tipo *t2*.

Se *t* é um termo, se *e* e *e1* são expressões, se *lamb* é uma abstração e se *tip* é um tipo-guia, as abstrações lambda do sistema têm as seguintes formas gerais:

a) l(t, e)

b) le(t, e)

c) s(t, e)

d) l(tip, lamb, e1)

Abstrações que apresentem uma das formas expressas nos itens *a*, *b* ou *c* são abstrações lambda do tipo primário. Nos três casos, *t* é chamado *cabeça* da abstração e reserva o local onde deve instalar-se o argumento ao qual ela será aplicada. Também nos três casos, *e* é chamada *corpo* da abstração e indica a forma da expressão que será retornada como resultado da aplicação.

Abstrações que apresentem a forma expressa no item *d* são abstrações lambda de tipo-guia *tip* que devem ser aplicadas ao argumento representado pela abstração *lamb* (cabeça). O resultado da aplicação deve ser da forma expressa em *e1* (corpo). Conseqüentemente, se *tip* representa o tipo composto *t1/t2*, então o tipo de *lamb* deve ser *t1* e o tipo de *e1* deve ser *t2*.

Piaget divide as estruturas que representam um fenômeno qualquer em dois grupos:

- **Grupo das abstrações empíricas:**

É constituído por um conjunto de abstrações fundamentais elaboradas a partir de um conjunto de propriedades que são inerentes ao fenômeno.

- **Grupo das abstrações reflexivas:**

É constituído de abstrações não fundamentais que são gerados pela aplicação das estruturas mães às abstrações empíricas, sem que para isso se observem as propriedades do fenômeno.

O fenômeno no qual se trabalha neste projeto é o fenômento lingüístico. Já se citou na secção 3.3 que as abstrações lambda fundamentais do sistema são as abstrações lambda dos casos e dos verbos. Conseqüentemente, tais abstrações representam as abstrações empíricas do sistema e devem ser elaboradas a partir de um conjunto de propriedades lingüísticas. As propriedades lingüísticas do sistema são

extraídas das propriedades dos verbos e dos casos de Latim, já que se escolheram essas categorias para comporem a gramática do sistema. Algumas dessas propriedades são:

Pa: Uma abstração lambda do verbo transitivo (tipo *vt*) é obtida pela composição de abstrações lambda de tipo primário com variáveis, de tal modo a se obter a representação do evento expresso no verbo. Tais variáveis representam, respectivamente, o evento, o agente do evento e o paciente do evento.

Pb: Uma abstração lambda do nominativo (tipo *nom*) é obtida pela composição de abstrações lambda de tipo primário com uma variável que represente um substantivo.

Pc: Uma abstração lambda do acusativo (tipo *acc*) deve ser aplicada a uma abstração lambda do verbo transitivo, sendo que o resultado da aplicação corresponde a uma abstração lambda do verbo intransitivo.

Pd: A abstração lambda do verbo intransitivo (tipo *vi*) deve ser aplicada a uma abstração lambda do nominativo, sendo que o resultado corresponde a uma abstração lambda da sentença (tipo *sent*), cujo significado é verdadeiro ou falso.

Pe: Uma abstração lambda do caso adjetivo no nominativo deve ser aplicada a uma abstração lambda do nominativo, sendo que o resultado corresponde a uma abstração lambda do nominativo. Isso significa que as abstrações lambda do caso adjetivo no nominativo conservam o caso do nominativo aos quais são aplicadas.

Pf: Uma abstração lambda do caso adjetivo no acusativo deve ser aplicada a uma abstração lambda do acusativo, sendo que o resultado corresponde a uma abstração lambda do acusativo. Isso significa que as abstrações lambda do caso adjetivo no acusativo conservam o caso do acusativo aos quais são aplicadas.

Pg: Sentenças relativas conservam o caso dos argumentos aos quais são aplicadas.

De acordo com essas propriedades lingüísticas, já se podem elaborar as formas gerais de algumas abstrações empíricas, tal como mostrado a seguir:

- A propriedade lingüística *Pa* permite que se escolha como representação do verbo transitivo a seguinte abstração lambda de tipo primário:

  le(t1, e),

  onde *t1* representa um termo e *e* representa uma expressão.

- A propriedade lingüística *Pb* permite que se escolha como representação do nominativo a seguinte abstração de tipo primário:

  l(t1, e),

onde *t1* é um termo e *e* é uma expressão.

- A propriedade lingüística *Pc* permite que se escolha como representação do acusativo a seguinte abstração lambda de tipo composto:

  l(acc, lamb1, lamb2),

onde *acc* corresponde ao tipo da abstração, *lamb1* corresponde a uma abstração do verbo transitivo e *lambd2* corresponde a uma abstração do verbo intransitivo.

- A propriedade lingüística *Pd* permite que se escolha como representação do verbo intransitivo a seguinte abstração lambda do tipo composto:

  l(vi, lamb3, exp)

onde *vi* corresponde ao seu tipo, *lamb3* corresponde a uma abstração de nominativo e *exp* é uma abstração de sentença.

A seguir serão introduzidos mais detalhes relativos às abstrações lambda do sistema.

Conforme já foi discutido, a cada categoria sintática deve-se associar uma abstração lambda que represente o seu significado. Isso implica que as abstrações lambda devem introduzir informações semânticas correspondentes às categorias as quais elas se associam.

A questão é: *como introduzir essas informações semânticas?*

No sistema implementado as informações semânticas são introduzidas através de casos semânticos presentes no corpo das abstrações lambda que representam as diversas categorias.

A seguir analisam-se os casos semânticos necessários ao verbo-transitivo, já que a maioria das exigências se estrutura em torno de verbos transitivos.

Um verbo transitivo precisa de dois casos semânticos: o agente e o paciente. Isso já assegura no mínimo dois argumentos para a abstração do verbo transitivo. Contudo, como diversas categorias sintáticas de uma sentença (exigência) devem fazer alusão ao evento expresso no verbo transitivo, foi acrescentado um terceiro elemento à lista de argumentos do verbo transitivo: um elemento que represente o evento a que o verbo se refere. Ora, como então representar estes três argumentos se, conforme já foi mostrado, as abstrações lambda do sistema estão preparadas para receber apenas um argumento?

A resposta a esta questão é: na verdade as abstrações lambda do sistema são operadores unários de Curry (veja secção 2.8.1). Desse modo, a abstração lambda do verbo transitivo, por exemplo, deve corresponder a uma abstração *e1* de tipo primário cuja cabeça seja um termo que represente o *evento* e cujo corpo seja uma abstração *e2* de tipo primário. A cabeça de *e2* deve ser um termo que represente o caso semântico *agente* e seu corpo deve ser uma abstração *e3*. Por sua vez, *e3* deve ter como cabeça um termo que represente o caso semântico *paciente* e um corpo que represente uma *descrição* (veja secção 3.3.1) que relacione eventos e casos semânticos. Logo, a forma geral final de uma abstração lambda de um verbo transitivo *Verbo* qualquer (tipo *vt*) deve ser:

```
le(Evento,
  l(Agente,
    l(Paciente,
      [Verbo, Evento, Agente, Paciente] & [ag, Evento, Agente] &
      [pac, Evento, Paciente])))
```

Como se falou há pouco, uma sentença que representa uma exigência é estruturada nos padrões do verbo transitivo. Logo, a forma geral final de uma abstração lambda que representa uma sentença (tipo *sent*) pode ser escolhida dentre as expressões do sistema como sendo da forma:

```
s(Evento1,
  l(Agente1,
    l(Paciente1,
      [Verbo, Evento1, Agente1, Paciente1] & [ag, Evento1, Agente1] &
      [pac, Evento1, Paciente1])))
```

Tentemos agora obter a forma final de uma abstração do nominativo. Ora, sabe-se através da propriedade lingüística *Pb* que uma abstração do nominativo deve associar variáveis a substantivos. Neste sistema, tal associação é representada através de uma *terminologia* (veja 3.3.1). Logo, a forma final de uma abstração de nominativo que represente um substantivo *Subs* qualquer deve ser:

```
l(N, [Subs, N])
```

Seguindo raciocínio análogo, *Pc* determina que a forma final de uma abstração que represente um acusativo *Acusativo* qualquer deve ser:

```
l(acc, Verbo,
 l(vi, Nominativo,
  s(Evento1,
   l(Agente1,
    l(Paciente1,
     [Verbo, Evento1, Agente1, Paciente1] & [Nominativo, Agente1] &
     [Acusativo, Paciente1] )))))
```

Por que é que no corpo dessa representação do acusativo são introduzidas aplicações representadas por terminologias tais como *[Verbo, Evento1, Agente1, Paciente1], [Nominativo, Agente1]* e *[Acusativo, Paciente1]* ? A resposta é simples e será mostrada em três etapas:

- É preciso que se faça uma associação entre as três variáveis do verbo transitivo (Evento, Agente e Paciente) e as três variáveis da sentença (Evento1, Agente1 e Paciente1). Tal associação fará com que as variáveis Evento, Agente e Paciente equivalham, respectivamente, às variáveis Evento1, Agente1 e Paciente1. Esta associação é efetuada durante a aplicação expressa na terminologia *[Verbo, Evento1, Agente1, Paciente1]*.

- Analogamente, a aplicação *[Nominativo, Agente1]* traça uma equivalência entre a variável N do nominativo e a variável Agente1 da sentença, que, por sua vez, após a aplicação *[Verbo, Evento1, Agente1, Paciente1]* equivale à variável Agente do verbo. Logo, após ambas as aplicações, as variáveis Agente, Agente1 e N são equivalentes.

- A aplicação *[Acusativo, Paciente1]* associa o grupo nominal expresso em Acusativo à variável Paciente1. Logo, após efetuadas as aplicações *[Verbo, Evento1, Agente1, Paciente1]* e *[Acusativo, Paciente1]*, as variáveis *Paciente* e *Paciente1* serão equivalentes entre si e ao mesmo tempo serão ambas associadas ao

grupo nominal expresso em *Acusativo*. Raciocínio análogo pode ser usado para se compreenderem as terminologias das demais abstrações lambda.

Detalhes e exemplos dessas aplicações são efetudas serão vistos na secção 3.3.3.2.

Continuando com as abstrações lambda, a propriedade lingüística *Pd* determina que a forma final geral que uma abstração lambda que represente o verbo intransitivo deve ser:

```
l(vi, Nominativo,
  s(Evento,
    l(Agente,
      l(Paciente,
        [Verbo, Evento, Agente, Paciente] & [Nominativo, Agente] &
        [Acusativo, Paciente] ))))
```

Cumpre observar que as abstrações lambda que representam as categorias têm tipos que realmente confirmam as expectativas das propriedades lingüísticas. Realmente, caso se assuma que os termos têm tipo *t*, as terminologias têm tipo *termin* e que as descrições têm tipo *descr*, então:

- *nom* (tipo do nominativo) corresponde ao tipo *Ot termin*, indicando que o nominativo aplicado a um termo produz uma terminologia.
- *vt* (tipo do verbo transitivo) corresponde ao tipo *OtOtOt descr*, indicando que um verbo transitivo aplicado a três termos (lembremos que tal aplicação é não simultânea, pois trabalhamos com operadores unários de Curry) produz uma descrição.
- *vi* (tipo do verbo intransitivo) corresponde ao tipo *O Ot termin sent*, indicando que um verbo intransitivo aplicado a um nominativo produz uma sentença.

- *acc* (tipo do acusativo) corresponde ao tipo *O Ot Ot Ot descr O Ot termin sent*, indicando que um acusativo aplicado a um verbo transitivo produz um verbo intransitivo.

Uma observação muito importante a ser feita é que as abstrações lambda que representam as categorias são supercombinadores (veja capítulo 2), o que vai fai simplificar muito o processo de "skolemização" da análise (veja secção 3.2). Tal facilidade é causada pela característica dos supercombinadores de já apresentarem todos seus quantificadores na forma prefixa.

3.3.2. Algoritmo de Análise

Já foi dito na secção 3.2 que o sistema de análise seria conduzido de modo a produzir uma fbf que corresponda aproximadamente a uma "skolemização" dos resultados das análises efetuadas pelo analisador ingês Alvey. Recorde-se que o resultado da análise do Alvey corresponde a uma fbf organizada em torno dos quantificadores introduzidos pelas referências hilbertianas (artigos definidos).

O algoritmo de análise foi concebido de modo a produzir o mais diretamente possível a meta desejada, ou seja, uma forma "skolemizada" tal qual a expressão F3 mostrada na secção 3.2 como resultado desejado para a análise da exigência *The ioi-gs shall control the computers on-board the space vehicle*, onde F3 pode ser vista a seguir:

F3: control(x1/τ(ioi-gs), x3/τ(computer & on-board(space-vehicle)))).

Basicamente a análise de uma exigência se processa em duas etapas: na primeira, gera-se uma pilha contendo as abstrações lambda correspondentes às categorias da exigência. Tal pilha armazena as *abstrações lambda empíricas* da análise (veja secção 3.3.3.1). Na segunda etapa, efetua-se a redução da pilha gerada, visando-se à formação de uma fbf cujo significado seja verdadeiro ou falso. Durante esta segunda etapa são geradas as *abstrações lambda reflexivas* da análise (ver ainda secção 3.3.3.1).

Mais especificamente, a análise é executada conforme os passos abaixo, onde o primeiro se refere à geração da pilha e os demais se referem à sua redução.

- A partir das regras da gramática, o algoritimo constrói uma pilha *L* que armazena as *abstrações lambda empíricas* correspondentes aos casos e verbos da exigência que deve ser analisada (predicado "build" mostrado abaixo).

- Ele escolhe (predicados "choose" e "ap" abaixo) em *L* uma abstração lambda de tipo composto *l(t1/t2, head, body)*, onde *t1/t2* indica o tipo composto, e retorna uma nova pilha *RL* correspondente à pilha *L* desfalcada da abstração escolhida *l(t1/t2, head, body)*.

- Ele escolhe em *RL* uma abstração lambda *A* de tipo *t1* a qual a abstração *l(t1/t2, head, body)* pode ser aplicada e retorna uma nova pilha *RL1* correspondente a *RL* desfalcada de *A*.

- Através dos predicados *beta* listados logo a seguir, ele efetua a aplicação (ou redução) expressa em *[l(t1/t2, head, body), A]*. Seja *S* o resultado dessa aplicação.

- O algoritmo é repetido recursivamente, a partir do segundo passo, para uma nova pilha *L* agora composta por *S* e *RL1*. O algorimo pára quando há apenas uma abstração na pilha. Tal abstração deve corresponder a uma abstração lambda de sentença, cujo significado é verdadeiro ou falso.

Notemos que os tipos atribuídos às abstrações lambda aceleram o processo de análise, pois funcionam como guia para o processo de combinação destas abstrações. Logo, as propriedade lingüísticas a partir das quais se constroem as abstrações lambda do sistema representam alguns dos critérios de coerência (veja secção 3.1.2). Os critérios de coesão são ditados pelas regras da gramática.

Abaixo mostram-se os predicados principais envolvidos no algoritmo de análise. Como a parte mais interessante da análise concentra-se na redução da pilha, a ênfase será dada aos predicados de redução. A parte do algoritmo que constrói a pilha

que armazena as abstrações lambda correspondentes aos casos e aos verbos da exigência a ser analisada será representada, resumidamente, pelo predicado "build". Na próxima seção serão feitos outros comentários referentes à construção da pilha.

```
analyse(Sentence, Res) :- build(Sentence, L), red(L, Res).

1. beta(X,X) :- var(X), !.
2. beta([X|R], [X|R])  :- var(X), !.
3. beta(X,X) :- atom(X), !.
4. beta([l(X,P), X], Q) :- !, beta(P,Q).
5. beta([l(_, X, P), X],  Q) :- !, beta(P,Q).
6. beta([l(X, P1), X | R], Q) :- !, beta(P1, P), beta([P | R], Q).
7. beta([le (X, P), X], Q) :- !, beta(P, Q).
8. beta([le(X,P1), X | R], Q) :- !, beta(P1,P), beta([P | R], Q).
9. beta([s (X, P), X], Q) :- !, beta(P, Q).
10. beta([s (X,P1), X | R], Q) :- !, beta(P1,P), beta([P | R], Q).
11. beta( [P | R], [P | R]) :- !.
12. beta( A & B, P & Q) :- !, beta(A, P), beta(B, Q).
13. beta(l(X, P), l(X, Q)) :- !, beta(P, Q).
14. beta(le(X, P), le(X, Q)) :- beta(P, Q).
15. beta(s(X,P), s(X, Q)) :- beta(P, Q).
16. beta(l(Tp, X, P), l(Tp, X, Q)) :- beta(P, Q).

choose([X|R], X, R).
choose([X|R], Y, [X|R1]) :- choose(R,Y,R1).

red([L], [L]) :- !.
red(L, Res) :- choose(L, l(Tp1, X1, P), RL),
               ap(Tp1, Q),
               choose(RL, Q, RL1),
               beta([l(Tp1, X1, P), Q], S),
               red([S | RL1], Res), !.
```

ap(acc, le(_, l(_, l(_, _)))) :- !.
ap(vi, l(_, _)) :- !.

Na próxima secção será visto em detalhes o funcionamento do nosso algoritmo de análise.

3.3.3. Exemplo de Análise

Em acordo com a descrição feita até aqui e já se conhecendo a forma das abstrações lambda que devem representar o significado das categorias dos casos e dos verbos, já se pode discutir em detalhes o processo de análise das exigências. Acreditamos que a melhor maneira de fazê-lo é através de um exemplo. Para tanto, considere-se a análise da exigência *E* abaixo:

E: The ioi-gs controls the space vehicle.

De acordo com o que se tem exposto, tal exigência pode ser analisada dentro dos moldes da seguinte gramática G1 resumida:

frase → snominativo, sverbal.

sverbal → verbo-transitivo, sacusativo.

snominativo → [the], nominativo.

sacusativo → [the], acusativo.

dic(substantivo, ioi-gs) → [ioi-gs].
dic(substantivo, space-vehihcle) → [space-vehicle].
dic(verbo-transitivo, control) → [control].

```
acusativo → dic(substantivo, Nm),
            push(l(acc, Verbo-transitivo,
                l(vi, Nominativo,
                  s(Evento,
                    l(Agente,
                      l(Paciente,
                        [Verbo-transitivo, Evento, Agente, Paciente] &
                          [Nominativo, Agente] & [Acusativo, Paciente] )))) ))

vt → dic(verbo-transitivo, Vb),
     push(l(E, l(X, l(Y, [Vb, E, X, Y] ))))

nominativo → dic(substantivo, Nm),
             push(l (N, [Nm, N])),
```

onde *snominativo*, *sverbal* e *sacusativo* representam um *sintagma nominal*, um *sintagma verbal* e um *sintagma acusativo*, respectivamente. No caso das exigências, normalmente os sintagmas que representam um grupo nominal (tais como *snominativo* e *sacusativo*) são introduzidos por artigos definidos que representam quantificadores hilbertianos. Para cada um desses grupos nominais será concebida uma abstração lambda que deve ter a forma correspondente à categoria sintática que ela representa. Tais abstrações introduzirão as variáveis que representarão, por exemplo, o *agente* (normalmente associado ao *nominativo*) e o *paciente* (nomalmente associado ao *acusativo*) das exigências. Analogamente, será concebida uma abstração lambda para o verbo de cada exigência. Tal abstração introduz um variável correspondente ao evento da exigência.

Observe-se que, conforme o que já se discutiu e conforme o que se constata na gramática acima, na maioria das línguas modernas, apesar de os casos terem praticamente desaparecido, eles ficam "indiretamente" marcados em função da posição dos diversos elementos na frase. No caso das exigências do sistema, que correspondem a sentenças da linguagem natural que devem ter uma forma bem definida (veja capítulo 1), pode-se, por exemplo, definir o *nominativo* como sendo o grupo nominal que

ocupa a primeira posição da frase e o *acusativo* como sendo o grupo nominal que se coloca após o verbo transitivo.

Observe-se também que o empilhamento das abstrações lambda dos casos e verbos é efetuado durante a execução da gramática, através do predicado "push" (obviamente, isso é feito na primeira etapa da análise, durante a execução do predicado "build").

A seguir mostram-se as duas etapas a serem efetuadas neste exemplo de análise, ou seja, a geração da pilha de abstrações e sua redução.

3.3.3.1. A Geração da Pilha de Abstrações Empíricas

De acordo com a gramática G1 descrita há pouco, nessa primeira etapa efetuam-se, para cada grupo de palavras da exigência analisada, os seguintes passos (suponha-se que se inicie tal etapa com uma pilha L vazia):

- Verifica-se se o grupo de palavras pertence à categoria gramatical estabelecida pela gramática em função da posição do grupo na exigência analisada. Caso a verificação seja bem sucedida, gera-se a abstração correspondente a este grupo em função da categoria que ele representa. Por exemplo, verifica-se se o grupo *the ioi-gs* que ocupa a primeira posição da exigência *E* é um substantivo; caso afirmativo, gera-se a abstração correspondente ao nominativo. Verifica-se se o grupo *controls* que ocupa a segunda posição da exigência *E* é um verbo transitivo; caso afirmativo, gera-se a abstração correspondente ao verbo transitivo. Analogamente, verifica-se se o grupo *the space vehicle* que ocupa a terceira posição da exigência *E* é um substantivo; caso afirmativo, gera-se a abstração correspondente ao acusativo.

  Obviamente, neste caso as abstrações lambda geradas para o nominativo, verbo transitivo e acusativo, de acordo com as formas gerais estabelecidas para tais categorias, são, respectivamente:

  l(Z1, [ioi-gs, Z1]), para o nominativo;

le(Y1,

l(Y2,

l(Y3,

[controls, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3]))), para o verbo transitivo;

l(acc, X1,

l(vi, X2,

s(X3,

l(X4,

l(X5,

[X1, X3, X4, X5] & [X2, X4] & [space-vehicle, X5]))))), para o acusativo.

- Caso o passo anterior seja bem sucedido, empilha-se na pilha L a abstração lambda nele gerada. Conseqüentemente, no exemplo analisado a pilha L gerada é:

[l(acc, X1,

l(vi, X2,

s(X3,

l(X4,

l(X5,

[X1, X3, X4, X5] & [X2, X4] & [space-vehicle, X5]))))),

le(Y1,

l(Y2,

l(Y3,

[controls, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3])))

l(Z1, [ioi-gs, Z1])]

Obviamente, as variáveis X1, X2, X3, X4 e X5, em função das características de um acusativo, representam, respectivamente, um verbo transitivo, um nominativo, um evento, um agente e um paciente. Da mesma forma, as variáveis Y1, Y2 e Y3, em função das características de um verbo transitivo, representam, respectivamente, um evento, um agente e um paciente. Por sua vez, a variável Z1, em função das características de um nominativo, representa um agente.

Como se pode observar, as abstrações-lambda que representam as categorias sintáticas introduzem os predicados binários portadores de informações sobre os casos semânticos do evento, bem como os predicados unários que caracterizam as variáveis. Logo, a própria análise feita pelo sistema se processa de forma a produzir diretamente uma representação para a frase analisada em que predominem os predicados binários e unários (vantagem da modularidade, discutida na secção 2.3.2).

É importante repetir que as abstrações armazenadas em L correspondem às *abstrações lambda empíricas* da estrutura de análise.

3.3.3.2. Redução da Pilha L

Para efeito de simplicação de notação, chamar-se-ão as abstrações do *acusativo*, do *verbo transitivo* e do *nominativo* da pilha L obtida em 3.3.3.1, respectivamente, de *eac*, *evt* e *enom*.

O mecanismo de redução da pilha corresponde a uma dedução (veja secção 3.3) a partir dos axiomas contidos na pilha (abstrações lambda), usando-se as regras *beta* de inferência. Durante essa dedução tenta-se compor as abstrações lambda da pilha no sentido de se produzir uma fbf cujo significado seja verdadeiro ou falso. O processo de dedução corresponde à escolha de um caminho numa árvore de busca em que os nós correspondem aos axiomas do sistema (abstrações lambda). Tal caminho é indicado pelos tipos das abstrações lambda. A partir dessas observações, chega-se a duas conclusões fundamentais:

- A estrutura mãe da álgebra do sistema implementado corresponde ao predicado beta.

- A estrutura mãe da ordem do sistema implementado é fornecida pelos tipos das abstrações lambda.

Durante esse processo de redução são geradas as *abstrações lambda reflexivas* da estrutura de análise.

De acordo com o algoritmo de redução mostrado em 3.3.2, a seqüência de redução da pilha L é da forma:

The ioi-gs	*controls*	*the space-vehicle*
nom	vt	((sent\ nom) \ vt)
nom	(sent \ nom)	
sent		

onde *sent*, *nom*, e *vt* representam os tipos *sentença*, *nominativo* e *verbo transitivo*, respectivamente. Observa-se que durante essa redução geram-se duas abstrações lambda reflexivas: uma do verbo intransitivo (tipo *sent* \ *nom*) e outra da sentença (tipo *sent*).

A seguir são recordados os tipos sintáticos de algumas expressões a serem usadas na redução. Recorde-se também que o símbolo O representa o operador da Gramática Aplicativa (veja secção 2.7) que gera recursivamente o sistema de tipos:

expressão	***tipo sintático***
termo	t
Lógica Terminológica	termin
sentença	sent
descrição	descr
nominativo	Ot termin (ou *nom*)
verbo transitivo	Ot Ot Ot descr (ou *vt*)
verbo intransitivo	O Ot termin sent (ou *vi*)
acusativo	O Ot Ot Ot descr O Ot termin sent (ou *acc*)

De acordo com tais definições, a redução da pilha anterior consistirá da seguinte seqüência de aplicações:

[eac evt] ⇒ expressão E1, ou seja,

```
[l(acc, X1,
 l(vi, X2,
  s(X3,
   l(X4,
    l(X5,
     [X1, X3, X4, X5] & [X2, X4] & [space-vehicle, X5] ))))),

 le(Y1,
  l(Y2,
   l(Y3,
    [control, Y1, Y2, Y3] & [ag, Y1, Y2] &
        [pac, Y1, Y3] ))) ] ⇒ expressão E1,
```

de onde se tira que E1 é a expressão:

```
l(vi, X2,
 s(X3,
  l(X4,
   l(X5,
    [le(Y1,
     l(Y2,
      l(Y3,
       [control, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3]))), X3, X4, X5] &
       [X2, X4] & [space-vehicle, X5] ))))
```

Ainda resta uma redução a ser efetuada em E1, que é a redução referente à aplicação da abstração do verbo transitivo às variáveis X3,X4 e X5, isso é:

[le(Y1,
l(Y2,
l(Y3,
[control, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3])))) X3 X4 X5] ⇒ expressão E2

A seqüência de aplicações que produzirá E2 é:

- A abstração do verbo transitivo, do tipo *Ot Ot Ot descr*, é aplicada ao termo X3 (tipo *t*), produzindo a expressão:

[l(Y2,
l(Y3,
[control, Y1, Y2, Y3] & [ag, X3, Y2] & [pac, Y1, Y3])) X4 X5].

cujo tipo *Ot Ot descr* é realmente o esperado pela primeira aplicação de *vt* a um termo de tipo *t*.

- Essa última abstração é aplicada a X4, produzindo expressão:

l(Y3,
[control, X3, X4, Y3] & [ag, X3, X4] & [pac, X3, Y3]) X5],

cujo tipo, confirmando a expectativa, é *Ot descr*;

- Essa última abstração é aplicada a X5 e produz o resultado final esperado para E2, isso é:

E2: [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3 X5].

O tipo *descr* de *E2* é realmente o previsto para o caso em que o verbo transitvo é aplicado a três termos.

Logo,o valor final da expressão *E1* (correspondente a um verbo intransitivo de tipo *O O t termin sent*) é:

```
l(vi, X2,
  s(X3,
    l(X4,
      l(X5,
        [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] & [X2, X4] &
        [space-vehicle, X5] ))))
```

Prosseguindo com a redução da pilha, o verbo intransitivo (*E1*) é aplicado ao nominativo, isso é, *[E1 enom]*, produzindo uma expressão Res que deve ser do tipo *sent*. Realmente:

[E1, enom] $\Rightarrow$ expressão Res,

isto é,

```
[l(vi, X2,
  s(X3,
    l(X4,
      l(X5,
        [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] & [X2, X4] &
        [space-vehicle, X5] )))),

  l(Z1,
  [ioi-gs, Z1] ) ] ⇒ expressão Res,
```

de onde se tira que a expressão *Res* é:

Res =

```
s(X3,
 l(X4,
  l(X5,
   [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] &
   [l(Z1, [ioi-gs, Z1], X4] &
   [vs, X5] )))
```

Res ainda pode ser reduzida pela aplicação de *enom* ao termo X4, o que produz a expressão definitiva de *Res*:

Res =

```
s(X3,
 l(X4,
  l(X5,
   [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] &
   [ioi-gs, X4] &
   [vs, X5] )))
```

Res corresponde ao resultado final da pilha (note-se que não há mais β-reduções pendentes). Conforme esperado, realmente *Res* apresenta a forma correspondente a uma frase (tipo *sent*).

O predicado beta descrito em 3.3.2 efetua a redução de seu primeiro argumento guardando o resultado em seu segundo argumento.

A seguir exemplifica-se o processo de aplicação dos predicados beta para a exigência do exemplo, isso é, *The ioi-gs shall control the space-vehicle*. Conforme já

se viu, a pilha gerada para esse exemplo contém eac, evt e enom, conforme se repete abaixo:

```
[l(acc, X1,
 l(vi, X2,
  s(X3,
   l(X4,
    l(X5,
     [X1, X3, X4, X5] & [X2, X4] & [space-vehicle, X5] ))))),

 le(Y1,
  l(Y2,
   l(Y3,
    [controle, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3])))

 l(Z1, [ioi-gs, Z1]) ]
```

Lembremos que *evi* corresponde ao corpo de *eac*. Abaixo faremos um rastreamento simplificado da redução dessa pilha. Deve-se entender uma notação tal como "1° arg do 2° arg do 3° arg evi" como: *o primeiro argumento do segundo argumento do terceiro argumento da expressão lambda evi*, o que obviamente corresponde a *X4*. O símbolo = representa a unificação e ⇔ representa equivalência.

```
?- red([eac, evt, enom, Res).

?- choose([eac, evt, enom], l(Tp1' , X1', P'), RL).

% "choose" procura na pilha uma abstração da forma l(Tp1' , X1', P')  e encontra eac.

% Logo,
% l(Tp1', X1', P') = eac e RL = [evt, enom], que é o que sobrou na pilha;
```

?- ap(acc, Q).

% Q = le(E, l(X, l(Y, P''')))

?- choose([evt, enom], le(E, l(X, l(Y, P''')')), RL1).

% le(E, l(X, l(Y, P''))) = evt
% RL1 = [enom]

?- beta([eac, evt], S). $\Rightarrow$ regra beta nro. 5
% 2° arg eac $\Leftrightarrow$ X1 = evt

?- beta(3° arg eac, S) $\Leftrightarrow$ beta(evi, S) $\Rightarrow$ regra beta nro. 16

% S = l(vi, 2° arg evi, Q1) $\Leftrightarrow$ l(vi, X2, Q1)

?- beta(3° arg evi, Q1) $\Rightarrow$ regra beta nro. 15

% Q1 = s(1° arg do 3° arg evi, Q2) $\Leftrightarrow$ le(X3, Q2)

?- beta(2° arg do 3° arg evi, Q2) $\Leftrightarrow$
?- beta(l(X4, l(X5, [evt, X3, X4, X5] & [X2, X4] &
[space-vehicle, X5])), Q2) $\Rightarrow$ regra beta nro. 13

% Q2 = l(1° arg do 2° arg do 3° arg evi, Q3) = l(X4, Q3)

?- beta(2° arg do 2° arg do3° arg evi, Q3). $\Leftrightarrow$

?- beta(l(X5, [evt, X3, X4] & [X2, X4] & [space-vehicle, X5]), Q3). $\Rightarrow$ regra beta nro. 13

% Q3 = l(1° arg do2° arg do 2° arg do 3° arg evi, Q4) = l(X5, Q4)

?- beta(2° arg do 2° arg do 2° arg do 3° arg evi, Q4) ⇔

?- beta([evt, X3, X4, X5] & [X2, X4] & [space-vehicle, X5], Q4). ⇒ regra beta nro. 12

% Q4 = P5 & Q5

?- beta([evt, X3, X4, X5], P5), beta([X2, X4] & [space-vehicle, X5], Q5).

Para facilitar futuras referências, a partir daqui numerar-se-ão algumas metas *beta* a se começar pelas duas últimas metas *beta* , a que se associarão, respectivamente, os números *1* (correspondente a *beta([evt, X3, X4, X5], P5)*) e *2* (correspondente a *beta([X2, X4] & [space-vehicle, X5], Q5])*) . Resolvendo a meta *1*:

?- beta([evt, X3, X4, X5], P5) ⇔

?- beta([le(Y1, l(Y2, l(Y3, [control, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3]))),
X3, X4, X5], PEÇA5). ⇒ regra beta nro. 8

% Y1 = X3

?- beta(l(Y2, l(Y3, [control, X3, Y2, Y3] & [ag, X3, Y2] & [pac, X3, Y3])), P6),
beta([P6, X4, X5], P5)

As duas últimas metas beta serão numeradas, respectivamente, como *1.1* e *1.2*. Resolvendo *1.1*:

?- beta(l(Y2, l(Y3, [control, X3, Y2, Y3] & [ag, X3, Y2] &
[pac, X3, Y3])), P6). ⇒ regra beta nro. 13

% P6 = l(Y2, Q6)

?- beta(l(Y3, [control, X3, Y2, Y3] & [ag, X3, Y2] &
[pac, X3, Y3]), Q6). ⇒ regra beta nro. 13

% Q6 = l(Y3, Q7)

?- beta([control, X3, Y2, Y3] & [ag, X3, Y2] & [pac, X3, Y3] , Q7). ⇒ regra beta nro. 12

% Q7 = P8 & Q9

?- beta([control, X3, Y2, Y3], P8), beta([ag, X3, Y2] & [pac, X3, Y3], Q9).

Numerem-se as duas últimas metas beta como *1.1.1* e *1.1.2*, respectivamente. Resolvendo-se *1.1.1*:

?- beta([control, X3, Y2, Y3], PEÇA8). ⇒ regra beta nro. 11

% P8 = [controle, X3, Y2, Y3].

% O cálculo de *Q9* em *1.1.2* é obtido através da regra beta nro. 12 que, ao
% ser aplicada, produz as metas *beta([ag, X3, Y2], P9)* e *beta([pac, X3, Y3], P10)*.
% A prova destas metas, por sua vez, produz:

% P9 = [ag, X3, Y2] e
% P10 = [pac, X3, Y2]

% Logo, Q9 = P9 & P10 (através da cláusula beta nro. 11). Como Q7 = P8 & Q9, então:
% Q7 = [control, X3, Y2, Y3] & [ag, X3, Y2] & [pac, X3, Y3]

Voltando-se na árvore recursiva, tem-se:

% Q6 = l(Y3, [control, X3, Y2, Y3] & [ag, X3, Y2] & [pac, X3, Y3]

% P6 = l(Y2, Q6) ⇔

% l(Y2, l(Y3, [control, X3, Y2, Y3] & [ag, X3, Y2] & [pac, X3, Y3])),

% que é a solução de *1.1*.

Seguindo-se raciocínio análogo, constata-se que a redução de *1.2* através das regras beta nro. 6, 12 e 13, provoca como unificações:

% Y2 = X4

% Y3 = X5

Logo, a meta *1* é resolvida com:

% P5 = [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5]

Analogamente, a meta *2* produz:

% Q5 = [X2, X4] & [space-vehicle, X5].

Como Q4 = P5 & Q5, então:

% Q4 = [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] & [X2, X4] &
[space-vehicle, X5]

Continuando regressivamente na recursividade, obtém-se:

% Q3 = l(X5, [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] &
[X2, X4] & [space-vehicle, X5])

% Q2 = l(X4, l(X5, [control, X3, X4, X5] & [ag, X3, X4] & [pac, X3, X5] &
[X2, X4] & [space-vehicle, X5]))

% Q1 = s(X3, l(X4, l(X5, [control, X3, X4, X5] & [ag, X3, X4] &
[pac, X3, X5] & [X2, X4] & [space-vehicle, X5])))

% S = l(vi, X2, s(X3, l(X4, l(X5, [control, X3, X4, X5] & [ag, X3, X4] &
[pac, X3, X5] & [X2, X4] & [space-vehicle, X5]

Finalmente, após o cálculo do resultado obtido através da aplicação do acusativo ao verbo transitivo, o resultado *Res* da redução da pilha *[eac, evt, enom]*, isso é, a meta *red([eac, evt, enom], Res)*, consiste na redução de uma nova pilha *NP* formada por *S* e pelas expressões que ainda não foram selecionadas na pilha original (no nosso caso, apenas *enom*). Analogamente ao que se viu até agora, tal redução vai selecionar em NP uma expressão da forma *l(Tp1, X1, P)* . A expressão selecionada é S, cujo tipo é *vi*, que por sua vez deve ser aplicada a expressões da forma *l(X, P)*, conforme indica o predicado *ap*. Logo, "choose" escolhe em NP a expressão *enom* a qual S deverá ser aplicada. Observando-se o predicado beta, constata-se que tal aplicação efetua a unificação da variável X2 de S com a expressão *enom* (ou seja, *l(Z1, [ioi-gs, Z1])*) e uma posterior aplicação de *enom* à variável X4 (isso é, *[X2, X4]* ⇔ *[l(Z1, [ioi-gs, Z1), X4]*). Esta última aplicação unifica a variável Z1 de *enom* com a variável X4 de S, produzindo *[ioi-gs, X4]*. Finalmente obtém-se o resultado *Res* esperado, ou seja:

Res ⇔ s(X3, l(X4, l(X5, [controls, X3, X4, X5] & [ag, X3, X4] &
[pac, X3, X5] & [ioi-gs, X4] & [space-vehicle, X5])))

Recorde-se que *Res* corresponde ao resultado da análise da exigência *The ioi-gs controls the space vehicle*.

Analisando-se os predicados de β-redução implementados, constata-se que eles atuam como combinadores. No caso da análise que se acaba de discutir, por exemplo, constata-se que o processo de redução através das regras beta pode ser descrito pela expressão:

B B acc vt nom,

onde B é o combinador de composição já estudado na secção 2.2. O desenvolvimento desta expressão é descrito a seguir:

B B acc vt nom ≥ B (acc vt) nom ≥ (acc vt) nom.

Logo, o resultado da avaliação da expressão *B B acc vt nom* corresponde à expressão *(acc vt) nom*, que indica que o resultado da aplicação do acusativo ao verbo transitivo deve ser aplicado ao nominativo. Como se vê, tal avaliação descreve exatamente os passos do exemplo de análise mostrado há pouco.

Note-se também que os tipos atribuído às abstrações lambda aceleram o processo de análise, pois funcionam como heurística que aponta para a a abstração que deve ser selecionada na pilha no momento da aplicação.

Uma outra observação muito importante a se fazer é que a representação das exigências através de fórmulas do Cálculo de Predicados auxilia muito na satisfação de um importante critério de qualidade de especificações: a rastreabilidade. Realmente, as variáveis introduzidas pelas referências hilbertianas permitem o reconhecimento de elementos idênticos dentro de uma mesma exigência (item *a* do tópico *rastreabilidade* do capítulo 1). Por exemplo, a variável X3 de *Res* representa sempre o evento *controls* da exigência *The ioi-gs controls the space vehicle*. Conseqüentemente, as terminologias *[ag, X3, X4]* e *[pac, X3, X5]* referem-se ambas ao evento controls expresso em X3, representando, respectivamente, o agente e o paciente de tal evento.

3.4. Organização da Base de Conhecimentos

As exigências ainda não são introduzidas na base de conhecimentos na forma em que são apresentadas após a segunda etapa da análise a que se submetem (tal como *Res* na secção anterior). Terminada a segunda etapa da análise, o resultado aí obtido se submete a uma "skolemização" adicional de modo a se transformar em uma conjunção de expressões que indicam o evento expresso no verbo da exigência e seus respectivos casos semânticos. A expressão do evento é do tipo:

pred(X1/exp1, X2/exp2, ..., X3/expn),

onde:

- *pred* representa um predicado n-ário (representando o verbo do evento);
- *X / exp* constitui-se de um átomo (variável ou constante) *X* anotado pela expressão *exp* da Lógica Terminólogica. Nesse caso, *exp* contém informações que particularizam o átomo *X* anotado por ela.

As demais expressões da seqüência de conjunções correspondem a expressões que representam os casos semânticos associados ao evento. Tais predicados são da forma:

caso(X1/exp1, X2/exp2),

onde:

- *caso* representa o caso semântico;
- *X/exp* tem o mesmo significado descrito acima.

Essa nova e definitiva representação da exigência (frase) é extremamente adequada ao sistema de prova (avaliação parcial) que implementamos e que será discutido nos próximos capítulos.

O algoritmo de "skolemização" utilizado assemelha-se ao descrito na secção 2.9. Obviamente, o fato de se terem concebido as abstrações lambda do sistema como combinadores onde os quantificadores já são expressos na forma prefixa simplifica bastante o processo de "skolemização". Assim sendo, para se produzir a conjunção final que representará a exigência, fica faltando apenas eliminar os quantificadores que foram introduzidos pelas referências hilbertianos da expressão final obtida na segunda etapa da análise. Isso é feito por funções análogas às funções "skolem" descritas na secção 2.9.

Finda a "skolemização", o resultado *Res* obtido na segunda etapa da análise da exigência E2 do nosso exemplo anterior, isso é:

Res: s(X3, l(X4, l(X5, [control, X3, X4, X5] & [ag, X3, X4] &
[pac, X3, X5] & [ioi-gs, X4] & [space-vehicle, X5]))),

se converterá na conjunção de expressões abaixo:

controls(e0/ev, X/ioi-gs, Y/space-vehicle) &
ag(e0/ev, X/ioi-gs) &
pac(e0/ev, Y/space-vehicle),

que é a forma normal conjuntiva final que representa a exigência.

Conforme esperado, nessa representação final, *controls(e0/ev, X/ioi-gs, Y/space-vehicle)* representa o predicado do evento cujos casos semânticos se encontram descritos nas expressões *ag(e0/ev, X/ioi-gs)* e *pac(e0/ev, Y/space-vehicle)*.

Observe-se que a representação final de uma exigência produzida pela análise corresponde a uma conjunção de proposições que retornariam ao estado de funções proposicionais caso se eliminassem as anotações de suas variáveis. Logo, a representação final de uma exigência é passível de ser interpretada como verdadeira ou falsa segundo a definição de verdade de Tarki (veja secção 3.1.4).

Neste ponto, o processo de skolemização está concluído, uma vez que as variáveis genéricas X3 (introduzida pelo evento), X4 (introduzida por referência hilbertiana) e X5 (também introduzida por referência hilbertiana) passam a ser caracterizadas (anotadas) por constantes que as restringem. Realmente, constata-se que a variável X3 introduzida pelo evento de *s* é substituída pela constante *e0/ev*, onde *e0* é anotada por *ev*. A anotação *ev* indica que o elemento por ela anotado é um evento. Assim sendo, no exemplo analisado, *e0* representa o evento *controlar* expresso em *controls*. Da mesma forma, as variáveis X4 e X5 introduzidas pelos dois outros quantificadores hilbertianos *l* são substituídas, respectivamente, pelas constantes *X/ioi-gs* e *Y/space-vehicle*.

Antes de se tentar introduzir na base de conhecimentos a forma normal conjuntiva (descrição) obtida pela análise de uma exigência, ela deve submeter-se a dois pequenos ajustes de notação que facilitarão a tarefa do provador de localizá-las na referida base: o primeiro ajuste consiste em omitir da notação o símbolo & das conjunções. O segundo ajuste consite em introduzir a palavra "any" no início da notação de cada disjunção que compõe a forma normal conjuntiva a ser introduzida. Deste modo, a descrição mostrada há pouco, por exemplo, seria introduzida na base com a seguinte notação:

any(controls(e0/ev, X/ioi-gs, Y/space-vehicle)).
any(ag(e0/ev, X/ioi-gs)).
any(pac(e0/ev, Y/space-vehicle)).

Característica híbrida da base de conhecimentos:

Observando-se a representação final de E2 (*descrição*) mostrada no exemplo anterior, pode-se observar o caráter híbrido da base de conhecimentos. Realmente, o evento e os casos semânticos da *descrição* representam o conhecimento *assercional* da base de conhecimentos e as anotações das variáveis representam o seu conhecimento *terminológico* (veja capítulo 2). Cada conceito atômico e cada "role" atômico da anotação correspondem, respectivamente, a um conceito primitivo e a um "role" primitivo. As conjunções entre conceitos e "roles" primitivos presentes nas anotações definem novos conceitos na terminologia (*conceitos definidos*). Obviamente, esses novos conceitos seguem a semântica descrita na secção 2.3. As variáveis da representação final correspondem a indivíduos do domínio que são instâncias dos conceitos expressos em suas respectivas anotações.

Já se discutiu no final da secção 3.3.3.2 a importância das variáveis usadas na representação das exigências no que se refere ao item *a* do tópico *rastreabilidade* (capítulo 1). Tal importância já pode ser estendida também ao item *b* do tópico rastreabilidade, isso é, o reconhecimento dos elementos de uma exigência nas diversas formas nas quais ela se converte até chegar a sua representação final. Neste caso, a rastreabilidade é garantida através do processo de "skolemização", onde as variáveis

são substituídas por constantes apropriadas. Por exemplo, a variável X3 de Res introduzida pelo evento *controls* da exigência *The ioi-gs controls the space vehicle*, após a "skolemização" é substituída pela constante e0/ev. Analogamente, a variável X4 é substituída pela expressão *constante X/ioi-gs*.

Subsunção: Um Critério de Simplificação na Construção Incremental da Base de Conhecimentos

Conforme já foi explicado, a base de conhecimentos é constituída por descrições que representam as exigências analisadas sintática e semanticamente. A construção da base de conhecimentos é feita de modo incremental [9, 55], ou seja, ela é construída por um processo repetitivo e seqüencial em que se analisa cada exigência e se introduz o resultado da análise na base de conhecimentos. Contudo, antes de se inserir uma exigência na base, deve-se analisar o que já se tem armazenado nela, para que se evite a inserção de coisas redundantes ou então contraditórias na base de conhecimentos. Com isso, evita-se a introdução de *barulho* na base ao mesmo tempo em que se assegura sua *coerência* (veja capítulo 1). Por exemplo, suponha-se que se queira inserir na base de conhecimentos a descrição:

controls(e1/ev, X/ioi-gs, Y/computers & on-board(space-vehicle)) &
ag(e1/ev, X/ioi-gs) &
pac(e1/ev, Y/computer & on-board(space-vehicle))

correspondente à exigência:

The ioi-gs controls the computers on-board the space vehicle,

Suponha-se também qu já se tenha armazenada na mesma base a descrição:

controls(e0/ev, X/ioi-gs, Y/equipments & on-board(space-vehicle)) &
ag(e0/ev, X/ioi-gs) &
pac(e0/ev, Y/equipments & on-board(space-vehicle))

correspondente à exigência:

The ioi-gs controls the equipments on-board the space-vehicle,

Obviamente, a inserção da primeira descrição introduzirá *barulho* na base de conhecimentos, uma vez que ela contém a segunda descrição. Isso é evitado porque, antes de se inserir uma nova exigência na base de conhecimentos, utilizam-se os critérios de subsunção para se verificar se a informação a ser inserida não é subsumida por alguma outra informação já presente na base. No caso do exemplo, a primeira descrição é subsumida pela segunda, pois todo elemento que é uma instância da primeira descrição o é também da segunda. De fato, o conceito definido *computers & on-board(space-vehicle)* é subsumido pelo conceito definido *equipments & on-board(space-vehicle)*, uma vez que o conceito primitivo *computers* é subsumido pelo conceito primitivo *equipments*. Logo, o sistema não introduzirá a primeira descrição na base de conhecimentos, pois ele detecta que ela é subsumida por uma outra descrição já inserida anteriormente. Essa comparação entre os elementos das exigências é efetuada por um algoritmo de unificação tradicional acrescido da semântica da subsunção. Os algoritmos de subsunção são extremamente complexos [50, 51], tornando-se impraticável o seu uso num contexto geral de fbfs. Por esta razão, nós restringimos seu uso às partes das fbfs que pertencem a um conjunto bem restrito da linguagem terminológica.

A unificação semântica baseada na subsunção é utilizada também na parte de recuperação de informação do sistema, conforme descrito em detalhes no capítulo 5.

Como se vê, na etapa de inserção de exigências na base de conhecimentos, a unificação efetua comparações entre elementos de uma exigência a ser inserida na base e as exigências já inseridas, detectando elementos que elas eventualmente mantenham em comum entre si. Daí se conclui que a unificação garante a rastreabilidade tal como prevista no item *c* do tópico de *rastreabilidade* (capítulo 1).

No capítulo 5 será estudado como o provador de teoremas explora a forma final das exigênicas produzida pela análise durante sua tarefa de responder perguntas.

4. TÓPICOS TEÓRICOS IMPORTANTES NO PROCESSO DE RECUPERAÇÃO DE INFORMAÇÃO IMPLEMENTADO

Introdução

O processo de recuperação de informação desencadeado quando uma pergunta ("query") é proposta ao sistema é efetuado por um provador de teoremas que atua avaliando parcialmente a pergunta com relação à base de conhecimentos. A parte terminológica correspondente às anotações das variáveis das asserções armazenadas na base de conhecimentos é fundamental no processo de unificação semântica adotado pelo provador de teoremas.

Neste capítulo introduzimos alguns tópicos teóricos que são ferramentas existentes importantes à implementação do sistema de recuperação de informação apresentado no próximo capítulo.

4.1. O Combinador de Ponto Fixo Y

Um ponto fixo de uma função *f* qualquer é um elemento *x* tal que $f(x) = x$. Por exemplo, *0* (zero) e *1* (um) são pontos fixos da função $\lambda x.\ *\ x\ x$.

Define-se combinador de ponto fixo Y como sendo a função Y que aplicada a uma função qualquer H produz um ponto fixo de H como resultado [57]. Assim sendo:

$Y\ H = H\ (Y\ H)$.

O combinador Y permite que se expressem funções recursivas através do Cálculo Lambda. Exemplifiquemos com a seguinte definição recursiva da função fatorial:

FAC = (λn. if (= n 0) 1 (* n (FAC (- n 1)))).

Esta definição se baseia na propriedade de se atribuir um nome a uma abstração lambda e de se referir a este nome dentro da própria abstração. Mas o Cálculo Lambda não é dotado desta propriedade, pois suas abstrações lambda são funções anônimas. A seguir é mostrado como resolver este impasse.

Pode-se reescrever a definição de FAC como:

FAC = (λfac. (λn. if (= n 0) 1 (* n (fac (- n 1))))) FAC,

ou, então:

FAC = H FAC,

onde:

H = (λfac. (λn. if (= n 0) 1 (* n (fac (- n 1))))).

Logo, FAC é um ponto fixo de H. Conseqüentemente:

FAC = Y H,

pois sabemos que (Y H) produz o ponto fixo de H. Realmente, no caso do fatorial:

FAC 1
= Y H 1
= H (Y H) 1
= (λfac. (λn. if (= n 0) 1 (* n (fac (- n 1))))) (Y H) 1
→ (λn. if (= n 0) 1 (* n(Y H (- n 1)))) 1
→ if (= 1 0) 1 (* 1 (Y H (-1 1)))
→ * 1 (Y H 0)
= * 1 (H (Y H) 0)

= * 1 ((λfac. (λn. if (= n 0) 1 (* n (fac (- n 1))))) (Y H) 0)

→ * 1 ((λn. if (= n 0) 1 (* n (Y H (- n 1)))) 0)

→ * 1 (if (= 0 0) 1 (* 0 (Y H (- 0 1))))

→ * 1 1

→ 1

Note-se que FAC = Y H é uma definição não recursiva de FAC.

Y pode ser representada pela abstração lambda:

Y = (λh. (λx. h(x x)) (λx. h(x x))).

Realmente:

Y H

= (λh. (λx. h(x x)) (λx. h(x x))) H

↔ (λx. H (x x)) (λx. H (x x))

↔ H ((λx. H (x x)) (λx. H (x x)))

↔ H (Y H)

4.2. O Símbolo ⊥

O símbolo ⊥ (bottom) é o valor atribuído a uma expressão que não tem uma forma normal. Lembremos que uma expressão está na forma normal quando não contém nenhuma avaliação pendente, isto é, quando não apresenta nenhuma redução a ser efetuada. Por exemplo, a expressão *(+ (* 3 4) (* 7 8))* pode ser reduzida a sua forma normal com a seqüência:

(+ (* 3 4) (* 7 8))

(+ 12 (* 7 8))

(+ 12 56)

68 (forma normal da expressão *(+ (* 3 4) (* 7 8))*)

Nem todas as expressões têm forma normal, como é o caso da expressão *(D D)*, onde *D* é *(λx. x x)*. A avaliação da expressão *(D D)* não termina nunca, ou seja:

$(D\ D) = \perp$

4.3. Cláusula

Uma cláusula [12] é uma disjunção de literais. Pode-se representar uma cláusula como um conjunto de literais, tal como mostrado abaixo:

A cláusula:

$P \vee Q \vee \neg R$

pode ser representada pelo conjunto:

$\{P, Q, \neg R\}$

Uma cláusula que não contenha nenhum literal é chamada de cláusula vazia e é representada por δ. A cláusula vazia (δ) é sempre falsa, uma vez que não pode ser satisfeita por nenhuma interpretação. Uma cláusula com um único literal é chamada de cláusula única.

Um conjunto S de cláusulas é entendido como uma conjunção de todas as cláusulas em S, onde toda variável em S é considerada governada por um quantificador universal. Exemplo:

A fórmula:

$\forall x\ \exists y\ \exists z\ ((\neg P(x, y) \wedge Q(x,z)) \vee R(x, y, z))$,

cuja forma "skolem" padrão é:

$\forall x((\neg P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x))))$,

pode ser representada pelo seguinte conjunto de cláusulas:

$\{\{\neg P(x, f(x)), R(x, f(x), g(x))\}, \{Q(x, g(x)), R(x, f(x), g(x))\}\}$

Teorema 1:

Seja S um conjunto de cláusulas representando a forma padrão de uma fórmula F. Então, F é inconsistente se, e somente se, S é inconsistente.

A prova deste teorema pode ser encontrada na página 48 de [12].

Lembremos que uma fórmula é inconsistente se e somente se ela é falsa em todas as suas interpretações.

Se F não é inconsistente, geralmente F não é equivalente a S, tal como mostrado no exemplo abaixo:

Seja a interpretação:

$F = \exists x\, P(x)$
Domínio: {1, 2}
a = 1
P(1) = falso
P(2) = verdadeiro

Neste exemplo, $S = \{\{P(a)\}\}$ e, na interpretação adotada, F é verdadeira e S é falsa (inconsistente).

Apesar de a "skolemização" trazer danos lógicos às fórmulas, o teorema 1 mostra que o processo de "skolemização" de uma fórmula não altera a propriedade da inconsistência. Tal constatação é fundamental no processo de se provar que uma

fórmula é verdadeira, pois, conforme discutido a seguir, a maioria dos provadores implementados se baseiam em procedimentos de refutação, isto é, ao invés de provarem que uma fórmula é válida, eles tentam provar que a negação desta fórmula é inconsistente.

4.4. Princípio da Resolução de Robinson

O princípio da Resolução de Robinson [12] é uma regra de inferência poderosíssima citada a seguir:

Para duas cláusulas C1 e C2 quaisquer, se há um literal L1 em C1 que seja a negação de um literal L2 em C2, então elimine L1 e L2 de C1 e C2, respectivamente, e construa a disjunção das cláusulas restantes.

A disjunção construída é chamada *resolvente* de C1 e C2.

De uma forma mais tradicional, essa inferência pode ser apresentada como:

$$\frac{\{\{L1, M1, N1, \ldots\}, \{\neg L1, M2, N2, \ldots\}\}}{\{\{M1, N1, \ldots, M2, N2, \ldots\}\}}$$

Teorema 2:

Se duas cláusulas C1 e C2 são verdadeiras, um resolvente C de C1 e C2 é conseqüência lógica de C1 e C2.

Prova:

Sejam C1, C2 e C denotadas, respectivamente, como:

$C1 = L \vee C1'$

$C2 = \neg L \vee C2'$

$C = C1' \vee C2'$,

onde C1' e C2' são disjunções de literais. Suponha-se que C1 e C2 sejam verdadeiras em uma interpretação I. Deseja-se provar que a resolvente C de C1 e C2 é também verdadeira em I. Para provar isto, note-se que L ou $\neg L$ tem que ser falso em I. Assuma-se que L seja falso em I. Então, C1 não pode ser cláusula unitária, caso contrário, C1 seria falsa em I. Logo, C1' tem que ser verdadeira em I. Conseqüentemente, o resolvente C, isso é, $C1' \vee C2'$ é verdadeiro em I. Similarmente, pode-se mostrar que se $\neg L$ é falso em I, então C2' deve ser verdadeiro em I. Conseqüentemente, $C1' \vee C2'$ deve ser verdadeira em I.

Exemplos:

$\underline{\{\{P, R\}, \{\neg P, Q\}\}}$

$\{R, Q\}$

$\underline{\{\{\neg P, Q, R\}, \{\neg Q, S\}\}}$

$\{\neg P, R, S\}$

Definição:

Dado um conjunto S de cláusulas, uma dedução (resolução) de C a partir de S é uma seqüência finita C1, C2, ..., Ck de cláusulas, onde cada Ci $(1 \leq i \leq k)$ é uma cláusula de S ou uma resolvente das cláusulas que precedem Ci e onde Ck = C.

Uma dedução de δ a partir de S é chamada de refutação ou prova de S. Exemplo:

$S = \{\{P, Q\}, \{\neg P, Q\}, \{P, \neg Q\}, \{\neg P, \neg Q\}\}$

(1) $P \vee Q$ (primeira cláusula de S)

(2) $\neg P \vee Q$ (segunda cláusula de S)

(3) $P \vee \neg Q$ (terceira cláusula de S)

(4) $\neg P \vee \neg Q$ (quarta cláusula de S)

(5) Q (Princípio da Resolução aplicado a (1) e (2))

(6) $\neg Q$ (Princípio da Resolução aplicado a (3) e (4))

(7) δ (Princípio da Resolução aplicado a (5) e (6))

Neste exemplo, δ é resolvente de S (logo, δ é conseqüência lógica de S). Como δ é falso, então, usando o teorema 2, S é inconsistente. Logo, a fórmula F representada por S é inconsistente, o que implica em $\neg F$ verdadeira.

4.5. Substituição e Unificação

4.5.1. Substituição

Uma substituição [12] é um conjunto finito da forma {t1/v1, ..., tn/vn}, onde cada *vi* ($1 \leq i \leq n$) é uma variável, cada *ti* é um termo diferente de *vi* e onde não há mais de um elemento com a mesma variável após a barra. Quando t1, ..., tn são termos básicos (isso é, termos sem variáveis), a substituição é chamada de substituição básica. A substituição desprovida de elementos é chamada de substituição vazia e é denotada por ε. Usar-se-ão letras gregas para indicarem as substituições.

Exemplo de substituição:

{g(y)/x, f(g(b))/z, a/w}.

Considerem-se a substituição θ = {t1/v1, ..., tn/vn} e a expressão E. Neste caso, $E\theta$ é uma expressão chamada instância de E e é obtida a partir de E pela substituição de cada variável *vi* de E, $1 \leq i \leq n$, pelo termo *ti*. Exemplo:

Sejam:

θ = {f(a)/x, b/y, g(c)/z}

e

$E = P(x, y, z)$.

Assim,

$E\theta = P(f(a), b, g(c))$.

Uma substituição θ é chamada *unificadora* para um conjunto {E1, E2..., Ek} se, e somente se, $E1\theta = E2\theta = ... = Ek\theta$.

O conjunto {E1, E2, ..., Ek} é chamado *unificável* se existe uma substituição unificadora para ele [12, 61].

4.5.2. Processo de Unificação Clássica de Dois Elementos

1. Se pelo menos um dos elementos for uma variável não ligada a valor algum, a unificação obterá êxito e a citada variável se unificará com o outro elemento.

2. Se ambos os elementos forem variáveis não unificadas, a unificação será bem sucedida e a primeira dessas variáveis se unificará com a segunda.

3. Se ambos os elementos forem functores, letras de predicados ou constantes atômicas, a unificação só obterá êxito se ambos os elementos forem idênticos.

4. Se ambos os elementos forem variáveis já unificadas, a unificação só obterá êxito se os objetos com os quais elas estiverem unificadas se unificarem também.

Para se verificar se duas expressões se unificam, devemos percorrê-las da esquerda para a direita testando se suas sub-expressões de mesma posição se unificam (segundo descrito no processo acima). Caso haja fracasso em alguma tentantiva de unificação de sub-expressões, o processo todo fracassa e, conseqüentemente, as duas

expressões não se unificam. Se, terminados os testes, não tiver ocorrido fracasso algum nas unificações das sub-expressões, então ambas as expressões terão se unificado com êxito.

Exemplo: verificar se as expressões abaixo se unificam:

E1 = P(a, x, f(g(y))) e
E2 = P(z, f(z), f(u)),

onde *x*, *y*, *z* e *u* são variáveis, *a* é constante, *P* é letra de predicado, *f* e *g* são functores.

Analisando-se da esquerda para a direita:

1. P se unifica com P (item 3 do processo de unificação);

2. A variável não unificada *z* se unifica com a constante *a* (item 1 do processo);

3. A variável não unificada *x* se unifica com a sub-expressão *f(z)*, ou seja, *f(a)*, uma vez que *z* está unificada com *a* (item 1 do processo).

4. A sub-expressão *f(g(y))* se unifica com a sub-expressão *f(u)*, pois *f* se unifica com *f* (item 3 do processo) e a variável não unificada *u* se unifica com *g(y)* (item 4 do processo).

Logo, a substituição final (também chamada *substituição mais geral*) decorrente da unificação do exemplo anterior é:

σ = {a/z, f(a)/x, g(y)/u}.

4.6. Determinação de Resolventes Para um Conjunto S de Cláusulas da Lógica de Primeira ordem:

O processo de determinação de resolventes para um conjunto S de cláusulas [12, 32] é análogo ao descrito para o cálculo sentencial, isso é, aplicamos o Princípio da Resolução de Robinson aos componentes de S. A diferença é que aqui se devem considerar os efeitos das unificações efetuadas ao se comparar um literal que aparece afirmado em uma cláusula com o mesmo literal que aparece negado na outra.

Exemplo: a seguir analisa-se o efeito das unificações na seqüência de resolventes correspondentes ao conjunto S de cláusulas abaixo:

$S = \{\{ \neg T(x, y, u, v), P(x, y, u, v)\}, \{\neg P(x, y, u, v), E(x, y, v, u, v, y)\},$
$\{T(a, b, c, d)\}, \{\neg E(a, b, d, c, d, b)\}\}.$

(1) $\neg T(x, y, u, v) \vee P(x, y, u, v)$.
(2) $\neg P(x, y, u, v) \vee E(x, y, v, u, v, y)$.
(3) T(a, b, c,d).
(4) $\neg E(a, b, d, c, d, b)$.
(5) $\neg P(a, b, c, d)$. (resolvente de 2 e 4)
(6) $\neg T(a, b, c, d)$. (resolvente de 1 e 5)
(7) δ (resolvente de 3 e 6)

4.7. Teorema da Completeza do Princípio da Resolução

Um conjunto S de cláusulas é não satisfactível se, e somente se, há uma dedução de δ a partir de S.

Informalmente, o Teorema da Completeza do Princípio da Resolução [12, 32] pode ser provado da seguinte forma: através do teorema 2, sabe-se que o resolvente de cláusulas verdadeiras é sempre verdadeiro. Daí pode-se concluir que, caso o resolvente de um conjunto de cláusulas S seja a cláusula vazia δ (sempre falsa), então, pelo menos

uma das cláusulas de S é falsa, o que torna o conjunto S de conjunção de cláusulas falso.

Conforme será visto na próxima secção, os provadores de teorema em geral são baseados neste Teorema da Completeza. Realmente, tais teoremas propõem que, caso se queira provar uma hipótese H a partir de axiomas verdadeiros A1, A2, A3, ..., An, deve-se tentar deduzir δ a partir do conjunto de cláusulas S = {¬H, A1, A2, A3, ..., An}. Caso δ seja um resolvente de S, então, pelo Teorema da Completeza do Princípio da Resolução, S é não satisfactível. Neste caso, como A1, A2, A3, ..., An são verdadeiros, ¬H tem que ser falso, o que prova a hipótese H.

4.8. Provadores de Teoremas da Lógica de Primeira Ordem

Durante muitos anos se tentou criar um procedimento capaz de interpretar qualquer fórmula da Lógica de Primeira Ordem. Em 1936, Church e Turing provaram que tal procedimento não existe. Entretanto, há procedimentos capazes de constatar que uma fórmula é válida se ela de fato o é. Para fórmulas não válidas, estes procedimentos podem não parar nunca.

Herbrand (1930) propôs um método de prova para sentenças de primeira ordem no qual se busca uma interpretação capaz de tornar falsa a sentença analisada. Contudo, se ela for válida, tal interpretação não existe e o algoritmo pára após um número finito de tentativas. O método de Herbrand é a base da maioria dos modernos procedimentos de prova automática.

Como uma fórmula é válida se, e somente se, sua negação é inconsistente, os provadores vêm seguindo uma linha na qual tentam detectar se a negação de uma dada fórmula é inconsistente com um conjunto de cláusulas ou não. Após diversas tentativas de implementação de provadores que se revelaram ineficientes, tais como o de Gilmore (1960) e o de Davis/Putnan (1960), Robinson (1965) propôs o Princípio da Resolução através do qual conseguiu aumentar significativamente a eficiência dos provadores. A seguir fornece-se os passos do algoritmo de um provador baseado em resolução linear [12]:

(1) Converta a pergunta P numa conjunção de literais.

(2) Considere como hipótese a disjunção de literais que representa a negação da pergunta. Logo, a hipótese é representada pelo literal $\neg P$.

(3) Coloque os axiomas da base de conhecimentos na forma normal conjuntiva (conjunção de disjunções).

(4) Escolha um literal da hipótese. Seja C este literal.

(5) Escolha uma disjunção R da base de conhecimentos onde o literal $\neg C$ esteja presente. Obviamente, esta disjunção é verdadeira, caso contrário, a base de conhecimentos seria falsa.

(6) Seja $R \vee \neg P$ a nova hipótese.

(7) Elimine $C \vee \neg C$ da nova hipótese, pois essa disjunção é sempre verdadeira.

(8) Se a eliminação de $C \vee \neg C$ produz uma contradição, então $\neg P$ é falsa e a pergunta, que é a negação da hipótese, é verdadeira.

(9) Se nenhuma contradição é obtida, retorne ao passo (4) do algoritmo.

Observe-se que no passo (7) efetua-se a unificação que produz o resolvente de R e da hipótese.

A seguir analisa-se um exemplo do algoritimo de prova.

Seja a base de conhecimentos formada pela conjunção entre as disjunções [d1], [d2] e [d3], tal como mostrado abaixo:

[d1] succ(V, s(V)) $\wedge$ % O sucessor de V é s(V).

[d2] p(X) $\vee \neg$succ(Y, X) $\vee \neg$p(Y) $\wedge$ % X é positivo se X é o sucessor de Y e Y é positivo.

[d3] p(s(0)) % O sucessor de 0 (zero) é positivo.

Considere-se a pergunta: *o sucessor do sucessor de 0 é positivo?*

A seqüência de prova produzida é:

(1) A pergunta é:

p(s(s(0)))?

(2) A hipótese é:

$\neg$p(s(s(0)))

(3) A base de conhecimentos já está na forma normal conjuntiva, isso é:

[d1] succ(V, s(V)) $\wedge$
[d2] p(X) $\vee$ $\neg$succ(Y, X) $\vee$ $\neg$p(Y) $\wedge$
[d3] p(s(0))

(4.1) Escolha um litoral da hipótese:

Literal escolhido: $\neg$p(s(s(0)))

(5.1) Escolha uma disjunção onde o literal p(s(s(0))) esteja presente. Escolha-se a disjunção [d2]. Para unificar p(X) em [d2] com p(s(s(0))), precisa-se fazer X = s(s(0))), onde o símbolo = expressa unificação lógica.

(6.1) A nova hipótese é:

[d2] $\vee$ $\neg$p(s(s(0))), isso é:

p(s(s(0))) $\vee$ $\neg$succ(Y, s(s(0))) $\vee$ $\neg$p(Y) $\vee$ $\neg$p(s(s(0))).

(7) Elimine $p(s(s(0))) \vee \neg p(s(s(0)))$ porque ela é uma conjunção sempre verdadeira. Tem-se, então:

$\neg succ(Y, s(s(0))) \vee \neg p(Y)$

(8.1) Nenhuma contradição foi encontrada.

(9.1) Retorne ao passo 4.

(4.2) Escolha o literal $\neg p(Y)$ na hipótese (que, agora, corresponde à nova hipótese).

(5.2) Escolha a disjunção [d3] com $Y = s(0)$.

(6.2) Nova hipótese:

$[d3] \vee \neg succ(s(0), s(s(0))) \vee \neg p(s(0))$,

isso é:

$p(s(0)) \vee \neg succ(s(0), s(s(0))) \vee \neg p(s(0))$.

(7.2) Elimine $p(s(0)) \vee \neg p(s(0))$. A nova hipótes se torna:

$\neg succ(s(0), s(s(0)))$.

(8.2) Encontra-se uma contradição entre:

$\neg succ(s(0), s(s(0)))$

e

succ(V, s(V))

em [d1], com V = s(0). Logo, a hipótese ¬p(s(s(0))) é falsa e a pergunta p(s(s(0))) é respondida com *verdadeiro,* indicando que o sucessor do sucessor de zero é positivo.

4.9. Extensões da unificação clássica:

A unificação, tal como foi proposta em sua forma clássica [61], eventualmente pode ser estendida para processos alternativos que permitem maior flexibilidade e eficiência computacional, tais como programação por restrição e unificação semântica, ambas discutidas a seguir.

4.9.1. Programação por Restrição

As linguagens de restrição [6] permitem a resolução de metas propostas de diversas maneiras, independentemente da forma na qual o conhecimento está armazenado na base de conhecimentos. Por exemplo, se a base de conhecimentos contiver a seguinte informação relacionando temperaturas em graus Celsius (C) e graus Fahrenheit (F):

C = (F - 32) * 5 / 9

e se o sistema tem como restrição que

C = 100,

pode-se converter F em C na meta:

conv(C, F), que, neste caso, corresponde a:

conv(100, F).

Analogamente,

se o sistema tem como restrição que

F = 0,

pode-se converter F em C na meta:

conv(C, F), que, neste caso, corresponde a:

conv(C, 0).

Além disso, pode-se calcular também a temperatura em que a retrição $F = C$ é satisfeita.

Citemos o seguinte meta-interpretador como exemplo de programação por restrição (alterado a partir de Cohen, 1990):

```
% resolve(Lista-de-alvos, Restrics-anteriores, Novas-restrics)

resolve([ ], C, C).
resolve([G|RG], PC, NC) :-
   resolve2(G, PC, TC),
   resolve(RG, TC, NC).

resolve2(G, PC, NC) :-
   rclause(G, B, CC),
   restr(PC, CC, TC),
   resolve(B, TC, NC).
```

O predicado "rclause" recebe um predicado no primeiro argumento e busca na base uma cláusula C que o defina, devolvendo em seu segundo argumento uma lista

contendo as eventuais chamadas recursivas do corpo de C e, em seu terceiro argumento, o corpo de C em forma de lista. O predicado "restr" retorna em seu terceiro argumento o resultado da aglutinação e da simplificação das restrições expressas em seu primeiro e segundo argumentos. É ele quem tenta satisfazer as restrições. Exemplo:

restr(A = B + 5, A = 8, B = 3),

onde:

```
restr(L1, L2, L) :-
   append(L1, L2, L3),
   simpl(L3, [ ], L).
```

A tentativa de simplificação em "simpl" pode utilizar duas técnicas:

- Propagação local: quando o sistema lida com equações de primeiro grau com uma variável;
- Resolução de sistemas de equações pelo método de Gauss (a ser usada, por exemplo, na resolução da meta que pergunta em que ponto temos uma igualdade entre graus "Celsius" e graus "Fahrenheit", ou seja, qual é o valor de F e C diante da restrição "F = C").

Ainda se considerando o exemplo de conversão de temperatura, caso a base de dados contenha a seguinte informação:

conv(C, F) :- F is (1.8 * C) + 32,

consegue-se provar qualquer uma das metas abaixo (por propagação local):

?- resolve([conv(C, F)], [C is 100.0], R)
?- resolve([conv(C, F)], [F is 212.0], R)

? - resolve([conv(100.0, F)], [ ], R)

4.9.2. Unificação Semântica

Na unificação tradicional [13] a equação g(t1, ..., tp) = g'(s1, ..., sq) não é passível de solução se g e g' não são idênticos ou se $p \neq q$, uma vez que g e g' são tratados como construtores. Distintamente disto, a unificação semântica [18] permite que g ou g' ou ambos (ou qualquer símbolo de função na equação) sejam funções definidas em EQN. Dois termos são semanticamente unificáveis se apresentarem formas semanticamente equivalentes (obtidas por redução (ões) em qualquer termo redutível da equação) que são unificáveis. Considere-se um axioma como algo que impõe restrições ao resultado das aplicações das operações mencionadas no axioma. Considere-se também um processo de unificação semântica cujo teor semântico se manifeste através de restrições (chamemo-lo de unificação restritiva). Dados uma expressão de entrada e um conjunto de axiomas, a unificação restritiva ("constraining unification") produz um conjunto de restrições, chamado restrições de transformadas ("transformation constraints"), a que todas as transformadas permitidas ("allowable transformation") da expressão de entrada devem satisfazer. Podem ocorrer casos em que mais de uma expressão satisfaz as restrições de transformadas, onde há as chamadas operações não determinísticas, ou, então, casos em que nenhuma expressão as satisfaz, onde se diz que o conjunto das restrições de transformadas é inconsistente (nestes últimos casos não existem transformadas permitidas da expressão de entrada). Uma restrição de transformada é produzida pela unificação da expressão de entrada com uma subexpressão de algum axioma, com posterior substituição da sub-expressão unificada por uma variável especial (não há restrição quanto ao lugar no axioma onde a unificação pode ser efetuada). Exemplos:

a) Seja a unificação da expressão de entrada:

square-root(4)

com a sub-expressão:

square-root(X)

no axioma:

X = square(square-root(X)),

produzindo a restrição de transformada

4 = square(V),

onde V é uma variável especial que não ocorre em outro ponto da restrição de transformada. Qualquer valor da variável especial que satisfaça simultaneamente a todas as restrições é uma transformada permitida da expressão de entrada original. No nosso exemplo, qualquer valor de V que satisfaça *4 = square(V)* é uma transformada permitida da expressão de entrada original *square-root(4).*

b) Seja a expressão de entrada:

plus(A, succ(0)),

referindo-se à soma e a sucessão numérica.

Considerem-se os axiomas de Peano:

b.1) plus(0, X) = X
b.2) plus(succ(X), Y) = succ(plus (X, Y))

Segundo o que foi exposto, a unificação da expressão de entrada com o primeiro axioma produz o primeiro conjunto de restrições de transformadas dado por:

{V1 = succ(0) }.

Analogamente, a unificação da expressão de entrada com o segundo axioma produz o segundo conjunto de restrições de transformadas dado por:

{ V2 = succ(plus(X, succ(0))) }.

Para que um dado valor da variável especial a torne uma transformada permitida da expressão de entrada, ele tem que satisfazer a todas as restrições de transformadas de um dos conjuntos de restrições de transformadas produzidos pela unificação. No caso, V1 deve valer 1 e V2 pode valer 2, 3, 4 ...

Note-se que a unificação semântica pode ser usada no sentido de dispensar uma ordem fixa para os axiomas, tal como no exemplo anterior, onde o método seguido não exige uma ordem fixa para os axiomas b.1 e b.2.

c) Ainda para ilustrar a unificação semântica, citemos a atuação de Prolog com Igualdade [18]. Tal Prolog permite as seguintes unificações:

6 = sucessor(5)
p(6) = p(sucessor(5))

Para ampliar o algoritmo de unificação tradicional, o intérprete de Prolog com Igualdade atua da seguinte maneira: suponha-se que se queira unificar ϕ e ψ. Primeiramente ele tenta unificar ϕ e ψ usando a unificação clássica. Se tal unificação fracassa, ele tenta provar a meta *(equals* ϕ ψ*)*. Se ele obtiver êxito, a unificação é bem sucedida e novas ligações de variáveis podem eventualmente ser produzidas. Se a prova fracassa, ele tenta provar *(equals* ψ ϕ*)*. Exemplo (verificação de igualdade entre dois números representados sob forma de fração, isso é, como uma função *quoc* de numeradores e denominadores):

```
(equals (quoc _num1 _denom1) (quoc _num2 _denom2)) :-
        (times _num1 _denom2 _intermediário)
        (times _num2 _denom1 _intermediário)
```

Pela unificação clássica, *(quoc 2 3) = (quoc _x 3)*, mas *(quoc 2 3)* não se unifica com *(quoc _x 6)*. Pela unificação com igualdade, a tentativa de unificar *(quoc 2 3)* com *(quoc _x 6)* produz a meta *(equals (quoc 2 3) (quoc _x 6))* que, por sua vez, liga a variável *_x* ao valor *4*.

Para se avaliar se *quoc* é um tipo inteiro e se tem valor *_n*, define-se:

```
(equals (quoc _num _denom) _n) :-
        (type _n integer)
        (if (and (variable _num) (variable _denom))
            (and (= _num _n) (= _denom 1))
            (times _denom _n _num)) )
```

Assim, por exemplo, a meta:

```
?- (member (quoc 4 _x) [2 3 (cons _y _z) (quoc _r _w) (quoc 2 7)])
```

produz: _x = 2; _r = 4; _x = _w; _x = 14

Pode-se ainda acrescentar a esse exemplo a seguinte definição de relação de grandeza entre duas frações:

```
(> (quoc _n1 _d1) (quoc _n2 _d2)) :-
        (times _n1 _d2 _x)
        (times _d1 _n2 _y)
        (> _x _y)
```

Desse modo, a meta:

```
?- (> (quoc 3 2) 1)
```

tenta unificar *1* com *(quoc _n2 _d2)*, o que produz as ligações *_n2 = 1* e *_d2 = 1* e sucesso para a meta proposta. Lembremos que em outras linguagens esse processo se chama *coerção automática de tipo* ("*automatic type coercion*").

Prolog com igualdade permite a definição de tipos de dados extensíveis e de operações genéricas, conforme mostramos no exemplo abaixo:

Definido o predicado que calcula a área de um polígono regular, de forma geral:

```
% (área (polígono-regular (_x _y _número-de-lados _extensão)) _área)
```

e dado que:

```
(equals (triângulo-equilátero _x _y _comprimento)
        (polígono-regular _x _y 3 _comprimento)),
```

pode-se resolver a meta:

```
?- (área (triângulo-equilátero _ _ 100) _resp)
```

ainda que não tenha sido definido o predicado *área* para triângulos equiláteros em particular.

Caso se introduza um predicado que defina a área de uma elipse, cuja forma geral é:

```
% (área (elipse _x _y _menor-eixo _maior-eixo) _área),
```

tal definição não vai interferir em metas que indaguem, por exemplo, sobre áreas de triângulos, pois não há nenhuma relação que afirme que triângulos e elipses sejam iguais.

Prolog com igualdade, sob certo aspecto, é mais flexível que as linguagens orientadas por objeto. Por exemplo, ainda seguindo as definições que acabamos de ilustrar, pode-se provar a meta:

```
?- (and (= _obj    (polígono-regular (_ _ _número-de-lados _comprimento)))
        (_número-de-lados 3)
        (predicate _obj)).
```

Isto mostra que em Prolog com igualdade pode-se prosseguir com a computação mesmo que ela esteja indeterminada em muitos de seus aspectos (no caso da meta acima, inicialmente o objeto *_obj* é unificado com um polígono regular de número de lados indeterminado. Posteriormente, o número de lados é instanciado com *3* e "predicate" é posto à prova. Se "predicate" tem asserções que permitam provas sobre triângulos, então a meta será resolvida com êxito, pois *_obj* será unificada com *triângulo-equilátero*). Ao contrário disso, nas linguagens orientadas por objetos (variáveis), a classe do objeto já deve ser conhecida no momento da instanciação.

Ainda neste capítulo serão discutidos os aspectos e critérios da unificação semântica implementada na tese. Adiante-se, entretanto, que ela será guiada pela semântica da subsunção da Lógica Terminológica.

4.10. Avaliação Parcial

A Avaliação Parcial foi concebida por Ershov [26] e teve continuidade com Futamura, no âmbito das linguagens funcionais e com Komorowski, Venken, Takeuchi/Furokawa [70], no âmbito das linguagens lógicas [77].

A seguir introduz-se a teoria de avaliação parcial na forma de um sistema formal [77].

Seja S um sistema formal (veja secção 3.1.3) com um conjunto C de axiomas. Seja *f* uma fórmula de S correspondente a uma hipótese. Diz-se que *fk* é uma computação de *f* se:

- *fk* foi inferida a partir de um conjunto de fórmulas que inclui *f*, ou se *fk* foi inferida a partir de um conjunto de fórmulas que inclui uma computação de *f*.
- Em cada passo, existe um critério rígido para se escolher a inferência aplicada.

Se *fk* é uma computação de *f*, então a seqüência *{f, f1, f2, ..., fk}* que permite deduzir *fk* é chamada história ou traço da computação. O conjunto de axiomas é chamado programa. A hipótese *f* é chamada estrutura de entrada de dados ou, simplesmente, de entrada. Ilustremos o conceito de computação através de um exemplo onde se representam as fórmulas por funções do cálculo lambda, conforme se segue:

Seja o seguinte conjunto de axiomas (ou programa, ou padrões sintáticos):

(D1) (λ(x y) E) = (λx(λy E))

(D2) verdadeiro = (λ(x y) x)

(D3) topo = (λp (p verdadeiro))

(D4) (E1 . E2) = (λf ((f E1) E2))

Sejam as regras de inferência:

(r1) Expansão sintática: o lado esquerdo de um padrão sintático pode ser substituído pelo seu lado direito.

(r2) Conversão beta: expressões na forma ((λv E1) E2) podem ser convertidas para E1[E2/v], onde E1[E2/v] representa a expressão E1 com todas as ocorrências de v substituídas por E2, desde que nenhuma variável livre de E1 se torne ligada com a substituição.

No sistema acima pode-se, por exemplo, computar E1 a partir da hipótese *(topo (E1 . E2))*, conforme mostrado a seguir:

- A partir de *(topo (E1 E2))* e usando-se (D3) e (r1), infere-se *((λp (p verdadeiro)) (E1 . E2))*

- A partir de *((λp (p verdadeiro)) (E1 . E2))* e usando-se (D2) e (r1), infere-se *((λp (p (λ (x y) x))) (E1 . E2))*.

- A partir de *((λp (p (λ (x y) x))) (E1 . E2))*, usando-se (D1) e (r1), infere-se *((λp (p (λx (λy x)))) (E1 . E2))*.

- A partir de *((λp (p (λx (λy x)))) (E1 . E2))* , usando-se (D4) e (r1), infere-se *((λp (p (λx (λy x))) (λf ((f E1) E2)))*.

- A partir de *((λp (p (λx (λy x))) (λf ((f E1) E2)))*, usando-se (r2), infere-se *((λf ((f E1) E2)) (λx (λy x)))*.

- A partir de *((λf ((f E1) E2)) (λx (λy x)))*, usando-se (r2), infere-se *(((λx (λy x)) E1) E2)*.

- A partir de *(((λx (λy x)) E1) E2)*, usando-se (r2), infere-se *((λy E1) E2)*.

- A partir de *((λy E1) E2)*, usando-se (r2), infere-se a computação E1 a partir do dado de entrada *(topo (E1 . E2))*.

Seja um sistema formal destinado a efetuar computações. Sejam D e P dois subconjuntos recursivos do conjunto Fs de fórmulas bem formadas do sistema, tal que todos os seus axiomas (ou procedimentos) pertençam ao conjunto P (programa). *(D ∩ P)* não é necessariamente vazio. Chamemos D de conjunto de dados. Sabe-se que a linguagem de um sistema formal é formada pelo seu alfabeto, pelo seu conjunto de

fórmulas bem formadas e pelas suas regras de inferência, sendo representada por *I(Ss, Fs, Rs)*. Seja uma linguagem particular *I(D, P, s)*. Se uma computação de *f* tem componentes somente em P, dizemos que ela é parcial. Se a computação de *f* pertence exclusivamente a D (sem componentes em P), dizemos que ela é total. Chama-se avaliador ao provador de teoremas que aplica as regras de inferência ao programa e aos dados de entrada a fim de realizar as computações. Chama-se avaliador parcial ao provador de teoremas que, ainda que não disponha de informações suficientes para produzir uma avaliação total para uma hipótese de entrada (ou dado de entrada, ou, ainda, "query"), isso é, ainda que não consiga produzir um dado como resposta (computação) para a hipótese, ele tenta executar uma avaliação parcial que retorne como resposta um axioma com o máximo possível de informações que ele conseguir reunir. A vantagem disto é que o agente que propôs a pergunta pode-se servir dessa resposta parcial para fazer novas inferências que eventualmente o coloquem ainda mais próximo de uma resposta mais específica, mais próxima a um dado.

O princípio básico da avaliação parcial consiste em avaliar as partes de um programa que sejam atendidas pelos dados de entrada, deixando inalteradas as demais. Isso pode ser visto nos exemplos das secções 4.10.1 e 4.10.2 a seguir. Na secção 4.10.1 veremos que tal técnica pode ser usada para aumentar a eficiência dos programs. Na secção 4.10.2 será visto que um avaliador parcial realmente corresponde a um demonstrador de teoremas que, dependendo de dispor ou não de todos os dados de entrada e de todas as informações necessárias para resolver a hipótese, produz como resposta, respectivamente, um dado ou um axioma.

4.10.1. Avaliação Parcial Como Ferramenta de Otimização de Programas

Considere-se o seguinte programa escrito em PROLOG [77]:

```
mostra_tela(Tela) :-
1           tela(Tela, Começo, Fim),
2           clear(Começo, Fim),
            enqto(
3 6 9               texto(Tela, L, C, Texto),
4 7 10      put(L, C, Texto)
5 8 11            ),
            enqto(
12 15               campo(Tela, Nome, L1, C1, Comp),
13 16               putd(L1, C1, Comp, '.')
14 17             ).

enqto(X, Y) :- X, Y, fail.
enqto(X, Y).
```

Onde, *clear*, *put* e *putd* são primitivas.

Considerem-se os dados do programa:

```
tela(teste, 5, 15).
texto(teste, 5, 15, 'Tela A - Programa').
texto(teste, 9, 15, 'Tela B - Comandos').
texto(teste, 11, 5, 'Tela C - Dados').

campo(teste, comando, 9, 13, 1).
campo(teste, dados, 11, 13, 60).
```

Para se executar o programa mostra_tela(teste) precisa-se efetuar as 17 inferências enumeradas à esquerda da definição do programa, onde as inferências de números 5, 8, 14 e 17 se referem ao predicado *fail* de *enqto*. Contudo, pode-se submeter o programa PROLOG acima a um avaliador parcial de programas que

aumenta sua eficiência especializando-o para os dados do programa fornecidos acima. O resultado dessa avaliação é o programa mais eficiente e especializado mostrado abaixo:

```
mostra_tela(teste) :-
1           clear(5,15),
2           (put(5, 15, 'Tela A - Programa '),
3           (put(9, 15, 'Tela B - Comandos '),
4           (put(11, 5, 'Tela C - Dados '),
5           (putd(9, 13, 1, '.'),
6           (putd(11, 13, 60, '.'),
7           true)).
```

Realmente, para se executar o novo programa avaliado *mostra_tela(teste)*, precisa-se efetuar apenas 7 inferências, o que comprova o aumento de eficiência propiciado pela avaliação.

4.10.2. Provador de Teoremas Efetuando Avaliações Parciais e Totais

Neste exemplo mostra-se como um provador de teoremas pode efetuar avaliações parciais, caso produza como respostas axiomas a partir dos quais se pode fazer novas inferências, ou, então, avaliações totais, caso produza dados como resposta. Obviamente, avaliações parciais são produzidas quando o provador não dispõe de informações suficientes para conduzir a prova até o nível dos dados.

Neste exemplo, quando uma pergunta ("query") não pode ser completamente resolvida, o provador (ou avaliador parcial) consegue traduzir a pergunta em uma rede semântica, desde que ele conheça a correspondência entre o predicado n-ário que representa a pergunta e alguns predicados binários que lhe equivalham. Note-se que uma rede semântica, na verdade, consiste de uma conjunção entre predicados binários. Assim sendo, neste exemplo, o avaliador parcial traduz predicados n-ários em binários, tal como discutido no capítulo 2.

Abaixo implementa-se em LISP uma versão simplificada deste avaliador, propõe-se uma pequena base de dados e introduz-se uma pergunta que ilustra como funciona a prova. De uma maneira geral, o programa abaixo implementa os passos do algoritmo de prova descrito na secção 4.8.

```
(defun final-value (X E)
 (let ( (S (assoc X E))) (if S (final-value (cdr S) E) X) ))

(defun deref(X E)
 (if (consp X) (cons (deref (car X) E) (deref (cdr X) E))
   (let ( ( S (assoc X E)))
     (if S (deref (final-value (cdr S) E) E) X))) )

(defun unif(X Y E)
 (if (eq E 'not-unify) 'not-unify
   (let ( ( A (final-value X E)) (B (final-value Y E)))
     (cond ( (eq A B) E)
           ( (varp A) (cons (cons A B) E))
           ( (varp B) (cons (cons B A) E))
           ( (or (atom A) (atom B)) 'not-unify)
           ( (unif (car A) (car B)
                 (unif (cdr A) (cdr B) E))) ))) )

(defun varp(x) (or (integerp x) (eq x 'x) (eq x 'y) (eq x 'z)))

(defvar global 0)

(defun nvar() (setq global (+ global 1)) global)

(defun crels(P) (eval (list (caar P)) ))
```

```
(defun rev-append(L H)
 (if (null L) H (rev-append (cdr L) (cons (car L) H))))

(defun  res(L R H Unif)
 (cons Unif (deref (rev-append L (append H R)) Unif)))

(defun semantic(x)
 (member (car x) '(elem agent acc dat)))
```

% *agent*, *acc* e *dat* representam *agente*, *acusativo* e *dativo*, respectivamente.

```
(defun expandable (x) (not (semantic x)))

(defun choice (L R)
 (if (expandable (car R)) (list L R)
    (choice (cons (car R) L) (cdr R))))

(defun solvers(P E)
 (let ( (LAR (choice () P)) (Rs ()) )
   (let ( (L (first LAR))
          (A  (first (second LAR)))
          (Rls (crels (second LAR)))
          (R (cdr (second LAR))))
     (dolist (Rn Rls)
        (let ( (Un (unif A (first Rn) E)))
           (when (listp Un) (push (res L R (cdr Rn) Un) Rs)) ))
   Rs)))

(defun complete(R)
 (if (null R) t
    (and (semantic (car R)) (complete (cdr R)))))
```

```
(defun proof(SOLVED WAIT)
 (do () ((null WAIT) SOLVED)
    (let ( (Y (pop WAIT)))
       (dolist (R (solvers (cdr Y) (first Y)))
          (if  (complete (cdr R))
             (push R SOLVED) (push R WAIT)) )) ))

(defmacro query(Padrao)
 `(mapcar (function (lambda(E) (deref (quote ,Padrao) (car E))))
       (proof () (list (cons () (quote (, Padrao))) ))) )

(defun partial(Padrao)
    (mapcar (function (lambda(E) (deref (cdr E) (car E))))
          (proof () `((() , Padrao))) ))
```

A seguir se discutem alguns dos predicados do provador.

A unificação prevista no passo 7 do algoritmo do provador (predicado *unif*) constrói ambientes contendo *ligações* (ou *"bindings"*) das variáveis envolvidas na prova. As variáveis serão representadas no ambiente como números inteiros. Exemplo de ambiente:

E1 = ((1 . ana) (2 . livro) (3 . 1) (4 . 3))

O valor final de uma variável é o resultado de se obter sucessivos valores até que se obtenha um que não seja uma variável. Por exemplo, a avaliação do valor final da variável 4 no ambiente E1 recai no valor 3 que também é uma variável. O valor de 3, por sua vez, é a variável 1. O valor de 1 é a constante *ana*, que corresponde, portanto, ao valor final da variável 4 em E1.

A operação de "dereferenciamento" de uma estrutura com relação a um ambiente, implementada no predicado *deref*, substitui as variáveis da expressão pelos

seus respectivos valores finais no ambiente. Exemplo: considerem-se o ambiente E1 e a estrutura X abaixo:

E1 = ((1 . mary) (2 . book) (3 . 1) (4 . 1))
X = (gave (agent john) (acc 2) (dat 4))

Logo:

(deref X E1) ⇒ (gave (agent john) (acc book) (dat mary))

A unificação de X com Y no ambiente E consiste em se encontrar a substituição mais geral das variáveis de X e Y de tal modo que X e Y se tornem iguais. Exemplo:

(unif '(see 1 2) '(see john book) ()) ⇒ ((1 . john) (2 . book))

O predicado *choice* executa o passo 4, ou seja, ele escolhe um literal na lista de hipóteses. Neste provador simplificado, *choice* escolherá sempre o primeiro literal da lista.

O predicado *solvers* leva como argumento uma lista de hipóteses P e um ambiente com as variáveis de P que já são ligadas. Seja P = (¬H1 ¬H2 ¬H3). A variável LAR corresponde a uma lista cujo *car* são os literais à esquerda do literal escolhido em *choice* (representado pela variável A) e cujo *cdr* são os literais à direita do mesmo literal. Esse *car* e esse *cdr* são representados pelas variáveis L e R, respectivamente. Logo, como *choice* escolhe sempre o primeiro literal (no exemplo, A = ¬H1), L é sempre a lista vazia. A variável Rls representa as disjunções da base de dados que são possíveis candidatas a resolver o litoral escolhido. A resolução consiste na construção de uma nova hipótese (passo 6) e na eliminação citada no passo 7.

Em *(dolist (Rn Rls) ...)*, Rn sofrerá uma iteração através das disjunções de Rls. Na primeira iteração, será escolhida a primeira disjunção, na segunda, a segunda disjunção e assim por diante. Para cada valor de Rn, o ambiente de unificação de A com ¬A (em Rn) é armazenada em Un.

Seja P = (¬H1 ¬H2 ¬H3). Então, L = (), A = ¬H1 e R = (¬H2 ¬H3). Seja Rls = ((H1 B1 B2) (H1 C1 C2)). Inicialmente ¬H1 se unifica com H1 da primeira disjunção de Rls, produzindo o ambiente Un1. A função *res*, que executa a resolução, constrói a estrutura:

(Un1 (¬H2 ¬H3 B1 B2)),

onde

(¬H2 ¬H3 B1 B2)

representa a nova hipótese.

A nova hipótese e seu ambiente são empilhados em Rs. Na próxima iteração, Rn = (H1 C1 C2) é usada. Então, *res* produzirá:

(Un2 (¬H2 ¬H3 C1 C2)),

que também é empilhada em Rs. Neste ponto tem-se que:

Rs = ((Un1 (¬H2 ¬H3 B1 B2))
(Un2 (¬H2 ¬H3 C1 C2)))

O predicado "*proof*" apresenta dois argumentos: SOLVED, contendo as hipóteses já provadas e WAIT, contendo as hipóteses que aguardam para serem provadas. Uma hipótese é extraída de WAIT e posta em Y. Executa-se:

(solvers (cdr Y) (first Y)),

que produz todas as novas hipóteses (todas as possibilidades dos passos 6 e 7) derivadas a partir de Y. Mais de uma nova hipótese pode surgir porque há diversas

escolhas possíveis no passo 5. As novas hipóteses são armazenadas em R (ainda predicado "proof"), que tem a seguinte forma:

((Un1 (¬H2 ¬H3 B1 B2)) (Un2 (¬H2 ¬H3 C1 C2)) ...),

onde se notam duas novas hipóteses, isso é, *(¬H2 ¬H3 B1 B2) e (¬H2 ¬H3 C1 C2)* .

Para fins de completude da prova, cada nova hipótese de R é submetida à prova. Após provada, ela é empilhada em SOLVED. Caso ainda haja trabalho remanescente, ele é empilhado em WAIT. Neste caso, a prova é concluída quando a lista de novas hipóteses é vazia, o que se detecta através do predicado *complete* que representa o passo 8.

A partir de agora, será mostrada a atuação deste provador como avaliador parcial.

Na verdade, o processo de prova parcial segue praticamente a mesma lógica descrita acima para o avaliador total, diferindo apenas no que diz respeito aos passos 4 (escolha de uma hipótese) e 8 (detecção da conclusão da prova). Logo, tais diferenças devem estar previstas nos predicados *choose* e *complete*, respectivamente, conforme discutido a seguir.

Quando o predicado *choice* receber uma lista de hipóteses onde deve escolher um literal, ele não escolherá sempre o primeiro da lista, tal como descrito no avaliador total, mas o primeiro literal que ainda não corresponda a um predicado binário adequado à rede semântica (que, no caso particular da implementação efetuada, seja uma hipótese que comece com *ag*, *dat*, *elem* ou *acc*).

O predicado *complete* do avaliador parcial vai detectar conclusão de prova quando ocorrer uma das duas situações abaixo:

- A lista das novas hipóteses for vazia (tal como para o avaliador total);
- Todas os literais da nova hipótese forem predicados binários.

A seguir mostra-se um exemplo de avaliação parcial em que se tenta converter o predicado ternário "give" em uma rede semântica (conjunção de predicados binários) a partir da seguinte base de dados:

```
(defun give()
 (list (let ((x (nvar)) (y (nvar)) (z (nvar)) (ev (gensym)) )
        '( (give ,x ,y ,z)
           (person ,x) (person ,y)
           (elem ,ev, givings)
           (agent ,ev ,x) (dat ,ev ,y) (acc ,ev ,z)) )))

(defun person()
 (list (let ((x (nvar)))
        '( (person ,x) (elem ,x persons))  )))
```

Rastreamento da avaliação parcial:

```
(partial '(give john x y))

(proof () '(( () '(give john x y) )))
%  SOLVED = ()
%  WAIT = ((() 'give john x y)))

(pop WAIT)
%  Y = ( () '(give john x y) )
%  WAIT = ()

(solvers ((give john x y)) () )

(choice () ((give john x y)))
%  LAR =  ( () ((give john x y)) )
```

% RS = ()

% L = ()

% A = (give john x y)

% R = nil

% Rls = (((give 1 2 3) (person 1) (person 2) (elem G1 givings)
(agent G1 1) (dat G1 2) (acc G1 3)))

% A seguir analisa-se a primeira iteração pelo *dolist* de *solvers*:

% Rn = ((give 1 2 3) (person 1) (person 2) (elem G1 givings) (agent G1 1) (dat G1 2)
(acc G1 3))

% Un = ((1 . john) (x . 2) (y . 3))

% Rs = ((((1 . john) (x . 2) (y . 3))
(person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3)))

% Primeira iteração pelo *dolist* de *proof*:

% R = ((((1 . john) (x . 2) (y . 3))
(person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3)))

(complete (cdr R)) = nil

% WAIT = ((((1 . john) (x . 2) (y . 3))
(person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3)))

(pop WAIT)

% Y = (((1 . john) (x . 2) (y . 3))
(person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3))

% WAIT = nil

(solvers (((person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3))
((1 . john) (x . 2) (y . 3)))

(choice () ((person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3)))
% LAR = (() ((person john) (person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3)))
% L = nil
% A = (person john)
% R = ((person 2) (elem G1 givings)
(agent G1 john) (dat G1 2) (acc G1 3))
% Rls = (((person 4) (elem 4 persons)))

% Primeira iteração pelo *dolist* de *solvers*:

% Rn = ((person 4) (elem 4 person))
% Un = ((4 . john) (1 . john) (x . 2) (y . 3))
% Rs = ((((4 . john) (1 . john) (x . 2) (y . 3))
(elem john persons) (person 2)
(elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3)))

% Primeira iteração pelo *dolist* de *proof*:

% R = (((4 . john) (1 . john) (x . 2) (y . 3))
(elem john persons) (person 2)
(elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3))

(complete (cdr R)) = nil
% WAIT = ((((4 . john) (1 . john) (x . 2) (y . 3))
(elem john persons) (person 2)

```
            (elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3) ))

(pop WAIT)
%  Y =  ( ((4 . john) (1 . john) (x . 2) (y . 3))
            (elem john persons) (person 2)
            (elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3) )
%  WAIT = nil

(solvers ( '((elem john persons) (person 2)
            (elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3) )
            '((4 . john) (1 . john) (x . 2) (y . 3)))

(choice ( () ((elem john persons) (person 2)
            (elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3) ))
%  LAR = (( (elem john persons)) ((person 2)
                (elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3) ))
%  L = ((elem john persons))
%  A = (person 2)
%  Rls = (((person 5) (elem 5 persons)))
%  R = ((elem G1 givings) (agent G1 john) (dat G1 2) (acc G1 3))

%  Primeira iteração pelo dolist de solvers:

%  Rn = ((person 5) (elem 5 persons))
%  Un = ((2 . 5) (4 . john) (1 . john) (x . 2) (y . 3))
%  Rs = (( ((2 . 5) (4 . john) (1 . john) (x . 2) (y . 3))
            (elem john persons) (elem 5 persons) (elem G1 givings)
            (agent G1 john) (dat G1 5) (acc G1 3)))

%  Primeira iteração pelo dolist de proof:

%  R =  ( ((2 . 5) (4 . john) (1 . john) (x . 2) (y . 3))
```

(elem john persons) (elem 5 persons) (elem G1 givings)
(agent G1 john) (dat G1 5) (acc G1 3))

(complete (cdr R)) = true
% SOLVED = ((((2 . 5) (4 . john) (1 . john) (x . 2) (y . 3))
(elem john persons) (elem 5 persons) (elem G1 givings)
(agent G1 john) (dat G1 5) (acc G1 3)))

(mapcar (function (lambda (E) (deref(cdr E)(car E))))
'((((2 . 5) (4 . john) (1 . john) (x . 2) (y . 3))
(elem john persons) (elem 5 persons) (elem G1 givings)
(agent G1 john) (dat G1 5) (acc G1 3))))

(deref '((elem john persons) (elem 5 persons) (elem G1 givings)
(agent G1 john) (dat G1 5) (acc G1 3))
'((2 . 5) (4 . john) (1 . john) (x . 2) (y . 3))) =

= ((elem john persons) (elem 5 persons) (elem G1 givings)
(agent G1 john) (dat G1 5) (acc G1 3))

Logo, a avaliação parcial de:

(give john x y)

produz:

((elem john persons) (elem 5 persons) (elem G1 givings)
(agent G1 john) (dat G1 5) (acc G1 3))

que realmente representa uma rede semântica.

A seguir mostra-se uma base de dados que permite uma avaliação total:

```
(defun father ()
  '( ((father a b)) ((father c d)) ((father d e)) ))

(defun mother ()
  '( ((mother e f))))

(defun grand-father()
  (list (let ((x (nvar)) (y (nvar)) (z (nvar)))
          '( (grand-father ,x ,y)
               (father ,x ,z) (father ,z ,y)))
        (let ((x (nvar)) (y (nvar)) (z (nvar)))
          '( (grand-father ,x ,y)
             (father ,x ,z) (mother ,z ,y))) ))
```

A hipótese:

```
(query (grand-father x y))
```

desencadeará uma avaliação total. Mas como a seqüência de prova segue a mesma lógica da avaliação parcial rastreada há pouco, diferindo apenas na execução dos predicados *choose* e *complete*, ela não será rastreada aqui. Adiantaremos apenas que o resultado produzido para a avaliação total da hipótese *grand-father* é:

```
((grand-father d f) (grand-father c e))
```

Com esse exemplo mostramos como um provador de teoremas é capaz de atuar tanto como avaliador parcial, quanto como avaliador total, dependendo da completeza das informações das quais dispõe.

4.11. Avaliação Parcial = Generalização Baseada em Explicação ("Explanation Based Generalisation" , ou, tal como é conhecida, EBG)

Nesta secção são apresentados os resultados de Harmlen e Bundy [36] mostrando que, na realidade, as técnicas de Avaliação Parcial e de Generalização Baseada em Explicação (EBG) são equivalentes.

A técnica EBG foi concebida inicialmente por De Jong [19] e Mitchell mas só recebeu essa designação com Mitchell et al [36].

Apesar de as técnicas de Avaliação Parcial e de EBG [36, 42, 47] terem sido concebidas com diferentes propósitos, elas representam um mesmo algoritmo, conforme mostrado a seguir.

4.11.1. A EBG

A EBG [36] consiste na formulação de conceitos gerais tendo como base um exemplo de treinamento específico.

As entradas esperadas pelo algoritmo EBG são as seguintes:

- Um conceito meta: define um conceito a ser aprendido.
- Um exemplo de treinamento: representa uma instância do conceito meta.
- Teoria do domínio: conjunto de regras que explicam por que o exemplo de treinamento é uma instância do conceito meta.
- Critério de operacionalidade: especifica a forma final sob a qual o conceito a ser aprendido deve se apresentar.

A meta da EBG é produzir uma generalização (ou reformulação) do exemplo de treinamento que defina suficientemente o conceito meta e que satisfaça o critério de operacionalidade.

Conforme será visto, o exemplo de treinamento é inconveniente quando não se deseja restringir demais o conjunto de generalizações possíveis que representam o conceito meta.

4.11.2. Outras Considerações sobre a Avaliação Parcial

Já foi visto que o principal objetivo da Avaliação Parcial é avaliar o máximo possível a computação de um programa em função das informações (valores de entrada) disponíveis.

Teoricamente, a Avaliação Parcial se fundamenta no teorema S-M-N de Kleene. Tal teorema afirma o seguinte [36]:

Dada uma função computável f de n variáveis qualquer, $f = f(x_1, ..., x_n)$, *e dados k valores de entrada* ($k \leq n$) $a_1, ..., a_k$ *para* $x_1, ..., x_k$, *pode-se efetivamente computar uma nova função f' tal que:*

$f'(x_{k+1}, ..., x_n) = f(a_1, ..., a_k, x_{k+1}, ..., x_n)$.

A função *f'* é uma especialização de *f* que, para os valores específicos de entrada adotados, é mais facilmente computável que *f*.

No domínio da programação lógica a Avaliação Parcial pode ser ainda mais abrangente, pois permite que as variáveis de entrada sejam também instanciadas parcialmente, isso é, que sejam instanciadas com termos que contenham variáveis.

É interessante notar que *k* pode também valer zero (0). Neste caso, nenhuma entrada é especificada e a Avaliação Parcial produz como resultado um programa que, apesar de ser mais otimizado que o original, não atinge o grau máximo de especialização que seria possível caso se particularizassem outros valores de entrada.

4.11.3. EBG = Avaliação Parcial

A reformulação da EBG em termos de dedução lógica provida por Kedar-Cabelli e McCarty precisou suficientemente o formalismo EBG de tal sorte a permitir comparações entre EBG e Avaliação Parcial [36].

Conclui-se de tais comparações que:

- O conceito meta da EBG corresponde ao nome do programa a ser parcialmente avaliado.
- A teoria do domínio da EBG corresponde às cláusulas (regras) e aos fatos que constituem o programa a ser parcialmente avaliado.
- O critério de operacionalidade corresponde ao critério que a Avaliação Parcial usa para interromper a construção da árvore de prova.
- O exemplo de treinamento da EBG corresponde aos valores das variáveis de entrada x_1, ..., x_k. Conforme já foi dito, não é imprescindível que se instanciem variáveis de entrada, isso é, *k* pode ser igual a zero. Isso corresponde a eliminar o exemplo de treinamento da EBG.

4.11.4. A Nível de Implementação, EBG = Avaliação Parcial

Seja o algoritmo de Avaliação Parcial de programas PROLOG abaixo [36]:

```
peval(Leaf, Leaf) :- clause(Leaf, true).

peval((Goal1, Goal2), (Leaves1, Leaves2)) :-
        peval(Goal1, Leaves1),
        peval(Goal2, Leaves2).

peval(Goal, Leaves) :-
        clause(Goal, Clause),
        peval(Clause, Leaves).
```

O avaliador parcial *peval* transforma o programa que representa o predicado de seu primeiro argumento em um conjunto de cláusulas unitárias que lhe equivalham, isso é, ele retorna em seu segundo argumento, um conjunto formado pelas folhas da árvore de prova obtida pela simulação da execução daquele predicado. Nesse caso, a montagem da árvore de prova termina quando se chega a uma cláusula unitária. Logo, as cláusulas unitárias definem o critério de operacionalidade da EBG e são detectadas implicitamente pela linguagem PROLOG (não precisando entrar como argumentos).

A entrada correspondente ao programa a ser avaliado (correspondente à teoria do domínio da EBG) é obtida implicitamente através do predicado primitivo *clause* de PROLOG.

Considere-se agora o algoritmo de EBG abaixo:

```
ebg(Leaf, GenLeaf, GenLeaf) :-
        operational(Leaf), !,
        call(Leaf).

ebg((Goal1, Goal2), (GenGoal1, GenGoal2), (Leaves1, Leaves2)) :-
        ebg(Goal1, GenGoal1, Leaves1),
        ebg(Goal2, GenGoal2, Leaves2).

ebg(Goal, GenGoal, Leaves) :-
        clause(GenGoal, GenClause),
        copy((GenGoal :- GenClause), (Goal :- Clause)),
        ebg(Clause, GenClause, Leaves).
```

O exemplo de treinamento entra no primeiro argumento do predicado *ebg*; o conceito meta entra em seu segundo argumento (logo, o primeiro argumento deve ser uma instância do segundo). O terceiro argumento de *ebg* corresponde à generalização desejada para a definição do conceito meta. Tal argumento é formado pelo conjunto de folhas da árvore de prova do predicado meta.

Também nesse algoritmo de EBG os critérios de operacionalidade e a teoria do domínio são providos pela base de dados de PROLOG.

Podem-se detectar três pequenas diferenças entre esses algoritmos de Avaliação Parcial e de EBG:

(1) O algoritmo *ebg* usa um critério de operacionalidade explícito para interromper a construção dos ramos da árvore de prova, enquanto que *peval* só pára quando

encontra uma cláusula unitária. Logicamente, *peval* também pode ter um critério de operacionalidade explícito, bastando, para isso, alterar sua primeira cláusula para:

```
peval(Leaf, Leaf) :-
        operational(Leaf), !,
        call(Leaf).
```

(2) O predicado *peval* inclui, no conjunto que retorna, a unificação com os predicados operacionais, o que restringe ainda mais a reformulação do conceito meta. Contudo, o algoritmo *ebg* não inclui tal unificação na lista que retorna. Isso é conseguido com a introdução de um argumento extra em *ebg* que corresponde a seu segundo argumento. De fato, o primeiro e o segundo argumentos de *ebg* mantêm exatamente a mesma árvore de prova, exceto na primeira cláusula, onde o primeiro argumento inclui a unificação com os predicados operacionais e o segundo argumento não a inclui. O terceiro argumento da primeira cláusula é igual ao segundo argumento e corresponde ao conjunto final de folhas. Logo, excluir a unificação com os predicados operacionais das folhas da árvore de prova corresponde a efetuar a generalização desejada na técnica de EBG. Segundo Kedar-Cabelli e McCarty [6, pág.8]:

A generalização é formada propagando-se substituições de regra mas ignorando-se substituições de fatos quando da criação da árvore de prova.

O algoritmo *peval* pode ainda ser modificado de modo a eliminar esta segunda diferença com relação a *ebg*, isso é, de modo a não inclui na lista devolvida a unificação com os predicados operacionais. Para tanto, basta que se substitua sua primeira cláusula por:

```
peval(Leaf, Leaf) :-
        operational(Leaf), !,
        not not call(Leaf).
```

onde o duplo predicado "*not*" de PROLOG desfaz as eventuais ligações de variáveis que tenham sido efetuadas durante unificações envolvendo predicado operacionais.

(3) A terceira diferença entre *ebg* e *peval* é introduzida pelo predicado "*copy*" na terceira cláusula de *ebg*. Tal predicado garante que a árvore generalizada reflita exatamente a árvore específica fazendo *GenGoal :- GenClause* e *Goal :- Clause* serem cópias exatas uma da outra, com novas variáveis. Entretanto, como o exemplo de treinamento não é imprescindível no algoritmo *ebg*, não se precisa manter dois argumentos distintos de árvore de prova, um com a versão específica e o outro com a versão generalizada, bastando, pois, conservar o argumento da árvore generalizada. Com isso, pode-se eliminar o predicado "*copy*" do algoritmo *ebg*.

Efetuadas essas pequenas alterações superficiais visando a eliminar as diferenças entre *peval* e *ebg*, ambos se tornam algoritmos equivalentes.

4.11.5. Exemplo

Apresenta-se aqui um exemplo clássico proposto originalmente por Borgida et al. [Harmlen] visando a avaliar a possibilidade de empilhar um objeto sobre o outro em função do peso de ambos. Considere-se a seguinte base de dados:

```
% Conceito meta:

safe_to_stack(X, Y) :-
        lighter(X, Y).

% Fatos que expressam o conhecimento do mundo (de onde deve ser extraído o
exemplo
% de treinamento):

on(box1, table1).
volume(box1, 10).
```

```
isa(box1, box).
isa(table1, table).
colour(box1, red).
colour(table1, blue).
density(box1, 10).

% Teoria do Domínio:

lighter(X, Y) :-
        weight(X, W1),
        weight(Y, W2),
        smaller(W1, W2).


weight(X, 500) :- isa(X, table).

weight(X, Y) :-
        volume(X, V),
        density(X, D),
        times(V, D, Y).

% Critério de Operacionalidade:

operational(Goal) :-
        member(Goal, [times(_, _), smaller(_, _),
        on(_, _), volume(_, _),
        isa(_, _), colour(_, _),
        density(_, _)]).
```

Pode-se aplicar *peval* e *ebg* a este exemplo através das metas que se seguem, representadas, respectivamente, por:

?- peval(safe_to_stack(X, Y), Reformulation).

e

?- ebg(safe_to_stack(box1, table1), safe_to_stack(X, Y), Reformulation).

Na primeira meta, *safe_to_stack(X, Y)* representa o programa a ser parcialmente avaliado e *Reformulation* indica a resposta dessa avaliação parcial. Na segunda meta, *safe_to_stack(box1, table1)* representa o exemplo de treinamento, *safe_to_stack(X, Y)* representa o conceito meta e *Reformulation* representa a generalização obtida pela técnica EBG.

Ambas as metas produzem como resultado:

Reformulation = (volume(X, VX), density(X, DX),
times(VX, DX, MX), isa(Y, table),
smaller(MX, 500),

isso é, *qualquer coisa com peso menor que 500 pode ser empilhada em uma mesa.*

Neste exemplo, o algoritmo *ebg* foi guiado pelo exemplo de treinamento, ao passo que *peval* executa uma busca não guiada. Como conseqüência disso, *ebg* cosegue computar somente a reformulação "*Reformulation*" mostrada acima, enquanto que peval consegue retornar, além dessa, as duas outras reformulações mostradas a seguir:

Reformulation = (isa(X, table), volume(Y, VY), density (Y, DY).
times(VY, DY, MY), smaller(500, MY)),

que indica que *uma mesa pode ser empilhada sobre qualquer coisa com peso acima de 500,*

e:

Reformulation = (volume(X, VX), density(X, DX), times(VX, DX, MX),
volume(Y, VY), density(Y, DY),
times(VY, DY, MY), smaller(MX, MY)),

que indica que *qualquer coisa pode ser empilhada sobre outra mais pesada do que ela*.

Conforme se vê nesse exemplo, enquanto *peval* produz uma reformulação completa do conceito meta, *ebg*, guiada por um exemplo de treinamento, produz uma única reformulação cujas condições podem ser, em geral, excessivamente fortes, o que restringe demais a definição produzida para o conceito meta. Logo, ao se usar a técnica EBG, mesmo que a introdução do exemplo de treinamento reduza o espaço de busca, simplificando com isso o trabalho computacional, ela somente deve ser efetivada quando estamos interessados apenas em um caso bem particular de definição do conceito meta, pois tal introdução particulariza muito essa definição, impondo-lhe, inclusive, condições desnecessárias.

5. SISTEMA DE RECUPERAÇÃO DE INFORMAÇÃO IMPLEMENTADO

Introdução

Neste capítulo apresenta-se o avaliador parcial que efetua a recuperação da informação a partir da base de conhecimentos [14, 15, 25].

A resposta a uma pergunta consiste de uma avaliação parcial da pergunta com relação a base de conhecimentos, onde o conjunto selecionado para expressar a resposta é reconhecido pelo conjunto $Pp - (Dp \cap Pp)$ representado pelas expressões da Lógica Terminológica.

A avaliação parcial é efetuada por um provador de teoremas baseado em resolução linear.

Acreditamos que a nossa contribuição a nível dessa etapa do sistema consiste no método de inferência adotado pelo provador, conciliando os métodos tradicionais de inferência do Cálculo de Predicados com o da Lógica Terminológica (subsunção).

5.1. Idéias de Siwek e Wittgeinstein na definição de Avaliação Parcial

Conforme tem sido discutido, o principal problema deste sistema é relacionado com a representação. A meta é produzir uma representação das exigências na base de conhecimentos de tal modo a facilitar a tarefa do provador de teoremas de lidar com elas. Similarmente, escolhe-se uma representação da resposta que seja esclarecedora para a pessoa que apresente a pergunta ao sistema.

De acordo com Siwek [67], a representação ou significação é baseada na associação de dois fenômenos paralelos. Um dos fenômenos deve ser acessível à unidade processadora da informação e deve conduzi-la à cognição do outro. O paralelismo de ambos os fenômenos é necessário para efetivar essa condução.

Wittgenstein [76] define complexidade lógica como a propriedade de uma proposição de ser sucessivamente quebrada em outras proposições menos complexas que a original. No final desse processo de quebra, devem-se obter apenas proposições

elementares (sem complexidade lógica). De acordo com Wittgeinstein, sempre que duas proposições são logicamente relacionadas, deve haver alguma complexidade lógica em pelo menos uma delas.

A seguir se fará uma adaptação de tais idéias aos sistemas de representação de conhecimento.

Quando se interroga um sistema de conhecimento, a informação que se deseja obter pertence a um fenômento inacessível que se pretende compreender (cognição). A resposta final produzida pela avaliação da pergunta deve pertencer a um subconjunto da linguagem de representação que seja *acessível* a quem propôs a pergunta. Entende-se por *acessível* o fato de o indagador reconhecer os conceitos representado pelas proposições da resposta e de fazer inferências lógicas a partir de qualquer dessas proposições que apresentem complexidade lógica.

Considera-se um dado como uma forma sem complexidade lógica a partir da qual não se pode deduzir nada mais.

Uma avaliação total é caracterizada por produzir um dado como resposta.

Uma avaliação parcial é caracterizada por produzir uma resposta dotada de complexidade lógica, ou seja, uma resposta a partir da qual podem-se extrair outras deduções.

Conforme explicado na introdução desta tese, um sistema de representação do conhecimento [46] é formado por um conjunto Fs de fórmulas e por um conjunto de regras de inferência. Há dois subconjuntos recursivos de Fs, D e P, tal que Fs = D $\cup$ P. Todas as fórmulas pertencentes a P são axiomas. Todas as fórmulas pertencentes a D são dados. O conjunto de regras de inferência é chamado de semântica do sistema de representação do conhecimento. Dentre as regras de inferência, há uma que especifica como e quando as outras regras de inferência serão usadas. Esta regra especial é denominada regra computacional.

Suponha-se que D e P sejam descritos por um grupo de regras sintáticas (isso é sempre possível porque D e P são conjuntos recursivos). Sejam Dp e Pp os conjuntos de regras de produção que descrevem D e P, respectivamente. Uma fórmula tem componentes somente em D se ela é descrita pelo conjunto Dp - (Dp $\cap$ Pp). Uma fórmula tem componentes somente em P se ela é descrita pelo conjunto Pp - (Dp $\cap$ Pp).

Considere-se o sistema de representação de conhecimento krs(D, P, semântica), onde *semântica* corresponde às *regras de inferência* do sistema. Se uma inferência f tem componentes somente em P, ela é chamada de *parcial*.

Chama-se inferência a qualquer dedução feita a partir do dado de entrada denominado *pergunta*.

O último nível de uma inferência é denominado *resposta*. Se a resposta é uma fórmula que pertença ao conjunto Dp - (Dp $\cap$ Pp), ela é chamada total. Se a resposta é uma fórmula pertencente ao conjunto Pp - (Dp $\cap$ Pp), ela é chamada parcial.

Geralmente, respostas parciais são usadas para aumentar a eficiência de futuras inferências totais, provendo um ponto de partida avançado. Aliás, o ganho de eficiência é o fator sobre o qual se apóiam teorias tais como Avaliação Parcial e Generalização Baseada em Explicação ("Explanation Based Generalisation").

Neste sistema, contudo, utilizamos a técnica da Avaliação Parcial para produzirmos respostas conceitualmente utilizáveis. Isso significa que a fórmulas da resposta deve introduzir novos conceitos ou identificar antigos conceitos. Seja Sp o conjunto de produções que descrevem a fórmula conceitualmente utilizável da resposta. Como essa fórmula deve ser obtida por avaliação parcial, Sp deve ser subconjunto de Pp - (Dp $\cap$ Pp). Uma resposta pertencente a Sp deve ser clara e esclarecedora, isso é, ela deve representar conceitos em uma forma bem nítida e deve prover um sistema formal que permita à pessoa que propõe a pergunta fazer inferências sem maiores dificuldades. Baseando-se em tais características desejadas para Sp, conclui-se que a Lógica Terminólogica é bem apropriada para expressar as respostas. E é por isso que ela foi adotada como formalismo representante de Sp.

Recorde-se que as exigências são representadas na base de conhecimentos como fórmulas do Cálculo de Predicados (reconhecidas por Pp). A associação entre o fenômeno acessível do Cálculo de Predicados com os fenômenos originalmente inacessíveis da base de conhecimentos nos permite obter a cognição almejada. Em outras palavras, a resposta expressa cognição à respeito do mundo descrito na base de conhecimentos.

5.2. Unificação Semântica do Sistema

As anotações das variáveis são expressões da Lógica Terminológica que podem ser consideradas como predicados. Os predicados das anotações devem ser verdadeiros para as variáveis que são anotadas por eles. Nesse caso, os resultados provindos das provas das fórmulas F1 e F2 abaixo devem ser os mesmos:

F1: τ(X1, ioi-gs(X1),
τ(X2, τ(X3, space-vehicle(X3),
computer(X2) &
on-board(X2, X3)) &
control(X1, X2)))

F2: control(X1/τ(ioi-gs), X2/τ(computer & on-board(space-vehicle)))

De fato, avaliar a forma:

r(X1/c1, X2/c2, X3/c3, ...) & ...

assegurando-se que:

(⌈c1 ⌋ X1), (⌈c2 ⌋ X2), (⌈c3 ⌋ X3) ... & ...

são todos verdadeiros, é o mesmo que avaliar a forma:

r(X1, X2, X3, ...) & c1(X1) & c2(X2) & c3(X3) & ...

Contudo, pode-se avaliar com mais eficiência a primeira forma, pois ela exige menos recursos do provador de teoremas do Cálculo de Predicados.

O provador de teoremas atua coletando conceitos relacionados a cada variável. Os conceitos coletados restringirão a variável de resposta e esta restrição, em si mesma, será a resposta à pergunta.

Por exemplo, se alguém pergunta *What does the ioi-gs control*, isso é:

?- control(X/ioi-gs, Y/C)

e o sistema responde:

Y/computer & on-board(space-vehicle),

obtém-se a informação de que o ioi-gs controla os computadores a bordo do veículo espacial. Mesmo que não se obtenha a lista dos computadores controlados pelo ioi-gs, tal resposta informa que eles devem satisfazer os seguintes predicados:

λ(X) computer(X)

e

λ(Y) on-board(Y, space-vehicle).

A coleta de conceitos é feita durante a unificação. A idéia é que, caso uma variável X/cx se unifique com uma variável Y/cy, a unificação entre X e Y se processe tal como a unificação clássica. Isso significa que todos os resultados da unificação clássica serão válidos no nosso algoritmo de unificação semântica (complexidade, correção etc). O processo de armazenagem dos conceitos *cx* e *cy* ocorre durante a unificação. Tal processo é conduzido de modo a permitir que no fim da prova se avalie se a variável X pode ser anotada pelo conceito *cx & cy*. Em outras palavras, X, após a unificação, sofrerá tanto as restrições da variável Y quanto as suas próprias. De fato, como:

r1(..., X/cx, ...)

é equivalente a:

r1(..., X, ...) & (⌈ cx ⌉ X)

e como:

r1(..., Y/cy, ...)

é equivalente a:

r1(..., Y, ...) & (⌈ cy ⌉ Y),

após a resolução e unificação, teremos:

X = Y & (⌈ cx ⌉ X) & (⌈ cy ⌉ Y)

Conseqüentemente, pela semântica de composição de dois conceitos, X se submeterá à restrição *cx & cy*. Obviamente, deve-se analisar a possibilidade de simplificação da fórmula resultante *cx & cy*. Nesse processo de simplificação, por exemplo, se:
cx = c1x & c2x & c3x

e se *c2x* subsume *cy*, então pode-se adotar a fórmula:

c1x & c3x & cy

no lugar da fórmula:

c1x & c2x & c3x & cy.

A seguir se apresentam algumas definições que devem ser conhecidas antes de se introduzir o algoritmo de unificação. Conforme veremos, a apresentação do algoritmo de unificação será feita de modo estreitamente próximo ao adotado por Robinson [61].

(D1) *Entidade* é uma estrutura de dados denotada por X/c cujos seletores são "*identifier*" e "*concept*", conforme mostrado a seguir:

identifier(X/c) $\Rightarrow$ X

concept(X/c) $\Rightarrow$ c

sendo que os identificadores ("identifiers") podem ser variáveis ou constantes.

(D2) "*Binding*" é uma estrutura de dados denotada por X/cx = Y/cy, onde X é uma variável. Seus seletores são *car* e *cdr*, de acordo com o que se segue:

car(X/cx = Y/cy) $\Rightarrow$ X/cx

cdr(X/cx = Y/cy) $\Rightarrow$ Y/cy

(D3) *Ambiente* é um conjunto de "bindings" cujas variáveis do *car* são diferentes umas das outras. Dada uma variável X e um ambiente E, a função:

(assoc X E)

retorna o "binding" cuja variável do *car* é X.

(D4) *Valor imediato* (vi) da variável X no ambiente E é definido como:

vi $\equiv \lambda X\, \lambda E$ cdr(assoc(X, E))

(D5) *Composição* ou *combinação* de dois conceitos *c1* e *c2* é uma *simplificação* de *c1 & c2*. Os predicados da Lógica Terminológica que são usados como anotações das variáveis representam as *restrições* ("constraints") do sistema. A composição das anotações é feita segundo a semântica da Lógica Terminológica, isso é, segundo noções tais como "subsunção", disjunção etc que verificam se as restrições são satisfeitas.

(D6) Se:

cl $\equiv$ cl1 & cm1 & cr1

e se:

cm1 subsume c2,

então:

cl1 & cr1 & c2

é uma simplificação de

c1 & c2.

Uma outra simplificação é a própria expressão *c1 & c2.*

(D7) *Extensão* de *c* é o conjunto:

$\{X \mid \lceil c \rceil (X)\}$

e é escrita como c^e.

(D8) Um conceito *c* subsume um conceito *d* se:

$c^e \supseteq d^e$.

Com essa definição de subsunção se prova que uma simplificação de *c1 & c2* é equivalente ao próprio *c1 & c2.*

(D9) O seguinte algoritmo de subsunção é *correto* mas não *completo*:

```
subsumes ≡ ( Y λ sub λ C1 λ C2
  (a) if C1 = X & Z then
        sub(X, C2) ∧ sub(Z, C2)
      else
        (b) if C2 = U & W then
              sub(C1, U) ∨ sub(C1, W)
            else
              (c) if C1 = R(D1) ∧ C2 = R(D2) then
                    sub(D1, D2)
                  else
                    (d) if C1 = R(D1) ∧ C2 = U & W then
                          sub(C1, U) ∨ sub(C1, W)
                        else false)
```

Saliente-se que Y é o combinador de ponto fixo (veja secção 4.1). É relativamente simples de se provar que este algoritmo é correto mas não completo. A prova de que ele pára decorre de indução sobre a extensão das expressões terminológicas. Por outro lado, verifica-se com facilidade que os itens (a), (b), (c) e (d) satisfazem a definição de subsunção.

Pode-se constatar que esse algoritmo não é completo, uma vez que ele indevidamente retorna *falso* em algumas situações em que C1 subsume C2. Exemplo disso é a situação em que C2 = U & W e C1 subsume C2, sem, contudo, subsumir nem U e nem W.

(D10) *Valor Final* de uma *entidade* X é definido por:

```
fvalue ≡ (Y λ  fv λ X λ E
  let V/CV = assoc(identifier(X), E) in
  if V/CV = ⊥ then X
  else
    fv(V/combine(concept(X), CV), E),
```

onde $\perp$ é o operador "*bottom*" (veja secção 4.2).

Este algoritmo deve produzir uma expressão VX/CX como *valor final* de X, onde VX é o *valor final* tal como proposto por Robinson e CX é uma *composição* de conceitos

(D11) Sejam G e H dois literais "flat", isso é, literais sem argumentos funcionais (fazemos esta restrição porque precisamos apenas desse tipo de literal. Contudo, não haveria dificuldades em se estender as idéias apresentadas aqui de modo a incluir literais contendo funções como argumento).

A *unificação* de G e H no ambiente E consiste em se encontrar a extensão mais geral E' de tal modo que G' = H', onde G' e H' correspondem ao "dereferenciamento" de G e H em E. Para "dereferenciar" um literal "flat", precisa-se somente substituir cada um de seus argumentos por seu respectivo valor final.

Deseja-se um algoritimo de unificação que armazene no ambiente todos os conceitos que anotam as variáveis dos literais que se tenta unificar. Os conceitos precisam estar organizados no ambiente de tal modo a permitir que o algoritmo do valor final de um provador de teoremas clássico consiga coletá-los. Através disso, assegura-se que nosso provador de teoremas final é um avaliador parcial de uma hipótese incompleta ("*query*" ou *pergunta*) com respeito à base de conhecimentos. O avaliador parcial responde perguntas coletando no ambiente construído a partir da pergunta e da base de conhecimentos todos os conceitos que representam restrições à variável da hipótese. Conforme já foi dito, tais conceitos pertencem ao conjunto:

$\{Pp - (Dp \cap Pp)\}$.

Em outras palavras, responder perguntas, neste trabalho, consiste em especializar o máximo possível a hipótese incompleta (pergunta) a partir de informações armazenadas na base de conhecimento.

A seguir mostra-se o algoritmo de unificação:

```
unify ≡ λ G λ H λ E
  if E = ⊥ then E
  else
    if  G = ¬Gp ∧ H = ¬Hp ∧ functor(Gp) =
        functor(Hp) ∧ arity(Gp) = arity(Hp) then
         unifyargs(Gp, Hp, 1, E)
    else
         if functor(G) = functor(H) ∧ arity(G) =
            arity(H) then
              unifyargs(G, H, 1, E)
         else ⊥


unifyargs  ≡ (Y λ uniargs λ G λ H λ I λ E
  if I > last_argument_of(G) then
   E
  else
   if E = ⊥ then
    ⊥
   else
    uniargs(G,H, I+1, unifyarg(fvalue(argument I of G, E), fvalue(argument I of H,
          E),E)))

unifyarg ≡  λ X  λ Z λ E
  if X =  ⊥ or  Z =  ⊥  then  ⊥
  else
   let VX/CX = X and W/CW= Z in
   if occurs(X,Z) then
   ⊥
   else
    if variablep(VX) then
     [X = Z|E]
```

else

if variablep(W) then

[Z = X | E]

else ...

O predicado *unify* tenta unificar duas funções G e H a partir do ambiente E. Podem-se antecipar duas condições necessárias ao sucesso da unificação: G e H devem ter um mesmo functor e um mesmo número N de argumentos. Verificadas tais condições, *unify* chama a execução de *unifyargs*.

Unifyargs, recursivamente, vai chamar *unifyarg* para cada par formado pelo valor final em E do i-ésimo argumento de G com o valor final do i-ésimo argumento de H, sendo que $1 \leq i \leq N$.

Unifyarg recebe de *unifyargs* um par de valores finais (o primeiro elemento do par é recebido em seu argumento X e o segundo elemento do par em seu argumento Z) e o ambiente E em seu terceiro argumento.

Caso o car de X seja uma variável, *unifyarg* insere o "binding" X = Z em E. Caso o car de Z seja uma variável, *unifyarg* insere o "binding" Z = X em E.

Conforme se pode observar, exceto pelas anotações, este algoritmo não é diferente do clássico algoritmo de unificação que foi provado ser correto. Sendo assim, temos que provar apenas que as anotações são coletadas corretamente. Pode-se fazer isso examinando-se a seqüência semântica que é construída durante a avaliação parcial. Varrendo-se esta seqüência e examinando-se a unificação das variáveis, nota-se que a unificação realmente constrói uma corrente de "bindings" que o algoritmo do valor final é capaz de seguir (veja seção 5.4).

De acordo com o que já foi mencionado, tanto a hipótese (pergunta) quanto as exigências são transformadas em uma conjunção de literais do tipo:

[P, r(X1/C1, X2/C2, ...), Q, ...]

que é equivalente a:

[P, r(X1, X2, ...), C1(X), C2(X), ..., Q].

A seqüência de resoluções executadas durante a prova apagará todos os literais da hipótese conservando, contudo, as anotações. Tais anotações constituirão o resultado da avaliação parcial.

No exemplo da secção 5.4 pode-se verificar a atuação da unificação como elemento capaz de efetuar a rastreabilidade entre os elementos da pergunta e da base de conhecimentos (item *d* da seção 1.1).

Na próxima seção todas essas idéias serão consolidadas através de exemplos de prova.

5.3. Algoritmo de Prova

Nesta secção introduz-se um esboço da estrutura geral do algoritmo de prova implementado. Não se introduzirão aqui detalhes ligados ao processo de prova. Tais detalhes serão vistos, preferencialmente, na próxima secção.

O provador é baseado na resolução linear (veja capítulo 4). Abaixo esboçam-se os principais predicados ligados à prova:

```
cprove(G, A) :- on(G, A).
cprove(G, A) :- sent(G, B),
                neg(G, NG),
                not(on(NG, A)),
                cpro(B, [NG|A]).

cpro([], _).
cpro([G|GS], A) :- neg(G, NG),
                   cprove(NG, A),
                   cpro(GS, A).

sent(G, T) :- any(Y), contra-posit(Y, G1, T), unify(G, G1).
```

Resumidamente, o predicado *cprove* tenta provar uma certa meta. Para tanto, *cprove* deve receber dois argumentos: a meta G a ser provada e uma lista A contendo a negação das metas já resolvidas no momento em que se pede a prova de G. A meta G estará provada caso um dos casos abaixo se verifique:

- se G for membro da lista A;

- se a negação de G não pertencer à lista A e se a resolvente resultante da unificação de G com uma das cláusulas da base de conhecimento que apresente o literal ¬G entre suas disjunções for o conjunto vazio.

O predicado *sent* busca na base de conhecimentos uma regra (cláusula) apropriada à resolução da meta G a ser provada e, através do predicado *contra-posit* tenta encontrar um literal G1 desta regra que se unifique com G (predicado *unify*). O predicado *sent* recebe em seu primeiro argumento a meta G e retorna em seu segundo argumento a lista correspondente ao resolvente da unificação efetuada.

O predicado *cpro* recebe a lista de metas a serem provadas e submete cada uma delas a cprove. A prova termina quando a lista de metas é vazia. O segundo argumento de *cpro* representa a lista A das metas já resolvidas.

A seguir dá-se um exemplo resumido do processo de prova desencadeado pela seguinte meta:

?- h(Z).

proposta diante da seguinte base de conhecimentos:

any([v(wang), m(wang)]).
any([h(X), ¬m(X)]).
any([h(Y, ¬v(Y)]).

onde os predicados *v*, *m* e *h* representam *homem*, *mulher* e *humano*, respectivamente. Obviamente, as vírgulas entre os literais das regras representam disjunções.

A seqüência de prova é:

?- cprove(h(Z), []).
% A = []

?- sent(h(Z), [¬m(Z)]).

?- neg(h(Z), ¬h(Z)).

?- cpro([¬m(Z), [¬h(Z)]).
% A = [¬h(Z)]

?- neg(¬m(Z), m(Z)).

?- cprove(m(Z), [¬h(Z)]).

?- sent(m(wang), [v(wang)]).
% A = [¬h(wang)]

?- neg(m(wang), ¬m(wang)).

?- cpro([v(wang)], [¬m(wang), ¬h(wang)]).

?- neg(v(wang), ¬v(wang)).

?- cprove(¬v(wang), [¬m(wang), ¬h(wang)]).
% A = [¬m(wang), ¬h(wang)]

?- sent(¬v(wang), [h(wang)]).

?- neg(¬v(wang), v(wang)).

?- cpro([h(wang)], [v(wang), ¬m(wang), ¬h(wang)]).
% A = [v(wang), ¬m(wang), ¬h(wang)]

?- neg(h(wang), ¬h(wang)).

?- cprove(¬h(wang), [v(wang), ¬m(wang), ¬h(wang)]).

?- on(¬h(wang), [v(wang), ¬m(wang), ¬h(wang)]).
% A prova termina aqui, pois a meta *¬h(wang)* pertence à lista A.

?- cpro([], _).
% A lista de metas está vazia. Logo, termina a prova da meta *h(Z)*, onde
% *Z* = *wang*, isso é, o problema encontrou *wang* como solução da meta
% humano.

Na próxima secção será mostrado um rastreamento detalhado do mecanismo de prova onde se inclui o processo de unificação e o processo de recuperação do valor final das variáveis no ambiente formado durante as unificações.

5.4. Exemplo Detalhado do Processo de Avaliação Parcial

Para melhor ilustrar o processo de recuperação de informação, faz-se nesta secção o rastreamento de um exemplo de prova. A atuação dos critérios de subsunção durante a unificação semântica efetuada pelo provador está embutida no predicado *combine* a ser visto durante o rastreamento. Tal predicado visa a verificar a possibilidade e a melhor maneira de se combinarem em conjunções os conceitos que anotam as variáveis envolvidas na unificação. *Combine* é implementado de tal maneira a considerar os critérios de simplificação discutidos na secção 5.2.

Suponha-se que a base de conhecimentos B armazene a descrição:

any([controls(e0/ev, X1/ioi-gs, Y1/space-vehicle)]).
any([ag(e0/ev, X1'/ioi-gs)]).
any([pac(eo/ev, Y1'/space-vehicle).

correspondente à exigência:

The ioi-gs shall control the space-vehicle.

Já foi dito que o provador de teoremas, ao tentar responder uma pergunta ("query"), efetua uma avaliação parcial da pergunta com relação à base de conhecimentos. Para tanto, ambas precisam estar representadas sob uma mesma forma. Conseqüentemente, além das exigências, também as pergunta devem ser submetidas à análise sintática/semântica de tal maneira a se transformarem em descrições.

Suponha-se que a seguinte pergunta seja apresentada ao sistema cuja base de conhecimentos seja a base B proposta há pouco:

Who controls the space vehicle?

Esta pergunta, após analisada sintática e semanticamente, transforma-se na seguinte descrição:

[control(e0/ev, X/who, Y/space-vehicle)] &
[ag(e0/ev, X/ioi-gs)] &
[pac(eo/ev, Y/space-vehicle)].

A seguir se rastreia o processo de resolução linear seguido durante a prova. Recorde-se que A é a lista que contém a negação das metas submetidas a *cprove*. Obviamente, a lista inicial *[G|GS]* de metas submetida a cpro é formada pela negação

de cada literal da descrição que representa a pergunta. Então, no caso desse exemplo [G|GS]:

[[¬control(e0/ev, X/who, Y/space-vehicle)],
[¬ag(e0/ev, X/ioi-gs)],
[¬pac(eo/ev, Y/space-vehicle)]].

A seguir exibe-se o rastreamento da prova:

?- cpro([¬[control(e0/ev, X/who, Y/space-vehicle)],
¬[ag(eo/ev, X/who],
¬[pac(eo/ev, Y/space-vehicle)]],
[], [], E).

?- cprove(control(eo/ev, X/who, Y/space-vehicle), [], [], E).
% A meta *control(e0/ev, X/who, Y/space-vehicle)* não está incluída na
% lista A, que, no caso, é a lista vazia. Logo, a prova continua.

?- sent(control(eo/ev, X/who, Y/space-vehicle), B, [], E2).
% B corresponde ao corpo da regra escolhida na base de conhecimentos.

?- any(Y).
% Y = control(eo/ev, X1/ioi-gs, Y1/space-vehicle), que é a regra
% escolhida na base de conhecimentos.

?- contra-posit(control(eo/ev, X1/ioi-gs, Y1/space-vehicle), H1, B)
% H1 = control(eo/ev, X1/ioi-gs, Y1/space-vehicle)
% B = [], que é a cauda da regra escolhida.

?- unify(control(e0/ev, X/who, Y/space-vehicle),
control(eo/ev, X1/ioi-gs, Y1/space-vehicle), [], E2).

```
?- functor(control(eo/ev, X/who, Y/space-vehicle), Fn, N).
% Fn = control
% N = 3

?- functor(control(e0/ev, X1/ioi-gs, Y1/space-vehicle), control, 3).

?- unifyargs(control(eo/ev, X/who, Y/space-vehicle,
            control(eo/ev, X1/ioi-gs, Y1/space-vehicle), 1, 3, [], E2).

?- arg(1, control(eo/ev, X/who, Y/space-vehicle), P).
%  P = eo/ev

?- fvalue(eo/ev, [], Z).
%  Z = eo/ev

?- arg(1, control(e0/ev, X1/ioi-gs, Y1/space-vehicle), T)
%  T = e0/ev

?- fvalue(e0/ev, [], B).
%  B = eo/ev

?- unifyarg(e0/ev, e0/ev, [], E3)
%  E3 = []

?-unifyargs(control(e0/ev, X/ioi-gs, Y/space-vehicle),
            control(e0/ev, X1/ioi-gs, Y1/space-vehicle), 2, 3, [], E2).

?- arg(2, control(e0/ev, X/who, Y/space-vehicle), P1).
%  P1 = X/who

?- fvalue(X/who, [], Z1).
%  Z1 = X/who
```

```
?- arg(2, control(e0/ev, X1/ioi-gs, Y1/space-vehicle), T1).
% T1 = X1/ioi-gs

?- fvalue(X1/ioi-gs, [], B1)
B1 = X1/ioi-gs

?- unifyarg(X/who, X1/ioi-gs, [], E4).
% E4 = [X/who = X1/ioi-gs]

?- unifyargs(control(e0/ev, X/who, Y/space-vehicle),
        control(e0/ev, X1/ioi-gs, Y1/space-vehicle), 3, 3,
        [X/who = X1/ioi-gs], E2).

?- arg(3, control(e0/ev, X/who, Y/space-vehicle), P2).
% P2 = Y/space-vehicle

?- fvalue(Y/space-vehicle, [X/who = X1/ioi-gs], Z2)
% Z2 = Y/space-vehicle

?- arg(3, control(e0/ev, X1/ioi-gs, Y1/space-vehicle), T2).
% T2 = Y1/space-vehicle

?- fvalue(Y1/space-vehicle, [X/who = X1/ioi-gs], B2).
% B2 = Y1/space-vehicle

?- unifyarg(Y/space-vehicle, Y1/space-vehicle, [X/who = X1/ioi-gs], E5).
% E5 = [Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]
?- unifyargs(control(e0/ev, X/ioi-gs, Y/space-vehicle),
            control(e0/ev, X1/ioi-gs, Y1/space-vehicle), 4, 3,
            [Y/space-vehicle = Y1/space-vehicle, X/who = X1/ ioi-gs],
            E2).
% E2 = [Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]
```

% Inclui-se a negação de *control(e0/ev, X/who, Y/space-vehicle)* na lista
% A e pede-se a prova da cauda B da regra escolhida (no caso, B = []).
% A = [¬control(e0/ev, X/who, Y/space-vehicle)]

?- cpro([], [¬control(e0/ev, X/who, Y/space-vehicle)],
[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E0).
% E0 = [Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- cpro([¬ag(e0/ev, X/who), ¬pac(e0/ev, Y/space-vehicle)], [],
[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E).

?- cprove(ag(e0/ev, X/who), [],
[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E6).
% A meta *ag(e0/ev, X/who)* não é membro da lista A, que, no caso, é a
% lista vazia. Logo, a prova continua.

?- unify(ag(e0/ev, X/who), ag(e0/ev, X1'/ioi-gs),
[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E7).

% É chamada a meta unifyargs para os argumentos e0/ev, o que não
% causa alteração alguma no ambiente.

?- unifyargs(ag(e0/ev, X/who), ag(e0/ev, X1'/ioi-gs), 2, 2,
[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
E7).

?- fvalue(X/who,
[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], Z3).

?- assoc(X, [Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
V/CV)
% V = X1

% CV = ioi-gs

?- combine(who, ioi-gs, C).

% C = who & ioi-gs

?- fvalue(X1/who& ioi-gs,

[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], Z3).

% Z3 = X1/who & ioi-gs

?- fvalue(X1'/ioi-gs,

[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], B3).

% B3 = X1'/ioi-gs

?- unifyarg(X1/who & ioi-gs, X1'/ioi-gs,

[Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E8).

% E8 = [X1/who & ioi-gs = X1'/ioi-gs,

Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- unifyargs(ag(e0/ev, X/who), ag(e0/ev, X1'/ioi-gs), 3, 2,

[X1/who & ioi-gs = X1'/ioi-gs,

Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],

E7).

% E7 = [X1/who & ioi-gs = X1'/ioi-gs,

Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- not(on(ag(e0/ev, X/who), []).

?- cpro([], [¬ag(e0/ev, X/who)], E7, E6).

% E6 = E7 = [X1/who & ioi-gs = X1'/ioi-gs,

Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- cpro([¬pac(e0/ev, Y/space-vehicle)], [],

```
        [X1/who & ioi-gs = X1'/ioi-gs,
            Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E).

?- cprove(pac(e0/ev, Y/space-vehicle), [],
          [X1/who & ioi-gs = X1'/ioi-gs,
           Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
           E9).

?- unify(pac(e0/ev, Y/space-vehiocle), pac(e0/ev, Y1'/space-vehicle),
          [X1/who & ioi-gs = X1'/ioi-gs,
           Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
           E10).


?- unifyarg(e0/ev, e0/ev,
            [X1/who & ioi-gs = X1'/ioi-gs,
             Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
             E11).
%  E11 = [X1/who & ioi-gs = X1'/ioi-gs,
             Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- unifyargs(pac(e0/ev, Y/space-vehicle), pac(e0/ev, Y1'/space-vehicle),
             2, 2,
             [X1/who & ioi-gs = X1'/ioi-gs,
              Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
              E10).

?- fvalue(Y/space-vehicle,
          [X1/who & ioi-gs = X1'/ioi-gs,
           Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
           Z4).
```

```
?- assoc(Y, [X1/who & ioi-gs = X1'/ioi-gs,
        Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
        V1/CV1).
%  V1 = Y1
%  CV1 = space-vehicle
?- combine(space-vehicle, space-vehicle, C1).
%  C1 = space-vehicle

?- fvalue(Y1/space-vehicle, [X1/who & ioi-gs = X1'/ioi-gs,
        Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
        Z4).
%   Z4 =  Y1/space-vehicle

?- fvalue(Y1'/space-vehicle, [X1/who & ioi-gs = X1'/ioi-gs,
        Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
         B4).
%  B4 = Y1'/space-vehicle

?- unifyarg(Y1/space-vehicle, Y1'/space-vehicle,
            [X1/who & ioi-gs = X1'/ioi-gs,
             Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
             E12).
%  E12 = [Y1/space-vehicle = Y1'/space-vehicle,
          X1/who & ioi-gs = X1'/ioi-gs,
          Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- unifyargs(pac(e0/ev, Y/space-vehicle), pac(e0/ev, Y1'/space-vehicle),
             3, 2,
             [Y1/space-vehicle = Y1'/space-vehicle,
              X1/who & ioi-gs = X1'/ioi-gs,
              Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
```

E10).

% E10 = [Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- not(on(pac(e0/ev, Y/space-vehicle), [])).

?- cpro([], [¬pac(e0/ev, Y/space-vehicle)],
[Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs], E9).

% E9 = [Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

?- cpro([], [], [Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs],
E).

% E = [Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle, X/who = X1/ioi-gs]

% E corresponde ao ambiente final produzido pela prova.

% A resposta final da prova é obtida extraindo-se o valor final da
% variável X da pergunta no ambiente final E.

?- fvalue(X/who, [Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle,
X/who = X1/ioi-gs], T1/W1).

```
?- assoc(X, [Y1/space-vehicle = Y1'/space-vehicle,
             X1/who & ioi-gs = X1'/ioi-gs,
             Y/space-vehicle = Y1/space-vehicle,
             X/who = X1/ioi-gs], V2/CV2).
% V2 = X1
% CV2 = ioi-gs

?- combine(who, ioi-gs, C2).
% C2 = who & ioi-gs

?- fvalue(X1/who & ioi-gs, [Y1/space-vehicle = Y1'/space-vehicle,
                            X1/who & ioi-gs = X1'/ioi-gs,
                            Y/space-vehicle = Y1/space-vehicle,
                            X/who = X1/ioi-gs], T1/W1).

?- assoc(X1, [Y1/space-vehicle = Y1'/space-vehicle,
              X1/who & ioi-gs = X1'/ioi-gs,
              Y/space-vehicle = Y1/space-vehicle,
              X/who = X1/ioi-gs], V3/CV3).
% V3 = X1'
% CV3 = ioi-gs

?- combine(who & ioi-gs, ioi-gs, C3).
% C3 = who & ioi-gs

?- fvalue(X1'/who & ioi-gs, [Y1/space-vehicle = Y1'/space-vehicle,
                             X1/who & ioi-gs = X1'/ioi-gs,
                             Y/space-vehicle = Y1/space-vehicle,
                             X/who = X1/ioi-gs], T1/W1).
% T1 = X1'
% W1 = who & ioi-gs
```

Este exemplo de prova mostra que o processo de recuperação de informação do sistema realmente concilia os métodos clássicos de inferência do Cálculo de Predicados com o método de inferência da Lógica Terminológica (subsunção). Esta conciliação é efetuada por um algoritmo de unificação que estende a unificação clássica adotando critérios semânticos de subsunção.

A unificação de variáveis é efetuada sob a ótica de critérios de subsunção que são aplicados às expressões da Lógica Terminológica que as anotam. Tais critérios dotam o processo de unificação de um caráter semântico que simplifica o traço de prova, uma vez que permite a eliminação de alguns ramos da árvore de prova (por exemplo, o algoritmo descarta rapidamente ramos onde se tenta unificar variáveis anotadas por conceitos disjuntos).

As ligações produzidas pelas sucessivas unificações que ocorrem durante um processo de prova são estocadas em um ambiente. A resposta a uma pergunta corresponde ao valor final da variável da pergunta (no caso do exemplo, a variável X) no ambiente final produzido pelas unificações. Tal resposta é representada pelas expressões da Lógica Terminológica que anotam a variável da pergunta. Logo, a resposta corresponde a uma avaliação parcial, isto é, ela pertence ao conjunto Pp - $\{Dp \wedge Pp\}$ que, no nosso caso, é constituído por expressões da Lógica Terminológica (tal qual a expressão who & ioi-gs que representa a resposta do exemplo de prova que acabamos de descrever).

6. CONCLUSÃO

Apresentamos neste trabalho um Sistema Híbrido de Representação de Conhecimento apropriado para lidar com contextos em que o conhecimento é expresso através de um subconjunto particular da linguagem natural. Neste subconjunto, o conhecimento é representado por frases sintaticamente simples e semanticamente precisas. Tais frases, apesar de serem estruturalmente independentes entre si, referem-se a elementos em comum, o que cria um vínculo semântico entre elas. No contexto em que trabalhamos, os elementos aos quais as frases se referem são conceitos primitivos (ou seja, elementos conhecidos no contexto em questão) introduzidos por referência de Hilbert através de artigos definidos (tal como em "the ioi-gs").

Escolhemos as Especificações de programa em linguagem natural como contexto de aplicação de nosso sistema. As especificações de programa são compostas por frases que devem respeitar as condições que acabamos de descrever. Cada frase de uma especificação de programas representa uma exigência que o programa deve cumprir.

Neste trabalho nós inicialmente criamos uma representação formal para nossas frases (exigências) centralizada nas referências de Hilbert. Com isso obtemos um formalismo que nos permite relacionar elementos semanticamente associados dentro de uma mesma exigência ou, então, em exigências distintas. Conseqüentemente, este formalismo fornece ferramentas úteis à tarefa de construir uma base de conhecimentos que represente o conhecimento expresso nas exigências, uma vez que permite a associação entre seus elementos.

A representação formal das exigências é obtida através de análise sintático/semântica das mesmas e corresponde a uma fórmula híbrida do Cálculo de Predicados. O hibridismo destas fórmulas provém do fato de elas apresentarem a estrutura clássica das fórmulas bem formadas do Cálculo de Predicado acrescida de uma peculiaridade: suas variáveis são caracterizadas (anotadas) por expressões da Lógica Terminológica. Conforme exposto neste texto, a porção terminológica destas fórmulas permite que se utilize a subsunção como critério de simplificação da base de conhecimentos. Além disso, as expressões da Lógica Terminológica são muito úteis como ferramenta de resolução do problema da referência de Hilbert, uma vez que são

um formalismo eficaz para especificar os elementos que são introduzidos por artigos definidos.

O analisador sintático e semântico implementado corresponde a uma estrutura (tal como definida por Bourbaki e Piaget) que define uma gramática aplicativa. O caráter estrutural do analisador se concentra em duas estruturas-mãe: a da álgebra e a da ordem. A estrutura-mãe da álgebra é representada por regras de beta redução que automaticamente geram novas regras semânticas da gramática a partir da combinação de regras semânticas fundamentais inicialmente definidas.

No nosso sistema, as regras semânticas fundamentais são aquelas que correspondem ao significado das categorias sintáticas dos casos e dos verbos da gramática.

A estrutura-mãe da ordem é fundamentada na heurística de algumas propriedades lingüísticas. O caráter estrutural do analisador apresenta a vantagem de permitir a geração automática de algumas regras semânticas, o que dispensa o lingüista da tarefa de defini-las.

A recuperação da informação, que representa a etapa final do sistema implementado, é efetuada por um demonstrador automático híbrido de teoremas que é capaz de efetuar avaliações parciais. Conseqüentemente, mesmo que o provador não disponha de conhecimento completo e abrangente que o habilite a produzir uma avaliação total (dado) como resposta a uma pergunta, ainda assim ele é capaz de produzir uma resposta conceitualmente útil.

A capacidade do provador de efetuar avaliações parciais provém da representação híbrida das exigências na base de conhecimento: a posição bem definida das expressões da Lógica Terminológica nas fórmulas que representam as exigências permite que o provador detecte e manipule facilmente estas expressões.

Uma outra vantagem que se obtém através da representação híbrida das exigências é a da simplificação do traço de prova. Conforme vimos, provar $r(X_1/cx_1, X_2/cx_2, ..., X_n/cx_n)$ garantindo que $(\lceil cx_1 \rceil\ X_1), (\lceil cx_2 \rceil\ X_2), ..., (\lceil cx_n \rceil\ X_n)$ são todas expressões verdadeiras, é equivalente a provar $r(X_1, X_2, ..., X_n) \& cx_1(X_1) \& cx_2(X_2) \& ... \& cx_n(X_n)$. Contudo, a primeira forma de provar é bem mais simples, pois requer a utilização de menos recursos do provador. Realmente, na segunda forma de prova, toda a rotina de prova é chamada recursivamente para provar as metas r, $cx_1, cx_2, ...,$

cx_n . Já na primeira forma, que é a que nós utilizamos, somente a prova da meta r é desencadeada. Durante a prova de r, o próprio algoritmo de unificação tenta verificar se X_1 pode ser uma instância de cx_1 (ou seja, se X_1 pode ser anotado por cx_1, ou, ainda, se $(/cx_1 \;) \; X_1)$ é verdadeiro), se X_2 pode ser uma instância de cx_2, ..., e se X_n pode ser uma instância de cx_n. Tal verificação dispensa e substitui as chamadas recursivas do algoritmo de prova para as metas cx_1, cx_2, ..., cx_n.

Finalmente, implementamos um protótipo do Sistema de Representação do Conhecimento descrito. Os resultados são satisfatórios para o domínio de nossas exigências, cujo principal problema é a referência de Hilbert.

Como trabalhos futuros, sugerimos um aprofundamento nas pesquisas das características lingüísticas de tal maneira a explorar mais os critérios da subsunção a serem aplicados à linguagem natural. Lembremos que qualquer aperfeiçoamento neste nível representará um avanço nos critérios de simplificação de bases de conhecimento e nos traços de prova. Sugerimos também uma reflexão mais profunda visando a descobrir caminhos ainda mais eficazes de armazenamento de ligações em ambientes e de coleta de valor final de variáveis nestes mesmos ambientes, uma vez que isto representaria um aprimoramento num ponto vital do nosso sistema de recuperação de informação: a unificação semântica.

REFERÊNCIAS

[1] Abas, C.A.A.P, *Descrição e Paraconsistência*, tese de doutorado PUC SP, 1985.

[2] Allen, J. *Natural Language Understanding*, USA, The Benjamin/Cummings Publishing Company, Inc., 1987.

[3] Árabe. A.M.F., Pereira, A.C., Testa, T.A.R., *Communicating with Databases in Natural Language: A Proposal of Simplify Interfaces*, Aceito para a 1995 Int.Conf. IEEE- SMC Vancouver.

[4] Aragon,D., Castro, I., Alcoforado, P., *Lógica Para Informática- Um Estudo Introdutório*, ILTC- Instituto de Lógica e ilosofia e Teoria da Ciência, Edição Prévia.

[5] Araribóia, G., *Inteligência Artificial- Um Curso Prático*, ILTC, 1989.

[6] Barros, M.P., *Introdução Automática de Controle em Programação Lógica*, tese de mestrado UFU, 1991.

[7] Beaugrande, R. & Dressler, M.U., *Introduction to Text Linguistics[trad.]*.London, Longman, 1981.

[8] Bittencout, G., *The Integration of Terminological and Logical Knowledge Representation Languages*, Elsevier Science Publishing Co., Inc. 1990.

[9] Borrajo, D., Veloso, M., *Incremental Learning of Control Knowledge for Nonlinear Problem Solving*, in Proceedings of the European Conference on Machine Learning, 1994.

[10] Bourbaki, N. *"L'architecture des mathématiques"*, in F. Le Lionnais, Les grands courants de la pensée mathématique, Paris, 1948.

[11] Chambreuil, M., *Grammaire de Montague*, Éditions ADOSA, B.P. 467, 1989.

[12] Chang,C. and Lee, R., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York San Francisco London, 1973.

[13] Clark, K.L, Tärnlund, A., (editores), *Logic Programming* - Academic Press inc. (London) Ltd, 1982.

[14] Da Silva, R.M., Pereira, A.C., Netto, M.L.A.: *A System of Knowledge Representation Based on Formulae of Predicate Calculus Whose Variables are Annotated by Expressions of a "Fuzzy" Terminological Logic*, Lecture Notes in Computer Science (n. 945), pp. 409-417 - Springer-Verlag, editada por Bernadette Bouchon-Meunier, Ronal R. Yager e Lotfi A. Zadeh. Igualmente na conferência "International Conference on Information Processing and Management of Uncertainty in knowledge-Based Systems", IPMU 1994, Paris, França, 4-8 Julho 1994.

[15] Da Silva, R.M., Pereira, A.C., Netto, M.L.A.: Information Retrieval by Means of a Semantic Unification Guided by Resources of Terminological Logic, International Conference IEEE- SMC (Systems, Man and Cybernetics), San Antonio, Texas, U.S.A., Outubro 1994.

[16] Da Silva Julia, R.M., Seabra, J.R., Sameghini, I.: *An Intelligent Parser that Automatically Generates Semantic Rules During Syntatic and Semantic Analysis*, "International Conference IEEE- SMC (Systems, Man and Cybernetics)", Vancouver, Britsh Columbia, Canada, 22-25 Outubro 1995, pp. 806-811.

[17] Davidson, D., *Verdade e Significado,* in Fundamentos Metodológicos de Lingüística, Marcelo Dascal - Vol. III.

[18] DeGroot, D. and Lindstrom, G., *Logic Programming* - Functions, Relations, and Equations - Prentice-Hall, Englewood Cliffs, New Jersey 07632.

[19] DeJong, G. F. and Gratch, J., Steve Minton, *Learning Search Control knowledge: An Explanation Based Approach*, in: Artificial Intelligence 50 (1991) 117 - 127.

[20] DeMarco 89 DeMarco, T., *Structured Analysis and System Specification.* Prentice-Hall, Youdon Press, Englewood Cliffs, 1979.

[21] Desclés, J.P., *Langages applicatifs langues naturelles et cognition*- Hermès, Paris, 1990.

[22] Donini, F.M., Lenzerini, M. and Nardi, D., *The Complexity of Existential Quantification in Concept Languages*, in: Artificial Intelligence 53 (1992) 309-327.

[23] Doyle, J. and Patil, R.S., *Two theses of knowledge representation: language restrictions, taxonomic classification, and the utility of representation services*, in: Artificial Intelligence 48 (1991) 261 - 297.

[24] Dowty, D., Wall, R. and Peters, S., *Introduction to Montague Semantics*, D.Reidel Publishing Company, 1985.

[25] Dubois, D.and Prade, H., *Théorie des Possibilités*, MASSON, 1985.

[26] Ershov, A.P., *Mixed Computation: Potential Applications and Problems for Study,* in: Theoretical Computer Science 18(1982) 41-67.

[27] Etzioni,O., *A structural theory of explanation -based learning*, in: Artificial Intelligence 60 (1993) 93-139.

[28] Fillmore, C.J., *The Case For Case*, in 'Universals in Linguistic Theory', Brech&Harms (eds), Hoet, Reinhart e Win. New York, 1968.

[29] Gazdar, G., *Phrase Structure Grammar*, in: Nature of syntatic representation, Jacobson et Pullum, Dordrecht: Reidel, 1979, pp 131-186.

[30] Gazdar, G., Klein, E., Pullum, G. and Sag.I, *Generalized Phrase Structure Grammar*, in: Harvard University Press, Cambridge, 1985.

[31] Gazdar, G. and Mellish, C.S., *Natural Language Processing in LISP.*

[32] Gochet, P. Gribomont, P., *Logique-Méthodes pour l'informatique fondamentale* - Hermes, Paris, 1990-1991.

[33] Grover,C., Briscoe, E., Carrolis, G. and Boguraev, *The Alvey Natural Language Tools Grammar, in: University of Cambridge*, Computer Lab.

[34] Guillaume, M., *Recherches sur le Symbole de Hilbert-* Clermont-Ferrand, 1960.

[35] Halliday, M.A.K. & HASAN, R. - *Cohesion in English.* London. Longman, 1976.

[36] Harmelen, F. and Bundy, A., *Explanation-Based Generalisation = Partial Evaluation*, in: Artificial Intelligence 36 (1988) 401-412.

[37] Hoey, M., *Patterns of Lexis in Text.* Oxford University Press, 1991.

[38] Hovy, E.H., *Pragmatics and natural Language Generation*, in: Artificial Intelligence 43 (1990) 153-197.

[39] [IEEE 83] IEEE, *An American National IEEE Standard Glossary of Software Engineering Terminology* n ° ANSI/IEEE Std 729-1983, 1983.

[40] [IEEE 84] IEEE *Guide to Software Requirements Specifications* - ANSI/IEEE Std 830 - 1984, IEEE, 1984.

[41] Jones,S.L.P., *The Implementation of Functional Programming Languages.*

[42] Keller, R., *Defining Operationality for Explanation-Based Learning*, in: Artificial Intelligence 35 (1988) 227-241.

[43] Kindermann, C., *Retraction in Terminologcial Knowledge Base*, ECAI 92, 10th European Conf. on Artificial Intelligence Edited by B. Neumann Published in 1992 by John Wisley & sons, Ltd.

[44] Leisenring, A.C., *Mathematical Logic and Hilbert's ε Symbol*, London MacDonald Technical & Scientific, 1969.

[45] Lewis, D., *General Semantics*, In Philosophical Papers I, Oxford University Press.

[46] Minsky, M., *A framework for representing knowledge.* In: WINSTON, P. (org.). The Psychology of Computer Vision. New York, McGraw-Hill, 1975.

[47] Minton,S., Carbonell, J.G., Knoblock, C.A., Kuokka, R., Etizioni, O., and Gil, Y., *Explanation Based Learning: A Problem Solving Perspective*, in:Artificial Intelligence 40 (1989) 63-118.

[48] Muller, A.L., *Ao Projeto Lingüistico da Gramática de Montague*, GEL Anais do XXXIX seminário.

[49] Napoli, A., *Subsumption and Classification-Based Reasoning in Object-Based Representations*, in ECAI 92.

[50] Nebel, B., *Terminological Reasoning is Inherently Intractable*, Artificial Intelligence 43 (1990) 235-249.

[51] Nebel, B., *Reasoning and Revision in Hybrid Representation-* Lectures Notes in Artificial Intelligence n.422.

[52] Nillson,N., *Principles of Artificial Intelligence*, Tioga Publishing Co; Palo Alto, California, 1980.

[53] Patel-Schneider, P.F., *A Four-Valued Semantics for Terminological Logics*, in:Artificial Intelligence, Vol. 39, 399.

[54] Patel-Schneider, P.F., *Undecidability of Subsumption in NIKL*, in: Artificial Intelligence 39 (1989) 263-272.

[55] Pereira,F.C.N., *Incremental Interpretation*, in: Artificial Intelligence 50 (1991) 37-82.

[56] Petöfi, J. , *Semantics. Pragmatics. Text Theory*. Università di Urbino, Worling Papers, A. n36, 1974.

[57] Peyton Jones,S.L., *The Implementation of Functional Programming Languages*, Prentice Hall.

[58] Piaget, J., *Le Structuralisme*, Coll. " Que sais-je?", n. 1311- Copyright by Presses Universitaires de France, Paris- 1970.

[59] Porter, B.X., *Concept Learning and Heuristique Classification in Weak-Theory Domains*, in:Artificial Intelligence 45 (1990) 229-263.

[60] Pressman, R.S., *Software Engineering- A Practitioner's Approach*- Mcgraw-Hill, 1992.

[61] Robinson, J.A., *Computational logical: the unification computation*, "Machine Intelligence", Vol.6(B.Meltzer and D.Michie,eds.), American Elsevier, New York, pp 63-72, 1971a.

[62] Rumelhart, D.E., *Schemata: The building blocks of coginition. In: SPIRO et akuu (orgs.) Theoretical issues in Reading Comprehension*. New Jersey, Lawrence Earlbaum Asso, 1980.

[63] Schank, R., *The Cognitive Computer on Language, learning and artificial intelligence*. Addison-Wesley Publishing Company, Inc., Reading, 1984.

[64] Schidt-Schaub, M., *Subsumption in KL-ONE is Undecidable*, in R.J. Brachman, H.J. Levesque & Reiter, editions, Proceedings of the lst International Conference on Principles of Knowledge Representattion and Reasoning, pages 421-431, Toronto, Canada, 1989.

[65] Schimidt, S.J., *Textheorie. Probleme einer Linguistik sprachlichen Kommunikation*, 1973.

[66] Shesov, S.D. and Myasnikov, A.G., *Logical-Semantic Structure of a Terminology and its Formal Properties*, in: Nauchno-Teckhnicheskaya Informatsiya, Seriya 2, Vol. 21, N.3, 1987.

[67] Siwek, P., *Psychologia Experimentalis*- Marietti Editori Ltd, Italy, 1958.

[68] Steedman, M., Categorial Grammar, *Lingua*, 90 (1993) 221-258, North Holland.

[69] Strawson, P.F., *Significado e Verdade*, in Fundamentos Metodológicos de Lingüistica, Marcelo Dascal- Vol III.

[70] Takeuchi, A, Furukawa, K., *Partial Evaluation of Prolog Programs and its Applications to Meta Programming*, ICOT Tech. Report, 1985.

[71] Tarski, A., *La Concepción semántica de la verdad y los fundamentos de la semántica*, Ediciones Nueva Visión, 1972.

[72] Tjandra, J.C., Bittencourt, G., *Mantra.: A Shell for Hybrid Knowledge Representation*, Proc. of the 1991 IEEE, International Conference on Tools for AI, San Jose, Ca- Nov 1991.

[73] Toussain,Y., *Methodes Informatiques et Linguistiques Pour L'aide a la Specification de Logiciel*, Tese de "doctorat d'Université", Université Paul Sabatier Toulouse, France, 1992.

[74] Vilain, M., *The Restricted Language Architecture of a Hybrid Representation System*, Proceedings IJCAI-85, Los Angeles, CA, 1985.

[75] Westerstâl,D., *Quantifiers in Formal and Natural Languages*, in: Handbook of Philosophical Logic, Gabbay, D. et Guenthner, F.,Eds.D.Reidel Publ. Co., 1989, chap.1, pp 1-131.

[76] Wittgeinstein, L., *Tratactus Logico-philosophicus*- Éditions Gallimard, 1993.

[77] Yamaki, C.K., *Combinando Avaliação Parcial e Introdução Automática de Controle para Aumentar a Eficiência de Sistemas Especialistas*, Tese de Mestrado apresentado à UNICAMP, 1990.

[78] Yamashita, M. *O Símbolo ε de Hilbert em Lógica Paraconsistente*, Tese de Doutorado PUC-SP-1985.

N° d'ordre: 2159

THESE

présentée

DEVANT L' UNIVERSITE PAUL - SABATIER DE TOULOUSE (SCIENCES) (EN COTUTELLE AVEC L' UNIVERSITE UNICAMP DE CAMPINAS AU BRESIL)

en vue de l'obtention

DU DOCTORAT D'UNIVERSITE

Spécialité : Informatique

par

Rita DA SILVA-JULIA

Un système hybride pour le traitement du langage naturel et pour la récupération de l'information

Soutenue le 28 Novembre 1995 au Brésil (UNICAMP) devant la /commission d'Examen

MM.	M.A. NETTO	Professeur à l'Université UNICAMP, Campinas	Président
	M. BORILLO	Directeur de Recherche au CNRS, IRIT, Toulouse	
MM.	E. FRANCOSO	Professeur à l'Université UNICAMP, Campinas	
	M. JINO	Professeur à l'Université UNICAMP, Campinas	
	D. NORDEMANN	Chercheur Titulaire à l'Institut de Recherches Spatiales à São José dos Campos (Brésil)	Rapporteur
	D. KAYSER	Professeur à l'Université de Paris XIII, Villetaneuse	Rapporteur

Directeurs de Thèse : Pr. Dr. M.A. Netto (Brésil), Dr. M. Borillo (France)
Université d'UNICAMP à Campinas (Brésil) et Université Paul Sabatier IRIT (France)
Co-responsable de Thèse: Pr.Dr. Antônio Costa Pereira (Univ. Fed. Uberlândia, Brésil)

**"Conserve a serenidade, minha filha,
tenha sempre em mente que as bênçãos
de Deus nos chegam em abundância,
e que as bênçãos do mundo nos chegam
na medida do indispensável. "**

ABEDENAGO NILLO DA SILVA

As almas gêmeas certamente se aninham e dançam no espaço abençoado onde razão e sentimento encontram o seu ponto de harmonioso equilíbrio.

A você, Stéphane Julia, ofereço todas as danças da minha vida, toda minha vida, a totalidade do que eu sou.

Sua mulher.

REMERCIEMENTS

A Dieu.

A Mario Borillo (UPS).

A Márcio Luiz de Andrade Netto (UNICAMP).

A Antônio Eduardo Costa Pereira (UFU).

A mon mari, Stéphane Julia.

A Simone Julia (UPS).

A Daniel Nordemann (INPE).

A Daniel Kayser (Université de Paris-Nord).

A Mario Jino (UNICAMP).

A Edson Françoso (UNICAMP).

A Rafael Mendes (UNICAMP).

A Borges (Universidade Federal de Curitiba).

A Secundino Soares Filho (UNICAMP).

A CAPES.

Au CNPq.

A mes parents, mon frère, mes soeurs, mes nièces et mes neveux.

A João Bosco (UFU).

A Gilberto Carrijo (UFU).

A Cristina et Mazé (CPG-UNICAMP).

A Graça et Nora (CETEC-UFU).

Aux amis et collègues: Myriam et Armando, Maristela et Jean-Paul, Jussara et Gonzaga, Ricardo et Luciana, Ronaldo et Célia, Gina Maira, Iva et Maurício, Ely, Olga, Valéria, Gugu, Ricardo Goodwin, Mohammed, Myriam Bras, Yannick Toussaint, Pierre, Michel, Bruno, Anne Condamines, Omar, Elias.

UN SYSTEME HYBRIDE POUR LE TRAITEMENT DU LANGAGE NATUREL ET POUR LA RECUPERATION DE L'INFORMATION

TABLE DES MATIERES

Résumé de la thèse de doctorat de Rita Maria da Silva Julia dans le cadre de la Convention de Cotutelle de Thèse qui existe entre l'UPS (Toulouse- France) et l' UNICAMP (Campinas- Brésil)

UN SYSTEME HYBRIDE POUR LE TRAITEMENT DU LANGAGE NATUREL ET POUR LA RECUPERATION DE L' INFORMATION

Nous proposons dans ce mémoire la mise en oeuvre d'un système hybride de liaison entre les ressources du Calcul des Prédicats et celles de la Logique Terminologique. Ce système est capable de:

- produire une représentation formelle d'exigences exprimées en Langage Naturel dans le cadre de spécifications de logiciel.
- introduire dans une base de connaissances la représentation formelle obtenue pour chaque exigence en prenant soin de détecter les possibles redondances et contradictions.
- répondre à des questions posées au système, par l'exécution d'un mécanisme d'inférence permettant la récupération de l'information stockée dans la base de connaissances.

Les exigences appartiennent à un sous-ensemble restreint du Langage Naturel qui se situe dans le contexte du domaine spatial.

La représentation formelle d'une exigence est obtenue par une analyse syntaxique et sémantique. Elle correspond à une formule du Calcul des Prédicats dont les variables sont annotées par des expressions de la Logique Terminologique qui les particularisent.

L'analyseur syntaxico-sémantique implémenté est un système formel construit selon la théorie du structuralisme. Ce système formel définit une Grammaire Applicative dont le mécanisme d'application est guidé par une méthode heuristique de l'Intelligence Artificielle.

Les réponses du système correspondent à l'évaluation partielle des questions par rapport à la base de connaissances. La récupération de l'information est effectuée par un démonstrateur de théorème basé sur la technique de la résolution linéaire. Ce démonstrateur utilise la sémantique de la Logique Terminologique pour guider son mécanisme d'inférence.

Les principales contributions de notre travail sont les suivantes:

- La mise en oeuvre d'un analyseur syntaxico-sémantique qui engendre automatiquement des règles sémantiques, ce qui dispense le linguiste de la tâche de les définir.
- L'utilisation d'une méthode heuristique de l'Intelligence Artificielle pour guider le processus d'analyse.
- L'utilisation d'une unification sémantique pour lier les méthodes d'inférence du Calcul des Prédicats et de la Logique Terminologique.
- L'utilisation de la subsomption pour simplifier la base de connaissances et le processus de récupération de l'information.

INTRODUCTION

Pour la résolution de problèmes complexes, il est nécessaire de posséder de puissants mécanismes permettant:

- le stockage de la connaissance.
- la manipulation de la connaissance stockée.

Lorsque l'ensemble de ces mécanismes est associé à l'Intelligence Artificielle (IA), on parle de représentation de la connaissance.

Les chercheurs travaillant dans le domaine de l'IA ont tout d'abord envisagé la mise en oeuvre d'un système automatique capable de résoudre des problèmes généraux. Par exemple, Newell, Shaw et Simon ont construit le "General Problem Solver" (GPS, [5]) qui est capable de déchiffrer des casse-têtes, de démontrer des théorèmes simples et de obtenir des intégrales indéfinies. Néanmoins, le GPS est limité à la résolution d'un ensemble de problèmes particuliers. Progressivement, les chercheurs ont réalisé que la mise en oeuvre d'un système automatique général de résolution des problèmes est excessivement difficile. C'est pour cela que, généralement, les systèmes automatiques de résolution de problèmes, même s'ils peuvent manipuler une grande quantité de connaissance, se rapportent uniquement à des domaines spécifiques et limités. Le DENDREL [5] est un des premiers systèmes de Représentation de la Connaissance qui a été construit selon cette stratégie. C'est un logiciel qui a été crée par Joshua Lederberg et Edward Feigenbaum et qui permet d'effectuer des calculs associés à des analyses chimiques.

Un système de Représentation de la Connaissance est nécessairement constitué d'au moins deux parties distinctes:

- une base de connaissance (BC);
- un mécanisme d'inférence qui, grâce à des techniques particulières de raisonnement, permet d'avoir accès à la connaissance implicite contenue dans la BC.

Nous proposons dans ce mémoire la mise en oeuvre d'un système hybride de Représentation de la Connaissance qui représente une liaison entre les ressources du Calcul des Prédicats et celles de la Logique Terminologique (LT) [14, 15, 16]. Ce système doit être capable de faire l'analyse syntaxique et sémantique d'exigences (exprimées dans un sous-ensemble restreint du Langage Naturel (LN)), de stocker les exigences analysées dans une BC et de

récuperer l'information stockée. Les exigences font partie d'un corpus de spécifications de logiciel qui se rapporte au domaine spatial.

L'analyse syntaxique et sémantique des exigences se fera selon certaines techniques du Traitement du Langage Naturel [16].

La récuperation de l'information se fera à travers d'un démonstrateur automatique de théorèmes.

Notre travail prend ses origines au sein du projet LESD ("Linguistic Engineering or Software Development" [73], financé par le Ministère de la Recherche et de l'Espace et par le CNRS), sans pour autant en être la continuation.

Ce mémoire est découpé en cinq parties. La première partie introduit la théorie de spécification de logiciel. Dans la deuxième partie, sont présentés les théories et les outils qui seront employés pour effectuer les analyses syntaxique et sémantique. La troisième partie est consacrée à la description de l'analyseur syntaxico-sémantique ainsi qu'à la construction de la BC. La quatrième partie résume les théories qui seront utilisés dans la construction du démonstrateur automatique de théorèmes. La dernière partie est consacrée à la présentation du démonstrateur de théorèmes qui doit permettre la récuperation de l'information se trouvant dans la BC.

1. LES SPECIFICATIONS DE LOGICIEL

Le génie logiciel est la discipline dont l'objectif est de fournir un ensemble de techniques appropriées à la mise en oeuvre de systèmes informatiques. Le cycle de vie qui est généralement adopté en génie logiciel pour la construction des systèmes informatiques est le suivant [60]:

- la phase de la planification ou de conception du système pendant laquelle les exigences du logiciel sont mises en évidence. Ces exigences sont décrites au sein d'un document spécifique que l'on désigne sous le nom de *spécifications de logiciel*.

- la phase de développement pendant laquelle les spécifications de logiciel sont traduites dans un formalisme compréhensible par l'ordinateur.

- la phase de maintenance pendant laquelle sont effectuées les modifications et les adaptations nécessaires au bon fonctionnement du logiciel.

Selon le guide "IEEE Guide to Software Requirements Specifications" [40], les principaux critères de qualité des spécifications de logiciel sont:

- la traçabilité.
- la non-ambiguïté
- la cohérence.
- la complétude.
- la vérifiabilité.
- la modifiabilité.
- la possibilité d'utiliser les spécifications pendant les phases de développement et de maintenance.

Les spécifications de logiciel en LN sont des structures composées par d'exigences. Une exigence est l'expression d'un besoin élémentaire de la foncionnalité d'un logiciel. Elle doit être une phrase simple, autant par sa structure syntaxique que par son vocabulaire.

Pour les spécifications de logiciel, le LN impose beaucoup moins de rigueur formelle qu'un langage formel (langage Z, par exemple, [73]). Il est en général beaucoup plus souple d'utilisation. Néanmoins, le LN comme outil de spécification peut présenter une contrapartie

regrettable: les spécifications sont susceptibles de présenter des irregularités associées à des contradictions ou à des ambiguités du langage utilisé. Pour éviter ces irrégularités dans les spécifications, il est necessaire de respecter au mieux les critères de qualité se trouvant dans le guide IEEE.

Ensuite, quelques exemples d'exigences du logiciel "ioi-gs" qui doit contrôler un véhicule spatial [73]:

- The ioi-gs shall provide space vehicle monitoring and control.
- The ioi-gs shall have the capability to control the automatic systems of the flight configuration.
- The ioi-gs shall have the capability to obtain and analyse memory dumps from any computer on-board the space-vehicle.

2. RESUME DES THEORIES RELATIVES A L'ELABORATION DE L'ANALYSEUR SYNTAXICO-SEMANTIQUE DU SYSTEME

2.1. Introduction

Cette section résume les théories utilisées pour la mise en oeuvre de l'analyseur syntaxico-sémantique de notre système de Représentation de la Connaissance.

2.2. Le Calcul Lambda

Le Calcul Lambda [57] est défini par des expressions appelées *expressions lambda* et par un mécanisme d'application de ces expressions.

Ensuite on décrit les expressions lambda (notation de Backus Naur):

<exp> ::= <constante>
| <variable>
| <exp> <exp> % Représente une application d'une expression à une autre expression.
| λ <variable>. <exp> % Représente une Abstraction lambda (<*variable*> représente le paramètre de l'abstraction et <*exp*> représente l'expression qui doit être produite lorsque l'abstracion est appliqué. <*exp*> correspond au *corps* de l'abstraction lambda).

L'opération d'application du Calcul Lambda est la β-réduction. Le résultat obtenu par l'application d'une abstraction lambda à un argument correspond à une instantiation du corps de l'abstraction. Dans cette instantiation, les occurences libres du paramètre sont remplacées par une copie de l'argument. Ce processus d'application est dénomé β-réduction.

Exemple: la β-réduction de l'expression lambda *(λx. + x 1) 4* produit l'expression *+ 4 1*.

2.3. Le Principe Applicatif de Schonfinkel

Ce principe [57] est enoncé de la façon suivante:

" Un opérateur à n-places f_n peut être representé par un opérateur équivalent *Curry* (f_n) à 1-place qui, lorsque il est apliqué à un opérande, il produit la représentation d'un opérateur à (n - 1)-places. Exemple: $Curry(f_2) = \lambda y.\ (\lambda x.\ f_2(x, y))$.

2.4. Les Supercombinateurs

Un supercombinateur [57] à n-places *$S* est une abstraction lambda de forme générique $\lambda x_1.\lambda x2...\ \lambda x_n$ E, où:

- $S ne posséde pas des variable qui figurent à titre d'occurence libre.
- toutes les abstractions lambda en E sont des supercombinateurs.
- $n \geq 0$.

Exemples: $\lambda f.\ f(\lambda x.\ +\ x\ x)$; 4; $\lambda x.\ +\ x\ 1$

2.5. Les Systèmes Hybrides de Représentation de la Connaissance et la LT

Dans cette section certaines définitions importantes dans le contexte de notre système sont présentées.

2.5.1. Les Concepts

Les concepts sont des éléments de la LT [43, 50, 51]. Ils sont divisés en deux groupes:

a) Les concepts génériques qui dénotent des classes d'individus. Ceux-ci sont classés en deux groupes:

a.1) Les concepts définis sont des concepts dont la signification est complètement déterminée par des descriptions (par exemple, le concept *humain* est un concept qui peut être completement défini par une description correspondante à la conjonction des concepts *animal* et *rationnel*).
a.2) Les concepts primitifs sont des concepts dont la signification est partiellement définies par des descriptions (ils ne peuvent pas être completement décrits). Par exemple, le concept *espèces-d'animaux* est un concept primitif. Dans les systèmes de représentation de la connaissance, les concepts primitifs correspondent à des concepts basiques qui sont bien determinés dans le contexte du système.

b) Les concepts individuels: ce sont des concepts utilisés pour représenter les individus du monde. Les concepts individuels ne peuvent pas être utilisés pour décrire d'autres concepts. Exemples: *Marie*, *Obelix*.

2.5.2. Les Rôles

Les rôles sont des éléments de la LT qui représentent des relations entre les individus du monde. Un concept peut être défini à partir de contraintes imposées sur les rôles. L'expression générale *r(c)* introduit un nouveau concept qui peut être traduit comme: *quel que soit le deuxième argument du rôle r, il faut qu'il satisfasse le concept c*. Par exemple, l'expression *membre-de-l'équipe (homme)* introduit un nouveau concept dont la signification est *tous les membres de l'équipe sont des hommes*.

2.5.3. Définition Formelle des Concepts (notation Backus Naur):

<concept> ::= <concept-atomique> |
<concept-atomique> & <concept> | % représente un concept non atomique.
<rôle>(<concept>)

<concept-atomique> ::= <identifieur>

2.5.4. Définition Formelle des Rôles (notation Backus Naur):

<rôle> ::= <rôle-atomique> |
<rôle-atomique> & <rôle> % représente un rôle non atomique
<rôle-atomique> ::= <identifieur>

2.5.5. Table des Valeurs Sémantiques des expressions de la LT

Concept	Type	Valeur Sémantique [74]
c	atomique	$\lambda X\ [\lceil c \rceil X]$
c1 & c2	non atomique	$\lambda X\ [\lceil c1 \rceil X] \& [\lceil c2 \rceil X]$
r(c)	non atomique	$\lambda X\ \forall Y\ [\lceil r \rceil X Y] \rightarrow [\lceil c \rceil Y]$

2.5.6. Formalisme Hybride de Représentation de la Connaissance

Un formalisme hybride de représentation de la connaissance est un formalisme qui divise la représentation de la connaissance en deux groupes: le groupe assertive et le groupe terminologique.

2.5.6.1. Représentation de la Connaissance Terminologique

De la même façon que les reseaux sémantiques et les cadres (en anglais, "frames"), la représentation de la connaissance terminologique est concentrée dans les objets. La connaissance terminologique se rapporte aux significations des concepts et des rôles.

Ensuite on montre un exemple de terminologie qui introduit les concepts définis *herbivore* et *carnivore*.

herbivore =.. animal & nourrit-de(végétal)
carnivore =.. animal & nourrit-de(viande),

où *animal*, *viande* et *végétal* sont des concepts primitifs et où *nourrit-de* est un rôle primitif.

2.5.6.2. Représentation de la Connaissance Assertive

Représente la connaissance sur le monde en associant les concepts et les rôles aux individus du monde (connaissance extensionelle). Dans un système de représentation hybride, l'information sur le monde (assertive) est donnée par des termes (des concepts et des rôles) décrits dans une terminologie.

Exemple de description du monde: *(herbivore lapin)*; *(auteur-de La-Peste Camus)*.

2.5.7. La Logique Terminologique (LT)

La LT est une logique du premier ordre qui traite des concepts et des rôles selon le méthode d'inférence de la *subsomption* [50, 51, 54]. L' *extension* d'un concept, dont notation est c^e, est définie par l'ensemble: $\{X / \lceil c \rfloor(X)\}$, où $\lceil c \rfloor$ représente la sémantique du concept c. Cet ensemble représente les individus du monde qui satisfont au concept c. Un concept c subsume un concept d si $c^e \supseteq d^e$. Par exemple, le concept *humain* subsume le concept *humain & homme*.

2.6. Grammaire

Une grammaire est définie par G = (V, T, P, S), où:

- V est l'ensemble non vide qui définit le vocabulaire total du langage (vocabulaire terminal et non terminal).
- T est l'alphabet terminal du langage (sous-ensemble non vide de V).
- P est l'ensemble fini des règles de production du type $a \rightarrow b$, où a appartient au vocabulaire non terminal et b appartient au vocabulaire terminal ou non terminal.
- S appartient à l'ensemble des non terminaux du langage (S est appelé symbole initial de la grammaire).

Une phrase n'est pas bien formée lorsqu'elle présente des terminaux qui n'appartienent pas à T ou lorsque les mots s'organisent dans la phrase dans un ordre qui ne soit pas prévu par les règles de production. Le langage représente l'ensemble des formules (phrases) bien formées de la grammaire.

2.7. Méthode Pour Produire les Formes Normales de Skolem des Formules du Calcul des Prédicats

Une formule propositionnelle α est sous la forme conjonctive normale (FCN) ssi α représente une conjonction $\beta_1 \wedge \ldots \wedge \beta_n$, $(n \geq 1)$ où, pour chaque i $(1 \leq i \leq n)$, β_i est ou bien une disjonction de littéraux, ou bien un littéral. Exemples: $(\neg p \vee r) \wedge (q \vee r)$, $p \wedge (q \vee r)$.

Une formule du Calcul des Prédicats γ est sous la *FCN prénexe* si et seulement si $\gamma = Q^1(x^1), \ldots, Q^n(x^n)M$, où $x^1, \ldots, x^n$ sont des variables distinctes, où chaque Q^j représente un quantificateur et où M est sous la FCN (il n'y a pas des quantificateurs en M).

La forme normale de Skolem d'une formule du Calcul des Prédicats correspond à une formule qui présente les propriétés suivantes:

- Elle se trouve sous la FCN prénexe.
- Sans changer les propriétés de l'inconsistence, les quantificateurs existentiels sont eliminés du préfixe de la formule originale en utilisant les fonctions Skolem.

Exemple: la forme normale de Skolem de la formule $\forall x\ \forall y\ \forall z\ \exists u\ (\neg P(x,z) \vee \neg P(y, z) \vee R(x, y, u))$ est la formule $\forall x\ \forall y\ \forall z(\neg P(x,z) \vee \neg P(y, z) \vee R(x, y, f(x, y, z)))$.

2.8. Les Structures

Une structure est un système de transformations [10, 58].

Les structures en général sont engendrées par un ensemble de structures-mère dont les principales sont:

- structure-mère de l'Algèbre: elle est responsable de la production de nouveaux éléments dans la structure à partir de la combinaison d'éléments qui existaient déjà dans la structure.
- structure-mère de l'ordre: elle est responsable du contrôle de l'ordre des combinaisons effectuées par la structure-mère de l'Algèbre.

2.9. Cohésion et Cohérence

La cohésion et la cohérence sont des critères qui garantissent la bonne formation syntaxique et sémantique des phrases d'un langage [7, 37, 62, 63].

2.10. Les Systèmes Formels

S est un système formel [5, 77] si S est composé:

- d'un alphabet Ss.

- d'un sous-ensemble récursif Fs de l'ensemble de toutes les sequences finies de Ss. Fs est appelé l'ensemble des formules biens formées de Ss.
- d'un sous-ensemble récursif Cs de Fs. Les éléments de Cs sont appelés *axiomes* de S.
- d'un sous-ensemble Rs de prédicats décidables appartenant à Fs. Les éléments de Rs sont appelés *règles d'inférence* de S.

Une déduction à partir d'une liste d'hypothèses $\{h_1, h_2, ...,h_n\}$ est une séquence finie de formules $f_1, f_2, ..., f_m$, telle que, quel que soit l'élément j qui appartient à l'ensemble $\{1, 2, ..., m\}$, un des cas suivants ait lieu:

- f_j est un axiome.
- f_j est une des hypothèses.
- fj est obtenue au travers d'applications des règles d'inférence à des formules qui sont antérieures à f_j dans la séquence.

2.11. La Sémantique de la Vérité Selon Tarski

Tarski [71] définit la vérité au travers d'une autre notion sémantique: la notion sémantique de la satisfaction. La satisfaction est une relation entre des objets quelconques et des fonctions propositionnelles [32]. Une fonction propositionnelle est un énoncé ϕx qui devient une proposition lorsque la variable x est associée à une signification determinée.

Exemple de fonction propositionnelle: *x est blanche*. Certains objets satisfont une fonction propositionnelle si cette fonction devient une proposition vraie lorsque ses variables qui figurent à titre d'occurence libre sont remplacées par ces objets (par exemple, l'objet *neige* satisfait la fonction propositionnelle *x est blanche*).

2.12. La Grammaire de Montague

Richard Montague (1930- 1970) [24, 48] a proposé une grammaire fondé sur les principes qui suivent:

- Il n'y a pas une différence qualitative entre le LN et les langages formels construits par le Logique. Donc, le LN peut être traité mathématiquement.
- La sémantique définit les limites d'une grammaire.

- Le lien entre un langage et le monde est fait par la définition de vérité (ce principe est également utilisé par Wittgenstein [76] et par Davidson [17].

2.13. Les Grammaires Applicatives Typées

En général, une grammaire applicative typée [21] est composée par:

- un ensemble d'expressions de base typées.
- un opérateur O capable de produire de manière récursive les types du système.
- une opération primitive (application) qui construit de nouvelles expressions typées en combinant des expressions déjà existentes.
- un ensemble récursif de formules bien formées (fbfs) denommées phrases.
- un ensemble de types dont les éléments peuvent être ou bien des types primaires ou bien des types composés. Un type primaire doit être appliqué à des variables. Si x et y sont des types, alors Oxy est un type composé où x correspond au type de l'argument auquel Oxy doit être appliqué et y correspond au type de l'expression qui doit être produite par l'application.
- une loi de combinaison des expressions permettant qu'une formule bien formée (fbf) soit produite.

3. L'EVOLUTION DE NOTRE ANALYSEUR SYNTAXICO-SEMANTIQUE

3.1. Introduction

Cette section présente l'analyseur syntaxico-sémantique qui produit la représentation formelle des exigences. L'analyseur est une structure [58] qui définit une Grammaire Applicative.

Le processus d'analyse est guidé pra une méthode heuristique de recherche de l'Intelligence Artificielle. Un ensemble de règles sémantiques est automatiquement engendré par l'ordinateur pendant l'analyse.

Avant de présenter l'analyseur que nous avons mis en oeuvre, nous allons résumer le fonctionnement de l'analyseur Alvey de la langue anglaise [33]. Ensuite nous présenterons notre analyseur à partir des résultats obtenus au moyen de l'Alvey. Finalement, nous présenterons la mise en oeuvre de notre analyseur.

3.2. Le Fonctionnement de l'Analyseur Alvey à Partir d'un Exemple d'Analyse

Considérons l'analyse de l'exigence E1: *"**The** ioi-gs shall control **the** computers on-board **the** space-vehicle"*. Dans les phrases du LN, les quantificateurs de Hilbert [1, 34, 44, 78] correspondent à des références de Hilbert du LN. Une référence de Hilbert du LN est introduite par un article défini [75] qui se rapporte à un objet connu dans le contexte analysé. Par exemple, les trois articles définis *"**the**"* de l'exigence E1 se rapportent, respectivement, aux objets ***ioi-gs***, ***computers*** et ***space-vehicle***, qui sont des concepts primitifs dans le context des exigences. Donc, ces trois articles définis introduisent des références de Hilbert.

L'analyseur Alvey effectue l'analyse d'une phrase en la traduisant en des prédicats ternaires qui représentent les quantificateurs de Hilbert de la phrase analisée.

Après l'analyse de E1, l'analyseur Alvey produit le résultat F1 suivant:

```
F1:  τ(x1, ioi-gs(x1),
        τ(x2, τ(x3, space-vehicle(x3),
           computer(x2) & on-board(x2, x3))
        control(x1, x2) )
       ),
```

où τ est le symbole qui représente les prédicats ternaires correspondants aux quantificateurs de Hilbert et où *x1*, *x2*, *x3* représentent les variables qui sont introduites par les références de Hilbert.

Nous présentons ci-dessous notre analyseur à partir des résultats obtenus par l'analyseur Alvey.

3.3. Description de l'Analyseur de Notre Système à Partir de l'Analyseur Alvey

L'analyse effectuée par notre analyseur a pour objectif de produire une représentation formelle pour les exigences. Cette représentation correspond à une forme normale de Skolem des formules produites par l'analyseur Alvey [12]. Conforme à ce qui a été décrit dans la section 2.7, tous les quantificateurs d'une formule qui est sous la forme normale de Skolem doivent être placés dans le préfixe des formules. Le processus qui engendre cette forme de Skolem est mis en oeuvre selon deux étapes:

- Initialement, les deuxièmes arguments des quantificateurs des formules dont les formes sont semblables à celles produites par l'analyseur Alvey sont transformées en des expressions de la LT (les expressions de la LT ne doivent pas presénter des quantificateurs). En reprenant l'exemple de la formule F1 antérieure, il est possible de remarquer qu'elle présente des quantificateurs dans les deuxièmes arguments de τ (tel que le quantificateur placé dans le deuxième argument du prédicat τ qui introduit la variable *x2*). L'application de ce premier pas à F1 produit l'expression F2 qui suit:

F2: τ(x1, ioi-gs(x1), τ(x2, computer(x2), τ(x3, space-vehicle(x3), on-board(x2,x3) & control(x1, x2))))

- Dans la deuxième étape du processus, le deuxième argument de chaque quantificateur (expressions de la LT) est utilisé comme annotation qui particularise la variable introduite dans le première argument du quantificateur. L'application à F2 de cette dèuxieme étape engendre l'expression F3 qui suit:

F3: control(x1/τ(ioi-gs), x3/τ(computer & part-of(space-vehicle)))).

La sémantique des expressions de la LT qui annotent les variables du système est la même proposée par Vilain [74], c'est-à-dire:

Expression	*Sémantique*
exp	⌈exp⌉
c1 & c2	$\lambda X(\lceil c1 \rceil X) \& (\lceil c2 \rceil X)$
r(c)	$\lambda X \forall Y(\lceil r \rceil X Y) \rightarrow (\lceil c \rceil Y))$
τ(c)	$\lambda X \| \{X \mid (\lceil c \rceil X))\} \| = 1$
Y/c	$\{Y \mid (\lambda X (\lceil c \rceil X) Y)\}$

Pour simplifier les notations, le symbole τ sera omis dorénavant.

3.4. La Mise en Oeuvre de l'Anayseur du Système

Les règles de la grammaire implementée dans notre système sont fondées sur les propriétés linguistiques des cas du Latin (néanmoins, nous aurions pu utiliser un autre système de cas standard quelconque) [19].

Considerons les règles de la grammaire simple (G) qui suit:

acte → verbe transitif, accusatif.
acte → verbe intransitif.
phrase → nominatif, acte.

Les éléments des règles grammaticales (tels que *acte*, *verbe transitif*, *accusatif*, *verbe intransitif*, *phrase*, *nominatif* etc) sont appelés *catégories syntaxiques* par Montague [11]. Dans sa grammaire, il propose que soit associée à chaque catégorie syntaxique une abstraction lambda qui correspond à sa signification. Les analyses syntaxique et sémantique sont effectuées en combinant deux à deux les catégories syntaxiques (analyse syntaxique) et les abstractions lambda qui sont associées à ces catégories (analyse sémantique). Ces combinaisons doivent continuer jusqu'à ce qu'une fbf dont la signification est vraie ou fausse soit produite. En conséquence, dans la grammaire de Montague, on doit définir autant d'abstractions lambda que de catégories syntaxiques.

A l'opposé de Montague, nous mettons en oeuvre, dans ce système, une grammaire applicative dans laquelle le linguiste doit définir seulement les abstractions lambda

correspondantes aux catégories syntaxiques des cas et des verbes (des catégories fondamentales). Les abstractions lambda qui correspondent aux autres catégories (des catégories non fondamentales) seront automatiquement engendrées par l'ordinateur pendant l'analyse.

L'analyseur correspond à une structure (voir section 2.8) representée par un système formel dont les axiomes sont des abstractions lambda typées des cas et dont les règles d'inférence sont des règles β de réduction. Le processus de l'analyse correspond à une sequence déductive f_1, f_2, ..., f_n, où f_n représente le résultat de l'analyse.

Durant l'analyse syntaxique, l'ordinateur combine les catégories syntaxiques jusqu'à ce qu'une phrase soit produite. Parallèlement, l'ordinateur combine les abstractions lambda associées à ces catégories syntaxiques jusqu'à ce qu'une formule hybride du Calcul des Prédicats, dont signification est vraie ou fausse, soit produite. Les combinaisons doivent respecter les critères de cohésion et de cohérence qui garantissent la correction syntaxique et sémantique des phrases analysées (voir section 2.9).

L'analyseur est fondé sur un algorithme de recherche de l'Intelligence Artificielle dans lequel les catégories non fondamentales sont engendrées au fur et à mesure qu'on parcourt un chemin sur un arbre de recherche. Le but de l'analyse est de découvrir un chemin sur l'arbre dont une feuille soit une fbf de signification vraie ou fausse.

Nous avons déjà dit que les structures sont engendrées par un ensemble de structures-mère (voir section 2.8). Notre analyseur comprend deux structures-mère: la structure-mère de l'Algèbre, qui indique comment combiner les abstractions lambda, et la structure-mère de l'ordre, qui indique dans quel ordre ces combinaisons doivent être effectuées. La structure-mère de l'Algèbre est introduite par des règles β de réduction qui combinent les abstractions lambda. La structure-mère de l'ordre est introduite par l'heuristique des types des abstractions lambda (celle-ci correspond à des chemins parcourus sur l'arbre de recherche). La structure-mère de l'ordre produit une séquence déductive, c'est-à-dire qu'elle impose une relation d'ordre qui établit la séquence des applications. Cette relation est établie par les types des abstractions lambda.

3.4.1. Définition des Expressions de l'Analyseur

Les expressions du système sont définies par (notation Backus-Naur):

<expression> ::= <terme> | <abstraction> | <application> | <description>

<terme> ::= <constante> | <variable>

<abstraction> ::= <lambda1> | <lambda2>

<lambda1> ::= l(<terme>, <expression>) |
le(<terme>, <expression>)
s(<terme>, <expression>)

<lambda2> ::= l(<type-guide>, <abstraction>, <expression>)

<type-guide> ::= nom | acc | dat | gen | ablat | vi | vt % en représentant le
nominatif, l'accusatif,
le datif, le genitif,
l'ablatif, le verbe intransitif
et le verbe transitif,
respectivement.

<description> ::= <événement> & < conjonction>

<événement> ::= [verbe <liste-de-termes>]

<liste-de-termes> ::= arg_1, arg_2, ..., arg_n,, onde arg_1, arg_2, ..., arg_n sont des termes.

<terminologie> ::= [<terme><liste-de-termes>]
<terminologie> ::= [<cas-sémantique> <lista-de-termes>]

<cas-sémantique> ::= ag | pat | instrum % *ag*, *pat*, *instrum* représentent
l' *agent*, le *patient* et l'*instrument*,
respectivement.

<conjonction> ::= <terminologie> |
[<terminologie>] & [<terminologie>]

<application> ::= <terminologie> |
[lambda1, <lista-de-termes>] |
[lambda2, abstraction]

<abstraction> représente les abstractions lambda des catégories de la grammaire et <type-guide> représente les types de ces abstractions.

3.4.2. Les Abstractions Lambda du Système

Les types des abstractions sont utilisés pour guider le processus déductif. Les types qui doivent être seulement appliqués à des termes sont appelés types primaires. Si *t1* e *t2* sont des types, alors *Ot1t2* est un type composé qui doit être appliqué à un type *t1*. Le résultat de l'application doit être un type *t2*.

Si *t* est un terme, si *e* et *e1* sont des expressions, si *lamb* est une <abstraction> et si *tip* est un <type-guide>, les abstractions du système ont les formes génériques suivantes:

a) l(t, e)
b) le(t, e)
c) s(t, e)
d) l(tip, lamb, e1)

Les formes décrites en *a*, *b* et *c* représentent des abstractions lambda de type primaire. Dans ces trois formes, *t* (paramètre) correspond à la tête de l'abstraction et *e* correspond au corps de l'abstraction (c'est l'expression produite lorsque l'abstraction est appliquée).

La forme décrite en *d* représente une abstraction lambda de type *tip* qui doit être appliquée à l'argument reçu à travers du paramètre *lamb* (tête). L'application doit produire une expression dont la forme est décrite en *e1* (corps). Donc, si *tip* représente le type composé *Ot1t2*, le type de *lamb* doit être *t1* et le type de *e1* doit être *t2*.

Piaget divise les structures qui représentent un phénomène en deux groupes:

- Le groupe des abstractions empiriques: c'est un groupe composé par des abstractions fondamentales définies à partir d'un ensemble de propriétés qui sont inhérentes au phénomène.
- Le groupe des abstractions réflexives: c'est un groupe composé par des abstractions non fondamentales qui sont engendrées à partir de l'application des structures-mère aux abstractions empiriques.

Le phénomène sur lequel notre système travaille est le phénomène linguistique. Nous avons déjà dit dans la section 3.4 que les abstractions fondamentales du système sont les

abstractions lambda des cas et des verbes. Ces abstractions représentent les abstractions empiriques du système et doivent être définies à partir d'un ensemble de propriétés linguistiques. Ces propriétés sont extraites des caractéristiques linguistiques des verbes et des cas du Latin. Nous résumons dans ce qui suit quelques propriétés linguistiques que l'on appelle *Pa*, *Pb*, *Pc* e *Pd*:

Pa: abstraction lambda de verbe transitif (type *vt*) est obtenue par la combinaison d'abstractions lambda du type primaire et des variables. Ces variables doivent représenter l'événement et les cas sémantiques de l'événement (tel que l'agent et le patient de l'événement).

Pb: abstraction lambda du nominatif (type *nom*) est obtenue par la combinaison d'une abstraction lambda de type primaire et d'une variable qui représente un substantif.

Pc: abstraction lambda de l'accusatif (type *acc*) doit être appliquée à une abstraction lambda du verbe transitif en produisant une abstraction lambda du verbe intransitif.

Pd: abstraction lambda du verbe intransitif (type *vi*) doit être appliquée à une abstraction lambda du nominatif en produisant une abstraction lambda d'une phrase (type *sent*) dont la signification est vraie ou fausse.

Les propriétés linguistiques correspondent à des heuristiques qui permettent la génération des abstractions lambda du système.

Exemples: la propriété *Pa* indique qu'une abstraction lambda du nominatif doit avoir la forme générique *l(t1, e)*, où *t1* est un terme et *e* est une expression. La propriété *Pc* indique qu'une abstraction de l'accusatif doit avoir la forme générique *l(acc, lamb1, lamb2)*, où *acc*, *lamb1* et *lamb2* représentent, respectivement, le type de l'abstraction, une abstraction du verbe transitif et une abstraction du verbe intransitif.

Nous avons déjà dit que les abstractions lambda représentent la signification (sémantique) des catégories syntaxiques. Le système représente cette sémantique à travers des cas sémantiques qui sont introduits dans les corps des abstractions lambda. Ces cas sémantiques sont représentés par des terminologies (voir section 3.4.1). Par exemple, la terminologie *[ag, Evénement, Agent]* représente le cas sémantique de l'agent du verbe transitif (voir l'abstraction du verbe transitif qui suit).

Considérons que *t*, *termin*, *sent* et *descr* représentent les types des *termes*, des *terminologies*, des *phrases* et des *descriptions*, respectivement. Nous pouvons définir la forme

des abstractions lambda du système. La propriété linguistique *Pa* définit la forme générique suivante pour le verbe transitif :

le(Événement,
  l(Agent,
    l(Patient,
      [Verbe, Événement, Agent, Patient] & [ag, Événement, Agent] &
      [pat, Événement, Patient]))), dont le type *vt* correspond à *OtOtOt descr.*

Les abstractions qui représentent les phrases du système sont structurées en fonction des éléments du verbe. Donc, la forme générique d'une phrase qui exprime un événement du verbe transitif est:

s(Événement1,
  l(Agent1,
    l(Patient1,
      [Verbe, Événement1, Agent1, Patient1] & [ag, Événement1, Agent1] &
      [pat, Événement1, Patient1])))

L'abstraction du nominatif est définie au travers de la propriété *Pb.* Elle a la forme générique:

l(N, [Subs, N]), dont type est *Ot termin.*

De façon analogue, *Pc* définit la forme générique suivante pour l'accusatif:

l(acc, Verbe,
 l(vi, Nominatif,
  s(Événement1,
   l(Agent1,
    l(Patient1,
     [Verbe, Événement1, Agent1, Patient1] & [Nominatif, Agent1] &
     [Accusatif, Patient1]))))), dont le type est *O Ot Ot Ot descr O Ot termin sent.*

La propriété *Pd* détermine la forme générique du verbe intransitif, c'est-à-dire.:

l(vi, Nominatif,
  s(Événement,
    l(Agent1,
      l(Patient1,
        [Verbe, Événement1, Agent1, Patient1] & [Nominatif1, Agent1] &
        [Accusatif, Patient1])))), dont le type est *O Ot termin sent*.

Il est intéressant de remarquer que les abstractions lambda du système correspondent à des supercombinateurs (voir section 2.4), ce qui va énormément simplifier la tâche de produire la forme normale de Skolem qui correspond aux exigences.

3.4.3. L'Algorithme d'Analyse

L'analyse est mise en oeuvre selon l'algorithme que suit:

- A partir des règles de la grammaire, l'algorithme construit une pile L qui emmagasine les abstractions lambda empiriques de la phrase (voir prédicat "build" plus en bas).
- L'algorithme choisit (voir prédicats "choose" et "ap" plus en bas) dans la pile L une abstraction lambda *l(Ot1 t2, tête, corps)* de type composé. Il produit ensuite une pile RL qui correspond à la pile L reduite de l'abstraction *l(Ot1 t2, tête, corps)*.
- L'algorithme saisit dans RL une abstraction lambda A de type *t1* à laquelle l'abstraction *l(Ot1 t2, tête, corps)* peut être appliquée. Il produit ensuite une nouvelle pille RL1 qui correspond à RL reduite de A.
- A travers des prédicats *beta* décrits plus en bas, l'algorithme effectue l'application (ou réduction) exprimée en *[l(Ot1t2, tête, corps), A]*. Soit S le résultat de cette application.
- L'algorithme est répété de façon récursive à partir du deuxième pas en considérant une pille L composée par S et RL1. L'algorithme s'arrête lorsqu'il ne reste qu'une abstraction dans la pile. Cette abstraction doit correspondre à une abstraction lambda d'une phrase.

Nous remarquons que les types des abstractions lambda accélèrent l'analyse, car ils jouent le rôle de guide dans le processus de combiner ces abstractions. Donc, les propriétés linguistiques à partir desquelles ont été construites les abstractions lambda du système

représentent des critères de cohérence (voir section 2.9). Les critères de cohésion sont fournis par la grammaire.

Dans ce qui suit, nous présentons les prédicats d'analyse du système (notations du langage PROLOG):

```
analyse(Sentence, Res) :- build(Sentence, L), red(L, Res).

1. beta(X,X) :- var(X), !.
2. beta([X|R], [X|R])  :- var(X), !.
3. beta(X,X) :- atom(X), !.
4. beta([l(X,P), X], Q) :- !, beta(P,Q).
5. beta([l(_, X, P), X],  Q) :- !, beta(P,Q).
6. beta([l(X, P1), X | R], Q) :- !, beta(P1, P), beta([P | R], Q).
7. beta([le (X, P), X], Q) :- !, beta(P, Q).
8. beta([le(X,P1), X | R], Q) :- !, beta(P1,P), beta([P | R], Q).
9. beta([s (X, P), X], Q) :- !, beta(P, Q).
10. beta([s (X,P1), X | R], Q) :- !, beta(P1,P), beta([P | R], Q).
11. beta( [P | R], [P | R]) :- !.
12. beta( A & B, P & Q) :- !, beta(A, P), beta(B, Q).
13. beta(l(X, P), l(X, Q)) :- !, beta(P, Q).
14. beta(le(X, P), le(X, Q)) :- beta(P, Q).
15. beta(s(X,P), s(X, Q)) :- beta(P, Q).
16. beta(l(Tp, X, P), l(Tp, X, Q)) :- beta(P, Q).

choose([X|R], X, R).
choose([X|R], Y, [X|R1]) :- choose(R,Y,R1).

red([L], [L]) :- !.
red(L, Res) :- choose(L, l(Tp1, X1, P), RL),
               ap(Tp1, Q),
               choose(RL, Q, RL1),
               beta([l(Tp1, X1, P), Q], S),
               red([S | RL1], Res), !.
```

```
ap(acc, le(_, l(_, l(_, _)))) :- !.
ap(vi, l(_, _)) :- !.
```

3.4.4. Un Exemple d'Analyse

Dans ce qui suit, l'analyse de l'exigence E: *The ioi-gs controls the space-vehicle*:

3.4.4.1. La Génération de la Pile d'Abstractions Empiriques

En fonction de l'ordre des mots dans l'exigence E et de la grammaire G, le système engendre les abstractions lambda suivantes:

- l(Z1, [ioi-gs, Z1]), pour le nominatif.

- le(Y1,
  l(Y2,
  l(Y3,
  [controls, Y1, Y2, Y3] & [ag, Y1, Y2] & [pac, Y1, Y3]))), pour le verbe transitif;

- l(acc, X1,
  l(vi, X2,
  s(X3,
  l(X4,
  l(X5,
  [X1, X3, X4, X5] & [X2, X4] & [space-vehicle, X5]))))), pour l'accusatif.

La pile L produite pendant l'exécution du premier pas de l'algorithme d'analyse est composée par ces trois abstractions lambda empiriques.

3.4.4.2. La réduction de la pile L

Pour simplifier les notations, nous désignerons les abstractions de l'*accusatif*, *du verbe transitif* et du *nominatif* par *eac*, *evt* et *enom*, respectivement.

L'algorithme d'analyse, en accord avec les propriétés linguistiques, produit en L la séquence de réduction qui suit :

- *[eac evt],* dont le résultat est:

```
l(vi, X2,
  s(X3,
    l(X4,
      l(X5,
        [control, X3, X4, X5] & [ag, X3, X4] & [pat, X3, X5] & [X2, X4] &
        [space-vehicle, X5] )))).
```

Ce résultat correspond à une abstraction lambda *vi* du verbe intransitif (type *O O t termin sent*). On peut remarquer que cette abstraction du verbe intransitif a été engendré automatiquement par le processus de analyse.

- [vi enom], dont le résultat est:

```
s(X3,
 l(X4,
  l(X5,
   [control, X3, X4, X5] & [ag, X3, X4] & [pat, X3, X5] &
   [ioi-gs, X4] &
   [vs, X5] ))),
```

qui représente une abstraction lambda d'une phrase (type *sent*).

A partir de cet exemple, on peut faire deux remarques importantes:

- les abstractions lambda du système sont des supercombinateurs qui opèrent comme des opérateurs de Curry pendant le processus de réduction (voir section 2.3 et 2.4).
- le traitement donné aux variables qui sont introduites par des références de Hilbert garantit le critère de qualité de la traçabilité (voir section 1) dans le contexte d'une exigence. Par exemple, la variable X3 de l'abstraction qui représente le résultat de l'analyse correspond, dans la totalité du contexte de l'abstraction, à l'événement *control.* Donc, les terminologies

[ag, X3, X4] et *[pat, X3, X5]* se rapportent à l'événement *control*, en représentant, respectivement, l'agent et le patient de cet événement.

3.4.5. Traitement de l'Abstraction Lambda Produite par la Réduction de la Pile: Représentation Finale d'une Exigence

Lorsque la réduction de la pile est achevée, le système met en oeuvre un algorithme qui traduit l'abstraction lambda produite par la réduction dans une conjonction d'expressions. Ces expressions correspondent à l'événement de l'exigence analysée et aux cas sémantiques de ces événements. Cet algorithme est semblable à l'algorithme classique qui produit la forme normale de Skolem d'une formule logique (voir section 2.7).

L'expression générique de l'événement est:

pred(X1/exp1, X2/exp2, ..., X3/expn), où:

- *pred* représente un prédicat à n-places (le verbe de l'événement).
- *X/exp* représente une variable *X* annotée par une expression *exp* de la LT.

L'expression générique des cas sémantiques de l'événement est:

cas(X/exp1, X2/exp2), où:

- *cas* représente le cas sémantique.
- X/exp représente une variable *X* annotée par une expression *exp* de la LT.

Donc, la forme normale de Skolem de l'abstraction lambda produite à la fin de la dernière section est:

controls(e0/ev, X/ioi-gs, Y/space-vehicle) &
ag(e0/ev, X/ioi-gs) &
pac(e0/ev, Y/space-vehicle),

qui est la représentation finale de l'exigence: *The ioi-gs controls the space-vehicle*. Observons que les variables de la représentation finale des exigences sont annotées par des expressions de la

LT (par exemple, les concepts *ioi-gs* et *space-véhicle*). Les annotations particularisent les variables des représentations finales. Par conséquent, les variables annotées ne sont plus des variables qui figurent à titre d'occurence libre et la forme finale produite correspond à une phrase dont la signification est vraie ou fausse selon la notion sémantique de Tarski (voir section 2.11).

On peut aussi remarquer que dans la forme normale de Skolem de l'exemple, la variable *X3,* qui représente l'événement *control* est remplacée par la constante *e0/ev*, où l'annotation *ev* indique que *e0* est un événement. De façon analogue, les variables X4 e X5 introduites par des références de Hilbert sont remplacées, respectivement, par les expressions constantes *X/ioi-gs* et *Y/space-vehicle*. Ces remplacements sont obtenus grâce à l'application des fonctions de Skolem [12].

Pendant tout le processus d'analyse, les exigences subissent des transformations. Néanmoins, dans toutes ces transformations, il est toujours possible d'identifier les éléments de l'exigence originale (par exemple, les éléments *ioi-gs* et *space-vehicle* sont identifiés dans toutes les formules logiques dans les quelles l'exigence *The ioi-gs controls the space-vehicle* est transformée). Donc, l'analyse des exigences garantit le critère de qualité de la traçabilité des spécifications de logiciel [40].

3.4.6. L'Hybridisme Dans la BC

Les exigences sont introduites dans la BC sous leur forme normale de Skolem. Dans ces formes de Skolem, les expressions de l'événement et des cas sémantiques représentent la connaissance assertive et les annotations des variables représentent la connaissance terminologique de la BC (voir section 2.5.6).

3.4.7. La Subsomption: un Critère de Simplification Pendant la Construction Incrémentale de la BC

La construction de la BC est effectuée de façon incrémentale: on introduit les exigences individuellement dans la BC en faisant attention de ne pas introduire des contradictions ou des redondances. On utilise la sémantique de la subsomption de la LT comme critère de construction de la BC (voir la définition de *subsumer* dans la section 2.5.7). Par exemple, supposons que l'on veuille introduire dans la BC les expressions:

controls(e1/ev, X/ioi-gs, Y/computers & on-board(space-vehicle)) &
ag(e1/ev, X/ioi-gs) &
pac(e1/ev, Y/computer & on-board(space-vehicle))

correspondantes à l'exigence: *The ioi-gs controls the computers on-board the space vehicle.* Supposons aussi que la BC contienne déjà les expressions:

controls(e0/ev, X/ioi-gs, Y/equipments & on-board(space-vehicle)) &
ag(e0/ev, X/ioi-gs) &
pat(e0/ev, Y/equipments & on-board(space-vehicle))

correspondantes à l'exigence: *The ioi-gs controls the equipments on-board the space-vehicle.* On remarque que le concept *equipments & on-board(space-vehicle)* subsume le concept *computers & on-board(space-vehicle),* car le concept *equipments* subsume le concept *computers.* En conséquence, l'exigence *The ioi-gs controls the equipments on-board the space-vehicle* subsume l'exigence *The ioi-gs controls the computers on-board the space-vehicle.* Le système, pour éviter des redondances, n'introduit pas la dernière exigence dans la BC.

4.PRINCIPES THEORIQUES IMPORTANTS POUR LA MISE EN OEUVRE D'UN DEMONSTRATEUR AUTOMATIQUE DE THEOREMES

4.1. Introduction

Cette section présente quelques notions théoriques nécessaires à la mise en oeuvre d'un système automatique de récuperation de l'information.

4.2. Clause

Une clause est une formule constituée d'une disjonction de littéraux. Mettre une formule en forme clausale correspond à la mettre en forme conjonctive normale [32].

4.3. Substituition et Unification

Une substitution est un ensemble fini $\{t_1/v_1, ..., t_n/v_n\}$, où chaque v_i $(1 \leq i \leq n)$ est une variable, chaque t_i est un terme différent de v_i et où $v_1 \neq v_2 \neq ... \neq v_n$ [12].

Soient la substitution $\theta = \{t_1/v_1, ..., t_n/v_n\}$ et l'expression E. $E\theta$ est une expression appelée instantiation de E qui est obtenue par le remplacement de chaque variable v_i de E par le terme t_i. Exemple: $\theta = \{f(a)/x, b/y, g(c)/z\}$; $E = P(x, y, z)$; $E\theta = P(f(a), b, g(c))$.

Une substitution θ est capable d'unifier un ensemble $\{E_1,E_2, ..., E_k\}$ si et seulement si $E_1\theta = E_2\theta = ... = E_k\theta$.

Un ensemble $\{E_1,E_2, ..., E_k\}$ est unifiable s'il existe une substitution θ capable de l'unifier.

4.4. Le Principe de Résolution de Robinson

Ce principe utilise comme entrée une paire de clauses C_1 et C_2, telle que C_1 et C_2 contiennent respectivement les littéraux complémentaires l et $\neg$l, et comme sortie la clause formée par les membres restants des deux clauses. Nous représenterons par C_i' la disjonction des membres restants de C_i. Une clause issue de C_1 et C_2 de cette manière est appelée résolvant de C_1 et C_2.

Formellement, la règle s'énonce ainsi:

$\underline{(C_1' \vee l) \wedge (C_2' \vee \neg l)}$,

$C_1' \vee C_2'$

où l désigne un littéral positif (p, q, r, ...) et ¬l un littéral négatif (¬p, ¬q, ¬r, ...):

Exemple:

$\underline{(\neg s \vee b \vee t) \wedge (\neg b \vee r)}$

$(\neg s \vee t \vee r)$

Si deux clauses C_1 et C_2 sont vraies, alors un résolvant de C_1 et C_2 est une conséquence logique de C_1 et C_2 [32].

Si l'on ajoute à un ensemble de clauses E la clause résolvante R, le nouvel ensemble E' obtenu de cette manière est logiquement équivalent à l'ensemble E. Cette opération d'adjonction est itérable. Elle se fonde sur le fait que la formule qui suit est valide [32]:

$((C_1' \vee l) \wedge (C_2' \vee \neg l) \equiv (C_1' \vee l) \wedge (C_2' \vee \neg l) \wedge (C_1' \vee C_2'))$

Si E est logiquement équivalent à E', E'est valide (respectivement simplement consistant, inconsistant) si et seulement si E est valide (respectivement simplement consistant, inconsistant) [32].

4.5. La Procédure de Résolution par Saturation de Niveau

Dans ce qui suit, une méthode automatique simple pour prouver qu'un ensemble de clauses est inconsistant:

Une manière directe d'effectuer la résolution sur l'ensemble E est de calculer tous les résolvants de paires de clauses dans E, d'ajouter ces résolvants à l'ensemble E, d'effectuer la résolution sur le nouvel ensemble et de répéter le processus jusqu'à ce que soit trouvée une paire de littéraux complémentaires que nous noterons **F** (le symbole **F** désigne la valeur de verité fausse) [32].

Normalement, les démonstrateurs automatiques de théorèmes sont fondés sur cette procédure de résolution. Pour démontrer une hypothèse H à partir d'un ensemble E d'axiomes vrais (sous la forme de clauses), les démonstrateurs automatiques appliquent la procédure de résolution par saturation de niveau à l'ensemble I constitué par la négation de H et par les clauses de E. Si à la fin de la procedure, une paire de littéraux complémentaires est produite, alors I est un ensemble inconsistant. Pour que I soit inconsistant, il faut que la négation de H soit inconsistante, car les autres clauses de I sont vraies. En conséquence, l'hypothèse H est vraie.

5. LA RECUPERATION DE L'INFORMATION

5.1. INTRODUCTION

Dans cette section nous présentons le démonstrateur automatique de théorèmes qui effectue la récupération de l'information, c'est-à-dire, qui répond aux questions posées au système [15, 16].

La réponse à une question posée correspond à une évaluation partielle [26, 70,77] des questions par rapport à la BC. Les réponses sont exprimées sous la forme d'expressions de la LT.

Le démonstrateur de théorèmes est fondé sur la méthode de la résolution linéaire dont le processus d'unification [12, 32] adopte des critères sémantiques de la subsomption (voir section 2.5.7).

Dans le domaine de la récupération de l'information, notre contribution concerne la mise en oeuvre d'un démonstrateur de théorèmes qui effectue un lien entre les méthodes d'inférence de la LT (la subsomption) et du Calcul des Prédicats. La subsomption est utilisée comme un critère de simplification du processus de récupération de l'information.

5.2. L'UNIFICATION SÉMANTIQUE DU DEMONSTRATEUR DE THEOREMES

Dans le processus d'unification traditionnel [12, 18, 32], deux termes g(X1, ..., Xp) et g'(Y1, ..., Yq) ne peuvent pas être unifiés si g et g' ne sont pas identiques ou si $p \neq q$.

Les limites de l'unification classique peuvent être étendues par l'introduction de critères sémantiques dans le contexte de l'unification. Par exemple, le langage PROLOG proposé par Kornfeld [18] utilise un mécanisme d'unification sémantique: tous les symboles de fonctions qui constituent une formule logique (tels que g et g') peuvent être considérés comme des fonctions définies par des relations d'égalité. Par exemple, même si les symboles g et g' ne sont pas identiques au niveau syntaxique, ils sont sémantiquement unifiables s'il existe une relation d'égalité appropriée entre eux. Normalement les relations d'égalité sont définies seulement pour les fonctions qui correspondent à des arguments d'un prédicat.

Une autre extension de l'unification classique est l'unification sous contrainte [18]. L'unification sous contrainte transforme successivement l'expression d'une question de façon à produire des transformations qui conduisent à une réponse compréhensible pour l'utilisateur du système.

Nous avons montré dans la section d'analyse syntaxique et sémantique que les exigences de notre système sont traduites par des formules du type $g(X_1/exp_1,..., X_n/exp_n)$, où $exp_1,..., exp_n$ représentent des expressions de la LT. Pour traiter ces formules, nous avons mis en oeuvre un démonstrateur de théorèmes qui combine l'unification sémantique et l'unification sous constrainte.

Nous donnons maintenant les définitions nécessaires à l'introduction de notre algorithme d'unification:

(D1) Une *entité* est une structure de données notée X/C et dotée des sélecteurs suivants:

- identifier(X/C) qui sélectione X. Les termes sélectionés par le sélecteur "identifier" peuvent être des variables ou des constantes.
- concept(X/C) qui sélectione l'expression C qui annote X..

(D2) Une liaison est une structure de données notée X/cx = Y/cy, où X est une variable. Les sélecteurs sont les suivants:

- car(X/cx = Y/cy) sélectione X/cx
- cdr(X/cx = Y/cy) sélectione Y/cy

(D3) Un environnement est un ensemble de liaisons dont les variables sélectionées par le sélecteur *car* sont distinctes. Si X est une variable et E est un environnement, alors l'expression *(assoc X E)* produit une liaison dont la variable sélectionée par le sélecteur *car* est X.

(D4) Une extension d'une expression *c* correspond à l'ensemble $\{X / \lceil c \rfloor (X)\}$. La notation de l'extension de *c* est c^e.

(D5) Une expression *c* subsume une expression *d* si $c^e \supseteq d^e$.

(D6) Si $c1 \equiv c11 \ \& \ cm1 \ \& \ cr1$ et si *cm1* subsume *c2*, alors $c11 \ \& \ cr1 \ \& \ c2$ est une simplification de $c1 \ \& \ c2$. Une autre simplification de $c1 \ \& \ c2$ est la propre expression $c1 \ \& \ c2$ (nous pouvons le démontrer au moyen de la définition de la subsomption).

(D7) La composition de deux expressions *c1* et *c2* est une simplification de l'expression $c1 \ \& \ c2$.

(D8) L'algorithme de subsomption qui suit est correct mais cependant incomplet:

subsume ≡ (Y λ sub λ C1 λ C2
(a) si C1 = X & Z alors
sub(X, C2) ∧ sub(Z, C2)
sinon
(b) si C2 = U & W alors
sub(C1, U) ∨ sub(C1, W)
sinon
(c) si C1 = R(D1) ∧ C2 = R(D2) alors
sub(D1, D2)
sinon
(d) si C1 = R(D1) ∧ C2 = U & W alors
sub(C1, U) ∨ sub(C1, W)
sinon faux),

où Y est l'opérateur de point fixe [57]. Nous pouvons remarquer que (a), (b), (c) et (d) satisfont la définition de subsomption.

(D9) La valeur finale d'une entité est définie par l'algorithme qui suit:

fvalue ≡ (Y λ fv λ X λ E
let V/CV = assoc(identifier(X), E) dans
si V/CV = ⊥ alors X
sinon
fv(V/compose(concept(X), CV), E),

(D10) La référence d'un littéral G dans un environnement E correspond à l'expression G' obtenue à travers du remplacement de chaque argument de G par la valeur finale de cet argument dans l'environnement E.

(D11) Considérons G et H deux littéraux dont les listes d'arguments ne comportent pas des fonctions. L'unification de G et de H dans l'environnement E correspond à la substitution (voir section 4.3) la plus générale E' [12, 32] telle que G' = H'. G' et H' correspondent, respectivement, à la référence de G et H dans l'environnement E.

L'algorithme d'unification doit être capable de stocker dans un même environnement tous les concepts qui correspondent aux annotations des variables des littéraux qu'il a unifiés. Les concepts doivent être organisés dans l'environnement de telle façon que l'algorithme de la valeur finale puisse les saisir.

Dans notre système d'unification, l'unification de deux variables X et Y est effectivée selon la méthode de l'unification classique [12, 32]. En conséquence, tous les critères de l'unification classique (tel que la complexité et la correction) sont valables dans notre algorithme d'unification. Le processus de composer et de stocker les concepts cx et cy est concourant du processus d'unification. En d'autres termes, pour vérifier que X/cx s'unifie avec Y/cy, l'algorithme d'unification doit faire une recherche dans l'environnement qui emmagasine les liaisons qui ont déjà été produites. L'algorithme suit la chaine des liaisons qui commence par X/cx jusqu'à ce qu'il trouve une variable qui figure à titre d'occurence libre ou une constante. L'algorithme saisit, compose et simplifie les concepts qui sont associés aux éléments de la chaine de liaisons. Appelons vx le résultat de cette opération. La valeur finale de X est FX/vx, où FX est sélectionné par l'application du sélecteur *identifier* à la dernière liaison de la chaine. De la même façon, l'algorithme essaie d'obtenir la valeur finale de la variable Y. Soit FY/vy la valeur finale de Y/cy. L'algorithme essaie d'unifier FX et FY et de composer vx et vy selon les notions de la LT (la subsomption, la disjonction, la simplification etc). Si la composition de cx et cy n'est pas possible, l'unification n'est pas possible. Si la composition est possible, la variable X sera annotée par une simplification de vx & vy. Dans ce cas, après l'unification, la variable X sera soumise aux restrictions de Y aussi bien qu'à ses propres restrictions. Le résultat de l'unification de X/cx et de Y/cy correspond à l'introduction, dans l'environnement, de la nouvelle liaison entre les valeurs finales de X et de Y. Cette liaison peut être considerée comme une contrainte. L'environnement représente alors l'ensemble de toutes les contraintes que les variables unifiées doivent satisfaire.

L'algorithme d'unification peut s'écrire:

```
unifie ≡ λ G λ H λ E
  si E = ⊥ alors E
  sinon
    si  G = ¬Gp ∧ H = ¬Hp ∧ foncteur(Gp) =
       foncteur(Hp) ∧ arité(Gp) = arité(Hp) alors
        unifieargs(Gp, Hp, 1, E)
    sinon
        si foncteur(G) = foncteur(H) ∧ arité(G) =
           arité(H) alors
             unifieargs(G, H, 1, E)
        sinon ⊥

unifieargs  ≡ (Y λ uniargs λ G λ H λ I λ E
  si I > dernier-argument-de(G) alors
   E
  sinon
   si E = ⊥ alors
    ⊥
   sinon
    uniargs(G,H, I+1, unifiearg(fvalue(argument I de G, E), fvalue(argument I of H,
          E),E)))

unifiearg ≡  λ X  λ Z λ E
  si X =  ⊥ ou  Z =  ⊥  then  ⊥
  sinon
   fais VX/CX = X and W/CW= Z dans
   si occur(X,Z) alors
    ⊥
   sinon
    si variablep(VX) alors
     [X = Z|E]
    sinon
     si  variablep(W) alors
      [Z = X | E] sinon ...
```

Le prédicat arité(G) produit le numéro qui corresponde à la quantité d'arguments de G.

Nous remarquons que, à l'exception des annotations, cet algorithme n'est pas différent de l'algorithme traditionnel d'unification [12,32] dont la démonstration a déjà été faite. Nous prouvons que les annotations sont saisies de façon correcte si, en parcourant l'arbre sémantique qui est construit pendant le processus de récupération de l'information (processus de preuve), nous verifions que le processus d'unification construit une chaine de liaisons que l'algorithme de la valeur finale est capable de suivre.

5.3. Mise en Oeuvre de l'Algorithme de Récupération de l'Information

Dans cette section nous allons présenter la notion d'évaluation partielle et montrer les avantages des résultats obtenus grâce à l'analyse syntaxique et sémantique dans le contexte de la récupération de l'information. A la fin de cette section, le démonstrateur de théorèmes de notre système est présenté.

5.3.1. Evaluation Partielle

Selon Wittgeinstein [76], la complexité logique est la proprieté d'une proposition d'être successivement decomposée en des propositions moins complexes que la proposition originale jusqu'à ce que soient obtenues seulement des propositions élémentaires (dépourvues de complexité logique).

Lorsqu'une question est posée à un système de représentation de la connaissance, la réponse produite doit être accessible à la personne qui a posé la question. En d'autres termes, la personne qui a posé la question doit avoir les moyens de reconnaître les concepts qui sont representés par les propositions de la réponse. En plus, elle doit avoir les moyens de faire des inférences à partir de toutes les propositions de la réponse qui sont dotées de complexité logique.

Une donnée est une formule dépourvue de complexité logique à partir de laquelle il ne peut plus être fait d'inférences. Une évaluation totale est caractérisée par le fait qu'elle produit une donnée comme réponse.

Une évaluation partielle est caractérisée par le fait qu'elle produit une réponse dotée de complexité logique, c'est-à-dire, une réponse à partir de laquelle nous pouvons faire d'autres inférences. Une description formelle des évaluations partielle et totale est presentée dans ce qui suit:

Un système de représentation de la connaissance (krs) est composé d'un ensemble récursif [32] FS = D $\cup$ P de formules et d'un ensemble IRs de règles d'inférence. Les formules du sous-ensemble récursif D de FS sont appelées données. Les formules du sous-ensemble récursif P de FS sont appelées axiomes. Comme D et P sont des ensembles récursifs, ils peuvent être décrits, respectivement, par des ensembles de règles de production Dp et Pp.

Une formule a des constituants seulement dans P si elle peut être décrite par l'ensemble Pp - (Dp $\cap$ Pp).

Considérons le système de représentation de la connaissance krs(D, P, IRs). Une inférence est une déduction obtenue à partir d'un axiome appelé question (ou hypothèse). Le dernier élément de la déduction est appelé réponse.

Si une réponse est une formule qui appartient à l'ensemble {Dp - (Dp $\cap$ Pp)}, elle est appelée évaluation totale. Une évaluation totale est caractérisée par le fait qu'elle produit une donnée comme réponse.

Si une réponse est une formule qui appartient à l'ensemble {Pp - (Dp $\cap$ Pp)}, elle est appelée évaluation partielle [26, 77]. Une évaluation partielle est caractérisée par le fait qu'elle produit comme réponse un axiome à partir du quel nous pouvons effectuer d'autres déductions. Généralement, les évaluations partielles sont utilisées pour augmenter l'efficacité des inférences totales, puisqu'une évaluation partielle permet qu'une évaluation totale ait un point de départ plus avancé. L'efficacité est le principal but de techniques telles que l'évaluation partielle et l'Apprentissage Basée Sur l'Explication (EBL) [36].

Dans notre système, nous utilisons l'évaluation partielle pour obtenir des réponses qui correspondent à des formules utiles au niveau conceptuel. En d'autres termes, les formules des réponses doivent introduire de nouveaux concepts aussi bien qu'identifier des concepts déjà connus. Ces formules sont décrites par un ensemble Sp de règles de production. Comme les réponses doivent correspondre à des évaluations partielles, Sp doit être un sous-ensemble de {Pp - (Dp $\cap$ Pp)}. Comme nous l'avons déjà dit au début de cette section, les réponses qui appartiennent à Sp doivent être accecibles à la personne qui a posé les questions. Nous remarquons que les expressions de la LT sont appropriées pour exprimer les réponses du système. Par conséquent, nous choisissons Sp dans le contexte de la LT.

5.3.2. Avantages du Résultat de l'Analyse Syntaxique et Sémantique au Niveau de la Récupération de l'Information

Les annotations des variables sont des expressions de la LT qui peuvent être considérées comme des prédicats que ces variables doivent satisfaire. Dans ce cas, les résultat des démonstrations de formules telles que *F1* et *F2* qui suivent sont équivalentes:

F1: τ(X1, ioi-gs(X1)
τ(x2, τ(X3, space-vehicle(X3)
computer(X2) &
on-board(X2, X3))
control(X1,X2)))

F2: control(X/ t(ioi-gs), X3/ t(computer & on-board(space-vehicle))).

Ces résultats sont équivalents parce que démontrer que *r(X1/c1, X2/c2, X3/c3, ...) & ...* est vrai, en garantissant que *(⌈c1⌋ X1), (⌈c2 ⌋ x2), (⌈c3 ⌋ X3...) & ...* sont tous vrais, est la même chose que démontrer que *r(X1, X2, X3, ...) & c1(X) & c2(X2) & c3(X3) &....* est vrai. Néanmoins, la première démonstration peut être effectuée plus efficacement, car elle utilise moins de ressources qu'un démonstrateur de théorèmes du Calcul des Prédicats. C'est ici que reside l'avantage des résultats de notre analyseur syntaxico-sémantique.

C'est par l'analyse syntaxique et sémantique que les questions posées au système et les exigences sont transformées en des conjonctions de littéraux du type [P, r(X1/C1, X2/C2, ...), Q,...], qui sont équivalentes à des conjonctions du type [P, r(X1, X2, ...), C1(X), C2(X2), ..., Q]. La séquence des résolutions effectuées durant une preuve effacera toutes les litteraux d'une question (hypothèse) à l'exception des annotations. Ces annotations correspondent à la réponse (ou à l'évaluation partielle de l'hypothèse par rapport à la BC).

5.3.3. Démonstrateur de Théorèmes

Toutes les liaisons produites par l'algorithme d'unification durant une démonstration sont emmagasinées dans un environnement. A la fin du processus d'unification, nous obtenons un environnement final. Le démonstrateur de théorèmes répond à une question en saisissant dans l'environnement final toutes les annotations qui correspondent à la variable de l'hypothèse (c'est-

à-dire, la valeur finale de cette variable dans l'environnement final). Ces annotations appartienent à l'ensemble {Pp - (Dp ∩ Pp)}, c'est-à-dire, elles correspondent à des expressions de la LT.

Récupérer l'information dans notre système correspond à spécialiser le plus possible les variables des hypothèses à partir des informations qui sont stockées dans la BC. Par exemple, supposons que la BC contienne les clauses:

controls(e0/ev, X1'/ioi-gs, Y1'/space-vehicle)] &
ag(e0/ev, X1'/ioi-gs)] &
pac(eo/ev, Y1'/space-vehicle),

qui correspondent à l'exigence: *"The ioi-gs shall control the space-vehicle"*. Supposons aussi que soit introduite l'hypothèse:

[control(e0/ev, X1/who, Y1/space-vehicle)] &
[ag(e0/ev, X1/who)] &
[pac(eo/ev, Y1/space-vehicle)],

qui correspond à la question *"Who controls the space vehicle?"*. A la fin de la preuve, le démonstrateur de théorèmes va essayer de saisir dans l'environnement final la valeur finale de la variable X1, i.e:

?- fvalue(X1/who & ioi-gs, [Y1/space-vehicle = Y1'/space-vehicle,
X1/who & ioi-gs = X1'/ioi-gs,
Y/space-vehicle = Y1/space-vehicle,
X/who = X1/ioi-gs]).

La réponse est donc l'annotation finale de la variable X1, c'est-à-dire, l'expression de la LT *who & ioi-gs*.

Le démonstrateur de théorèmes qui a été mis en oeuvre est un démonstrateur de théorèmes fondé sur la résolution linéaire [12] dont l'algorithme d'unification suit les critères sémantiques de la subsumptiom (LT). La subsomption simplifie le processus de preuve en guidant l'algorithme d'unification. Les réponses produites par le démonstrateur corrspondent à des évaluations partielles des questions par rapport à la BC. Par exemple, si nous demandons au système:

?- control(X/ioi-gs, Y/C)

et si la réponse produite est *Y/ computer & on-board(space-vehicle)*, nous pouvons comprendre que le ioi-gs doit controler tous les ordinateurs qui sont à bord du véhicule spatial. Même si le système n'a pas assez d'information pour produire comme réponse une liste des ordinateurs qui sont controlés par le véhicule spatial, il peut, à travers d'une évaluation partielle, répondre qu'ils doivent satisfaire les prédicats: *λ(X)computer(X)* et *λ(Y)on-board(Y, space-vehicle)*.

Dans ce qui suit, nous présentons une version simplifiée de l'algorithme du démonstrateur de théorèmes.

```
prouve(G, A) :- on(G, A).
prouve(G, A) :- sent(G, B),
                neg(G, NG),
                not(on(NG, A)),
                cpro(B, [NG|A]).
cpro([], _).
cpro([G|GS], A) :- neg(G, NG),
                   prouve(NG, A),
                   cpro(GS, A)
sent(G, T) :-    saisit-clause(C),  contre-posit(C, G1, T), unifie(G, G1).
```

où le prédicat *prouve* essaie de démontrer une hypothèse G. Le prédicat *sent* cherche dans la BC une clause C qui soit appropriée à la résolution de G. Le prédicat *contre-posit* essaie de trouver en C un littéral qui s'unifie avec l'hypothèse G.

CONCLUSION

Dans le cadre de cette thèse, nous avons mis en oeuvre un système hybride de représentation de connaissance qui représente une liaison entre les ressources du Calcul de Prédicats et de la Logique Terminologique. Ce système est destiné au traitement de contextes linguistiques dont les caractéristiques sont similaires à celles des spécifications de logiciel. Le système est capable de:

- produire une représentation formelle d'exigences exprimées en Langage Naturel dans le cadre de la spécification du logiciel.

- introduire la représentation formelle obtenue pour chaque exigence dans une base de connaissances en prenant soin de détecter les possibles redondances et contradictions.

- répondre à des questions posées au système à travers de l'exécution d'un mécanisme d'inférence permettant la récupération de l'information stockée dans la base de connaissances.

La représentation formelle d'une exigence est produite par un analyseur syntaxique et sémantique. Elle correspond à une formule du Calcul des Prédicats mise sous forme normale de Skolem. Les variables de cette formule sont annotées par des expressions de la LT qui imposent des contraintes sur les variables qu'elles annotent.

L'analyseur est une grammaire construite selon la théorie du structuralisme.

Notre contribution au niveau de l'analyse concerne la mise en oeuvre d'un analyseur capable:

- d'engendrer automatiquement un ensemble de règles sémantiques, ce qui dispense le linguiste de la tâche de les définir.
- de produire une représentation formelle pour les exigences dont partie terminologique est utilisée dans la simplification de la BC (à travers des critères de la subsumption).

La récuperation de l'information est obtenue par un démonstrateur automatique de théorèmes du Calcul de Prédicats qui répond aux questions en évaluant partiellement les questions par rapport à la BC. Le mécanisme d'inférence du démonstrateur est guidé par la sémantique de la subsumption (LT).

Nos contributions au niveau de la récuperation de l'information sont les suivantes:

- la mise en oeuvre d'un démonstrateur automatique de théorèmes qui effectue un lien entre les méthodes d'inférence du Calcul des Prédicats et de la LT. Ce lien permet une simplification du mécanisme de preuve.
- la capacité du démonstrateur de théorèmes d'effectuer des évaluations partielles lui permet de fournir des réponses conceptuellement utiles même s'il ne dispose pas de toutes les informations necessaires pour produire une évaluation totale.

Nous avons mis en oeuvre un logiciel dans lequel nous avons intégré tous les algorithmes composent le système de représentation de la connaissance décrit dans le cadre de cette thèse. Les résultats montrent que les techniques utilisés sont appropriées au traitement de contextes linguistiques dont principal problème est la référence de Hilbert (tel que le contexte des spécifications de logiciel).

BIBLIOGRAPHIE

[1] Abas, C.A.A.P, *Descrição e Paraconsistência*, thèse de doctorat, PUC SP, Brésil, 1985.

[2] Allen, J. *Natural Language Understanding*, USA, The Benjamin/Cummings Publishing Company, Inc., 1987.

[3] Árabe. A.M.F., Pereira, A.C., Testa, T.A.R., *Communicating with Databases in Natural Language: A Proposal of Simplify Interfaces*, Int.Conf. IEEE- SMC Vancouver, 1995.

[4] Aragon,D., Castro, I., Alcoforado, P., *Lógica Para Informática- Um Estudo Introdutório*, ILTC- Instituto de Lógica e ilosofia e Teoria da Ciência, Edição Prévia.

[5] Araribóia, G., *Inteligência Artificial- Um Curso Prático*, ILTC, 1989.

[6] Barros, M.P., *Introdução Automática de Controle em Programação Lógica*, thèse de master, UFU, Brésil, 1991.

[7] Beaugrande, R. & Dressler, M.U., *Introduction to Text Linguistics[trad.]*.London, Longman, 1981.

[8] Bittencout, G., *The Integration of Terminological and Logical Knowledge Representation Languages*, Elsevier Science Publishing Co., Inc. 1990.

[9] Borrajo, D., Veloso, M., *Incremental Learning of Control Knowledge for Nonlinear Problem Solving*, in Proceedings of the European Conference on Machine Learning, 1994.

[10] Bourbaki, N. *"L'architecture des mathématiques"*, in F. Le Lionnais, Les grands courants de la pensée mathématique, Paris, 1948.

[11] Chambreuil, M., *Grammaire de Montague*, Éditions ADOSA, B.P. 467, 1989.

[12] Chang,C. and Lee, R., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York San Francisco London , 1973.

[13] Clark, K.L, Tärnlund, A., (editores), *Logic Programming* - Academic Press inc. (London) Ltd, 1982.

[14] Da Silva, R.M., Pereira, A.C., Netto, M.L.A.: *A System of Knowledge Representation Based on Formulae of Predicate Calculus Whose Variables are Annotated by Expressions of a "Fuzzy" Terminological Logic*, Lecture Notes in Computer Science (n. 945), pp. 409-417 - Springer-Verlag, editée par Bernadette Bouchon-Meunier, Ronal R. Yager e Lotfi A. Zadeh. Egalement dans la conférence "International Conference on Information Processing and Management of Uncertainty in knowledge-Based Systems", IPMU 1994, Paris, France, 4-8 Julho 1994.

[15] Da Silva, R.M., Pereira, A.C., Netto, M.L.A.: Information Retrieval by Means of a Semantic Unification Guided by Resources of Terminological Logic, International Conference IEEE- SMC (Systems, Man and Cybernetics), San Antonio, Texas, U.S.A., Octobre 1994.

[16] Da Silva Julia, R.M., Seabra, J.R., Sameghini, I.: *An Intelligent Parser that Automatically Generates Semantic Rules During Syntatic and Semantic Analysis*, "International Conference IEEE- SMC (Systems, Man and Cybernetics)", Vancouver, Britsh Columbia, Canada, 22-25 Octobre 1995, pp. 806-811.

[17] Davidson, D., *Verdade e Significado*, in Fundamentos Metodológicos de Lingüística, Marcelo Dascal - Vol. III.

[18] DeGroot, D. and Lindstrom, G., *Logic Programming* - Functions, Relations, and Equations - Prentice-Hall, Englewood Cliffs, New Jersey 07632.

[19] DeJong, G. F. and Gratch, J., Steve Minton, *Learning Search Control knowledge: An Explanation Based Approach*, in: Artificial Intelligence 50 (1991) 117 - 127.

[20] DeMarco 89 DeMarco, T., *Structured Analysis and System Specification*. Prentice-Hall, Youdon Press, Englewood Cliffs, 1979.

[21] Desclés, J.P., *Langages applicatifs langues naturelles et cognition*- Hermès, Paris, 1990.

[22] Donini, F.M., Lenzerini, M. and Nardi, D., *The Complexity of Existential Quantification in Concept Languages*, in: Artificial Intelligence 53 (1992) 309-327.

[23] Doyle, J. and Patil, R.S., *Two theses of knowledge representation: language restrictions, taxonomic classification, and the utility of representation services*, in: Artificial Intelligence 48 (1991) 261 - 297.

[24] Dowty, D., Wall, R. and Peters, S., *Introduction to Montague Semantics*, D.Reidel Publishing Company, 1985.

[25] Dubois, D.and Prade, H., *Théorie des Possibilités*, MASSON, 1985.

[26] Ershov, A.P., *Mixed Computation: Potential Applications and Problems for Study*, in: Theoretical Computer Science 18(1982) 41-67.

[27] Etzioni,O., *A structural theory of explanation -based learning*, in: Artificial Intelligence 60 (1993) 93-139.

[28] Fillmore, C.J., *The Case For Case*, in 'Universals in Linguistic Theory', Brech&Harms (eds), Hoet, Reinhart e Win. New York, 1968.

[29] Gazdar, G., *Phrase Structure Grammar*, in: Nature of syntatic representation, Jacobson et Pullum, Dordrecht: Reidel, 1979, pp 131-186.

[30] Gazdar, G., Klein, E., Pullum, G. and Sag.I, *Generalized Phrase Structure Grammar*, in: Harvard University Press, Cambridge, 1985.

[31] Gazdar, G. and Mellish, C.S., *Natural Language Processing in LISP*.

[32] Gochet, P. Gribomont, P., *Logique-Méthodes pour l'informatique fondamentale* - Hermes, Paris, 1990-1991.

[33] Grover,C., Briscoe, E., Carrolis, G. and Boguraev, *The Alvey Natural Language Tools Grammar, in: University of Cambridge*, Computer Lab.

[34] Guillaume, M., *Recherches sur le Symbole de Hilbert-* Clermont-Ferrand, 1960.

[35] Halliday, M.A.K. & HASAN, R. - *Cohesion in English.* London. Longman, 1976.

[36] Harmelen, F. and Bundy, A., *Explanation-Based Generalisation = Partial Evaluation,* in: Artificial Intelligence 36 (1988) 401-412.

[37] Hoey, M., *Patterns of Lexis in Text.* Oxford University Press, 1991.

[38] Hovy, E.H., *Pragmatics and natural Language Generation,* in: Artificial Intelligence 43 (1990) 153-197.

[39] [IEEE 83] IEEE, *An American National IEEE Standard Glossary of Software Engineering Terminology* n ° ANSI/IEEE Std 729-1983, 1983.

[40] [IEEE 84] IEEE *Guide to Software Requirements Specifications* - ANSI/IEEE Std 830 - 1984, IEEE, 1984.

[41] Jones,S.L.P., *The Implementation of Functional Programming Languages.*

[42] Keller, R., *Defining Operationality for Explanation-Based Learning,* in: Artificial Intelligence 35 (1988) 227-241.

[43] Kindermann, C., *Retraction in Terminologcial Knowledge Base,* ECAI 92, 10th European Conf. on Artificial Intelligence Edited by B. Neumann Published in 1992 by John Wisley & sons, Ltd.

[44] Leisenring, A.C., *Mathematical Logic and Hilbert's ε Symbol,* London MacDonald Technical & Scientific, 1969.

[45] Lewis, D., *General Semantics,* In Philosophical Papers I, Oxford University Press.

[46] Minsky, M., *A framework for representing knowledge.* In: WINSTON, P. (org.). The Psychology of Computer Vision. New York, McGraw-Hill, 1975.

[47] Minton,S., Carbonell, J.G., Knoblock, C.A., Kuokka, R., Etizioni, O., and Gil, Y., *Explanation Based Learning: A Problem Solving Perspective,* in:Artificial Intelligence 40 (1989) 63-118.

[48] Muller, A.L., *Ao Projeto Lingüistico da Gramática de Montague,* GEL XXXIX seminaire.

[49] Napoli, A., *Subsumption and Classification-Based Reasoning in Object-Based Representations,* in ECAI 92.

[50] Nebel, B., *Terminological Reasoning is Inherently Intractable,* Artificial Intelligence 43 (1990) 235-249.

[51] Nebel, B., *Reasoning and Revision in Hybrid Representation-* Lectures Notes in Artificial Intelligence n.422.

[52] Nillson,N., *Principles of Artificial Intelligence,* Tioga Publishing Co; Palo Alto, Californie, 1980.

[53] Patel-Schneider, P.F., *A Four-Valued Semantics for Terminological Logics,* in:Artificial Intelligence, Vol. 39, 399.

[54] Patel-Schneider, P.F., *Undecidability of Subsumption in NIKL*, in: Artificial Intelligence 39 (1989) 263-272.

[55] Pereira,F.C.N., *Incremental Interpretation*, in: Artificial Intelligence 50 (1991) 37-82.

[56] Petöfi, J. *,Semantics, Pragmatics, Text Theory*. Università di Urbino, Worling Papers, A. n36, 1974.

[57] Peyton Jones,S.L., *The Implementation of Functional Programming Languages*, Prentice Hall.

[58] Piaget, J., *Le Structuralisme*, Coll. " Que sais-je?", n. 1311- Copyright by Presses Universitaires de France, Paris- 1970.

[59] Porter, B.X., *Concept Learning and Heuristique Classification in Weak-Theory Domains*, in:Artificial Intelligence 45 (1990) 229-263.

[60] Pressman, R.S., *Software Engineering- A Practitioner's Approach-* Mcgraw-Hill, 1992.

[61] Robinson, J.A., *Computational logical: the unification computation*, "Machine Intelligence", Vol.6(B.Meltzer and D.Michie,eds.), American Elsevier, New York, pp 63-72, 1971a.

[62] Rumelhart, D.E., *Schemata: The building blocks of coginition. In: SPIRO et akuu (orgs.) Theoretical issues in Reading Comprehension*. New Jersey, Lawrence Earlbaum Asso, 1980.

[63] Schank, R., *The Cognitive Computer on Language, learning and artificial intelligence*. Addison-Wesley Publishing Company, Inc., Reading, 1984.

[64] Schidt-Schaub, M., *Subsumption in KL-ONE is Undecidable*, in R.J. Brachman, H.J. Levesque & Reiter, editions, Proceedings of the Ist International Conference on Principles of Knowledge Representattion and Reasoning, pages 421-431, Toronto, Canada, 1989.

[65] Schimidt, S.J., *Textheorie. Probleme einer Linguistik sprachlichen Kommunikation*, 1973.

[66] Shesov, S.D. and Myasnikov, A.G., *Logical-Semantic Structure of a Terminology and its Formal Properties*, in: Nauchno-Teckhnicheskaya Informatsiya, Seriya 2, Vol. 21, N.3, 1987.

[67] Siwek, P., *Psychologia Experimentalis-* Marietti Editori Ltd, Italy, 1958.

[68] Steedman, M., Categorial Grammar, *Lingua*, 90 (1993) 221-258, North Holland.

[69] Strawson, P.F., *Significado e Verdade*, in Fundamentos Metodológicos de Lingüistica, Marcelo Dascal- Vol III.

[70] Takeuchi, A, Furukawa, K., *Partial Evaluation of Prolog Programs and its Applications to Meta Programming*, ICOT Tech. Report, 1985.

[71] Tarski, A., *La Concepción semántica de la verdad y los fundamentos de la semántica*, Ediciones Nueva Visión, 1972.

[72] Tjandra, J.C., Bittencourt, G., *Mantra.: A Shell for Hybrid Knowledge Representation*, Proc. of the 1991 IEEE, International Conference on Tools for AI, San Jose, Ca- Nov 1991.

[73] Toussain,Y., *Methodes Informatiques et Linguistiques Pour L'aide a la Specification de Logiciel*, thèse de doctorat d'université, Université Paul Sabatier Toulouse, France, 1992.

[74] Vilain, M., *The Restricted Language Architecture of a Hybrid Representation System*, Proceedings IJCAI-85, Los Angeles, CA, 1985.

[75] Westerstâl,D., *Quantifiers in Formal and Natural Languages*, in: Handbook of Philosophical Logic, Gabbay, D. et Guenthner, F.,Eds.D.Reidel Publ. Co., 1989, chap.1, pp 1-131.

[76] Wittgeinstein, L., *Tratactus Logico-philosophicus*- Éditions Gallimard, 1993.

[77] Yamaki, C.K., *Combinando Avaliação Parcial e Introdução Automática de Controle para Aumentar a Eficiência de Sistemas Especialistas*, thèse de msater, UNICAMP, Brésil, 1990.

[78] Yamashita, M. *O Símbolo ε de Hilbert em Lógica Paraconsistente*, Thèse de doctorat, PUC-SP-Brésil, 1985.