SIMONE NASSER MATOS FERREIRA

ESPECIFICAÇÃO FORMAL E IMPLEMENTAÇÃO DE UM PROTÓTIPO PARA A LINGUAGEM *PARALOG*

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

Orientador: Prof. Martin A. Musicante

CURITIBA
2001

UFPR
Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática da aluna ***Simone Nasser Mattos Ferreria***, avaliamos o trabalho intitulado ***"Projeto e Implementação de Protótipo para a Linguagem PARALOG",*** cuja defesa foi realizada no dia 26 de julho de 2001, às treze horas, na sala de vídeo do departamento de Informática da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação da candidata.

Curitiba, 26 de julho de 2001.

Prof. Dr. Martin Alejandro Musicante
DINF/UFPR - Orientador

Prof^a. Dra. Mariza Bigonha
DCC/UFMG

Prof. Dr. Bráulio Coelho Ávila
PUC/PR

Prof. Dr. Michel Gagnon
DINF/UFPR

A Deus,

fonte inesgotável de amor.

Às minhas amigas,

Maria e Dalmi, in memoriam.

Aos meus pais,

João e Sônia, amores inigualáveis.

Ao meu marido,

Marcos, pela dedicação, apoio e carinho.

AGRADECIMENTOS

Ao ser supremo, Deus, que nos dá proteção, saúde e resistência.

À minha família, que apoiou e incentivou esta etapa da minha vida, a de realização do mestrado.

Ao professor orientador Martin A. Musicante, que muito mais que um orientador, foi amigo paciente e dedicado, que como uma luz no meio das trevas, guiou-me em toda esta jornada.

Aos professores Décio Krause e Bráulio Coelho Ávila, pelos comentários, discussões, apoio e orientações que contribuiram para o enriquecimento deste trabalho.

Aos professores Wagner Machado Nunan Zola e Bráulio Coelho Ávila, pelas observações e contribuições na banca de qualificação.

Ao professor Jair Minoro Abe, pela indicação de alguns materiais sobre lógica paraconsistente.

Aos professores do Departamento de Pós-Graduação em Informática, pelas contribuições nas disciplinas ministradas.

Aos colegas da Coordenação de Informática do CEFET-PR Unidade de Ponta Grossa, pela oportunidade concedida.

Aos dirigentes do Centro Federal de Educação Tecnológica do Paraná - Unidade de Ponta Grossa, que me concederam licença para cursar o Mestrado, no primeiro ano, em regime parcial.

À amiga Glória Silva, pelas horas divididas e pelo transporte com destino à rodoviária.

Aos colegas de viagem Clodis Boscarioli, Geraldo Ranthum, Ezequiel Gueber, Ivo Mario Mathias e Mônica H. Pietruchinki, pelas horas passadas juntas com destino à Universidade Federal do Paraná.

A todos que direta ou indiretamente contribuíram para a conclusão deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

ASF	—	Algebraic Specification Formalism
BNF	—	Backus-Naur Form
HG	—	Horn Generalizado
GIPE	—	Generation of Interative Programming Environments
INRIA	—	Institut National de Recherche en Informatique et en Automatique
nh	—	não-Horn
ParaLog	—	Prolog Paraconsistente
ParaLog_e	—	Prolog Paraconsistente Estendido
PHG	—	Programa Horn Generalizado
Prolog	—	Programação Lógica
SIS	—	Structural Inductive Semantics
SLD	—	Resolução Linear com Função de Seleção para as Cláusulas Horn
SLD-nh	—	Resolução Linear com Func. de Seleção para as Cláusulas não-Horn
SDF	—	Syntax Definition Formalism
TransLog	—	Transformação de Programas Lógicos
UMG	—	Unificador mais geral

RESUMO

O formalismo de semântica de transição define o significado dos programas de uma linguagem por meio de *regras indutivas*. Esta característica faz com que ele possa ser usado tanto no projeto quanto na implementação de protótipos de linguagens de programação. Essas regras podem ser implementadas no meta-ambiente ASF+SDF, o qual permite o desenvolvimento e geração automática de sistemas interativos que manipulam programas, especificações ou outros textos escritos em uma linguagem formal, gerando dessa forma um interpretador para a linguagem. Com base em tais pressupostos, desenvolvemos uma especificação formal para a linguagem ParaLog, utilizando semântica de transição. O protótipo de um ambiente operacional ParaLog, implementado no ASF+SDF, funciona como um interpretador para programas paraconsistentes capaz de raciocinar na presença de uma base de conhecimento que contenha inconsistências.

Palavras-chave: semântica de transição, especificação formal, ASF+SDF, formalização de programas paraconsistentes.

ABSTRACT

Transition semantics is useful for the definition and construction of interpretors, as it allows the syntactic and semantic definition of a programming language by means of inductive rules. These rules can be implemented in the meta-environment ASF+SDF, which makes possible the development and automatic generation of interactive systems that manipulate programs, specifications or other texts written in a formal language, thus generating an interpreter for the language. On the basis of such assumptions, we develop a formal specification for the ParaLog language using transition semantics. The prototype of a ParaLog operational environment, implemented in ASF+SDF, is an interpreter for paraconsistent programs which is capable of reasoning in the presence of a knowledge base containing inconsistencies.

Key words: transition semantics, formal specification, ASF+SDF, formalization of paraconsistent programs.

CAPÍTULO 1

Introdução

Na atualidade, lógicas paraconsistentes [CSV89, CM87, CS89] têm sido alvo de estudos de pesquisa, principalmente no campo da ciência da computação, no âmbito da inteligência artificial. Estudos nessa área podem ser encontrados em: Inteligência Artificial Distribuída [PA98], Lógica Modal e Multimodal [AA98, BP93, Sak92, APA97], Linguagens de Programação Lógica Paraconsistente [BS88, BS87, Sub87b], Robótica [FAT+99], Provadores de Teoremas [CHLS90], entre outros [AF98, NAS99].

O uso da programação lógica paraconsistente abre novos temas de pesquisa, pois reúne conceitos de programação lógica clássica com os conceitos de inconsistência, paracompleteza ou ambos, visando ao tratamento de incertezas. Um exemplo de linguagem de programação paraconsistente é o ParaLog [CPA+95], que utiliza conceitos de lógica paraconsistente para o tratamento de conflitos de crença em suas bases de conhecimento.

No que se refere às linguagens de programação, vários métodos têm sido propostos para sua descrição formal. Entre eles, os métodos denotacional e operacional são os mais populares.

A Semântica Denotacional [Wat91, Win93] tem como característica principal definir o significado dos programas com funções matemáticas. O significado de cada frase é definido em termos dos significados de suas sub-frases.

A Semântica Operacional [Win93, Ast91] tem por objetivo declarar "como" uma frase é executada. Pode descrever transformações sintáticas que imitam a execução do programa em uma máquina abstrata ou definir uma relação entre o programa e os seus possíveis resultados [Mus96].

A característica comum de todos os formalismos operacionais é a definição indutiva de uma relação de transição [Plo81, Ast91].

Conforme apresentado em [BHK89], a implementação da semântica operacional de uma linguagem pode ser feita por meio de vários sistemas de software como: Lex [LS86], Yacc [Joh86], Metal (linguagem de especificação do sistema Mentor [KLMM83]), SSL (especificação da linguagem Synthesizer Generator [Rep82, RT89]), PSG (especificação da linguagem do sistema PSG [BS86]) e SDF [Kli92, HHKR92]. Em [HK89] encontramos a comparação entre SDF com Lex, Yacc e Metal.

Nessa comparação destaca-se o formalismo dado pelo SDF [HHKR92] que é um formalismo de definição sintática comparável à BNF em alguns aspectos, mas possui um amplo escopo para definição da sintaxe léxica e abstrata. Além disso, as definições sintáticas para Lex / Yacc / Metal e SSL são restritas para gramática LALR, enquanto SDF permite gramáticas livres de contexto arbitrárias. Portanto, em SDF:

- há uma melhor integração entre sintaxe léxica e sintaxe livre de contexto;
- nomes de tokens são, por exemplo, gerados automaticamente;
- é fixada correspondência entre regras de sintaxe livre de contexto e regras de sintaxe abstrata.

Podemos observar que o formalismo do SDF apresenta várias vantagens e é por isso que neste trabalho usaremos seus conceitos para desenvolver as regras de semântica de transição da linguagem ParaLog. A implementação das regras será feita no meta-ambiente ASF+SDF [Kli92, BK00], que utiliza os conceitos do SDF para gerar um interpretador para linguagens de programação.

Muitas vezes, ao escrever em uma linguagem de programação, o programador deseja representar suas crenças e não verdades absolutas [BS88]. Ele pode, por exemplo, escrever certas cláusulas porque acredita que elas sejam verdadeiras e não porque elas sejam obrigatoriamente verdadeiras. Por meio do ParaLog isso pode ser representado de maneira simples, uma vez que o mesmo permite a definição de constantes anotacionais [Abe92, Abe93, CAS91] e da resolução-SLDnh[1] [BS87].

Desenvolver um ambiente para uma linguagem de programação paraconsistente amplia o escopo da aplicação da programação lógica em ambientes onde ocorrem conflitos de

[1] Os referidos conceitos serão definidos no capítulo 4.

crença e informações contraditórias, uma vez que, em linguagens de programação clássica, esses conflitos são difíceis de ser tratados.

Dessa forma, nossas principais contribuições serão:

- especificação formal da linguagem ParaLog, usando semântica de transição;
- uso da especificação acima para gerar um ambiente de programação.

Cabe destacar que, apesar de existirem outras descrições (inclusive formais) de ParaLog, a definição aqui proposta é a primeira definição formal, usando semântica de transição da linguagem em todos os seus detalhes.

1.1 Organização da Dissertação

Esta dissertação está organizada em sete capítulos, descritos a seguir:

Capítulo 2 - Semântica Operacional. Neste capítulo descrevemos o funcionamento da semântica de transição [Ast91]. Desenvolvemos também a especificação formal das expressões inteiras e booleanas, usando semântica de transição.

Capítulo 3 - ASF+SDF. Neste capítulo descrevemos o meta-ambiente ASF+SDF[2] [Kli92, BK00], o qual pode ser usado na criação de uma especificação formal. Incluimos também um exemplo de como ficaria a implementação de expressões inteiras e booleanas, vistas no Capítulo 2, nesse meta-ambiente.

Capítulo 4 - Programação Lógica Paraconsistente. Descrevemos conceitos sobre Lógica Paraconsistente [CSV89, CM87, CS89], bem como explicitamos sua programação lógica [BS88, BS87, Sub87b]. Apresentamos também a linguagem de programação paraconsistente ParaLog [CPA+95]. Tal linguagem, além de englobar o Prolog padrão, estende seu escopo permitindo manipular inconsistência e/ou paracompleteza, intratáveis no Prolog padrão.

[2]Desenvolvido pelo European Community's *Esprit program* no projeto Gipe e Gipe II.

Capítulo 5 - Ambiente Operacional Interativo Prolog. Descrevemos aqui nosso protótipo de um ambiente operacional interativo Prolog, que servirá de base para a construção do protótipo de ParaLog. O desenvolvimento do ambiente interativo será feito em ASF+SDF, descrito no Capítulo 3.

Capítulo 6 - Ambiente Operacional Interativo ParaLog. A partir da formalização de programas Prolog obtida no Capítulo 5, descrevemos neste capítulo nosso protótipo de um ambiente operacional interativo ParaLog. A sintaxe e semântica propostas estão baseadas nos conceitos introduzidos no Capítulo 4, em conformidade [Abe92, Abe93, CAS91, CPA+95, BS87]. O desenvolvimento do ambiente também será feito em ASF+SDF.

Capítulo 7 - Conclusão. Neste capítulo são apresentadas as conclusões, bem como alguns tópicos de pesquisa futura que podem dar continuidade ao presente trabalho.

CAPÍTULO 2

Semântica de Transição

Neste capítulo faremos uma breve introdução aos formalismos conhecidos como *semântica operacional.* Descrevemos com mais detalhes características da semântica de transição, que é considerada um marco na evolução do formalismo operacional [Plo81].

Para comparação e um estudo mais aprofundado sobre semântica operacional, o leitor pode buscar referências em [Ast91, Win93, Wat91, IFI91, Gun91, Mus96, Sil92].

Neste capítulo trataremos apenas da semântica de execução ou semântica dinâmica das linguagens de programação, devendo mencionar a existência da semântica estática dessas linguagens aonde o comportamento estático, principalmente de tipos, é estudado.

2.1 Semântica Operacional

A semântica de uma linguagem de programação descreve a relação entre a *sintaxe* dos programas e um modelo computacional para eles. Ela preocupa-se com a *interpretação* ou *compreensão* de programas.

Costuma-se construir a definição operacional de uma linguagem organizando-a em sintaxe abstrata[1] e um conjunto de regras de avaliação das frases da linguagem[2].

A característica comum a todos os formalismos operacionais é a definição indutiva de uma relação de transição, pela qual o significado de frases sintáticas é definido. Definições indutivas [Acz77] e sistemas de transição [Plo81, Ast91] são usados em descrições de semântica operacional.

Consideramos a seguir um exemplo de uma linguagem de expressões aritméticas e booleanas com variáveis e identificadores, a qual será denominada de Exp. Após, definire-

[1] Define a gramática (árvore sintática) sem preocupar-se com regras de precedência, associatividade e pontuação.

[2] Podem ser definidas de diversas formas, por exemplo, mediante uma máquina abstrata ou funções recursivas.

mos sua sintaxe e semântica.

O conteúdo das seções a seguir foram adaptados de [Ast91].

2.1.1 Sintaxe

Como representação de sintaxe abstrata, adotaremos a representação usual infixa com parênteses. Assumimos que Num denota um conjunto de representações de números naturais; $true$ e $false$ são símbolos sintáticos para verdade e falsidade; Id e Bid denotam dois conjuntos de símbolos, o conjunto de identificadores inteiros e booleanos, respectivamente. Os elementos genéricos Num, Id e Bid serão denotados por n, id e bid, isto é, n, id e bid são variáveis sobre Num, Id e Bid respectivamente, e assim metavariáveis de Exp. Os conjuntos $Iexp$ e $Bexp$ são definidos representando-se as expressões inteiras e booleanas da linguagem Exp. Usando a notação BNF(Backus-Naur Form), temos:

$$\langle Exp \rangle ::= \langle Iexp \rangle \mid \langle Bexp \rangle$$
$$\langle Iexp \rangle ::= \langle Num \rangle \mid \langle Id \rangle \mid \langle (Iexp\ iop\ Iexp) \rangle$$
$$\langle Bexp \rangle ::= \langle true \rangle \mid \langle false \rangle \mid \langle Bid \rangle \mid$$
$$\langle (Bexp\ bop\ Bexp) \rangle \mid \langle (Iexp\ rop\ Iexp) \rangle \mid \langle\ \neg(Bexp)\ \rangle$$
$$\langle iop \rangle ::= + \mid * \mid -$$
$$\langle bop \rangle ::= \wedge \mid \vee$$
$$\langle rop \rangle ::= > \mid < \mid <= \mid >=$$

A notação BNF acima indicada é uma notação compacta para a correspondente definição indutiva da linguagem Exp. O nome à esquerda do símbolo ::= indica o conjunto a ser definido.

Por exemplo, a expressão $5 + 2 >= X$ pode ser construída usando a gramática acima, aplicando-se sucessivamente as regras.

Na seção seguinte, apresentaremos uma definição semântica para Exp.

2.1.2 Semântica

Em Exp, temos duas categorias de objetos: $Iexp$ e $Bexp$, que são chamados de domínios semânticos. Note-se que a semântica associada a uma expressão inteira, dita e

(conjunto de metavariáveis de $Iexp$), não é um valor inteiro, mas uma função. Ou seja: para qualquer atribuição de valores de identificadores das variáveis de e, nós conseguiremos um valor inteiro. Assim, teremos que distinguir entre: conjuntos auxiliares de valores, conjuntos básicos, conjuntos estruturados e os conjuntos de valores correspondentes para categorias sintáticas de interesse, aqui $Iexp$ e $Bexp$.

Vejamos como pode ser escrita uma definição da semântica operacional de Exp usando o formalismo da Semântica Indutiva Estruturada (SIS).

Primeiramente definimos o domínio básico de valores inteiros e booleanos. Assumimos que:

- Int e $Bool$ denotam representações de valores inteiros e booleanos, respectivamente, para qualquer símbolo de operação op (incluindo símbolos de aridade 0);
- para qualquer $n \in Num$, $\underline{n}$ denota o correspondente valor em Int;
- $true$ e $false$ denotam verdade e falsidade em $Bool$;
- para valores n_1 e n_2, a representação do resultado de aplicar a operação "op" para esses os valores $\underline{n_1}$ e $\underline{n_2}$ é escrita por $\underline{n_1 \ op \ n_2}$.

Definimos agora o domínio semântico das expressões. Chamamos de *state* uma atribuição de valores para conjuntos finitos de identificadores que preservam o tipo; isto é, o conjunto de estados denotados por $State$ é o conjunto de mapeamentos finitos de subconjuntos de $Id \cup Bid$ conforme: se $s \in State$, $s(id) \in Int$ se $id \in Id$ e $s(bid) \in Bool$ se $bid \in Bid$. Então, a semântica de uma expressão inteira e, notada $I[e]$, é uma função que recebe o estado s, cujo domínio contém identificadores de e, devolvendo um valor inteiro $I[e]s$, ou seja, um elemento de Int; analogamente para expressões booleanas. Formalmente, dados quaisquer dois conjuntos A e B, denotaremos por $(A \to B)$ o conjunto das funções parciais de A em B; se $f \in (A \to B)$, então $dom(f) = \{a \mid f(a) \in B\}$; por $(A \to B)_f$ indicamos o conjunto de funções parciais *finitas* de A em B.

Assim temos:

$$State = ((Id \cup Bid) \to (Int \cup Bool))_f$$

$$I : Iexp \to (State \to Int)$$

$$B : Bexp \rightarrow (State \rightarrow Bool)$$

Podemos escrever $e \xrightarrow[s]{I} v$ para $I[e]s = v$; esta notação simplesmente enfatiza que para qualquer s nós conseguimos uma relação binária $\xrightarrow[s]{I}$, que pode ser interpretada "dado s, avalia para o valor inteiro". Note-se, contudo que $\xrightarrow{I} \subseteq Iexp$ x $State$ x Int; portanto, $\xrightarrow{I}$ é uma relação ternária. Analogamente, podemos fazer o mesmo para a avaliação semântica das expressões booleanas, denotadas por B; podendo definir a relação $\xrightarrow{B} \subseteq Bexp$ x $State$ x $Bool$.

Temos assim duas funções semânticas correspondendo às duas categorias principais de objetos sintáticos. Agora, uma definição SIS consiste simplesmente na definição de relações $\xrightarrow{I}$ e $\xrightarrow{B}$ por meio de uma definição indutiva, cujas cláusulas são dirigidas pela estrutura sintática de expressões.

A definição indutiva da SIS pode ser feita em dois estilos:

- semântica de passo grande (*big step*) [Ast91, Gun91];
- semântica de passo pequeno (*small step*) [Ast91, Kah87].

Neste trabalho abordaremos somente a semântica de passo pequeno (também conhecida como Semântica de Transição), pois o desenvolvimento dos ambientes Prolog e ParaLog, descritos nos Capítulos 5 e 6 respectivamente, ocorre no meta-ambiente ASF+SDF[Kli92, HHKR92, BK00], o qual utiliza naturalmente conceitos de semântica de transição para a criação de um ambiente operacional relativo a uma determinada linguagem. Para maiores informações sobre semântica de passo grande, o leitor pode referenciar-se em [Ast91, Mus96, Win93, Sil92].

2.2 Semântica de Transição

A semântica de transição é um formalismo que permite a definição de uma relação de passo pequeno (*small step*) entre frases de uma linguagem, visto que a semântica de um programa é caracterizada pela relação *step* e seu fecho reflexivo e transitivo.

Para definirmos a linguagem *Exp* em semântica de transição consideramos que:

- n e bc denotam respectivamente quaisquer constantes inteiras e booleanas pertencentes a Num e a $\{true, false\}$;

- *id* e *bid* representam respectivamente identificadores inteiros e booleanos pertencentes a Id e Bid;
- b, b_1, b_2 e e_1, e_2 representam respectivamente expressões do tipo booleana e inteiras pertencentes a $Bexp$ e a $Iexp$;
- bv, bv_1, bv_2 e v, v_1, v_2 denotam respectivamente valores booleanos e inteiros pertencentes a $Bool$ e a Int;
- *iop*, *bop* e *rop* representam respectivamente símbolos matemáticos $(+, *, -)$, booleanos $(\wedge, \vee)$ e relacionais $(>, <, <=, >=)$.

Vejamos agora, por exemplo, as regras para definição de Exp:

[01] $n \xrightarrow[s]{} \underline{n}$

[02] $bc \xrightarrow[s]{} \underline{bc}$

[03] $id \xrightarrow[s]{} s(id)$ onde $id \in dom(s)$

[04] $bid \xrightarrow[s]{} s(bid)$ onde $bid \in dom(s)$

[05] $(v_1\ iop\ v_2) \xrightarrow[s]{} v$ onde $v = \underline{v_1\ iop\ v_2}$

[06] $(bv_1\ bop\ bv_2) \xrightarrow[s]{} bv$ onde $bv = \underline{bv_1\ bop\ bv_2}$

[07] $(v_1\ rop\ v_2) \xrightarrow[s]{} bv$ onde $bv = \underline{v_1\ rop\ v_2}$

[08] $\neg(bv) \xrightarrow[s]{} bv'$ onde $bv' = \underline{\neg bv}$

[09] $$\frac{e_1 \xrightarrow[s]{} e_1'}{(e_1\ iop\ e_2) \xrightarrow[s]{} (e_1'\ iop\ e_2)}$$

[10] $$\frac{e_2 \xrightarrow[s]{} e_2'}{(v\ iop\ e_2) \xrightarrow[s]{} (v\ iop\ e_2')}$$

[11] $$\frac{e_1 \xrightarrow[s]{} e_1'}{(e_1\ rop\ e_2) \xrightarrow[s]{} (e_1'\ rop\ e_2)}$$

[12] $$\frac{e_2 \xrightarrow[s]{} e_2'}{(v\ rop\ e_2) \xrightarrow[s]{} (v\ rop\ e_2')}$$

[13] $$\frac{b_1 \xrightarrow[s]{} b_1'}{(b_1\ bop\ b_2) \xrightarrow[s]{} (b_1'\ bop\ b_2)}$$

[14] $$\frac{b_2 \xrightarrow[s]{} b_2'}{(bv\ bop\ b_2) \xrightarrow[s]{} (bv\ bop\ b_2')}$$

$$[15]\ \frac{b \xrightarrow[s]{} b'}{\neg(b) \xrightarrow[s]{} \neg b'}$$

As regras acima são essencialmente divididas em duas categorias:

- regras [01] - [08] definem um passo de avaliação elementar e são chamadas de axiomas;
- regras [09] - [15] definem a transformação de uma expressão correspondente aplicando *um passo de avaliação* elementar e são chamadas de regras de inferência. O que está acima da linha é chamado numerador (premissas); e o que está abaixo da linha, algumas vezes é chamado de denominador (conseqüências).

Podemos notar, neste exemplo, que cada regra corresponde a um passo de avaliação elementar; cada passo transforma uma expressão em outra pela reescrita de suas subexpressões, até a obtenção de uma expressão que não possa mais ser reescrita. Supondo que $s = \{(\mathrm{X} \rightarrow 3)\}$, a avaliação da expressão $X - 3 >= 0$, utilizando as regras de semântica de transição desta Seção, pode ser dada pela seguinte tabela abaixo:

Termo	**Regra Usada**
$X - 3 >= 0$	
$\xrightarrow[s]{}\ 3 - 3 >= 0$	[03]
$\xrightarrow[s]{}\ 0 >= 0$	[05]
$\xrightarrow[s]{}\ true$	[07]

2.3 Considerações Finais

Este capítulo descreveu algumas características do formalismo de semântica de transição, o qual é útil para a construção de projetos de interpretadores, uma vez que nos permite a definição sintática e semântica de uma linguagem de programação por meio de regras, como foi o caso da avaliação das expressões booleanas nele mostrado.

A implementação dessas regras pode ser feita usando-se o meta-ambiente ASF+SDF, que descreveremos no capítulo a seguir.

CAPÍTULO 3

ASF+SDF

Descrevemos neste capítulo o meta-ambiente ASF+SDF, o qual é útil para especificação de tipos abstratos de dados arbitrários, tradicionalmente chamada de especificação algébrica, bem como para a definição formal de qualquer linguagem de programação.

Este capítulo está baseado em [BCD+88, Kli92, HHKR92, BK00].

3.1 ASF+SDF

O ASF+SDF [BCD+88, Kli92, BK00] é um gerador genérico de ambientes interativos para linguagens de programação, o qual, dada a especificação formal de uma linguagem de programação, sintaxe e semântica, produz um ambiente para ela. Esse ambiente contém um:

- editor;
- interpretador ou compilador;
- *type checker*, e
- depurador.

O ambiente de especificação é baseado na combinação do formalismo *Algebraic Specification Formalism* (ASF) com o formalismo *Syntax Definition Formalism* (SDF) [Kli92, HHKR92, BK00]. O meta-ambiente fornece um editor de módulos para o desenvolvimento de especificações modulares. Cada módulo consiste de duas partes: uma para a sintaxe da linguagem e uma para sua semântica.

O estilo de definição da sintaxe dos programas assemelha-se à BNF, mas também trata da definição léxica e da sintaxe abstrata da linguagem. Por exemplo, a regra BNF:

$$\langle Exp \rangle ::= \langle Iexp \rangle \mid \langle Bexp \rangle$$

pode ser escrita no meta-ambiente ASF+SDF como:

$$Iexp = Exp$$
$$Bexp = Exp$$

A definição de uma linguagem em tal meta-ambiente consistirá das seguintes partes:

- definição da sintaxe concreta, isto é, *lexical* e *context-free*, bem como da sintaxe abstrata da linguagem;
- definição de suas semânticas estáticas e dinâmicas.

O SDF permite a definição da sintaxe concreta e abstrata em um único sistema. Seu projeto e implementação são adaptados para o projetista da linguagem, o qual deseja desenvolver novas linguagens ou implementar as existentes de maneira altamente interativa.

O ASF é baseado na noção de um modelo que consiste de uma *signature* definindo a sintaxe abstrata das funções que operarão sobre termos da linguagem e um conjunto de equações condicionais. As equações definem a semântica operacional de transição da linguagem e são implementadas como um sistema de reescrita de termos [Klo92], produzindo especificações executáveis.

A especificação da linguagem tipicamente inclui características como:

- edição orientada pela sintaxe;
- verificação de tipos;
- execução de programas, incluindo execução passo-a-passo.

O uso do meta-ambiente traz algumas vantagens como o aumento de uniformidade e a redução do esforço de implementação, pois ele provê: formalismo de especificação algébrica baseada na lógica equacional; estruturação modular de especificações; integrada definição de sintaxe léxica, livre de contexto e abstrata; escrita de suas especificações usando suas próprias notações (sintaxe definida pelo usuário), completa integração sintática e semântica [BK00].

Em resumo, o ambiente ASF+SDF permite:

- escrever uma especificação formal para alguns problemas de forma interativa;
- criar um ambiente interativo para uma linguagem;
- analisar e transformar programas.

Na seção a seguir descrevemos a implementação da semântica de transição da linguagem *Exp* no meta-ambiente ASF+SDF. Essa seção está baseada nas definições sintáticas e semânticas descritas nas Seções 2.1.1 e 2.2.

3.2 Implementação da Linguagem de Expressões (*Exp*)

A implementação de um ambiente interativo para *Exp* consiste da definição sintática (SDF) e semântica (ASF) da linguagem, que gera, para cada módulo desenvolvido, respectivamente, os arquivos $\langle module \rangle.sdf2$ e $\langle module \rangle.eqs$.

Para facilitar a criação do ambiente interativo da linguagem *Exp*, a implementação usará os módulos[1] abaixo:

1. *Layout*: define algumas sintaxes padrão do ambiente ASF+SDF, como: espaço, tabulação, caractere de nova linha, comentários (ver Apêndice A.1.6).
2. *Booleans*: é responsável por fazer avaliações de expressões booleanas. Este módulo teve alterações em relação à versão original, pois foram criadas as funções *and*, *or* e *neg*, as quais fazem avaliações do tipo *e*, *ou* e *não*, respectivamente (ver Apêndices A.1.1 e A.1.2).
3. *Integers*: permite fazer avaliações de expressões inteiras. Este módulo também sofreu alterações. Criamos as funções *plus*, *sub* e *mult* para realizar cálculos aritméticos, e as funções *gt*, *ge*, *lt* e *le* para realizar operações relacionais (ver Apêndices A.1.4 e A.1.5).
4. *Value-EnvInt* : define o mapeamento de estados para identificadores inteiros (ver Apêndice A.1.11 e A.1.12).

[1] Os referidos módulos fazem parte da implementação da linguagem exemplo denominada PICO [BK00].

5. *Identifiers*: define os identificadores inteiros (*Id*) e booleanos (*Bid*). Este módulo foi alterado porque sua implementação original somente define identificadores inteiros. Sua nova versão está listada abaixo[2]:

```
module Identifiers
imports Layout
exports
sorts Id Bid
lexical syntax
        [a-z] [a-z0-9]*    →  Id
        [A-Z] [a-z0-9]*    →  Bid
variables
        "id" [0-9\']*      →  Id
        "bid" [0-9\']*     →  Bid
```

Além dos módulos acima, criamos também os módulos *Value-EnvBool* e *LingExp*. O primeiro módulo representa o mapeamento de estados para identificadores booleanos. O segundo módulo conterá as definições sintáticas e semânticas de *Exp*, descritas nas Seções 2.1.1 e 2.2.

O módulo de expressões inteiras e booleanas possui dois arquivos: *LingExp.sdf2* e *LingExp.eqs*. O primeiro deles define a sintaxe de expressões, e é listado a seguir:

```
module LingExp
imports Value-EnvBool Value-EnvInt
exports
sorts Iexp Bexp Exp
lexical syntax
context-free syntax
        Bexp              →  Exp
        Iexp              →  Exp

        Id                →  Iexp
        Num               →  Iexp
        Iexp "+" Iexp     →  Iexp{left}
        Iexp "-" Iexp     →  Iexp{left}
        Iexp "*" Iexp     →  Iexp{left}
        "(" Iexp ")"      →  Iexp{bracket}
```

[2]A sintaxe dos módulos será explicada em detalhes no próximo exemplo.

```
        Bid                 →  Bexp
        Bool                →  Bexp
        Bexp "&" Bexp       →  Bexp{left}
        Bexp "|" Bexp       →  Bexp{left}
        Iexp ">" Iexp       →  Bexp
        Iexp ">=" Iexp      →  Bexp
        Iexp "<" Iexp       →  Bexp
        Iexp "<=" Iexp                          →  Bexp
        "not" "(" Bexp ")"                      →  Bexp
        "(" Bexp ")"                            →  Bexp{bracket}

        "step" "(" VENV "," VENVB "," Exp ")"   →  Exp
        "evalExp" "(" VENV "," VENVB "," Exp ")" →  Exp
        "isnumber" "(" Exp ")"                  →  Bool
variables
        "v"[0-9\']*         →  Num
        "e"[0-9\']*         →  Iexp
        "bc"[0-9\']*        →  Bool
        "v"[0-9\']*         →  Num
        "b"[0-9\']*         →  Bexp
        "IB"[0-9\']*        →  Exp
context-free priorities
        Bexp "&" Bexp       →  Bexp >
        Bexp "|" Bexp       →  Bexp >
        Iexp "*" Iexp       →  Iexp >
        {left: Iexp "+" Iexp →  Iexp {left}
        Iexp "-" Iexp       →  Iexp {left}
        }
```

A listagem acima é composta pelas seguintes seções:

- *sorts* : nomes de domínios ou símbolos não terminais que podem ser usados em outras seções da especificação. De acordo com o módulo listado acima, nesta seção temos as definições de *Iexp*, *Bexp* e *Exp*, que representam respectivamente expressões inteiras, expressões booleanas e expressões inteiras e booleanas da linguagem *Exp*.

- *lexical syntax* : regras de sintaxe léxica. Neste seção os símbolos correspondentes a *iop*, *bop*[3] e *rop* da linguagem *Exp* (Capítulo 2) foram definidos diretamente para

[3]Os símbolos & e | representam respectivamente os símbolos ∧ e ∨ definidos na Seção 2.1.1.

facilitar a implementação na seção *context-free.*

- *context-free syntax* : regras de sintaxe da linguagem a ser especificada.

  As regras definidas nesta seção correspondem à BNF da linguagem. Por exemplo, a regra de produção:

  $$Iexp ::= Iexp \text{ “+” } Iexp$$

  é definida como:

  $$Iexp \text{ “+” } Iexp \rightarrow Iexp$$

  Nesta seção, também é definida a sintaxe das regras de avaliação (semântica) da linguagem e algumas prioridades de parametrização e associatividade (“evalExp” e “step”, listadas anteriormente, são alguns exemplos de regras de avaliação).

  A função *not*[4] é um exemplo de sintaxe da linguagem, enquanto a função *step* é um exemplo de função de avaliação e representa um passo de avaliação das expressões. Esta função possue como argumentos um mapeamento (para expressões inteiras e booleanas — *VENV*, *VENVB*), e uma expressão (*Exp*) que passará por um processo de avaliação (*small step*) para devolver uma expressão.

- *variables* : definição das meta-variáveis da linguagem, as quais podem ser usadas de dois modos: como variáveis em definições semânticas adicionadas para definição SDF e como regras em programas durante edição dirigida pela sintaxe.

- *context-free priorities* : define relações de prioridades e associatividades entre os operadores. O símbolo > (Seção **context-free priorities**) indica que a expressão à esquerda tem maior prioridade do que a expressão à direita. Também definimos um grupo de função associativa (Seção **context-free priorities** - *Iexp*), a qual define como aceitar ou rejeitar árvores contendo ocorrências relacionadas de funções diferentes com a mesma prioridade. A palavra **left** é utilizada para fazer associações da esquerda para a direita.

[4] Representa o símbolo ¬ definido na Seção 2.1.1.

Em síntese, a definição SDF tem o propósito de definir:

- um conjunto de sentenças (*strings*);
- um conjunto de árvores sintáticas (árvore de *parsing*);
- uma relação entre sentença (árvore de *parsing*) e árvore de sintaxe abstrata.

Além do arquivo *LingExp.sdf2*, já descrito, tem-se também o arquivo *LingExp.eqs*. Este arquivo consiste da definição *small step* da semântica da linguagem *Exp* definida na Seção 2.2, sendo transcrito parcialmente a seguir:

equations

```
[01]   step(Venv, VenvB, n)          =  n
[02]   step(Venv, VenvB, bc)         =  bc
[03]   step(Venv, VenvB, id)         =  lookup(id , Venv)
[04]   step(Venv, VenvB, bid)        =  lookupB(bid, VenvB)
[09a]  step(Venv, VenvB, e1 + e2)    =  e1' + e2 when step(Venv, VenvB, e1) = e1'
[10a]  step(Venv, VenvB, v + e2)     =  v + e2' when step(Venv, VenvB, e2) = e2'
[05a]  step(Venv, VenvB, v1 + v2)    =  v when plus(v1, v2) = v

[ev1]  evalExp(Venv, VenvB, n)       =  n
[ev2]  evalExp(Venv, VenvB, IB)      =  true when IB = true
[ev3]  evalExp(Venv, VenvB, IB)      =  false when IB =false
[ev4]  evalExp(Venv, VenvB, IB)      =  IB" when
                                        IB != false, IB != true, isnumber(IB) = false,
                                        step(Venv, VenvB, IB) = IB',
                                        evalExp(Venv, VenvB, IB') = IB"
```

De acordo com a listagem[5] acima, a regra[6] [01] faz a avaliação de um número natural, devolvendo o próprio número como resultado final. A regra [02] define a avaliação de um valor booleano (*true* ou *false*), devolvendo o próprio valor booleano. As regras [03] e [04] representam a avaliação de um identificador inteiro e booleano respectivamente, e usam dois módulos: *Value-Int* e *Value-Bool*. A regra [05a] permite fazer avaliação de adição

[5] A listagem completa do programa está no Apêndice A.1.8.

[6] Não estão na ordem descritas no Capítulo 2, devido a problemas de ambigüidade da gramática no meta-ambiente ASF+SDF.

quando se têm dois valores numéricos (exemplo: 1 e 2). Para que a avaliação seja efetuada na regra [05a], torna-se necessário o uso do módulo *Integers*. A regra [09a] representa a avaliação quando se tem um conjunto de expressões inteiras tanto à esquerda quanto à direita do sinal de "+". A regra [10a] faz avaliação quando se tem um valor numérico à esquerda e uma expressão inteira à direita do sinal de "+". As quatro últimas regras [ev1], [ev2], [ev3] e [ev4] permitem fazer a avaliação de uma expressão inteira ou booleana, devolvendo um resultado final. A primeira regra [ev1] é utilizada quando a expressão já é um valor numérico. A segunda e a terceira regra ([ev2] e [ev3]) permitem devolver um valor *true* ou *false* quando a expressão é do tipo booleano. A última regra [ev4] faz a avaliação de uma expressão, até que um valor final seja encontrado para essa expressão.

Em resumo, em ASF o significado de operadores é definido por meio de um procedimento de normalização, isto é, um procedimento de expansão textual que elimina todas as estruturas modulares de uma especificação e consegue uma simples especificação, que consiste de uma *signature* e *equations*. Outros exemplos de especificações ASF podem ser encontradas em [BHK89, Hen91].

3.3 Considerações Finais

O formalismo ASF+SDF permite a definição tanto sintática quanto semântica de linguagens. Ele pode ser usado para a definição de linguagens de programação, especificação de reescrita, consulta de banco de dados, processamento de texto e aplicações dedicadas, ou seja, pode ser usado para especificação formal de uma ampla variedade de problemas. Seu sistema de avaliação permite gerar um interpretador baseado nas regras da semântica da linguagem.

O ASF+SDF nos permite também o desenvolvimento e geração automática de sistemas interativos, manipulando programas, especificações ou outros textos que são escritos em uma linguagem formal. É, pois, por meio desse ambiente de desenvolvimento e dos conceitos de semântica de transição que nossos protótipos de especificação das linguagens Prolog e ParaLog, descritos nos Capítulos 5 e 6, foram desenvolvidos.

CAPÍTULO 4

Programação Lógica Paraconsistente

Neste capítulo descrevemos alguns conceitos importantes sobre Lógica Paraconsistente [BS88, CS89, CSV89, Cos93], bem como uma variação da linguagem Prolog baseada nas Lógica Anotadas [Abe92, Abe93, CAS91], denominada ParaLog [CPA+95, BS87].

4.1 Histórico da Lógica Paraconsistente

O manuseio de inconsistência em bases de conhecimento tem se tornado um problema prático [CS89]. Na construção dessas bases de conhecimentos recebemos informações de várias fontes, o que pode levar-nos a fatos contraditórios. Um exemplo disso seria a construção de um sistema médico especialista. No caso de um médico $\mathcal{M}_1$ concluir que o paciente $\mathcal{P}$ possui uma infecção viral, e ao mesmo tempo o médico $\mathcal{M}_2$ concluir que o mesmo paciente $\mathcal{P}$ possui rejeição alérgica, esses fatos contraditórios podem levar o paciente $\mathcal{P}$ a tomar decisões, como por exemplo, consultar-se com um terceiro médico $\mathcal{M}_3$. Se fossemos construir um sistema com as opiniões dos médicos $\mathcal{M}_1$, $\mathcal{M}_2$ e $\mathcal{M}_3$, teríamos provavelmente a presença de inconsistências.

Na lógica clássica não possuímos meios para lidar com essas inconsistências. Tecnicamente, o motivo é que se em um sistema baseado em uma tal lógica houver uma contradição, todas as expressões bem formadas de sua linguagem (ditas "fórmulas" da linguagem) podem ser demonstradas, ou seja, nessa lógica "prova-se tudo".

Os principais precursores das idéias de paraconsistência são Jan Lukasiewicz e Nicolaj Vasil'év. Ambos em 1910, independentemente, apresentaram algumas perspectivas lógicas e resultados sob a lógica tradicional Aristotélica, os quais hoje podem ser incluídos no domínio de Lógica Paraconsistente. Em 1948, sob influência de Stanislaw Jaskowski,

Lukasiewicz propôs o primeiro Cálculo Proposicional Paraconsistente, mas somente em 1950, independentemente dos trabalhos de Lukasiewicz, Vasil'év e Jaskowski, N.C.A. da Costa, desenvolveu várias lógicas paraconsistentes que possuem os seguintes níveis lógicos [Arr80, CM87, Cos93]:

- cálculo proposicional;
- cálculo de predicados;
- cálculo de predicados com igualdade;
- cálculo de descrições e teoria dos conjuntos.

4.2 Lógicas Paraconsistentes, Paracompletas e Não-Aléticas

Os conceitos apresentados nesta seção foram extraídos de [Á96, Cos93].

Seja considerada uma teoria dedutiva $\mathcal{T}$ fundada sobre uma lógica $\mathcal{L}$, e suponha-se que uma linguagem de $\mathcal{T}$ e de $\mathcal{L}$ possua um símbolo para a negação: se existir mais de uma negação, uma delas deve ser escolhida. Uma teoria $\mathcal{T}$ diz-se inconsistente se seus teoremas contêm ao menos dois deles, um dos quais é a negação do outro. Nesse caso, sendo A e $\neg A$ tais teoremas, normalmente deriva-se em $\mathcal{T}$ uma contradição, isto é, uma expressão da forma $A \wedge \neg A$; caso contrário, $\mathcal{T}$ é consistente.

Uma teoria $\mathcal{T}$ é dita trivial se todas as fórmulas de $\mathcal{L}$ — ou todas as fórmulas fechadas de $\mathcal{L}$ — forem teoremas de $\mathcal{T}$, ou seja, se todos os enunciados sintaticamente corretos do ponto de vista da linguagem de $\mathcal{T}$ puderem ser provados em $\mathcal{T}$. Se esse for o caso, a teoria não permite que se distinga o "**demonstrável**" do "**não demonstrável**", não apresentando interesse algum, uma vez que não se poderá separar o falso do verdadeiro. Caso contrário, $\mathcal{T}$ é não trivial.

Uma Lógica Paraconsistente permite a convivência de contradições em uma teoria e substitue eficazmente a lógica convencional em algumas situações reais, quando esta se mostra ineficiente ou incapaz de ser aplicada. Dessa forma, uma lógica é considerada **Paraconsistente** se pode ser utilizada como lógica subjacente a teorias inconsistentes mas não triviais [Cos93].

Um outro conceito importante dentro das lógicas não clássicas são as lógicas **Paracompletas**. Uma lógica $\mathcal{L}$ diz-se **Paracompleta** se nela não for válido o princípio do terceiro excluído, isto é, não se pode ter em geral que $A \vee \neg A$ seja um teorema da lógica, dado uma fórmula A.

Uma lógica $\mathcal{L}$, que é simultaneamente Paraconsistente e Paracompleta, é dita **Não-Alética**.

4.3 Lógica Anotada

Nesta seção, com base nas referências [Abe93, Abe92, CAS91, Á96], faremos um apanhado geral da lógica de primeira Ordem Anotada. Vale a pena ressaltar que as lógicas anotadas são lógicas paraconsistentes e, em geral, paracompletas e não-aléticas.

Essas lógicas foram estudadas do ponto de vista da programação lógica por Blair e Subrahmanian, e algumas de suas aplicações podem ser encontradas em [NAS99].

Seja $\tau = \langle |\tau|, \leq \rangle$ um reticulado finito fixo, no qual fixa-se um operador

$$\sim: |\tau| \to |\tau|$$

Esse reticulado é conhecido como *reticulado de valores-verdades* e $\sim$ constitui o "significado" do símbolo de negação $\neg$ dos sistemas que serão considerados. Associados a esse reticulado τ tem-se, ainda, os seguintes símbolos e operações:

- $\bot$ indica o mínimo de τ;
- $\top$ indica o máximo de τ;
- **sup** indica a operação de supremo — em relação a subconjuntos de τ;
- **inf** indica a operação de ínfimo — em relação a subconjuntos de τ.

A linguagem $\mathcal{L}_\tau$ é definida medicante os seguintes símbolos primitivos e construtores (léxicos):

1. variáveis individuais: $x, y, z, w, x_1, x_2, \ldots$;
2. para cada número natural n , símbolos funcionais n-ários (os símbolos funcionais 0-ários constituem o conjunto das constantes individuais);

3. para cada número natural n , símbolos de predicados n-ários;

4. o símbolo de igualdade $=$;

5. cada membro de τ é uma constante anotacional;

6. os símbolos $\neg, \vee, \wedge, \rightarrow, \exists$ e $\forall$;

7. os símbolos auxiliares (,) e ,.

Os termos da linguagem $\mathcal{L}_\tau$ são definidos de maneira usual. São utilizados a, b, c e d — com ou sem índices — como meta variáveis para os termos.

DEFINIÇÃO. 4.3.1 (FÓRMULA) Uma *fórmula básica* é uma expressão do tipo

$$p(a_1, ..., a_n),$$

em que p é um símbolo predicativo n-ário e $a_1, ..., a_n$ são termos de $\mathcal{L}_\tau$. Se $p(a_1, ..., a_n)$ é uma fórmula básica e $\mu \in \tau$ é uma constante anotacional, então $p_\mu(a_1, ..., a_n)$ e $a = b$ — onde a e b são termos — chamam-se *fórmulas atômicas*. As fórmulas têm a seguinte definição indutiva generalizada:

1. uma fórmula atômica é uma fórmula;

2. se A é uma fórmula, então $\neg A$ é uma fórmula;

3. se A e B são fórmulas, então $A \vee B$, $A \wedge B$ e $A \rightarrow B$ são fórmulas;

4. se A é uma fórmula e x é uma variável individual, então $(\exists x)A$ e $(\forall x)A$ são fórmulas;

5. uma expressão de $\mathcal{L}_\tau$ constitui uma fórmula se e somente se for obtida aplicando-se uma das regras — 1 a 4 — anteriores.

A fórmula $\neg A$ é lida "a negação — ou negação fraca — de A"; $A \vee B$ "a disjunção de A e B"; $A \wedge B$ "a conjunção de A e B"; $A \rightarrow B$ "a implicação de B por A"; $(\exists x)A$ "a instanciacão de A por x"; $(\forall x)A$ "a generalização de A por x".

Introduz-se alguns símbolos definidos:

DEFINIÇÃO. 4.3.2 (NEGAÇÃO FORTE E EQUIVALÊNCIA) Sejam A e B fórmulas quaisquer de $\mathcal{L}_\tau$, define-se:

$$(A \leftrightarrow B) =_{def} ((A \rightarrow B) \wedge (B \rightarrow A))$$

e

$$(\neg_{\shortmid} A) =_{def} (A \rightarrow ((A \rightarrow A) \wedge \neg(A \rightarrow A)))$$

O símbolo $\neg_{\shortmid}$ denomina-se *negação forte* ; portanto, $(\neg_{\shortmid} A)$ deve ser lido a *negação forte* de A. A fórmula $(A \leftrightarrow B)$ é lida, usualmente como, a *equivalência* de A e B.

4.3.3 DEFINIÇÃO. Seja A uma fórmula. Então:

- $\neg_{\shortmid}^{0} A$ indica A;
- $\neg_{\shortmid}^{1} A$ indica $\neg_{\shortmid} A$ e
- $\neg_{\shortmid}^{k} A$ indica $\neg_{\shortmid}(\neg_{\shortmid}^{k-1} A)$, $(K \in N,\ K > 0)$

Também, se $\mu \in \tau$, convenciona-se que:

- $\sim^{0} \mu$ indica μ;
- $\sim^{1} \mu$ indica $\sim \mu$ e
- $\sim^{k} \mu$ indica $\sim (\sim^{k-1} \mu)$, $(K \in N, K > 0)$

DEFINIÇÃO. 4.3.4 (LITERAL) Seja $p_{\mu}(a_1, ..., a_n)$, uma fórmula atômica. Qualquer fórmula do tipo $\neg_{\shortmid}^{k} p_{\mu}(a_1, ..., a_n)(K \geq 0)$ denomina-se uma fórmula *hiper-literal* ou, simplesmente, *literal*. As demais fórmulas denominam-se *fórmulas complexas*.

A seguir, é apresentada uma descrição semântica para as linguagens $\mathcal{L}_{\tau}$.

DEFINIÇÃO. 4.3.5 (ESTRUTURA) Uma estrutura v para uma linguagem $\mathcal{L}_{\tau}$ consiste dos seguintes objetos:

1. um conjunto não-vazio $|v|$, denominado o *universo* de v. Os elementos de $|v|$ chamam-se *indivíduos* de v;
2. para cada símbolo funcional n-ário f de $\mathcal{L}_{\tau}$, uma operação n-ária f_v de $|v|^n$ em $|v|$ — em particular, para cada constante individual e de $\mathcal{L}_{\tau}$, e_v é um indivíduo de v;
3. para símbolo predicativo p de peso n de $\mathcal{L}_{\tau}$, uma função

$$p_v : |v|^n \rightarrow |\tau|.$$

Seja v uma estrutura para $\mathcal{L}_\tau$. A *linguagem-diagrama* $\mathcal{L}_\tau(v)$ é obtida de modo habitual. Dado um termo livre de variável a de $\mathcal{L}_\tau(v)$, define-se, também, de modo comum, o indivíduo $v(a)$ de v. Utilizam-se i e j como meta-variáveis para denotar nomes.

Define-se, agora, o valor-verdade $v(A)$ de uma fórmula fechada A de $\mathcal{L}_\tau(v)$. A definição é obtida por indução sobre o comprimento de A. Por abuso de linguagem, utilizam-se os mesmos símbolos para meta-variáveis de termos da linguagem diagrama.

4.3.6 DEFINIÇÃO. Seja A uma fórmula fechada e v uma interpretação para $\mathcal{L}_\tau$.

1. Se A é atômica da forma $p_\mu(a_1, ..., a_n)$, então:

$$v(A) = 1 \text{ se e somente } p_v(v(a_1), ..., (a_n)) \geq \mu$$
$$v(A) = 0 \text{ se e somente } p_v(v(a_1), ..., (a_n)) \ngeq \mu$$

2. Se A é atômica da forma $a = b$, então:

$$v(A) = 1 \text{ se e somente se } v(a) = v(b)$$
$$v(A) = 0 \text{ se e somente se } v(a) \neq v(b)$$

3. Se A é da forma $\neg^k(p_\mu(a_1, ..., a_n))(K \geq 1)$, então:

$$v(A) = v(\neg^{k-1}(p_{\sim\mu}(a_1, ..., a_n))$$

4. Sejam A e B fórmulas fechadas quaisquer. Então,

$$v(A \wedge B) = 1 \text{ se e somente se } v(A) = v(B) = 1$$
$$v(A \vee B) = 1 \text{ se e somente se } v(A) = 1 \text{ ou } v(B) = 1$$
$$v(A \rightarrow B) = 1 \text{ se e somente se } v(A) = 0 \text{ ou } v(B) = 1$$

5. Se A é uma fórmula fechada complexa, então:

$$v(\neg A) = 1 - v(A)$$

6. Se A é da forma $(\exists x)B$, então:

$$v(A) = 1 \text{ se e somente se } v(B_x[i]) = 1 \text{ para algum } i \text{ em } \mathcal{L}_\tau(v)$$

onde $B_x[i]$ denota a fórmula obtida pela substituição simultânea de x em B por i.

7. Se A é da forma $(\forall x)B$, então:

$$v(A) = 1 \text{ se e somente se } v(B_x[i]) = 1 \text{ para todo } i \text{ em } \mathcal{L}_\tau(v)$$

onde $B_x[i]$ denota a fórmula obtida pela substituição simultânea de x em B por i.

4.3.7 TEOREMA. Sejam A, B e C fórmulas quaisquer de $\mathcal{Q}_\tau$. Os conectivos $\rightarrow, \vee, \wedge, \neg$, junto com os quantificadores $\forall$ e $\exists$, possuem todas as propriedades da implicação, disjunção, conjunção e negação clássicas, bem como dos quantificadores $\forall$ e $\exists$ clássicos, respectivamente. Por exemplo, tem-se que:

1. $\vdash \neg\forall x A \leftrightarrow \exists x \neg A$
2. $\vdash \neg\exists x B \vee C \leftrightarrow \exists x(B \vee C)$
3. $\vdash \neg\exists x B \vee \exists x C \leftrightarrow \exists x(B \vee C)$

O sistema de postulados — esquemas de axiomas e regras de inferência — para $\mathcal{Q}_\tau$, que é apresentado a seguir, será denominado de $\mathcal{A}_\tau$.

A, B, C denotam fórmulas quaisquer; F e G denotam fórmulas complexas; p denota uma fórmula básica; e $\mu, \mu_j (1 \leq j \leq n)$ denotam constantes anotacionais; $x, x_1, ..., x_n$, $y_1, ..., y_n$ são variáveis individuais.

$(\rightarrow_1)$ $A \rightarrow (B \rightarrow A)$

$(\rightarrow_2)$ $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$(\rightarrow_3)$ $((A \rightarrow B) \rightarrow A) \rightarrow A)$

$(\rightarrow_4)$ $\dfrac{A, A \rightarrow B}{\mathrm{B}}$

$(\wedge_1)$ $(A \wedge B) \rightarrow A$

$(\wedge_2)$ $(A \wedge B) \rightarrow B$

$(\wedge_3)$ $A \rightarrow (B \rightarrow (A \wedge B))$

$(\vee_1)$ $A \rightarrow (A \vee B)$

$(\vee_2)$ $B \rightarrow (A \vee B)$

$(\vee_3)$ $(A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow ((A \vee B) \rightarrow C))$

$(\neg_1)$ $(F \rightarrow G) \rightarrow ((F \rightarrow \neg G) \rightarrow \neg F)$

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

$(\neg_3)$ $F \vee \neg F$

(τ_1) $p_\perp$

(τ_2) $(\neg^k p_\mu) \leftrightarrow (\neg^{k-1} p_{\sim\mu})K \geq 1$

(τ_3) $p_\mu \rightarrow p_\lambda$, onde $\mu \geq \lambda$

(τ_4) $p_{\mu 1} \wedge p_{\mu 2} \wedge ... \wedge p_{\mu n} \rightarrow p_\mu$, onde $\mu = \mathbf{sup}\ \mu_j$, $j = 1, 2, ..., n$

$(\forall_1)$ $\dfrac{B \rightarrow A(x)}{B \rightarrow \forall x A(x)}$

$(\forall_2)$ $\forall x A(x) \rightarrow A(t)$

$(\exists_1)$ $A(t) \rightarrow \exists x A(x)$

$(\exists_2)$ $\dfrac{A(x) \rightarrow B}{\exists x A(x) \rightarrow B}$

$(=_1)$ $x = x$

$(=_2)$ $x_1 = y_1 \rightarrow ... \rightarrow x_n = y_n \rightarrow f(x_1, ..., x_n) = f(y_1, ..., y_n)$

$(=_3)$ $x_1 = y_1 \rightarrow ... \rightarrow x_n = y_n \rightarrow p_\lambda(x_1, ..., x_n) \rightarrow p_\lambda(y_1, ..., y_n)$

4.3.8 DEFINIÇÃO. Uma *estrutura* $\mathcal{A}$ diz-se *não-trivial* se existe um átomo anotado fechado $p_\lambda t_1, ..., t_n$ tal que $\mathcal{A}(p_\lambda t_1, ..., t_n) = 0$.

Portanto, uma estrutura $\mathcal{A}$ é não trivial se e somente se existir algum átomo anotado fechado que não é válido em $\mathcal{A}$.

4.3.9 DEFINIÇÃO. Uma *estrutura* $\mathcal{A}$ diz-se *inconsistente* se existe um átomo anotado fechado $p_\lambda t_1, ..., t_n$ tal que

$$\mathcal{A}(p_\lambda t_1, ..., t_n) = 1 = \mathcal{A}(\neg p_\lambda t_1, ..., t_n)$$

Portanto, uma estrutura $\mathcal{A}$ é inconsistente se e somente se existir algum átomo anotado fechado tal que ele e a sua negação sejam válidos em $\mathcal{A}$.

4.3.10 DEFINIÇÃO. Uma *estrutura* $\mathcal{A}$ diz-se *paraconsistente* se $\mathcal{A}$ é inconsistente e não-trivial. O sistema $\mathcal{Q}_\tau$ diz-se *paraconsistente* se existe uma estrutura $\mathcal{A}$ para $\mathcal{Q}_\tau$ tal que $\mathcal{A}$ seja paraconsistente.

4.3.11 DEFINIÇÃO. Uma *estrutura* $\mathcal{A}$ diz-se *paracompleta* se existe um átomo anotado fechado $p_\lambda t_1, ..., t_n$ tal que

$$\mathcal{A}(p_\lambda t_1, ..., t_n) = 0 = \mathcal{A}(\neg p_\lambda t_1, ..., t_n)$$

O sistema $\mathcal{Q}_\tau$ diz-se *paracompleto* se existe uma estrutura $\mathcal{A}$ para $\mathcal{Q}_\tau$ tal que $\mathcal{A}$ seja paracompleto.

4.3.12 TEOREMA. $\mathcal{Q}_\tau$ é paraconsistente se e somente se $\#\tau \geq 2$. (O símbolo $\#$ indica o número cardinal de τ).

4.3.13 TEOREMA. Se $\mathcal{Q}_\tau$ é paracompleto, então $\#\tau \geq 2$. Se $\#\tau \geq 2$, existem sistemas $\mathcal{Q}_\tau$ que são paracompletos e existem $\mathcal{Q}_\tau$ que não são paracompletos.

4.3.14 TEOREMA. Se $\mathcal{Q}_\tau$ é não-alética então $\#\tau \geq 2$. Se $\#\tau \geq 2$, existem sistemas $\mathcal{Q}_\tau$ que são não-aléticos e sistemas $\mathcal{Q}_\tau$ que não são não-aléticos.

Por conseguinte, vê-se que os sistemas $\mathcal{Q}_\tau$ são em geral paraconsistentes, paracompletos e não-aléticos.

4.3.15 TEOREMA. O cálculo $\mathcal{Q}_\tau$ é não-trivial.

Em [Abe92] foram demonstrados teoremas de correção e de completeza para os cálculos $\mathcal{Q}_\tau$. Em [CAS91] evidencia-se que a teoria anotada dos conjuntos é extraordinariamente forte, envolvendo, como caso particular, a teoria dos conjuntos difusos — *fuzzy sets*.

4.4 Programação Lógica Paraconsistente

Blair e Subrahmanian [BS88, Sub87a] introduziram o uso de lógicas anotadas em programação lógica. Seus trabalhos mostraram que é possível e conveniente associar anotações às cláusulas Horn [Bra86].

Esta seção contém um apanhado geral de tais trabalhos. Para fixar, considera-se um retículado finito $\tau = \langle |\tau|, \leq \rangle$, onde $\tau = \{ \bot , t, f, lt, lf, \top \}$, como foi definido em [AS87]. Os elementos que compõem o reticulado, conhecidos como constantes anotacionais, representam respectivamente: *indefinido, verdade, falso, quase verdadeiro, quase falso e sobre definido* — em uma lógica com 6 anotações. O símbolo $\top$, de forma intuitiva, pode ser visto como inconsistente. Na Figura 4.1 temos o diagrama de Hasse e podemos observar a ordem $\leq$ subjacente.

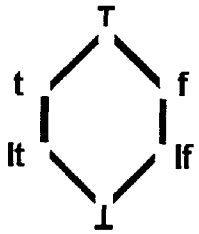

Figura 4.1: Reticulado Seis

Considera-se, também, o operador $\sim$: $|\tau| \to |\tau|$, definido a seguir (a exposição que se segue pode ser adaptada para um reticulado finito arbitrário)

4.4.1 DEFINIÇÃO. O operador $\sim$: $\tau \to \tau$ é definido como [Sub87a, CPA+95]:

- $\sim (t) = f$;
- $\sim (f) = t$;
- $\sim (lf) = lt$;
- $\sim (lt) = lf$;
- $\sim (\bot) = \bot$;
- $\sim (\top) = \top$.

4.4.1 Sintaxe

DEFINIÇÃO. 4.4.2 (LITERAL ANOTADO) Se p é uma fórmula básica e $\mu \in \{t, f\}$, então diz-se que p_μ é um literal *bem-anotado*, e que μ é uma *boa-anotação* de p.

Às vezes escreve-se $p{:}\mu$ em vez de p_μ; as demais noções sintáticas são introduzidas como na seção anterior.

Intuitivamente, o átomo anotado qualquer $p{:}\mu$ pode ser lido como: "crê-se em p com o grau de crença menor do que ou igual a μ".

DEFINIÇÃO. 4.4.3 (CLÁUSULA HORN GENERALIZADA) Se $p_0{:}\mu_0,...,p_n{:}\mu_n$ são literais bem-anotados, então

$$p_0{:}\mu_0 \Leftarrow p_1{:}\mu_1 \wedge ... \wedge p_n{:}\mu_n$$

chama-se *cláusula horn generalizada* — ou cláusula-hg, $p_0{:}\mu_0$ chama-se *cabeça da cláusula*, enquanto que $p_1{:}\mu_1 \wedge ... \wedge p_n{:}\mu_n$ denomina-se *corpo da cláusula*.

A aplicação de uma substituição θ para uma fórmula básica p de um literal anotado $p{:}\mu$ resulta no literal $p\theta{:}\mu$. A noção de substituição é estendida de forma óbvia para uma conjunção de literais anotados e para cláusulas-hg.

Uma *unificação mais geral* — umg — de fórmulas básicas p e q é a unificação θ, tal que para qualquer unificação ξ dos literais p e q, há uma substituição γ, tal que $\theta\gamma = \xi$. Se dois literais anotados são unificáveis, então eles possuem uma unificação mais geral.

DEFINIÇÃO. 4.4.4 (UNIFICAÇÃO) Se $p{:}\mu$ e $q{:}\rho$ são literais, então diz-se que $p{:}\mu$ e $q{:}\rho$ são unificáveis se p e q são unificáveis.

DEFINIÇÃO. 4.4.5 (PROGRAMA HORN GENERALIZADO) Um Programa Horn Generalizado — PHG — é qualquer conjunto finito, não vazio, de cláusulas-hg.

4.4.2 Semântica

Todas as interpretações têm como domínio a base de Herbrand, isto é, o universo de indivíduos da interpretação consiste dos termos básicos da linguagem que está sendo interpretada. Uma interpretação é uma função $I{:}B_L \rightarrow \tau$, onde B_L é a base Herbrand em consideração e τ é o reticulado subjacente.

4.4.6 DEFINIÇÃO. As notações $(\forall)F$ e $(\exists)F$ são usadas para denotar *fechamento universal* e *fechamento existencial* da fórmula F.

4.4.7 DEFINIÇÃO. Diz-se que a interpretação I satisfaz um Programa Horn Generalizado G se ela satisfaz todas as cláusulas-hg C pertencentes a G.

4.4.8 DEFINIÇÃO. Se I satisfaz um Programa Horn Generalizado G, então diz-se que I é um *modelo* para G.

A ordenação $\leq$ sobre as constantes anotacionais é estendida, de forma óbvia, para as interpretações. De fato, dado o PHG G e as interpretações Herbrand I_1 e I_2, diz-se que

$$I_1 \leq I_2 \text{ se e somente se } (\forall p \in B_G) I_1(p) \leq I_2(p)$$

onde B_G é a base Herbrand de G. Note-se que o conjunto das interpretações constitui um reticulado completo em relação à ordem $\leq$.

4.4.9 DEFINIÇÃO. Se C é uma cláusula-hg em G, então o resultado da substituição de todo literal negado $\neg p{:}\mu$ em C por $p :\sim \mu$ chama-se *contraparte positiva* C^{pos} de C.

4.4.10 DEFINIÇÃO. O PHG obtido pela substituição de cada cláusula-hg C no PHG G por C^{pos} chama-se *contraparte positiva* de G e denota-se por G^{pos}.

4.4.11 LEMMA. $I \models (\exists)\neg p{:}\mu$ se e somente se $I \models (\exists)p :\sim \mu$. Analogamente, $I \models (\forall)\neg p{:}\mu$ se e somente se $I \models (\forall)p :\sim \mu$.

4.4.12 TEOREMA. I é um modelo de G se e somente se I é um modelo de G^{pos}.

O teorema acima assegura que o dispositivo de anotação é poderoso o suficiente para tornar o uso de átomos negados desnecessários. Conseqüentemente, a partir desse ponto, será assumido que os PHGs não possuem negação de literais.

4.4.13 DEFINIÇÃO. Supondo-se que G é um PHG, define-se T_G como uma aplicação de interpretações Herbrand de G em interpretações Herbrand de G, onde

$$T_G(I)(p) = \textbf{sup}\ \{\mu | p{:}\mu \Leftarrow q_1{:}\mu_1 \wedge ... \wedge q_k{:}\mu_k \text{ é uma instância básica de uma cláusula-hg em } G \text{ e } I \models q_1{:}\mu_1 \wedge ... \wedge q_k{:}\mu_k\}$$

4.4.14 DEFINIÇÃO. A interpretação I de um PHG diz-se *normal* se verifica a seguinte condição:

$$(\forall p \in B_G) T_G(p) \neq \top$$

4.4.15 DEFINIÇÃO. Diz-se que um PHG G é *bem-comportado* se as cláusulas-hg de G satisfazem às seguintes condições:

1. se C_1 e C_2 são cláusulas-hg em G, sendo suas cabeças $p_1{:}\mu_1$ e $p_2{:}\mu_2$, respectivamente e
2. se p_1 e p_2 são unificáveis, então μ_1 é comparável com μ_2

4.4.3 A Semântica Operacional dos PHGs

4.4.16 DEFINIÇÃO. Se G é um PHG, p é um átomo na linguagem de G e μ uma anotação, define-se uma árvore *e/ou* $T(G, p{:}\mu)$ da seguinte forma:

1. a *raiz* de $T(G, p{:}\mu)$ é um *nó-ou* rotulado $p{:}\mu$;
2. se N é um *nó-ou* , então ele é rotulado por um literal anotado simples;
3. cada *nó-e* é rotulado por uma cláusula-hg de G e por uma substituição;
4. descendentes de um *nó-ou* são *nós-e* e os descendentes de *nós-e* são *nós-ou* ;
5. se N é um *nó-ou* rotulado por $q{:}\alpha(\alpha \neq \bot)$, e se $C\theta$ é uma instância de uma cláusula-hg C em G da seguinte forma:

$$q{:}\beta \Leftarrow D_1{:}\psi_1 \wedge ... \wedge D_k{:}\psi_k$$

onde, $\beta \geq \alpha$, então há um descendente de N rotulado por C e θ. Um *nó-ou* sem descendente chama-se *nó não-informativo*;

6. se N é um *nó-e* rotulado por uma cláusula C e a substituição θ, então para todo literal anotado $q{:}\gamma$ no corpo de C, há um *nó-ou* descendente rotulado $q\theta{:}\gamma$. Um *nó-e* sem descendente chama-se *nó-sucesso*.

Associado a todo nó N da árvore *e/ou* $T(G, p{:}\mu)$ existe uma constante anotacional $\nu(N)$, chamada *valor do nó*.

Define-se ν assim:

1. se N é um nó sucesso rotulado $q{:}\psi$, então $\nu(N) = \psi$;
2. se N é um nó não-informativo, então $\nu(N) = \bot$;
3. se N é um *nó-ou* que não é não-informativo e seus descendentes são $N_1, ..., N_k$, então $\nu(N) = \mathbf{sup}\ \{\nu(N_1), ..., \nu(N_k)\}$;
4. se N é um *nó-e* e não-terminal com a cláusula-hg $p{:}\rho \Leftarrow q_1{:}\psi_1 \wedge ... \wedge q_k{:}\psi_k$, e se o valor $\nu(N_i)$ de cada um de seus nós descendentes N_i rotulados q_i é tal que $\nu(N_i) \geq \psi_i$ para todo $1 \leq i \leq k$, então $\nu(N) = \rho$; senão $\nu(N) = \bot$.

4.4.17 DEFINIÇÃO. Um PHG G chama-se *coberto* se toda variável que ocorre no corpo de uma cláusula-hg $C \in G$ ocorre também na cabeça de C. Caso contrário, diz-se que G é *não-coberto*.

4.4.18 TEOREMA. Se G é um PHG coberto, $p \in B_G$ e se $T(G; p{:}\mu)$ é finito com raiz R, então $\nu(R) \leq T_G \uparrow w(p)$.

4.4.19 TEOREMA. Se G é um PHG não-coberto, $p \in B_G$ e se $T(G; p{:}\mu)$ é finito com raiz R, então $\nu(R) \geq T_G \uparrow w(p)$.

1 COROLÁRIO. Se G é um PHG coberto, $p \in B_G$ e se $T(G; p{:}\mu)$ é finita, tendo R como raiz, então $\nu(R) = T_G \uparrow w(p)$.

O procedimento de resolução-SLD padrão, como descrito em [Llo84], não pode ser aplicado a PHGs, devido a algumas peculiaridades do cálculo de $T_G \uparrow w(p)$ para $p \in B_G$. Por esse motivo, em [BS87] propõe-se um novo procedimento de resolução, denominado *resolução-SLDnh*, onde *nh* é a abreviação para "não-Horn" — que pode ser aplicado a PHGs.

4.4.20 DEFINIÇÃO. Um *resolvente-nh* em relação a $p_i{:}\rho_i$ do questionamento Q dado por $p_1{:}\rho_1 \wedge ... \wedge p_k{:}\rho_k$ e a cláusula-hg C da forma $d{:}\beta \Leftarrow q_1{:}\psi_1 \wedge ... \wedge q_r{:}\psi_r$ é o questionamento

$$(p_1{:}\rho_1 \wedge ... \wedge p_{i-1} \wedge q_1{:}\psi_1 \wedge ... \wedge q_r{:}\psi_r \wedge p_{i+1}{:}\rho_{i+1} \wedge ... \wedge p_k{:}\rho_k)\theta$$

onde $\beta \geq \rho_i$ e θ é o unificador mais geral (umg) de d e p_i.

4.4.21 DEFINIÇÃO. Uma *dedução-SLDnh* de um questionamento inicial Q_0 a um PHG G é a seqüência finita

$$\langle Q_0, C_1, \theta_1\rangle, ..., \langle Q_i, C_{i+1}, \theta_{i+1}\rangle, ..., \langle Q_n, C_{n+1}, \theta_{n+1}\rangle$$

onde Q_{i+1} é o resolvente-nh de Q_i e C_{i+1}; C_{i+1} é alguma cláusula-hg de G que foi renomeada para não ter símbolos de variáveis em comum com algum questionamento $Q_0, ..., Q_i$ ou alguma cláusula-nh $C_0, ..., C_i$ e θ_{i+1} é o unificador mais geral de Q_i e C_{i+1}

4.4.22 DEFINIÇÃO. Uma *refutação-SLDnh* do questionamento inicial Q_0 é uma dedução-SLDnh finita tal que o resolvente de Q_i e C_{i+1} é uma cláusula-hg vazia.

4.4.23 TEOREMA. Se existe uma *refutação-SLDnh* do questionamento inicial

$$Q_0 = p_1{:}\rho_1 \wedge ... \wedge p_m{:}\rho_m$$

a um PHG G, então

$$T_G \uparrow w \models \exists(Q_0)$$

4.4.24 TEOREMA. Se G é um PHG coberto e se Q_0 é um questionamento satisfeito por $T_G \uparrow w$, então há uma refutação-nh de Q_0 a partir de G.

A *resolução-SLDnh* não pode ser aplicada diretamente a um PHG que não seja bem-comportado. Para tais programas, é proposto em [BS87] um dispositivo técnico chamado *fechamento*.

4.4.25 DEFINIÇÃO. Um PHG G diz-se *fechado* se para quaisquer duas cláusulas-hg C_1 e C_2 pertencentes a G da forma

$$p_1{:}\rho_1 \Leftarrow q_1{:}\mu_1 \wedge ... \wedge q_k{:}\mu_k (K \geq 0)$$
$$p_2{:}\rho_2 \Leftarrow d_1{:}\psi_1 \wedge ... \wedge d_m{:}\psi_m (m \geq 0)$$

tais que, p_1 e p_2 são unificáveis — através de uma umg θ — e ρ_1 e ρ_2 são não comparáveis, há uma cláusula-hg

$$p_1\theta : \mathbf{sup}\ \{\rho_1, \rho_2\} \Leftarrow q_1{:}\mu_1 \wedge ... \wedge q_k{:}\mu_k \wedge d_1{:}\psi_1 \wedge ... \wedge d_m{:}\psi_m$$

representada por $\lambda(C_1, C_2)$.

4.4.26 DEFINIÇÃO. Seja G um PHG, ponha-se:

$$A_1(G) = G,$$
$$A_{n+1}(G) = A_n(G) \cup \{\lambda(C_1, C_2) | C_1, C_2 \in A_n(G)\} (n \geq 1)$$

Para todo PHG G há um inteiro n tal que $A_n(G) = A_{n+1}(G)$. $A_n(G)$ chama-se fechamento de G e é denotado por $CL(G)$.

4.4.27 TEOREMA. Se G é um PHG fechado, então:

1. I é um modelo de G se e somente se I é um modelo de $CL(G)$,

2. $T_G = T_{CL(G)}$.

4.4.28 DEFINIÇÃO. Um PHG sobre τ denomina-se *inconsistente por negação* se há alguma cláusula-hg $p : \mu$, $(p \in B_G)$ tal que:

1. $T_G \uparrow w \models p{:}\mu$, e

2. $T_G \uparrow w \models p{:}\neg(\mu)$

4.4.29 DEFINIÇÃO. Um PHG denomina-se *não-trivial* se há alguma cláusula-hg $p{:}\mu$ tal que $T_G \uparrow w$ não satisfaz $p{:}\mu$.

4.4.30 DEFINIÇÃO. Um PHG chama-se *paraconsistente* se G for inconsistente por negação e não-trivial.

4.5 ParaLog

Nesta seção faremos um apanhado geral da linguagem lógica ParaLog, com base no trabalho de Da Costa,Prado, Abe, Ávila e Rillo [CPA+95].

A linguagem ParaLog, ou Prolog Paraconsistente, é uma variação da linguagem Prolog baseada na lógica Anotada $\mathcal{Q}_\tau$, descrita na seção anterior, que permite manipular a inconsistência, a paracompleteza ou ambas.

A semântica de um programa escrito em ParaLog está baseada na semântica do modelo mínimo de Herbrand. Assim, os novos recursos introduzidos pelo ParaLog são independentes da aplicação e são embasadas na lógica de primeira ordem, que é correta e completa em relação à semântica utilizada.

4.5.1 Sintaxe de ParaLog

A implementação do Prolog Paraconsistente está baseada na sintaxe do Prolog de Edinburgh [War79] conhecido como sintaxe $[DEC - 10]$. Essa sintaxe recebeu alguns novos elementos relacionados com a programação lógica paraconsistente, descrita na seção anterior.

DEFINIÇÃO. 4.5.1 (ALFABETO) O alfabeto básico do Prolog Paraconsistente possui o mesmo conjunto de símbolos do Prolog padrão acrescido do símbolo ":" e dos símbolos anotacionais. Os símbolos do Prolog Paraconsistente são os seguintes:

1. letras: *a, b,..., z, A, B,...,Z*;

2. digitos: *0, 1,..., 9*;

3. símbolos especiais: $\_, +, -, /, *$ e *espaço em branco*;

4. símbolos de pontuação: *(,)* , . , “, ” ;

5. símbolos de conectivos: conjunção (&), implicação (<--) , negação (*not*);

6. símbolo de anotação: “:”,

7. símbolos anotacionais: *t, f ,lt, lf, tf, **.

DEFINIÇÃO. 4.5.2 (EXPRESSÃO) Uma expressão do Prolog Paraconsistente é qualquer sequência finita de símbolos de seu alfabeto.

DEFINIÇÃO. 4.5.3 (ÁTOMO) Um átomo do Prolog Paraconsistente é:

1. toda expressão formada de letras e digitos, iniciada com uma letra minúscula;

2. uma expressão formada por digitos com no máximo uma ocorrência do símbolo “.”;

3. toda expressão — incluindo o branco — delimitada por aspas.

DEFINIÇÃO. 4.5.4 (CONSTANTE E VARIÁVEL) :

→ Uma constante do Prolog Paraconsistente é:

1. um átomo; ou

2. um elemento do reticulado τ, chamado constante anotacional.

→ Uma variável no Prolog Paraconsistente é:

1. uma expressão de letras e digitos cujo primeiro elemento é uma letra maiúscula ou

2. o símbolo “_”, chamado variável anônima.

4.5.5 DEFINIÇÃO. O conjunto dos termos do Prolog Paraconsistente é definido indutivamente, como se segue:

1. uma variável é um termo;

2. uma constante é um termo;

3. se f é um átomo e f tem o papel de um símbolo funcional n-ário e $t_1, ..., t_n$ são termos, então $f(t_1, ..., t_n)$ é um termo;

4. Uma expressão do Prolog Paraconsistente constitui um termo se e somente se for obtida aplicando-se uma das condições 1, 2, 3 anteriores.

4.5.6 DEFINIÇÃO. Uma fórmula atômica anotada é uma expressão da forma $p(t_1, ..., t_n):\mu$, para $n \geq 0$ e $\mu \in \tau$, onde $t_1, ..., t_n$ são termos e p é um átomo no papel de um símbolo predicativo n-ário. Quando n for igual a zero, para simplificar, escrevemos $p():\mu$ como $p:\mu$.

Note-se que um átomo pode representar simultaneamente o papel de um ou mais símbolos funcionais ou predicativos de aridades diferentes. Esse múltipo do mesmo átomo não causa ambigüidade, pois o contexto de um programa indica sempre o papel que o mesmo representa.

DEFINIÇÃO. 4.5.7 (CLÁUSULAS ANOTADAS) O conjunto das cláusulas anotadas do Prolog Paraconsistente é definido indutivamente como:

1. se p é uma fórmula atômica anotada, então p é uma cláusula anotada, chamada *cláusula anotada unitária* ;

2. se $p:\mu$ e $q_1:\mu_1$,..., $q_n:\mu_n$ são fórmulas atômicas anotadas, então a expressão $p:\mu$ <-- $q_1:\mu_1$ & ... & $q_n:\mu_n$ é uma cláusula anotada, chamada de *cláusula anotada não unitária* , onde $p:\mu$ é a cabeça e $q_1:\mu_1$ & ... & $q_n:\mu_n$ é o corpo da cláusula;

3. se $p_1:\mu_1$, ... , $p_n:\mu_n$ são fórmulas atômicas anotadas, então <-- $p_1:\mu_1$ & ... & $p_n:\mu_n$ é uma cláusula anotada, chamada *cláusula objetivo* . Uma consulta Prolog é uma cláusula objetivo,

4. uma expressão do Prolog Paraconsistente constitui uma cláusula se e somente se foi obtida aplicando-se umas das definições 1, 2 ,3 anteriores.

DEFINIÇÃO. 4.5.8 (PROGRAMA PROLOG PARACONSISTENTE) Um programa Prolog Paraconsistente é um conjunto finito de cláusulas anotadas unitárias e não unitárias.

4.5.9 EXEMPLO. Um programa paraconsistente de relações familiares com definição de projenitor [CPA+95]

```
progenitor(bianca,bruna):t.
progenitor(braulio,bruna):t.
progenitor(malu,yago):t.
progenitor(ricardo,yago):t.
progenitor(carlos,maria):t.
progenitor(maria,gustavo):t.
avo(X,Y):t <--
        progenitor(X,Z):t &
        progenitor(Z,Y):t.
```

No referido exemplo, o fato de que Ricardo é progenitor de Yago é representado como:

```
progenitor(ricardo,yago):t.
```

Essa cláusula pode ser lida como: "Acredita-se com uma crença menor ou igual a t que Ricardo é progenitor de Yago". Observe-se que os fatos são escritos em letras minúscula, o que se faz necessário devido à definição sintática da linguagem. O uso de uma letra maiúscula indica a intenção de se definir uma variável — por exemplo, uma variável denominada RICARDO — e não um átomo como é o desejado.

4.5.10 DEFINIÇÃO. A definição da gramática da linguagem Prolog Paraconsistente na notação Backus-Naur Form (BNF) é apresentada abaixo:

⟨programa⟩ ::= ⟨cláusula⟩ ⟨resto-programa⟩

⟨cláusula⟩ ::= ⟨fato⟩ | ⟨regra⟩

⟨resto-programa⟩ ::= ⟨cláusula⟩ | ϕ[1]

⟨fato⟩ ::= ⟨átomo⟩ : ⟨anotação⟩ | ⟨átomo⟩ (⟨ argumento⟩) : ⟨anotação⟩

⟨regra⟩ ::= ⟨cabeça⟩ <-- ⟨corpo⟩

⟨cabeça⟩ ::= ⟨fato⟩

⟨corpo⟩ ::= ⟨fato⟩ | ⟨fato⟩ ⟨resto-corpo⟩

⟨resto-corpo⟩ ::= , ⟨fato⟩ | , ⟨fato⟩ ⟨resto-corpo⟩ | ϕ

⟨argumento⟩ ::= ⟨átomo⟩ ⟨resto-argumento⟩ | ⟨variável⟩ ⟨resto-argumento⟩

⟨resto-argumento⟩ ::= , ⟨argumento⟩ | , ⟨argumento⟩ ⟨resto-argumento⟩ | ϕ

⟨átomo⟩ ::= ⟨letra-minúscula⟩ ⟨nr-letras⟩

⟨variável⟩ ::= ⟨letra-maiúscula⟩ ⟨nr-letras⟩

[1] O símbolo ϕ representa a expressão vazia.

⟨nr-letras⟩ ::= ⟨digito⟩ ⟨nr-letras⟩ ⟨letra⟩ ⟨nr-letras⟩ | ϕ

⟨letra⟩ ::= ⟨letra-minúscula⟩ | ⟨letra-maiúscula⟩

⟨anotação⟩ ::= t | f | lt | lf | tf | *

⟨digito⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨letra-minúscula⟩ ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | p | q | r
| s | t | u | w | x | y | z

⟨letra-maiúscula⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
| P | Q | R | S | T | U | W | X | Y | Z

Observe-se que a definição do símbolo não terminal ⟨anotação⟩ está relacionada com o reticulado τ adotado nesta implementação. Para o uso de outros reticulados, basta redefinir esse símbolo não terminal.

Para realizar uma consulta em Prolog Paraconsistente, podemos seguir a notação BNF proposta abaixo:

⟨meta⟩ ::= ⟨fato⟩ ⟨resto-meta⟩

⟨resto-meta⟩ ::= & ⟨fato⟩ | & ⟨fato⟩ ⟨resto-meta⟩ | ϕ

4.5.2 Semântica ParaLog

Os resultados e os conceitos da semântica declarativa e procedimental utilizadas nos Programas de Horn Generalizados (PHG), vistos na Seção 2.4, podem ser aplicados aos programas em Prolog Paraconsistente.

Cumpre lembrar que o reticulado utilizado nesta versão do Prolog Paraconsistente é o reticulado definido em [AS87] e, portanto, toda interpretação I tem como conjunto imagem o reticulado $\tau = \{t, f, lt, lf, *, tf\}$[2] , como se descreveu anteriormente.

Da mesma forma que um programa Prolog padrão, um programa $\mathcal{P}_0$ e uma consulta $\mathcal{Q}_0$ em Prolog Paraconsistente devem passar por um processo de regularização, no qual:

- toda variável anônima é substituída por uma nova variável;
- todos os átomos que ocorrem em mais de um papel serão renomeados, de forma que ao final desse processo, não há átomos com o mesmo nome em papéis diferentes.

[2]Os símbolos * e tf são representações para $\bot$ e $\top$ mostrados na Figura 4.1.

O programa $\mathcal{P}_1$ e a consulta $\mathcal{Q}_1$ resultantes desse processo de regularização são equivalentes a $\mathcal{P}_0$ e $\mathcal{Q}_0$ respectivamente, como se demonstra [CGF87]. Diferentemente do Prolog Padrão, um programa em Prolog Paraconsistente deve passar, ainda, por mais duas transformações sintáticas:

- Eliminação da negação; e
- Fechamento.

As transformações sintáticas são necessárias para se eliminar as facilidades sintáticas introduzidas na linguagem, que não fazem parte da sintaxe dos PHG.

O programa $\mathcal{P}_1$ e a consulta $\mathcal{Q}_1$ passam, então, pelo processo de eliminação da negação. Nesse processo, todas as ocorrências das fórmulas atômicas anotadas da forma *not* $p{:}\mu$ são substituídas por $p :\sim \mu$. A eliminação da negação de $\mathcal{P}_1$ e $\mathcal{Q}_1$ resulta no programa $\mathcal{P}_2$ e na consulta $\mathcal{Q}_2$.

A última transformação sintática é o fechamento como descrito em [BS87], e o programa $\mathcal{CL}(P_2)$ resultante dessa transformação é logicamente equivalente a $\mathcal{P}_2$, isto é, I é um modelo de $\mathcal{P}_2$ se e somente se I for um modelo de $\mathcal{CL}(P_2)$.

O programa $\mathcal{CL}(P_2)$ e a consulta $\mathcal{CL}(Q_2)$ resultante desse processo de transformação sintática são equivalentes ao programa e à consulta $\mathcal{P}_0$ e $\mathcal{Q}_0$ em Prolog Paraconsistente. O programa $\mathcal{CL}(P_2)$ é equivalente a um PHG $\mathcal{G}$ e a consulta $\mathcal{CL}(Q_2)$ é equivalente a uma consulta $\mathcal{C}$, se e somente se:

1. uma cláusula $p{:}\mu$ ocorre em $\mathcal{CL}(P_2)$ se e somente se existe uma cláusula $p{:}\mu$<-- em $\mathcal{G}$;
2. uma cláusula não-unitária Prolog $p{:}\mu$<--$q_1{:}\mu_1$&...&$q_n{:}\mu_n$ ocorre em $\mathcal{CL}(P_2)$ se e somente se existe uma cláusula $p{:}\mu$<--$q_1{:}\mu_1$&...&$q_n{:}\mu_n$ em $\mathcal{G}$,
3. $\mathcal{CL}(Q_2)$ é da forma <--$\mathcal{C}_1{:}\mu_1$&...&$\mathcal{C}_n{:}\mu_n$ se e somente se $\mathcal{C}$ for da forma $\mathcal{C}_1{:}\mu_1 \wedge ... \wedge \mathcal{C}_n{:}\mu_n$.

Ao término dessas transformações sintáticas, o programa e a consulta resultantes podem ser manipulados como PHG's *bem-comportados*, nos quais a resolução-SLDnh pode ser aplicada.

4.5.3 Avaliação dos Programas em ParaLog

O mecanismo usado pelo motor de inferência do Prolog Paraconsistente baseia seu funcionamento no método de resolução-SLDnh [BS87].

A função de seleção f mapeia uma meta $C_1{:}\mu_1$ & ... & $C_n{:}\mu_n$ no literal $C_i{:}\mu_i$, (onde $1 \leq i \leq n$ e $\mu_i = \mathbf{sup}\ \{\ \mu_1 ... \mu_n\ \}$). Cumpre ressaltar que a função f empregada no motor de inferência do Prolog Paraconsistente não é a função de seleção padrão descrita em [Llo84].

O procedimento de seleção das cláusulas do programa também não segue a estratégia padrão. Nesse motor de inferência, as cláusulas do programa são selecionadas de tal forma que a cláusula $C{:}\mu$ selecionada possui anotação μ igual ao supremo do conjunto formado pelas anotações das cláusulas candidatas.

Essas estratégias de seleção fazem com que o procedimento de refutação do motor de inferência simule uma busca semelhante ao *best-first* na árvore de refutação para as cláusulas do programa $\mathcal{CL}(P_2)$.

Dados um programa $\mathcal{P}$ e uma consulta $\mathcal{Q}$, o motor de inferência dá como resposta uma crença ψ tal que $\psi \in \tau$. Nos casos em que $\psi \neq *$, a resposta também inclui uma substituição θ para as variáveis de $\mathcal{Q}$.

4.5.4 Programando com o Prolog Paraconsistente

Além da capacidade de manipular bases de conhecimento inconsistentes, o prolog paraconsistente possui outros aspectos, inerentes ao fato de estar assentado em um lógica anotada, que devem ser explorados ao se escrever programas nessa linguagem. Para a melhor compreensão, tais aspectos estão divididos em dois grupos:

1. Aspectos Semânticos;
2. Aspectos de Controle.

Aspectos Semânticos

O uso de anotação permite a definição de cláusulas com diferentes constantes anotacionais. Essa é uma característica muito importante da linguagem, pois muitas vezes o programador deseja representar apenas suas crenças e não verdades absolutas [BS88]. Ele

pode, por exemplo, escrever certas cláusulas porque acredita que elas sejam verdadeiras e não que elas sejam obrigatoriamente verdadeiras. Por exemplo, ao analisar o Exemplo 4.5.11, intuitivamente, pode-se presumir — com um certo grau de crença — que Gustavo é o progenitor de Fernando, uma vez que é sabido que Gustavo é casado com Ana e que esta é progenitora de Fernando.

4.5.11 EXEMPLO. Relações Familiares [CPA+95]

```
homem(ricardo):t.
homem(braulio):t.
homem(gustavo):t.
homem(fernando):t
homem(yago):t.
mulher(malu):t.
mulher(bianca):t.
mulher(ana):t.
mulher(bruna):t.
casado(bianca,braulio):t.
casado(malu,ricardo):t.
casado(ana,gustavo):t.
progenitor(bianca,bruna):t.
progenitor(braulio,bruna):t.
progenitor(malu,yago):t.
progenitor(ricardo,yago):t.
progenitor(carlos,maria):t.
progenitor(maria,gustavo):t.
progenitor(ana,fernando):t.
```

É possível generalizar esse raciocínio com a seguinte cláusula prolog paraconsistente:

```
progenitor(X,Y):lt <--
        progenitor(Z,Y):t &
        casado(X,Z):t.
```

Quando consultado, o motor de inferência associa à cláusula progenior(ricardo,yago) uma crença maior que a associada à cláusula progenior(gustavo,fernando) — conforme o reticulado $lt \sqsubset t$ — indicando uma crença maior na paternidade de Ricardo do que na de Gustavo.

Outra importante característica do prolog paraconsistente é que seu motor de inferência não trata a negação como a "impossibilidade de prova", ou seja, ele não segue o princípio do mundo fechado, descrito por [Llo84]. Dada uma consulta $\mathcal{M}$ e um programa $\mathcal{P}$, se não for possível provar $\mathcal{M}$ a partir de $\mathcal{P}$, o motor de inferência retornará à crença * — desconhecido — para $\mathcal{M}$ e não ao valor falso como ocorre com o Prolog padrão.

Quando necessário, o grau de crença f deverá ser associado a uma cláusula de forma explícita, pelo programador. Por exemplo, a regra "um homem não é uma mulher" pode ser assim definida:

```
homem(X):f <--
        mulher(X):t.
```

Aspectos de Controle

Em uma linguagem de programação lógica ideal, os aspectos imperativos da programação — ou seja, o controle — deveriam ser deixados a cargo do motor de inferência da linguagem e os programas deveriam expressar apenas o componente lógico do algoritmo [Kow79]. Contudo, na grande maioria das linguagens de programação lógica existentes, entre elas o Prolog, o programador deve — em troca de uma maior eficiência dos programas — preocupar-se com certos aspectos de controle.

Ao escrever seus programas em prolog paraconsistente o programador, pode, além de contar com mecanismos de controle existentes no Prolog padrão, tirar proveito de anotação e da resolução-SLDnh, utilizando-os como um componente a mais de controle da linguagem.

4.5.12 EXEMPLO. Programa em Prolog Padrão [CPA+95]

```
casado(bianca,braulio).
casado(malu,ricardo).
casado(ana,gustavo).
casado(X,Y):-
        casado(Y,X).
```

Se for feita a seguinte consulta ao programa prolog do Exemplo 4.5.12:

```
?- casado(ana,fernando).
```

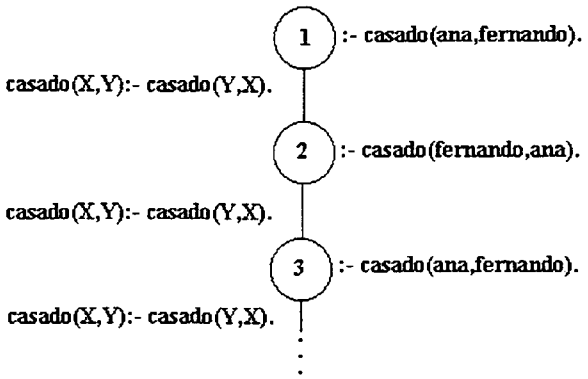

Figura 4.2: Árvore de Resolução do Prolog Padrão

o motor de inferência do Prolog padrão não retornará qualquer resultado, uma vez que o mesmo fará uma busca por um ramo infinito da árvore de resolução, mostrado na Figura 4.2.

No prolog paraconsistente, o programador pode usar a anotação para evitar a criação desse ramo infinito na árvore de resolução; o programa descrito anteriormente ficaria como no Exemplo 4.5.13.

4.5.13 EXEMPLO. Representação do Exemplo 4.5.12 em Prolog Paraconsistente

```
casado(bianca,braulio):t.
casado(malu,ricardo):t.
casado(ana,gustavo):t.
casado(X,Y):lt <--
        casado(Y,X):t.
```

O programador pode tirar proveito da função de seleção do motor de inferência do prolog paraconsistente, para aumentar a eficiência de seu programa. Como essa função seleciona sempre a cláusula com a maior anotação, o programador pode controlar a busca na árvore de resolução, fazendo com que o motor de inferência explore primeiro os ramos mais críticos, nos quais há maior possibilidade de insucesso — o que inviabiliza todo o processo de resolução. Para fazer isso, basta que ele associe graus de crença maiores para as cláusulas com maior propensão à falha. A anotação, como no caso anterior, não possui valor semântico, sendo apenas mais um componente de controle.

Deve ser lembrado, ainda, que a seleção das cláusulas e das regras durante o processo de resolução é feita tendo-se em vista o grau de crença associado à cláusula ou à regra e, depois — no caso de valores de crença iguais —, a ordem pela qual estão declarados.

4.6 Considerações Finais

A Lógica Paraconsistente foi proposta por da Costa para fornecer meios de raciocinar sobre inconsistência, a qual é um fenômeno natural que surge na descrição de aspectos do mundo real. Os conceitos introduzidos pela lógica paraconsistente serão referência para a implementação do ambiente interativo de programas lógicos paraconsistentes, descrito no Capítulo 6.

A Programação Lógica Paraconsistente encontrou, em anos recentes, aplicações em Ciência da Computação, onde podemos encontrar situações de conflitos de crença e informações contraditórias e/ou paracompletas. Tais situações de conflitos podem ser encontradas em Inteligência Artificial Distribuída, Base de Dados e Bases de Conhecimentos Distribuídas, Tratamento de Hierarquias de Herança em Sistemas Centrados em Objetos, Ambiente Multi-agentes, Integração Sensorial e outros.

CAPÍTULO 5

Ambiente Operacional Interativo Prolog

Neste capítulo descrevemos o desenvolvimento de um ambiente interativo para linguagem Prolog, utilizando semântica de transição. Para o desenvolvimento do protótipo Prolog, usamos como referência o trabalho desenvolvido por Brunekreef [Bru96], apresentado a seguir.

5.1 Construção do Ambiente Prolog

O protótipo proposto por Brunekreef, TransLog, é uma ferramenta que suporta a transformação interativa de (uma parte de) um programa Prolog por meio de botões que representam passos de transformação.

A ferramenta TransLog foi desenvolvida por meio de especificação algébrica e foi implementada no meta ambiente ASF+SDF [Kli92]. O ambiente Prolog de Brunekreef não faz o tratamento de metas em Prolog, ou seja, dado uma base de fatos e de regras ele não é capaz de responder a uma pergunta (meta).

Nosso objetivo foi criar um novo protótipo a partir da implementação de Brunekreef, sem nos preocuparmos com a questão de desempenho do programa lógico por ele tratado. Assim sendo, nosso protótipo é constituído por um ambiente interativo que, dado um programa $\mathcal{P}$ e uma consulta $\mathcal{C}$ em Prolog, fornece ao usuário verificações sintáticas e semânticas.

A partir dessas verificações conseguimos gerar um interpretador para programas lógicos com especificação algébrica, uma vez que a implementação desenvolvida no meta-ambiente ASF+SDF [BK00] produz como resultado final um interpretador para a linguagem.

A especificação formal completa consiste de doze módulos que serão apresentados nas

seções subseqüentes. O conjunto dos módulos do ambiente Prolog pode ser dividido em quatro diferentes subconjuntos:

1. Básicos: este subconjunto contém dois módulos:
   - *Layout*,
   - *Booleans*.
2. Sintáticos: apresenta a sintaxe do Prolog e é especificada em dois módulos:
   - *PrologTerms*,
   - *PrologProgram*.

   A especificação do ambiente usa um número de funções sobre um programa Prolog, como por exemplo, normalização de listas, verificação de ocorrência e substituição de cláusulas. Essas funções são agrupadas em dois módulos:
   - *NormalisationFunction*,
   - *PrologFunction*.
3. Unificação: realiza o processo de unificação em Prolog, por isso conta com os seguintes módulos:
   - *Substitution*;
   - *Equations*,
   - *Unification*.
4. Avaliação: dado uma base de conhecimento em Prolog, este subconjunto fornece uma resposta a partir de uma meta (pergunta). Para isso contém os módulos:
   - *TermSets*;
   - *VariantClause*,
   - *EvalProlog*.

A relação de dependência entre os módulos é mostrada nas Figuras 5.1 e 5.2. A descrição mais detalhada de cada módulo está nas seções seguintes.

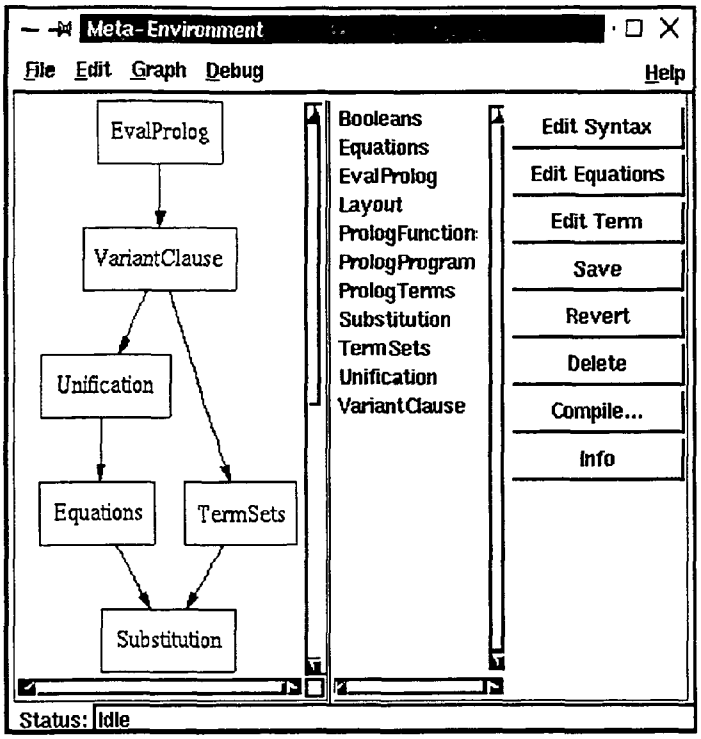

Figura 5.1: Relação de dependência entre os módulos

5.2 Módulos Básicos

Alguns dos módulos aqui utilizados são usados em todas especificação ASF+SDF. Esses módulos são:

5.2.1 *Layout*

Este módulo define alguns elementos sintáticos padrões do meta-ambiente ASF+SDF. Comentários são definidos como %%, finalizando com um novo caractere de linha ($\sim [\backslash n] * [\backslash n]$). O caractere de espaço, o caractere de tabulação ($[\backslash t]$) e o caractere de nova linha ($[\backslash n]$) são definidos também como caracteres padrões (ver Apêndice A.2.7).

5.2.2 *Booleans*

Permite definir as funções booleanas como: *and, or* e *not*, bem como seu critério de prioridades. Este módulo é responsável pelas avaliações booleanas e será utilizado aqui em alguns módulos, por exemplo, *PrologFunctions* (ver Apêndices A.2.1 e A.2.2).

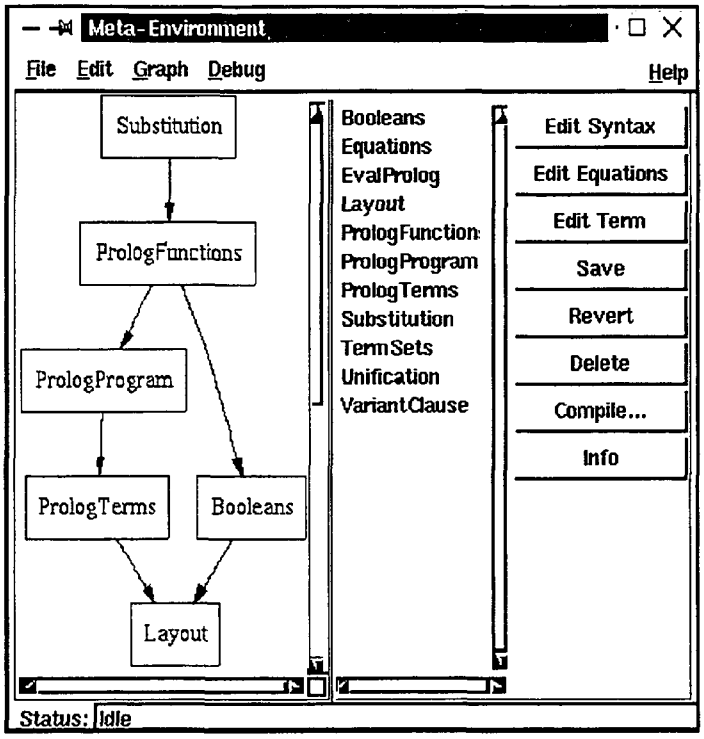

Figura 5.2: Relação de dependência entre os módulos (cont.)

5.3 Módulos Sintáticos

O ambiente interativo Prolog opera sobre programas escritos em sintaxe Prolog (Edinburg). A definição desta seção está baseada na definição utilizada por Brunekreef para especificação do TransLog, com algumas modificações nos módulos *PrologProgram* e *PrologFunctions*. As modificações foram necessárias para o tratamento dos objetivos em Prolog, não abordado por Brunekreef.

5.3.1 *PrologTerms*

Este primeiro módulo contém a definição da sintaxe léxica de vários "tokens": átomos, números (inteiros), símbolos de operação e variáveis. Desses elementos básicos, o termo genérico Prolog (*sort* TERM) é definido. Um termo com operação infixa é permitido. São definidos também construtores de listas (*sort* LIST). O *sort* TERMLIST denota uma seqüência de um ou mais termos em Prolog (ver Apêndice A.2.13).

5.3.2 *PrologProgram*

Este módulo apresenta os *sorts*: PROGRAM CLAUSE OBJECTIVE. Um programa é definido como um conjunto de zero ou mais cláusulas seguidos por uma lista de objetivos. Uma cláusula é uma cláusula unitária com somente uma cabeça ou uma cláusula com uma cabeça e um corpo. O símbolo ":-" é definido como conectivo de separação entre a cabeça e o corpo. O corpo consiste de um TERMLIST (um ou mais termos, separados por vírgula). O objetivo é definido como um termo ou um conjunto de termos (TERMLIST). O símbolo "?-" é usado para indicar o ínicio de um objetivo e representa também um objetivo vazio.

module PrologProgram
imports PrologTerms
exports
sorts PROGRAM CLAUSE OBJECTIVE
context-free syntax

TERM "."	→	CLAUSE
TERM ":-" TERMLIST "."	→	CLAUSE
"?-" TERMLIST "."	→	OBJECTIVE
"?-"	→	OBJECTIVE
CLAUSE* OBJECTIVE	→	PROGRAM

variables

"C"[\']*	→	CLAUSE
"C*"[\']*	→	CLAUSE*
"P"[\']*	→	PROGRAM
"O"[\']*	→	OBJECTIVE

5.3.3 *NormalizationFunction*

Em Prolog, diferentes termos podem representar a mesma lista, por exemplo, $[a,b,c]$ e $[a|[b,c]]$. Neste módulo uma função *norm* é definida, a qual transforma todas as listas em uma forma normal padrão. Nessa forma normal, uma lista é construída por meio de um átomo (dois pontos[1]) com dois argumentos, sendo o primeiro elemento da lista e uma lista com os argumentos restantes. A mesma função é usada para normalizar um termo com um operador infixo para notação prefixa (ver Apêndices A.2.8 e A.2.9).

[1]Usamos ".." ao invés de '.' devido a problema de ambiguidade no ambiente ASF+SDF.

5.3.4 *PrologFunctions*

Este módulo (ver Apêndices A.2.10 e A.2.11) contém um número de funções sobre termos e programas Prolog. Essas funções são necessárias em alguns módulos, por exemplo no módulo *Unification*. Elas podem ser agrupadas em dois subconjuntos:

- Funções Booleanas para testar a propriedade de uma certo termo, por exemplo: É um termo ou uma variável? É uma lista vazia? Um variável ocorre em um termo?. A função *contem* usa uma função auxiliar *contemh*, a qual investiga se uma variável ocorre ou não em um termo normalizado.
- Função sobre uma cláusula (*head*) tem como saída um termo somente, ou seja, a cabeça da cláusula.

5.4 Módulo de Unificação

Nesta seção apresentamos três módulos que são importantes no processo de unificação. No módulo *Unification*, o unificador mais geral (umg) de uma lista de equações de termos é construído. A função unificação está baseada no algoritmo de unificação de Martelli-Montanari [MM82], o qual tem um conjunto de equações $T_1 = T_2$ como entrada. Em um número de passos, esse conjunto é transformado em novos conjuntos de equações. Quando não há mais passos para serem executados e nenhuma falha é detectada, o conjunto final contém o unificador mais geral. A saída da função de unificação não será o conjunto de equações, mas o conjunto de substituições ou uma falha. No módulo *Substitution* a sintaxe da substituição é definida, bem como o resultado de uma seqüência de substituições sobre uma cláusula, um corpo, etc. No módulo *Equations* a sintaxe de um conjunto de equações é definida. Finalmente, no módulo *Unification* o algoritmo de Martelli-Montanari é especificado.

Nas subseções seguintes apresentamos as principais características de cada módulo.

5.4.1 *Substitution*

Neste módulo, a substituição de uma variável por um termo, $V \mapsto T$ é definida. Tal substituição é considerada como uma forma normal do sort SUB. Além disso, um termo

do sort SUBS denota uma seqüência de zero ou mais substituições. A constante *fail* do sort SUBS denota uma falha do algoritmo de unificação.

Dois operadores binários sobre seqüência de substituição são especificados. O operador *inteligente* une duas seqüências de substituições de um modo *smart*: conflitos entre duas substituições, isto é, duas diferentes substituições para uma mesma variável são detectados e manipulados. Além disso, se uma das seqüências é igual à constante *fail*, o resultado do operador *inteligente* é igual ao próprio *fail*. O operador *simples* liga duas seqüências de substituições em um modo simples: as duas seqüências são simplesmente unidas em uma grande seqüência. O operador *simples* usa duas funções auxiliares: *isMember*, que investiga se uma variável é parte ou não de uma seqüência de substituição; e a função *getTerm*, que produz o termo que está sendo substituído por uma dada variável.

Na parte final deste módulo, o efeito de, uma seqüência de substituições sobre vários construtores Prolog (CLAUSE, TERMLIST (corpo) e TERM) é especificado. Uma lista de substituições sobre uma cláusula (C) é denotado por [c S] (ver Apêndices A.2.14 e A.2.15).

5.4.2 *Equations*

Este módulo define a sintaxe de um conjunto de equações de termos Prolog, conjunto esse que é a entrada básica do algoritmo de Martelli-Montanari, a ser especificada na próxima subseção.

O sort *EQ* é relacionado para uma simples equação, enquanto o sort *EQS* relaciona uma seqüência de equações. Em relação a ambos sorts, uma função de substituição é definida. Na definição de equações essas substituições são reduzidas para substituições sobre termos separados (ver Apêndices A.2.3 e A.2.4).

5.4.3 *Unification*

Como mencionado anteriormente, a unificação de termos Prolog é especificada de acordo com o algoritmo de Martelli-Montanari [MM82]. O algoritmo original toma um conjunto de equações de termos como entrada e produz um modificado conjunto de equações como saída.

O algoritmo consiste de seis passos condicionais. Em um processo iterativo, esses passos são aplicados, produzindo um resultado de saída. Tal resultado pode ser uma falha (não foi encontrado nenhum umg).

O algoritmo de unificação toma uma equação de uma seqüência e realiza as seguintes ações:

1. $a(\ t_1, ..., t_n) \equiv a(\ v_1, ..., v_n)$: substituir a equação por equações $t_1 \equiv v_1, ..., t_n \equiv v_n$;
2. $a(\ t_1, ..., t_n) \equiv b(\ v_1, ..., v_m)$ com $a \neq b$ ou $n \neq m$: metade com falha;
3. $v \equiv v$: eliminar a equação;
4. $t \equiv v$ onde t não é uma variável: substituir a equação pela equação $v \equiv t$;
5. $v \equiv t$ onde $v \neq t$, v não ocorre em t e v ocorre em outro lugar: realizar a substituição de $v \mapsto t$ em toda parte exceto na equação corrente;
6. $v \equiv t$ onde $v \neq t$ e v ocorre em t: metade com falha.

O algoritmo termina quando nenhuma ação mais puder ser realizada ou quando resultar em uma falha.

No módulo Unificação, três funções são especificadas. A função *mgu* determina o unificador mais geral, seqüência de substituições, para uma lista de equações que é dada em seus argumentos de entrada. A função *mgu-nv* determina o unificador mais geral de dois termos não variáveis. A função local *mgu*, com dois argumentos, é usada enquanto produz o resultado da exportação da função *mgu*. As equações para a função *mgu* fielmente obedecem ao algoritmo descrito acima (ver Apêndices A.2.18 e A.2.19).

5.5 Módulo de Avaliação

Neste módulo definimos a máquina Prolog [Bra86], a qual aceita como entrada um programa $\mathcal{P}$ e uma consulta Prolog $\mathcal{C}$, e produz como saída **falha** ou **sucesso**. No segundo caso, a saída inclui também uma substituição θ para as variáveis de $\mathcal{C}$. A máquina comporta-se de tal forma que, se a saída é **falha**, não há respostas corretas de $\mathcal{C}$ a $\mathcal{P}$ e, se a saída é **sucesso**, θ é uma resposta correta de $\mathcal{C}$ a $\mathcal{P}$.

Para o desenvolvimento da máquina abstrata Prolog foi criado o módulo *EvalProlog*[2] o qual utiliza o módulo *VariantClause* que, por sua vez, utiliza o módulo *TermSets*. Esses módulos serão descritos a seguir.

5.5.1 *EvalProlog*

Neste módulo a máquina abstrata Prolog foi implementada. Essa máquina, quanto ao resultado **sucesso**, fornece uma lista de substituições (LISTSUBS), ou seja, respostas alternativas são apresentadas forçando o processo de retrocesso (*backtracking*).

equations

[ev1] eval(C* O) = LS' **when**
listvar(O,{}) = TS,
variant-P(C* O, {}) = C*' O',
eval-K(C*', {}, C*' O')= LS,
verify-LS(LS,TS) = LS'

[evk4a] eval-K(C*,S,C' C*' O) = LS **when**
head(C') = T',
head(O) = T,
mgu-nv(T', T) = S', S'= fail,
eval-K(C*, S, C*' O) = LS

[evk4b] eval-K(C*,S, C' C*' ?- T, T+.) = LS"' **when**
head(C') = T',
mgu-nv(T',T) = S', S' != fail,
S simples S' = S",
tailC(C') = T+',
append(T+';T+) = T+",
T+"[tl S"] = T+"',
eval-K(C*,{},C* ?-T+"'.) = LS,
insertSinLS(S', LS) = LS',
eval-K(C*,S,C*' ?- T, T+.) = LS",
conc(LS', LS") = LS"'

A função *eval*, regra [ev1], possui como entrada um programa Prolog, conjunto de cláusulas e uma lista de objetivos, e fornece como saída uma lista de substituições, *fail*, *yes*

[2] O módulo completo está nos Apêndices A.2.5 e A.2.6.

ou *sucess*. A função denominada *eval-K*, regras [evk4a] e [evk4b], possui como argumentos o conjunto de cláusulas, uma substituição (inicializada com vazio) e o próprio programa. Nesse caso, guardamos o programa para que o processo de *backtracking* seja efetuado. A regra [evk4a] é utilizada quando não existe unificação entre o programa e o objetivo, e a regra [evk4b] é utilizada quando existe a unificação.

Outras funções auxiliares são utilizadas, como:

- *verify-LS*: é definida para verificar o valor final da lista de substituições, ou seja, se o valor for: [ ] devolve *fail*, [{ }] devolve *yes*, se não for nenhum dos casos anteriores devolve o conjunto de substituições que será gerado por uma função auxiliar *find*.

```
[vls1] verify-LS(LS,TS) = fail when LS= [ ]
[vls2] verify-LS(LS,TS) = yes when LS=[{ }]
[vls4] verify-LS(LS,TS) = yes when LS!=[ ], LS != [{ }], TS={ }
[vls5] verify-LS(LS,TS) = LS' when
                          LS!=[ ], LS != [{ }], TS!={ },
                          find(TS,LS) = LS'
```

- *find*: verifica se as variáveis do objetivo pertencem a uma lista de substituições e produz uma nova lista de substituições que será a resposta final mostrada no vídeo. A função *find* usa a função auxiliar *findS*.

```
[f1] find(TS,[]) = []
[f2] find(TS,[S,LS*]) = LS" when
                        findS(TS,S,{}) = S',
                        find(TS,[LS*]) = LS',
                        conc([S'],LS') = LS"
[fs1] findS(TS, {}, S)                                  = S
[fs2] findS({T*, T', T*'} , {(V |- > T),S*}, S)         = S' when
        T' != V,
        findS({T*, T', T*'}, {S*},S)= S'
[fs3] findS({T*, T', T*'}, {(V |- > T),S*}, S)          = S"' when
        T' = V,
        S simples {(V |- > T)} = S",
        findS({T*, T', T*'},{S*},S") = S"'
```

- *headO*: encontra a cabeça do objetivo;
- *tailC*: encontra o corpo de uma cláusula;
- *append*: utilizada para unir o corpo da cláusula com o corpo do objetivo e produz como resultado um novo objetivo a ser avaliado;
- *conc*: concatena as listas de substituições geradas pela máquina prolog e devolve o resultado para *eval-K* que, por sua vez, devolve para *eval*;
- *listvar*: permite retirar todas as variáveis do objetivo. Usa a função *var* a qual passa como argumento o termo para a função *verify-var* que verifica se o termo é uma variável, um inteiro ou um átomo, e devolve como saída uma lista contendo somente as variáveis do objetivo.

```
[lo1a]  listvar(?-T.,TS)        =  TS" when
                                   var(T) = TS',
                                   TS uniao TS'= TS"
[lo1b]  listvar(?-T, T+.,TS)    =  TS"' when
                                   var(T) = TS',
                                   listvar(?- T+.,TS') = TS",
                                   TS uniao TS" = TS"'
```

5.5.2 *VariantClause*

Este módulo, desenvolvido por Brunekreef, teve algumas alterações por nos realizadas pois, seu objetivo era criar uma variante somente para as cláusulas, sem envolver os objetivos. Mudamos seu código, a fim de que a especificação pudesse renomear as variáveis das cláusulas de tal forma que o resultado obtido e a lista de objetivos não possuissem variáveis em comum. Nessa renomeação a variável é substituída por uma nova variável, acrescentando-se $\_n$.

A função que gera as cláusulas variantes é *variant-P*, a qual toma como argumentos de entrada o programa e um conjunto de termos. Com a função *varset*, um conjunto é criado com as variáveis de uma cláusula. Com a função *varsub*, uma seqüência de substituições é deduzida de dois conjunto de variáveis (ver Apêndices A.2.20 e A.2.21).

5.5.3 *TermSets*

Neste módulo, o sort TERM-SET e algumas operações básicas como: pertence, união, interseção e diferença de conjuntos são especificadas. Também uma função substituição sob conjunto de termos é definida. Tais funções roteiam uma substituição '$[ts\ S]$' sobre um conjunto de termos para uma substituição '$[t\ S]$' em cada um dos elementos do conjunto (ver Apêndices A.2.16 e A.2.17).

5.6 Funcionamento do Ambiente Interativo Prolog

A seguir, apresentamos os passos para o funcionamento do ambiente Prolog:

1. Ir ao diretório prolog (lugar onde estão todos os módulos descritos nas Seções 5.2, 5.3, 5.4 e 5.5);
2. Digitar o comando **meta**, que dá acesso ao meta-ambiente ASF+SDF. A janela principal do meta-ambiente aparecerá como mostrou a Figura 5.1 da Seção 5.1.
3. Adicionar o módulo *EvalProlog*, selecionando o menu *File* e escolher o botão *Open*. Na janela de diálogo, o sistema perguntará o nome do módulo que se deseja abrir. Ele apresenta uma lista de todos os arquivos com extensão 'sdf2'. Pressionar sob o 'EvalProlog.sdf2' e escolher a opção *Open*. Este carregará o módulo *EvalProlog* (ambas suas sintaxe e equações) dentro do sistema.
4. Verificar que o módulo *EvalProlog* aparece com um retângulo, bem como os módulos por ele importados, como é mostrado nas Figuras 5.1 e 5.2 da Seção 5.1.
5. Selecionar o módulo *EvalProlog* e pressionar o botão *EditTerm*. Entrar com qualquer nome de arquivo, por exemplo, 'progenitor.prolog'.
6. Digitar o seguinte termo no editor:

```
eval(

    progenitor(carla,bruna).
    progenitor(marcelo,bruna).
    progenitor(luiza,cristina).
    progenitor(jose,cristina).
    progenitor(carlos,maria).
```

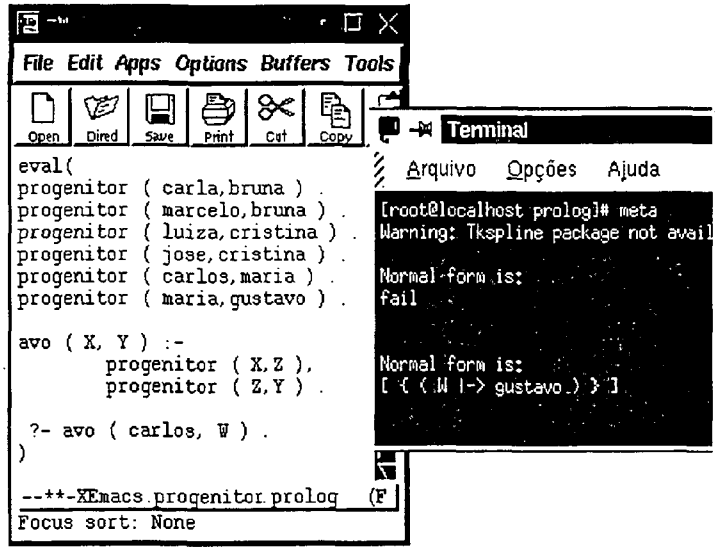

Figura 5.3: Programa de Relação Familiar em Prolog

```
progenitor(maria,gustavo).
avo(X,Y) :-
                progenitor(X,Z),
                progenitor(Z,Y).
?- avo(jose, W).)
```

Observar como o texto que se digitou, o qual é chamado de *focus*, fica com uma cor diferente.

7. Pressionar no *focus*, ir no menu *Meta-Environment* e escolher a opção *Parse*. O texto ficará com uma cor diferente novamente.

8. Por último, pressionar no menu *Meta-Environment* e escolher a opção *Reduce* para reduzir um termo no editor de termos. O resultado aparecerá na janela do terminal onde o meta-ambiente foi iniciado. Para o exemplo do 'progenitor.prolog' o resultado será *fail*, como mostra a Figura 5.3. na primeira *Normal form is.* Se mudarmos o objetivo para:

   ```
   ?- avo(carlos, W).
   ```

o resultado será [{ ($W \mapsto$ *gustavo*) }], como mostra a segunda *Normal form is* da Figura 5.3.

5.7 Considerações Finais

Neste capítulo descrevemos os módulos e o funcionamento do ambiente operacional interativo Prolog. Alguns módulos apresentados aqui foram adaptados da implementação de Brunekreef [Bru96]. As adaptações foram:

1. nomes de variáveis, constantes e funções passaram a ser delimitadas pelo caractere aspas(""), pois em [Bru96] os identificadores não eram delimitados por aspas. Todos os módulos descritos neste capítulo sofreram estas alterações;
2. representação da relação de prioridades, do maior para o menor (por exemplo no módulo *Booleans*). Em [Bru96] esta relação não existia;
3. soluções para problemas de ambigüidades (*PrologTerms*, *NormalizationFunction*, *TermSets*);
4. eliminação de algumas funções sintáticas do módulo *PrologFunction*;
5. as regras semânticas representadas por esquemas do estilo:

$$\begin{array}{c} \langle \text{ premissa}_1 \rangle \\ . \\ . \\ \langle \text{ premissa}_n \rangle \\ \hline\hline \\ \langle \text{ conclusão} \rangle \end{array}$$

passaram a ser representadas como:

$$\begin{array}{l} \langle \text{ conclusão} \rangle \textbf{ when} \\ \qquad \langle \text{ premissa}_1 \rangle \\ \qquad . \\ \qquad . \\ \qquad \langle \text{ premissa}_n \rangle \end{array}$$

A mudança de representação das regras foi realizada em todos os módulos, devido à mudança de versão do meta-ambiente.

6. adição do *sort* OBJECTIVE no módulo *PrologProgram* e acréscimo da função *variant-P* no módulo *VariantClause*. Na versão de [Bru96], os programas não possuiam objetivo, pois não eram avaliados;
7. inclusão da seção **context-free restrictions** no módulo PrologTerms para evitar problemas de ambigüidade;
8. as regras [m], [m1], [g] e [g1] do módulo *Substitution* foram alteradas pois, não realizavam corretamente a substituição de uma variável no corpo das cláusulas se a variável fosse a mesma. Isto é um *bug* da especificação de Brunekreef;
9. o módulo *EvalProlog* foi adicionado, pois é o módulo de avaliação de programas Prolog, sendo que implementa a parte principal da semântica de avaliação da linguagem.

As adaptações e inclusões realizadas neste capítulo serviram como base para o desenvolvimento do ambiente ParaLog, descrito no capítulo seguinte.

CAPÍTULO 6

Ambiente Operacional Interativo ParaLog

Descrevemos neste capítulo o desenvolvimento de um ambiente interativo para linguagem ParaLog utilizando semântica de transição. O ambiente proposto suporta verificações sintática e semântica de um programa escrito em ParaLog. Tais verificações serão possíveis respeitando-se a sintaxe e a semântica da linguagem ParaLog, introduzidas no Capítulo 4.

6.1 Implementando o ParaLog

Combinando os conceitos da semântica de transição, o meta-ambiente de desenvolvimento ASF+SDF, o protótipo para programas Prolog[1], juntamente com os recursos oferecidos pela programação lógica paraconsistente, foi possível implementar um ambiente interativo para o ParaLog.

A implementação desse ambiente interativo também foi divida em vários módulos, descritos nas próximas seções.

6.1.1 Módulos Básicos

Estes módulos são os mesmos usados na especificação formal para programas Prolog descritos no Capítulo 5, Seção 5.2. O módulo *Layout* não teve nenhuma alteração; já o módulo *Booleans*, listado abaixo, sofreu alterações no nome de duas funções. A função:

- & passou a chamar-se $\wedge$;
- *not* passou a chamar-se *neg*.

[1] Descrito no Capítulo 5.

Essa troca de nomes foi necessária para evitar problemas de ambigüidade, pois o & e *not* são símbolos padrões da linguagem ParaLog.

```
module Booleans
imports Layout
exports
sorts BOOL
context-free syntax
        "true"                  →  BOOL
        "false"                 →  BOOL

        BOOL "|" BOOL           →  BOOL {left}
        BOOL "∧" BOOL           →  BOOL {left}
        "neg" "(" BOOL ")"      →  BOOL

        "(" BOOL ")"            →  BOOL {bracket}
variables
        "B"[\ ']*  →  BOOL
context-free priorities
        BOOL "&" BOOL  →  BOOL  >
        BOOL "|" BOOL  →  BOOL
equations
        [o1]  true | B   =  true
        [o2]  false | B  =  B

        [a1]  true ∧ B   =  B
        [a2]  false ∧ B  =  false

        [n1]  neg(false) =  true
        [n2]  neg(true)  =  false
```

6.1.2 Módulo Sintático

O módulo sintático, *ParaLogSyntax*, opera sob a sintaxe definida no Capítulo 4, Seção 4.5.1. Nesse módulo definimos inteiros, listas, termos, variáveis, átomo, programa, cláusula, objetivo e constante anotacional da linguagem ParaLog.

Na listagem abaixo temos um trecho do módulo *ParaLogSyntax*; nela, um átomo é uma cadeia de letras e dígitos, iniciando com uma letra minúscula (linha 7) ou uma cadeia de

símbolos do alfabeto básico Prolog, podendo incluir o branco, começando por apóstrofos (linha 8). Implementamos o conceito de fórmula (linhas 10 e 11) conforme a Definição 4.5.5, descrita no Capítulo 4. Uma cláusula é definida como uma fórmula seguida de um "." (linha 13) ou pode ter uma cabeça (uma fórmula) e um corpo (conjunto de uma ou mais fórmulas) (linha 14). A representação de um conjunto de uma ou mais fórmulas é feita pela definição de FORMBODY (linha 12). Um objetivo é um conjunto de uma ou mais fórmulas (linha 15) ou é um objetivo vazio (linha 16). Por último, definimos um programa (linha 17) em ParaLog como um conjunto de cláusulas (CLAUSE*) seguido de um objetivo (OBJECTIVE).

```
[1]  module ParaLogSyntax
[2]  imports Layout
[3]  exports
[4]  sorts TERM PROGRAM INTEGER VARIABLE LIST1 ATOM CLAUSE
[5]      OPSYM LIST CONST-ANOT TERMLIST FORMBODY OBJECTIVE
[6]  lexical syntax
[7]      [a-z][a-zA-Z0-9\_]*          →  ATOM
[8]      "'" ~[\\ '\ n]+ " ' "        →  ATOM
[9]  context-free syntax syntax
[10]     ATOM"("TERMLIST")" ":" CONST-ANOT        →  FORMULA
[11]     "not"ATOM"("TERMLIST")" ":"CONST-ANOT    →  FORMULA
[12]     {FORMULA "&"}+                           →  FORMBODY
[13]     FORMULA "."                              →  CLAUSE
[14]     FORMULA "<--" FORMBODY "."               →  CLAUSE
[15]     "<--" FORMBODY "."                       →  OBJECTIVE
[16]     "<--"                                    →  OBJECTIVE
[17]     CLAUSE* OBJECTIVE                        →  PROGRAM
```

6.1.3 Módulos de Regularização e Transformação

Conforme a semântica do ParaLog descrita no Capítulo 4, Seção 4.5.2, um programa ParaLog passa por três processos: *regularização*, *eliminação da negação* e *fechamento*. Para cada um desses processos existe um módulo em nossa implementação.

O módulo *VariantClause* recebe como entrada um programa $\mathcal{P}_0$ e uma consulta $\mathcal{Q}_0$, e fornece como saída um programa $\mathcal{P}_1$ e uma consulta $\mathcal{Q}_1$, que são equivalentes ao programa e à consulta inicial. Esse módulo é o mesmo módulo desenvolvido para o ambiente interativo Prolog com o acréscimo de duas funções: *varset-F* e *tvarset-F*. Abaixo apresentamos a listagem do módulo *VariantClause*, que sofreu alteração. Nessa listagem, a primeira função criada foi *varset-F*, que possui como argumento de entrada um conjunto de uma ou mais fórmulas (FORMBODY) e gera como saída um conjunto de váriaveis pertencentes a essas fórmulas. Essa função possui duas regras semânticas:

- [sv5] : executada quando uma cláusula tiver somente uma cabeça;
- [sv6] : executada quando uma cláusula tiver uma cabeça e um corpo.

A segunda função criada, *tvarset-F*, tem como objetivo avaliar somente uma fórmula e retirar dela as variáveis que serão repassadas para a função *varset-F*. Esta função possui duas regras semânticas: a primeira [st1] será executada quando a fórmula for do tipo *A(T+):CA* e a segunda [st2] será executada quando a fórmula for do tipo *not A(T+):CA*.

```
module VariantClause
imports Unification TermSets
hiddens
context-free syntax
        "varset-F" "(" FORMBODY ")"    → TERM-SET
        "tvarset-F" "(" FORMULA ")"    → TERM-SET
equations
        [sv5]  varset-F (F)            = tvarset-F(F)
        [sv6]  varset-F (F & F+)       = tvarset-F(F) uniao varset-F(F+)
        [st1]  tvarset-F(A(T+):CA)     = tvarset(T+)
        [st2]  tvarset-F(not A(T+):CA) = tvarset(T+)
```

No módulo *EliminationOfNegation*, o programa $\mathcal{P}_1$ e a consulta $\mathcal{Q}_1$, resultantes do processo de regularização, passam pelo processo de eliminação da negação. Nesse processo, todas as ocorrências das fórmulas atômicas anotadas da forma *not* $p{:}\mu$ são substituídas por $p :\sim \mu$, fornecendo como resultado um novo programa $\mathcal{P}_2$ e uma nova consulta $\mathcal{Q}_2$, sem ocorrências de cláusulas do tipo *not* $p{:}\mu$.

module EliminationOfNegation

imports ParaLogSyntax

exports

context-free syntax

"~" "(" CONST-ANOT ")" → CONST-ANOT

"elimination" "(" PROGRAM ")" → PROGRAM

"elimination-C" "(" CLAUSE ")" → CLAUSE

equations

[op1] ~(t) = f

[op2] ~(f) = t

[op3] ~(lf) = lt

[op4] ~(lt) = lf

[op5] ~(*) = *

[op6] ~(tf) = tf

[e1] elimination(O) = <-- F+. **when**

elimination-O(O) = F+

[e2] elimination(C C* O) = C' C*' O' **when**

elimination-C(C) = C',

elimination(C* O) = C*' O'

[ec1] elimination-C (F.) = F'. **when**

verify-form(F) = F'

[ec2] elimination-C (F <-- F+.) = F' <-- F+'. **when**

verify-form(F) = F',

verify-body(F+) = F+'

Acima temos uma listagem do trecho do módulo *EliminationOfNegation*, na qual a função *elimination* é responsável por efetuar o processo de eliminação descrito no parágrafo anterior. Essa função possui duas regras de semântica de transição: a primeira regra [e1] será executada quando o programa estiver vazio e existir uma lista de objetivos a serem avaliados; a segunda regra [e2] está em ASF+SDF, que é a notação para a regra:

$$\frac{\text{verify-form(F) = F', verify-body(F+) = F+'}}{\text{elimination-C(F <-- F+.) = F' <-- F+'.}}$$

A segunda regra será executada quando o programa for formado por uma ou mais cláusulas e uma lista de objetivos; nesse caso, avalia-se a primeira cláusula originando

uma nova cláusula e depois o processo continua recursivamente, até que se tenha somente uma lista de objetivos a serem avaliados. A função "$\sim$" permite efetuar a troca do valor de μ para $\sim \mu$ por meio das regras semânticas: [op1], [op2], [op3], [op4], [op5], [op6]. Essas regras semânticas serão executadas quando as cláusulas ou a lista de objetivos tiverem o operador *not*.

O último módulo, *Closed*, é responsável por efetuar o processo de fechamento como descrito em [BS87]. Nesse módulo o programa $\mathcal{P}_2$, resultante do processo de *elimination of negation*, é considerado *closed* quando dado $\mathcal{C}_1$, $\mathcal{C}_2$ (cláusulas pertencentes ao programa $\mathcal{P}_2$) são cláusulas anotadas distintas como:

$$\rho_1{:}\mu_1 \texttt{<--} q_1{:}\nu_1 \& ... \& q_n{:}\nu_n \quad (\mathcal{C}_1)$$
$$\rho_2{:}\mu_2 \texttt{<--} r_1{:}\phi_1 \& ... \& r_m{:}\phi_m \quad (\mathcal{C}_2)$$

Portanto, se ρ_1 é unificável via mgu θ e o $\sqcup \{\mu_1, \mu_2\} \notin \{\mu_1, \mu_2\}$; então a cláusula:

$$(\rho_1{:}\ \sqcup \{\mu_1, \mu_2\} \texttt{<--}\ q_1{:}\nu_1 \& ... \& q_n{:}\nu_n \& r_1{:}\phi_1 \& ... \& r_m{:}\phi_m)\theta \quad (\mathcal{C}_3)$$

está também em $\mathcal{P}_2$. Se um programa $\mathcal{P}_2$ não estiver fechado, então um programa $\mathcal{P}_3$ é construído da seguinte maneira: para todos os pares de cláusulas-hg[2] na forma $\mathcal{C}_1$, $\mathcal{C}_2$ em $\mathcal{P}_2$, uma cláusula-gh $\mathcal{C}_3$ é construída. Se nenhuma cláusula-hg em $\mathcal{P}_2$ é uma mutante de $\mathcal{C}_3$, então $\mathcal{C}_3$ é incluído em $\mathcal{P}_2$. O programa $\mathcal{P}_3$ é obtido pela inserção de todas as cláusulas-hg necessárias em $\mathcal{P}_2$. Se $\mathcal{P}_3$ não estiver *closed*, então o programa $\mathcal{P}_4$ é construído pela repetição do processo acima para todas as cláusulas em $\mathcal{P}_3$. É fácil verificar que esse processo eventualmente termina, resultando um programa fechado $\mathcal{P}_n$, e é denotado por $\mathrm{CL}(\mathcal{P}_2)$, que é equivalente ao programa inicial $\mathcal{P}_0$.

```
module Closed
imports VariantClause
exports
sorts CLAUSE-UNION
context-free syntax
        "closed" "(" PROGRAM ")"   →   PROGRAM
```

[2] Cláusulas de Horn Generalizadas na forma $\rho_0{:}\mu_0 \texttt{<---} \rho_1{:}\mu_1 \& ... \& \rho_n{:}\mu_n$ onde $\rho_0{:}\mu_0$ é chamada *head* da cláusula, enquanto $\rho_1{:}\mu_1 \& ... \& \rho_n{:}\mu_n$ é chamado *body* da cláusula.

hiddens

context-free syntax

"lub" "(" CONST-ANOT "," CONST-ANOT ")" → CONST-ANOT

"belong-lub" "(" CONST-ANOT ","

"{" CONST-ANOT "," CONST-ANOT "}" ")" → BOOL

"mutant" "(" CLAUSE "," CLAUSE-UNION ")" → BOOL

equations

[c1a] closed(C O) = C O

[c1b] closed(C C' C* O) = C' C* O **when** C = fail

[c1c] closed(C C' C* O) = C*"' O **when**
C != fail,
closed-P(C C' C*) = C" C*',
verify-CP(C", C*', C C' C*) = C"' C*",
closed(C"' C*" O) = C*"' O

[su1a] lub(*, lt) = lt

[su1b] lub(*, lf) = lf

[su1c] lub(t, t) = t

[su1d] lub(f, f) = f

[su1e] lub(t, tf) = tf

[su1f] lub(f, tf) = tf

[bl1] belong-lub(CA, {CA', CA"}) = true **when** CA = CA'

[bl2] belong-lub(CA, {CA', CA"}) = true **when** CA = CA"

[bl3] belong-lub(CA, {CA', CA"}) = false **when** CA != CA', CA != CA"

[mut1a] mutant(C, C') = B **when**
mutant-C(C, C') = B

[mut1b] mutant(C, C' C" C*) = B | B' **when**
mutant-C(C, C') = B,
mutant(C, C" C*) = B'

Acima apresentamos um techo do módulo *Closed*. Nesse módulo a função principal é *closed*, representada pelas regras semânticas [c1a], [c1b] e [c1c]. Essa função recebe como entrada um conjunto de cláusulas (C*) e retorna como saída um novo conjunto de cláusulas (C*') fechadas. As funções internas *lub*, *belong-lub* e *mutant* são algumas das funções utilizadas por esse módulo. A função *lub* retorna o supremo entre duas anotações através das regras semânticas [su1a], [su1b], [su1c], [su1d], [su1e] e [su1f]. As regras

```
[1]    p(X):t <--
[2]       not q(X,Y):f.
[3]    p(X): f <--
[4]       r(Y):f.
[5]
[6]    q(a, b):t.
[7]    q(a, c):t.
[8]    r(b):t.
[9]    r(c):f.
[10]
[11]   <-- p(a):t.
```

Figura 6.1: Programa ParaLog valor de p

```
[1]    p(X):t <--
[2]       q(X,Y):t.
[3]    p(X): f <--
[4]       r(Y):f.
[5]
[6]    q(a, b):t.
[7]    q(a, c):t.
[8]    r(b):t.
[9]    r(c):f.
[10]
[11]   <-- p(a):t.
```

Figura 6.2: Programa ParaLog valor de p após processo de eliminação

semânticas [bl1], [bl2] e [bl3] compõem a função *belong-lub*, que permite verificar se uma anotação já pertence ao conjunto de anotações, sendo seu retorno do tipo booleano (B). A função *mutant*, regras semânticas [mut1a] e [mut1b], verifica se duas cláusulas são mutantes uma da outra. Nessa função, o retorno é **TRUE** para cláusulas mutantes e **FALSE** para cláusulas não mutantes.

Para exemplificar os módulos de regularização e transformação criamos um exemplo (Figura 6.1) de um programa escrito em linguagem ParaLog e depois apresentamos o mesmo programa após os processos de eliminação (Figura 6.2), fechamento (Figura 6.3) e renomeação de variáveis (Figura 6.4) [3].

Observa-se que na Figura 6.1, na linha 2, temos o operador *not* que será eliminado

[3] No início desta seção, a ordem destes processos era renomeação de variáveis, eliminação da negação e fechamento conforme descrito em [CPA+95]. Para exemplificar tais processos, a ordem de execução dos módulos foi mudada para eliminação, fechamento e renomeação de variáveis, com o objetivo de facilitar a implementação. Esta mudança não altera a semântica dos programas.

```
[1]    p(X):tf <--
[2]        q(X , Y):t &
[3]        r(Y ):f.
[4]    p(X):t <--
[5]        q(X,Y):t.
[6]    p(X): f <--
[7]        r(Y):f.
[8]
[9]    q(a, b):t.
[10]   q(a, c):t.
[11]   r(b):t.
[12]   r(c):f.
[13]
[14]   <--p(a):t.
```

Figura 6.3: Programa ParaLog valor de p após processo de fechamento

```
[1]    p(X_n):tf <--
[2]        q(X_n, Y_n):t &
[3]        r(Y_n):f.
[4]    p(X_n_n):t <--
[5]        q(X_n_n,Y_n_n):t.
[6]    p(X_n_n_n): f <--
[7]        r(Y_n_n_n):f.
[8]
[9]    q(a, b):t.
[10]   q(a, c):t.
[11]   r(b):t.
[12]   r(c):f.
[13]
[14]   <--p(a):t.
```

Figura 6.4: Programa ParaLog valor de p após processo de regularização

após a execução do módulo *EliminationOfNegation*. O operador *not* será eliminado substituindo a constante anotacional f por t (linha 2, Figura 6.2).

Nas linhas 1 e 3 da Figura 6.2 temos as cláusulas *p(X)* as quais são unificáveis e posuem um grau de crença contraditório (t e f respectivamente). Por essa razão, após a execução do módulo *Closed* insere-se uma nova cláusula como mostra a Figura 6.3 nas linhas 1-3. A nova cláusula inserida terá um grau de crença *tf*, pois o supremo entre t e f é a inconsistência. Na Figura 6.3, linhas 1-7, as variáveis X e Y foram trocadas por *X_n*, *X_n_n*, *X_n_n_n* e *Y_n*, *Y_n_n*, *Y_n_n_n* (linhas 1-7, Figura 6.4) após a execução do módulo *VariantClause*. É importante ressaltar que o módulo *Unification* realiza o controle dos átomos que ocorrem em mais de um papel, por esse motivo não foi desenvolvido um

módulo específico que executasse esta tarefa.

6.1.4 Módulo de Unificação

O processo de unificação é responsável pela obtenção do umg de uma lista de equações de termos. A obtenção de um umg para um par de fórmulas é similar ao empregado na linguagem Prolog. Por esse motivo, o módulo desenvolvido para o "Ambiente Prolog" pôde ser aplicado também para o "Ambiente ParaLog", com a inserção da função *mgu-nv*.

Abaixo segue uma parte do módulo *Unification*; na listagem apresentada, as regras semânticas para *mgu-nv* são [nf1], [nf2] e [nf]. A regra [nf1] tenta unificar duas fórmulas com o argumento sendo um termo (TERM). A regra [nf2] tenta unificar duas fórmulas quando seu argumento é um conjunto de termos (TERMLIST). As regras [nf1] e [nf2] fornecem como saída uma substituição (SUBS). A última regra [nf] é utilizada quando a unificação entre duas fórmulas não é possível, fornecendo dessa forma a resposta **fail**.

module Unification
imports Equations
exports
context-free syntax
"mgu" "(" EQS ")" → SUBS
"mgu-nv" "(" TERM "," TERM ")" → SUBS
"mgu-f" "(" FORMULA "," FORMULA ")" → SUBS
hiddens
context-free syntax
"mgu" "(" EQS "," SUBS ")" → SUBS
equations
[nf1] mgu-f(A(T):CA, A(T'):CA') = mgu({T equivalente T'})
[nf2] mgu-f(A(T, T+):CA, A(T', T+'):CA') =
S inteligente mgu-f(A(T+[tl S]):CA, A(T+' [tl S]):CA') **when**
mgu(T equivalente T') = S
[nf] mgu-f(F, F') = fail

6.1.5 Módulo de Avaliação

Este módulo define o motor de inferência do ParaLog [CPA+95]. A máquina ParaLog requer um programa $\mathcal{P}_0$ e uma consulta $\mathcal{Q}_0$ como entrada e produz uma crenção $\psi \in \tau$

como resposta. Para os casos onde $\psi \neq *$ a resposta também inclui uma substituição θ para as variáveis do objetivo.

```
equations
[1]  [ev1]  eval(C* O)   =   LS' when
[2]                          listvar(O,{}) = TS,
[3]                          elimination(C* O) = C*' O',
[4]                          closed(C*' O') = C*" O",
[5]                          variant-P(C*" O", {}) = C*"' O"',
[6]                          eval-K(C*"', {}, C*"' O"' , false)= LS,
[7]                          verify-LS(LS,TS) = LS'
```

Acima segue uma parte do módulo *EvalParaLog*; nesta listagem apresentada, a regra semântica da função *eval* (linhas 1-7) implementa o motor de inferência do ParaLog, tendo como entrada um programa (C* O) e retornando como saída uma lista de substituições[4] (LS'). Note que o programa C* e a consulta O são respectivamente $\mathcal{P}_n$ e $\mathcal{Q}_n$ descritos na Seção 6.1.3. Esta função é responsável por chamar as funções: *listvar* (linha 2), *elimination* (linha 3), *closed* (linha 4), *variant-P* (linha 5), *eval-K* (linha 6) e *verify-LS* (linha 7).

A função *listvar* remove todas as variáveis do objetivo para que seja gerado a lista de substituição final. As funções *elimination*, *closed* e *variant-P* foram explicadas na Seção 6.1.3. A função *eval-K* possui um conjunto de cláusulas, uma substituição (inicialmente vazia), o mesmo programa e uma variável do tipo booleana (inicialmente possui o valor *false*) como argumentos. Nesse caso, guarda-se o programa no primeiro argumento de *eval-K* para que o processo de *backtracking* seja realizado.

Com a implementação de todos os módulos descritos nas Seções 6.1.1, 6.1.2, 6.1.3, 6.1.4 e 6.1.5 foi possível gerar um ambiente de programação para a linguagem ParaLog no meta-ambiente ASF+SDF.

O ambiente é capaz de retornar uma resposta dado um programa ParaLog e uma consulta. Na Figura 6.1 mostrou-se um exemplo de programa ParaLog, se executado neste ambiente, será capaz de retornar uma crença igual a inconsistente (tf) como resposta para a pergunta <-- p(a):t como mostra a Figura 6.5.

[4]Essa lista contem também o grau de crença.

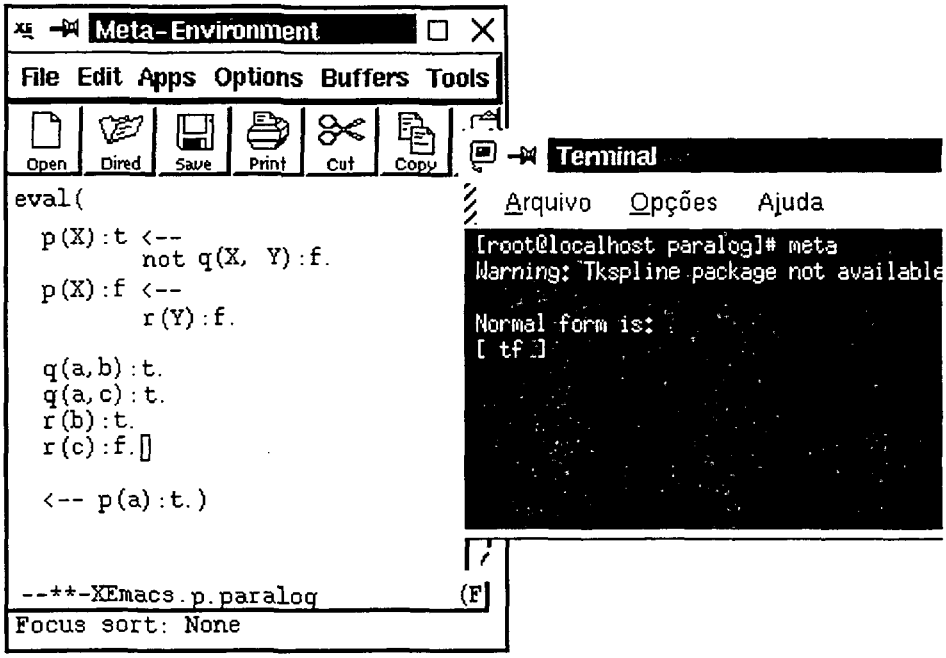

Figura 6.5: Programa ParaLog valor de p após processo de avaliação

6.2 Funcionamento do Ambiente Interativo ParaLog

O ambiente de desenvolvimento ParaLog possui funcionamento similar ao do Ambiente Prolog descrito na Seção 5.6. A diferença existente entre os ambientes é que no ParaLog as cláusulas e objetivos possuem uma anotação e a resposta quando a crença é diferente de * fornece uma lista de substituições contendo seu grau de crença.

O módulo responsável pela avaliação de programas em ParaLog é o *EvalParaLog*. Supondo um exemplo composto de fatos e de regras escrito no editor de termos do Ambiente Operacional Interativo ParaLog:

```
eval(

    doenca1(X):t <--
                    sintoma1(X):t &
                    sintoma2(X):t.
    doenca2(X):t <--
                    sintoma1(X):t &
                    sintoma3(X):t.
    doenca1(X):f <--
                    sintoma4(X):t.
```

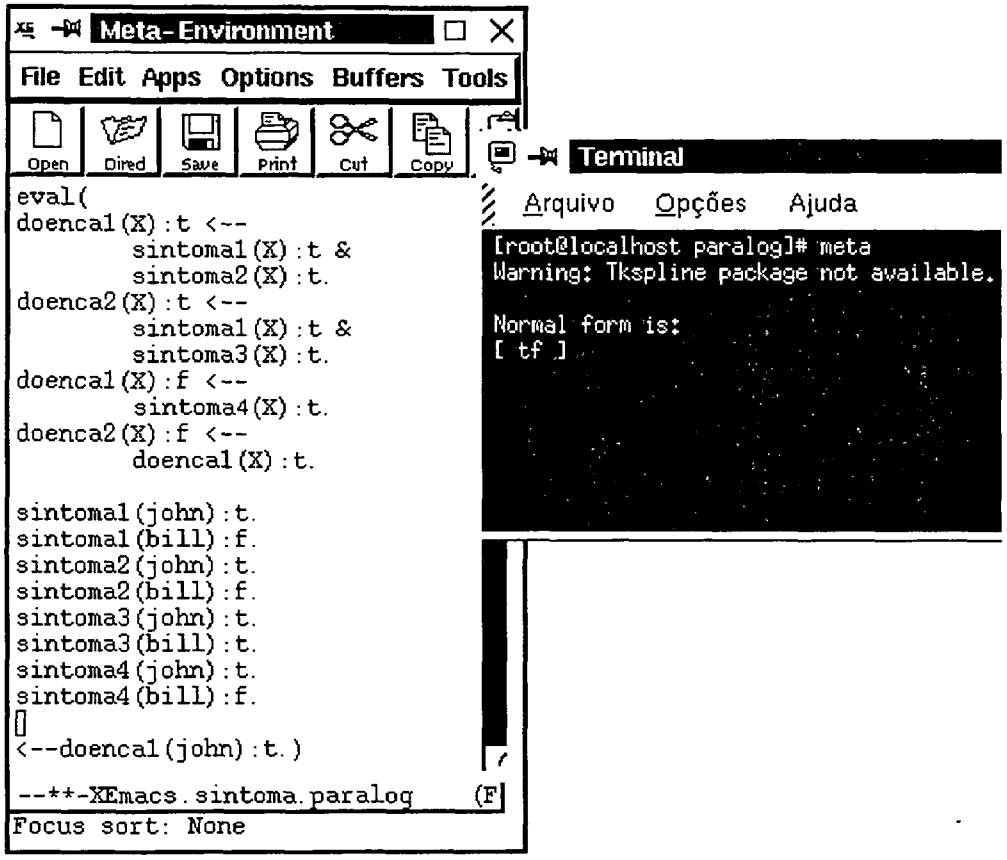

Figura 6.6: Programa *doença* após o processo de avaliação

```
doenca2(X):f <--
        doenca1(X):t.

sintoma1(john):t.
sintoma1(bill):f.
sintoma2(john):t.
sintoma2(bill):f.
sintoma3(john):t.
sintoma3(bill):t.
sintoma4(john):t.
sintoma4(bill):f.

<-- doenca1(john):t.)
```

Neste exemplo acima, após escolher a opção *Parse* e *Reduce*, do meta-ambiente ASF+SDF, o interpretador ParaLog proposto executará os processos de eliminação da negação, fechamento, renomeação de variáveis e avaliação. Ele retornará como resposta à pergunta <-- doenca1(john):t. uma crença inconsistente (tf) como mostra a Figura 6.6.

6.3 Considerações Finais

Neste capítulo descrevemos os módulos[5] que compõem a implementação de um interpretador para a Linguagem ParaLog. Essa implementação contém a especificação formal do ParaLog, bem como um ambiente de desenvolvimento para ela.

Tal ambiente é capaz de funcionar como um interpretador, pois combina os conceitos de programação paraconsistente com os conceitos de semântica de transição, implementados em um ambiente interativo de desenvolvimento.

[5] A listagem completa de todos os módulos está no Apêndice A.3.

CAPÍTULO 7

Conclusões

A semântica operacional é uma das técnicas para descrição formal que permite, por exemplo, a formalização de uma linguagem de programação. Essa semântica possue duas instanciações que são: semântica de passo grande (*big step*) [Ast91, Gun91] e semântica de passo pequeno (*small step*) [Ast91, Kah87].

Na semântica *big step*, ou semântica relacional, dada uma expressão e um estado, produz um valor diretamente para essa expressão [Win93].

A semântica *small step*, ou semântica de transição, permite quebrar a avaliação de uma expressão dentro de uma sequência de avaliações parciais, onde cada passo é transformado em uma expressão pela reescrita de expressões elementares.

Neste trabalho utilizamos os conceitos de semântica de transição para formalizarmos a linguagem ParaLog [CPA+95], a qual está baseada em uma lógica não clássica — paraconsistente anotada. Sua semântica está baseada na semântica do modelo mínimo de Herbrand.

Para formalizarmos essa linguagem utilizamos o meta-ambiente ASF+SDF[BK00], o qual permite a definição sintática bem como a semântica de uma linguagem. O ASF+SDF é um ambiente interativo que oferece características como sintaxe, verificação de tipos e um compilador.

Portanto, o objetivo principal deste trabalho foi mostrar que, por meio da Lógica Paraconsistente, da semântica de transição e de um meta-ambiente de programação, podemos implementar um ambiente interativo para a linguagem, contendo toda especificação formal e sendo capaz de funcionar ainda como um interpretador.

7.1 Principais Resultados Obtidos

O desenvolvimento de um "Ambiente Operacional Interativo ParaLog" [FM01] está fundamentado em uma variação da linguagem Prolog, denominada Prolog Paraconsistente, baseada nas lógicas de primeira ordem $\mathcal{Q}\tau$. O reticulado (τ) utilizado nessa versão do protótipo é aquele definido em [AS87] e, portanto, as constantes anotacionais utilizadas são *t,f,lt,lf,tf, **.

Nosso protótipo implementa o processo de regularização e as duas transformações sintáticas do Prolog Paraconsistente. O módulo *Unification*[1] e *VariantClause* são responsáveis pelo processo de regularização e os módulos *EliminatioOfNegation* e *Closed* são responsáveis pelo processo de transformação sintática da eliminação da negação e do fechamento. Depois desses processos, o programa (conjunto de cláusulas e uma lista de objetivos) é manipulado como PHG's bem comportado no qual a resolução SLD-nh pode ser aplicada.

O motor de inferência, implementado no módulo *EvalParaLog*, seleciona as cláusulas do programa de tal forma que a cláusula $C{:}\mu$ selecionada possui anotação μ igual ao supremo conjunto formado pelas anotações das cláusulas candidatas. Esse motor simula também uma busca semelhante a *best-first* na árvore para as cláusulas do programa e fornece como resposta uma crença $\psi \in \tau$. Nos casos em que $\psi \neq *$, a resposta também inclui uma substituição θ para as variáveis da lista de objetivos.

É importante ressaltar que, quando existir mais de uma resposta válida para a lista de objetivos, o motor de inferência as fornece, pois ele implementa o processo de *backtracking*.

Dessa forma, pode-se concluir que, com o desenvolvimento deste protótipo, o programador — além de possuir um interpretador para a linguagem ParaLog que efetua o processo de *backtracking* automaticamente — contém também toda a especificação formal dessa linguagem, visto que esse foi desenvolvido no meta-ambiente ASF+SDF, o qual permite a definição sintática e semântica de uma linguagem usando as regras de semântica de transição.

Vale salientar que, no desenvolvimento do ambiente proposto, não existiu preocupação com o desempenho do protótipo. O ambiente interativo com semântica operacional

[1]Este módulo é responsável também por encontrar o unificador mais geral(umg).

de transição para linguagem ParaLog possui a vantagem da formalização de programas lógicos paraconsistentes. A especificação aqui definida é a primeira formalização da linguagem ParaLog em todos os seus detalhes. Existem outras especificações anteriores [CPA+95], porém elas não cobrem todos os detalhes da linguagem. Também nossa especificação é a primeira especificação em semântica de transição de ParaLog de que temos conhecimento e implementa fielmente o que é especificado na bibliografia [CPA+95, Sub87b].

No que se refere à qualidade do nosso protótipo temos:

- uma ferramenta que proporciona um ambiente de edição dirigida pela sintaxe (editor de termos) e um mecanismo de execução de programas,
- possibilidade de mudar tanto a sintaxe quanto a semântica dos programas; constituíndo-se num verdadeiro laboratório de linguagem.

Em relação à complexidade de avaliação dos programas temos um algoritmo exponencial, classe $P = NP$, similar à avaliação de programas Prolog [Vel89].

7.2 Sugestões de Trabalhos Futuros

Existem várias linhas de pesquisas que podem dar continuidade e complementar aspectos abordados neste trabalho. Algumas delas são descritas a seguir:

- criar avaliação de expressões funcionais (tipo "is" [2]) em Prolog, a qual será também utilizada na linguagem ParaLog;
- desenvolver um protótipo para transformação de programas ParaLog, sem a preocupação com consultas e respostas, obtendo resultados similares aos de Brunekreef;
- estender o nosso protótipo para o $ParaLog_e$ [AAP97], que é uma extensão do ParaLog proposto que utiliza conceito de Programação Lógica Evidencial [Sub87b];
- desenvolver uma aplicação eficiente baseada neste protótipo,
- criar uma interface interativa para este protótipo.

[2] Este comando resulta por exemplo, na instanciação de uma variável pelo resultado da avaliação de uma expressão. Ex: X *is* 5+6 unifica X com 11.

BIBLIOGRAFIA

[Á96] Bráulio Coelho Ávila. *Uma Abordagem Paraconsistente Baseada em Lógica Evidencial para Tratar Exceções em Sistemas de Frames com Múltipla Herança.* PhD thesis, Escola Politécnica da Univerisade de São Paulo, 1996.

[AA98] Seiki Akama and Jair Minoro Abe. Many-valued logics and annotated modal logics. Technical Report 38, IEA (Instituto de Estudos Avançados), USP, jun 1998. Lógica e Teoria da Ciência.

[AAP97] Bráulio Coelho Ávila, Jair Minoro Abe, and José Pacheco Almeida Prado, editors. *$ParaLog_e$: A Paraconsistent Evidential Logic Programming Language.* IEEE, 1997.

[Abe92] Jair Minoro Abe. *Fundamentos da Lógica Anotada.* PhD thesis, FFLCH/USP, 1992.

[Abe93] Jair Minoro Abe. On annotated model theory. Technical Report 11, IEA (Instituto de Estudos Avançados), USP, jun 1993. Lógica e Teoria da Ciência.

[Acz77] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 739–782. North-Holland Publishing Company, 1977.

[AF98] Jair Minoro Abe and Joao I. Silva Filho. Algoritmo para-analisador - parte ii. Technical Report 44, IEA (Instituto de Estudos Avançados), USP, jun 1998. Lógica e Teoria da Ciência.

[APA97] Jair Minoro Abe, José Pacheco Almeida Prado, and Bráulio Coelho Ávila. On a class of paraconsistent multimodal systems for reasoning. Technical Report 24, IEA (Instituto de Estudos Avançados), USP, jun 1997. Lógica e Teoria da Ciência.

[Arr80] A. I. Arruda. *A Survey of Paraconsistent Logic.* A.I. Arruda and R. Chuaqui and N.C.A. da Costa, North-Holland, Amsterdam, 1980. am Mathematical Logic in Latin America.

[AS87] R. Anand and V. S. Subrahmanian. *Flog: A Logic Programming System Based on a Six-Valued Logic.* AAAI/Xerox Second Intl. Symp. on Knowledge Eng., Madri, Espanha, 1987.

[Ast91] Edigio Astesiano. Inductive and operacional semantics. In *Formal Description of Programming Concepts* [IFI91], pages 51–136.

[BCD+88] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, and V. Pascual, editors. *Centaur: the system*, Boston, USA, 1988. Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3).

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. Algebraic specification. Technical report, ACM Press Frontier Series, 1989.

[BK00] M.G.J. Van Den Brand and P. Klint. Asf+sdf meta-environment user manual revision: 1.53. Technical report, CWI, Centrum voor Wiskunde en Informatica, Amsterdam, 2000. Available at URL: http://www.cwi.nl/projects/MetaEnv/meta.

[BP93] Arthur Ronald Vallauris Buschsbaum and Tarcísio Haroldo Cavalcante Pequeno. Uma família de lógicas paraconsistentes e/ou paracompletas com semânticas recursivas. Technical Report 14, IEA (Instituto de Estudos Avançados), USP, set 1993. Lógica e Teoria da Ciência.

[Bra86] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley Publishers Limited, 1986.

[Bru96] Jacob Brunekreef. A transformation tool for pure prolog programs. In *LOPSTR'96 - Sixth International Workshop on Logic Program Synthesis and Transformation*, 1996.

[BS86] R. Bahlke and G. Snelting. *The PSG system: from formal language de-*

finitions to interactive programming environments. ACM Transactions on Programming Languages and Systems, 1986.

[BS87] Howard A. Blair and V. S. Subrahmanian. Paraconsistent logic programming. In *Proc. 7^{th} Conf. on Foundations of Software Technology & Theoretical Computer Science*, pages 340–360, Springer-Verlag, 1987. Journal Theoretical Computer Science. Lectures Notes in Computer Science.

[BS88] Howard A. Blair and V. S. Subrahmanian. Paraconsistent foundations for logic programming. *Journal of Non-Classical Logic,5,2*, pages 45–73, 1988.

[CAS91] Newton C. A. Costa, Jair Minoro Abe, and V. S. Subrahmanian. Remarks on annotated logic. Technical Report 8, IEA (Instituto de Estudos Avançados), USP, jun 1991. Lógica e Teoria da Ciência.

[CGF87] Marco A. Casanova, Fernando A. C. Giorno, and Antonio L. Furtado. *Programação em Lógica e a Linguagem Prolog.* Edgard Blucher Ltda, São Paulo, Brazil, 1987.

[CHLS90] Newton C. A. Costa, Lawrence J. Henschen, James J. Lu, and V. S. Subrahmanian. Automatic theorem proving in paraconsistent logics: Theory and implementation. Technical Report 03, IEA (Instituto de Estudos Avançados), USP, mai 1990. Lógica e Teoria da Ciência.

[CM87] Newton C. Costa and D. Marconi. *An overview of Paraconsistent Logic in the 80's*, volume 5. Monografias da Soc. Paranaense de Matemática, 1987. aparecer em Lógica Nova, 1988.

[Cos93] Newton C. A. Costa. *Sistemas Formais Inconsistentes.* Clássicos n. 03. Editora UFPR, 1993.

[CPA+95] Newton C. A. Costa, José Pacheco Almeida Prado, Jair Minoro Abe, Bráulio Coelho Ávila, and Márcio Rillo. Paralog: Um prolog paraconsistente baseado em lógica anotada. Technical Report 18, IEA (Instituto de Estudos Avançados), USP, abr 1995. Lógica e Teoria da Ciência.

[CS89] Newton C. A. Costa and V. S. Subrahmanian. Paraconsistent logics as a formalism for reasoning about inconsistent knowledge bases. Technical Re-

port 02, IEA (Instituto de Estudos Avançados), USP, nov 1989. Lógica e Teoria da Ciência.

[CSV89] Newton C. A. Costa, V. S. Subrahmanian, and Carlo Vago. The paraconsistent logics p_t. Technical Report 01, IEA (Instituto de Estudos Avançados), USP, nov 1989. Lógica e Teoria da Ciência.

[FAT+99] Joao Inácio Silva Filho, Jair Minoro Abe, Cláudio Rodrigo Torres, Alexandre M. César, Maurício C. Mário, Ari Mendes Santos, Israel J. Cancino Júnior, and Danilo Mendonça Salles. Emmy: Robô móvel autônomo paraconsistente - protótipo 1. Technical Report 59, IEA (Instituto de Estudos Avançados), USP, mai 1999. Lógica e Teoria da Ciência.

[FM01] Simone Nasser Matos Ferreira and Martin A. Musicante. Using operational semantics for the development of a paralog environment. *V Simpósio Brasileiro de Linguagens de Programação 2001*, pages C16–C31, may 2001. SBC - Sociedade Brasileira de Computação.

[Gun91] Carl A. Gunter. Forms of semantic specification. *Bulletin of the European Association for Theoretical Computer Science*, 45:98–113, oct 1991. The Logic in Computer Science Column, by Yury Gurevich.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HHKR92] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF: Reference Manual*, dec 1992.

[HK89] J. Heering and P. Klint. *PICO revisited.* ACM Press Frontier Series, pages 359-379, 1989. The ACM Press in operation with Addison-Wesley.

[IFI91] IFIP. *Formal Description of Programming Concepts*, IFIP State-of-the-Arte Report. Spring-Verlag, 1991.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler.* Bell Laboratories, unix programmer's supplementary documents, volume 1 edition, 1986.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. Symp. on Theoretical Aspects*

of Computer Science, number 247 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[Kli92] P. Klint. The asf+sdf meta-environment. Technical report, CWI, Centrum voor Wiskunde en Informatica, Amsterdam, 1992. Available at URL: ftp://ftp.cwi.nl/pub/gipe/repots/SysManual.ps.Z.

[KLMM83] G. Kahn, B. Lang, B. Melese, and E. Morcos. *Meta: a formalism to specify formalisms.* Science of Computer Programming, 3:151-188, 1983.

[Klo92] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume 2, pages 1–166. Oxford University Press, New York, 1992.

[Kow79] R. A. Kowalski, editor. *Algorithm - Logic - Control*, 1979. CACM, 22, 7.

[Llo84] J.W. Lloyd, editor. *Foundations of Programming.* Spring-Verlag, 1984.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator.* Bell Laboratories, unix programmer's supplementary documents, volume 1 edition, 1986.

[MM82] A. Martelli and U. Montanari. *An efficient unification algorithm.* ACM Transactions on Programming Language and Systems, 4:258-282, 1982.

[Mus96] Martin A. Musicante. *On the Relational Semantics of Interleaving Constructors.* PhD thesis, UFPE, Departamento da Ciência da Computação, 1996.

[NAS99] Kazumi Nakamatsu, Jair Minoro Abe, and Atsuyuki Suzuki. Aplicações de programação anotada. Technical Report 60, IEA (Instituto de Estudos Avançados), USP, mai 1999. Lógica e Teoria da Ciência.

[PA98] José Pacheco Almeida Prado and Jair Minoro Abe. Uma arquitetura para inteligência artifical baseada em lógica paraconsistente anotada. Technical Report 33, IEA (Instituto de Estudos Avançados), USP, jan 1998. Lógica e Teoria da Ciência.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Lecture notes, Aarhus University, Computer Science Departament, 1981. Now available only from University of Edinburg.

[Rep82] T. Reps. Generating language-based environments. Technical Report 82-514, Cornell University, Ithaca, 1982.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[Sak92] Chiaki Sakama. Extended well-founded semantics for paraconsistent logic programs. In *Proccedings of the 1992 International Conference on Fifty Generation Computer Systems*, pages 592–599, Ohmsha, 1992.

[Sil92] Fábio Queda Bueno Silva. *Correctness proofs of compilers and debuggers: An approach based on Structural Operational Semantics.* Ph.d.thesis, Dept. of Computer Science, The University of Edinburgh, 1992.

[Sub87a] V. S. Subrahmanian. On the semantics of quantitative logic programs. In 4^{th} *IEEE Symposium on Logic Programming*, pages 173–182, San Francisco, set 1987. Computer Society Press.

[Sub87b] V. S. Subrahmanian. Towards a theory of evidencial reasoning in logic programming. In *Logic Colloquim'87 - The European Summer Meeting of the Association for Symbolic Logic*, Granada, Spain, jul 1987.

[Vel89] André Vellino. *The Complexity of Automated Reasoning.* PhD thesis, University of Toronto, 1989.

[War79] D.H.D. Warren. *Prolog on the DECsystem-10.* em Expert System in Micro Eletronica Age, D. Michie (ed.), Edinburg, University Press, 1979.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall International, Series in Computer Science. Prentice Hall, 1991.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing Series. MIT Press, 1993.

APÊNDICE A

Programas em ASF+SDF

Neste trabalho as regras de semântica de transição para as linguagens *Exp*, Prolog e ParaLog foram implementadas no meta-ambiente ASF+SDF. Nas seções seguintes apresenta-se as listagens dos programas desenvolvidos nos Capítulos 3, 5 e 6 respectivamente.

A.1 Capítulo 3 - Linguagem *Exp*

A.1.1 Booleans.sdf2

module Booleans
imports Layout
exports
sorts Bool
context-free syntax

"nil-value"	→	Bool
"true"	→	Bool
"false"	→	Bool
"and" "(" Bool "," Bool ")"	→	Bool
"or" "(" Bool "," Bool ")"	→	Bool
"neg" "(" Bool ")"	→	Bool

variables

"bv"[0-9\']* → Bool

A.1.2 Booleans.eqs

equations

[b1]	or(true,bv)	=	true
[b2]	or(false, bv)	=	bv

[b3] and(true, bv) = bv

[b4] and(false,bv) = false

[b5] neg(true) = false

[b6] neg(false) = true

A.1.3 Identifiers.sdf2

module Identifiers

imports Layout

exports

sorts Id Bid

lexical syntax

[a-z] [a-z0-9]* → Id

[A-Z] [a-z0-9]* → Bid

variables

"id" [0-9\']* → Id

"bid" [0-9\']* → Bid

A.1.4 Integers.sdf2

module Integers

imports Layout Booleans

exports

sorts Num

lexical syntax

[0-9]+ → Num

context-free syntax

"nil-value" → Num

"(" Num ")" → Num

"+" Num → Num

"-" Num → Num

Num "-/" Num → Num

"plus" "(" Num "," Num ")" → Num

"sub" "(" Num "," Num ")" → Num

"mult" "(" Num "," Num ")" → Num

"gt" "(" Num "," Num ")" → Bool

"ge" "(" Num "," Num ")" → Bool

"lt" "(" Num "," Num ")" → Bool

"le" "(" Num "," Num ")" → Bool

variables

"n" [0-9\']* → Num

"c" [0-9]* → CHAR

"x" [0-9]* → CHAR*

"y" [0-9]* → CHAR+

hiddens

context-free syntax

"agt" "(" Num "," Num ")" → Bool

Num ">-" Num → Num

"subaux" "(" Num "," Num ")" → Num

A.1.5 Integers.eqs

equations

[1] num("0" y) = num(y)

%% - Addition -

[2] plus(0, n) = n

[3] plus(n, 0) = n

[4]	plus(1, 1)	=	2	[5]	plus(1, 2)	=	3	[6]	plus(1, 3)	=	4
[7]	plus(1, 4)	=	5	[8]	plus(1, 5)	=	6	[9]	plus(1, 6)	=	7
[10]	plus(1, 7)	=	8	[11]	plus(1, 8)	=	9	[12]	plus(1, 9)	=	10
[13]	plus(2, 1)	=	3	[14]	plus(2, 2)	=	4	[15]	plus(2, 3)	=	5
[16]	plus(2, 4)	=	6	[17]	plus(2, 5)	=	7	[18]	plus(2, 6)	=	8
[19]	plus(2, 7)	=	9	[20]	plus(2, 8)	=	10	[21]	plus(2, 9)	=	11
[22]	plus(3, 1)	=	4	[23]	plus(3, 2)	=	5	[24]	plus(3, 3)	=	6
[25]	plus(3, 4)	=	7	[26]	plus(3, 5)	=	8	[27]	plus(3, 6)	=	9
[28]	plus(3, 7)	=	10	[29]	plus(3, 8)	=	11	[30]	plus(3, 9)	=	12
[31]	plus(4, 1)	=	5	[32]	plus(4, 2)	=	6	[33]	plus(4, 3)	=	7
[34]	plus(4, 4)	=	8	[35]	plus(4, 5)	=	9	[36]	plus(4, 6)	=	10
[37]	plus(4, 7)	=	11	[38]	plus(4, 8)	=	12	[39]	plus(4, 9)	=	13
[40]	plus(5, 1)	=	6	[41]	plus(5, 2)	=	7	[42]	plus(5, 3)	=	8
[43]	plus(5, 4)	=	9	[44]	plus(5, 5)	=	10	[45]	plus(5, 6)	=	11
[46]	plus(5, 7)	=	12	[47]	plus(5, 8)	=	13	[48]	plus(5, 9)	=	14
[49]	plus(6, 1)	=	7	[50]	plus(6, 2)	=	8	[51]	plus(6, 3)	=	9
[52]	plus(6, 4)	=	10	[53]	plus(6, 5)	=	11	[54]	plus(6, 6)	=	12
[55]	plus(6, 7)	=	13	[56]	plus(6, 8)	=	14	[57]	plus(6, 9)	=	15
[58]	plus(7, 1)	=	8	[59]	plus(7, 2)	=	9	[60]	plus(7, 3)	=	10
[61]	plus(7, 4)	=	11	[62]	plus(7, 5)	=	12	[63]	plus(7, 6)	=	13

[64] plus(7, 7) = 14 [65] plus(7, 8) = 15 [66] plus(7, 9) = 16

[67] plus(8, 1) = 9 [68] plus(8, 2) = 10 [69] plus(8, 3) = 11

[70] plus(8, 4) = 12 [71] plus(8, 5) = 13 [72] plus(8, 6) = 14

[73] plus(8, 7) = 15 [74] plus(8, 8) = 16 [75] plus(8, 9) = 17

[76] plus(9, 1) = 10 [77] plus(9, 2) = 11 [78] plus(9, 3) = 12

[79] plus(9, 4) = 13 [80] plus(9, 5) = 14 [81] plus(9, 6) = 15

[82] plus(9, 7) = 16 [83] plus(9, 8) = 17 [84] plus(9, 9) = 18

[85a] plus(num(c1), num(c2)) = num(x c),
plus(num(y1), num(y2)) = num(x1 y3),
plus(num(x1 y3) , num("0" x)) = num(y)
==========
plus(num(y1 c1), num(y2 c2))= num(y c)

[85b] plus(num(c1), num(c2)) = num(x c),
plus(num("0" x1), num(y2)) = num(x2 y3),
plus(num(x2 y3), num("0" x)) = num(y)
==========
plus(num(x1 c1), num(y2 c2)) = num(y c)

[85c] plus(num(c1), num(c2)) = num(x c),
plus(num(y1), num("0" x2)) = num(x3 y3),
plus(num(x3 y3), num("0" x)) = num(y)
==========
plus(num(y1 c1), num(x2 c2)) = num(y c)

%%– Auxiliary Subtraction –

[86] subaux(num(c), num(c)) = 0

[87] subaux(num(c), 0) = num(c)

[88] subaux(2, 1) = 1

[89] subaux(3, 1) = 2 [90] subaux(3, 2) = 1

[91] subaux(4, 1) = 3 [92] subaux(4, 2) = 2 [93] subaux(4, 3) = 1

[94] subaux(5, 1) = 4 [95] subaux(5, 2) = 3 [96] subaux(5, 3) = 2

[97] subaux(5, 4) = 1

[98] subaux(6, 1) = 5 [99] subaux(6, 2) = 4 [100] subaux(6, 3) = 3

[101] subaux(6, 4) = 2 [102] subaux(6, 5) = 1

[103] subaux(7, 1) = 6 [104] subaux(7, 2) = 5 [105] subaux(7, 3) = 4

[106] subaux(7, 4) = 3 [107] subaux(7, 5) = 2 [108] subaux(7, 6) = 1

[109] subaux(8, 1) = 7 [110] subaux(8, 2) = 6 [111] subaux(8, 3) = 5

[112] subaux(8, 4) = 4 [113] subaux(8, 5) = 3 [114] subaux(8, 6) = 2

[115] subaux(8, 7) = 1

[116] subaux(9, 1) = 8 [117] subaux(9, 2) = 7 [118] subaux(9, 3) = 6

[119] subaux(9, 4) = 5 [120] subaux(9, 5) = 4 [121] subaux(9, 6) = 3

[122] subaux(9, 7) = 2 [123] subaux(9, 8) = 1

[124] subaux(10,1) = 9 [125] subaux(10,2) = 8 [126] subaux(10,3) = 7

[127] subaux(10,4) = 6 [128] subaux(10,5) = 5 [129] subaux(10,6) = 4

[130] subaux(10,7) = 3 [131] subaux(10,8) = 2 [132] subaux(10,9) = 1

%% – Subtraction –

[133] sub(n,0) = n

[134] sub(num(c1), num (c2)) = subaux(num(c1), num(c2))

[135a] subaux(num(c1), num(c2)) = num(c3),
sub(num(y1), num("0" x2)) = num(y)
==========
sub(num(y1 c1), num(x2 c2)) = num(y c3)

[135b] subaux(num(c2), num(c1)) = num(c3),
subaux(10, num(c3)) = num(c),
plus(num("0" x2), 1) = n,
sub(num(y1), n) = num(y)
==========
sub(num(y1 c1), num(x2 c2)) = num(y c)

[136] subaux(num(c1), num(c2)) = num(c),
num(c) != 0
==========
agt(num(c1), num(c2)) = true

[137] agt(num(y1 c1), num(c2)) = true

[138] agt(num(y c1), num(y c2)) = agt(num(c1), num(c2))

[139] agt(num(y1), num(y2)) = true
==========
agt(num(y1 c1), num(y2 c2)) = true

%% – Cut off subtraction -/

[140] agt(n1, n2) = true === n1 -/ n2 = sub(n1, n2)

[141] agt(n1, n2) != true === n1 -/ n2 = 0

%% – Subtraction when n2 gt n1

[142] sub(n1, n2) = -n when agt(n2,n1) = true, sub(n2,n1) = n

%% – Multiplication of naturals –

[143] mult(n, 0) = 0

[144] mult(n, 1) = n

[145] agt(num(c), 1) = true
==========
mult(n, num(c)) = plus(n, mult(n, subaux(num(c),1)))

[146] mult(num(y1), num(y2)) = num(y)

==========

mult(num(y1), num(y2 c)) = plus(num(y "0"), mult(num(y1), num(c)))

%% – addition, subtraction, and multiplication of integers

[147] plus(n1, -n2) = sub(n1, n2)

[148] plus(-n1, n2) = sub(n2, n1)

[149] plus(n1, n2) = n === plus(-n1, -n2) = -n

[150] sub(n1, -n2) = plus(n1, n2)

[151] plus(n1, n2) = n === sub(-n1, n2) = -n

[152] sub(-n1, -n2) = sub(n2, n1)

[153] mult(n1, n2) = n === mult(n1, -n2) = -n

[154] mult(n1, n2) = n === mult(-n1, n2) = -n

[155] mult(-n1, -n2) = mult(n1,n2)

%% – relational operators

[156] agt(n1, n2) = true === gt(n1, n2) = true

[157] agt(n1, n2) != true === gt(n1, n2) = false

[158] gt(n1, -n2) = true

[159] gt(-n1, n2) = false

[160] gt(-n1,-n2) = gt(n2, n1)

[161] n1 != n2 === ge(n1, n2) = gt(n1, n2)

[162] ge(n, n) = true

[163] lt(n1, n2) = neg(ge(n1,n2))

[164] le(n1, n2) = neg(gt(n1, n2))

A.1.6 Layout.sdf2

module Layout

exports

lexical syntax

[\ \t\n] → LAYOUT

"%%" ~ [\n]* "\n" → LAYOUT

"%" [\%\\n]+ "%" → LAYOUT

context-free restrictions

LAYOUT? -/- [\ \t\n\%]

A.1.7 LingExp.sdf2

module LingExp

imports Value-EnvBool Value-EnvInt

exports

sorts Iexp Bexp Exp

context-free syntax

Bexp	→	Exp
Iexp	→	Exp
Id	→	Iexp
Num	→	Iexp
Iexp "+" Iexp	→	Iexp**{left}**
Iexp "-" Iexp	→	Iexp**{left}**
Iexp "*" Iexp	→	Iexp**{left}**
"(" Iexp ")"	→	Iexp**{bracket}**
Bid	→	Bexp
Bool	→	Bexp
Bexp "&" Bexp	→	Bexp**{left}**
Bexp "|" Bexp	→	Bexp**{left}**
Iexp ">" Iexp	→	Bexp
Iexp ">=" Iexp	→	Bexp
Iexp "<" Iexp	→	Bexp
Iexp "<=" Iexp	→	Bexp
"not" "(" Bexp ")"	→	Bexp
"(" Bexp ")"	→	Bexp**{bracket}**
"step" "(" VENV "," VENVB "," Exp ")"	→	Exp
"evalExp" "(" VENV "," VENVB "," Exp ")"	→	Exp
"isnumber" "(" Exp ")"	→	Bool

variables

"v"[0-9\']*	→	Num
"e"[0-9\']*	→	Iexp
"bc"[0-9\']*	→	Bool
"v"[0-9\']*	→	Num
"b"[0-9\']*	→	Bexp
"IB"[0-9\']*	→	Exp

context-free priorities

Bexp "&" Bexp → Bexp >
Bexp "|" Bexp → Bexp >
Iexp "*" Iexp → Iexp >
{**left**: Iexp "+" Iexp → Iexp **{left}**
Iexp "-" Iexp → Iexp **{left}**
}

A.1.8 LingExp.eqs

equations

[01] step(Venv, VenvB, n) = n

[02] step(Venv, VenvB, bc) = bc

[03] step(Venv, VenvB, id) = lookup(id , Venv)

[04] step(Venv, VenvB, bid) = lookupB(bid, VenvB)

[09a] step(Venv, VenvB, e1 + e2) = e1' + e2 **when** step(Venv, VenvB, e1) = e1'
[09b] step(Venv, VenvB, e1 - e2) = e1' - e2 **when** step(Venv, VenvB, e1) = e1'
[09c] step(Venv, VenvB, e1 * e2) = e1' * e2 **when** step(Venv, VenvB, e1) = e1'

[10a] step(Venv, VenvB, v + e2) = v + e2' **when** step(Venv, VenvB, e2) = e2'
[10b] step(Venv, VenvB, v - e2) = v - e2' **when** step(Venv, VenvB, e2) = e2'
[10c] step(Venv, VenvB, v * e2) = v * e2' **when** step(Venv, VenvB, e2) = e2'
[11a] step(Venv, VenvB, e1 > e2) = e1' > e2 **when** step(Venv, VenvB, e1) = e1'
[11b] step(Venv, VenvB, e1 >= e2) = e1' >= e2 **when** step(Venv, VenvB, e1) = e1'
[11c] step(Venv, VenvB, e1 < e2) = e1' < e2 **when** step(Venv, VenvB, e1) = e1'
[11d] step(Venv, VenvB, e1 <= e2) = e1' <= e2 **when** step(Venv, VenvB, e1) = e1'

[12a] step(Venv, VenvB, v > e2) = v > e2' **when** step(Venv, VenvB, e2) = e2'
[12b] step(Venv, VenvB, v >= e2) = v >= e2' **when** step(Venv, VenvB, e2) = e2'
[12c] step(Venv, VenvB, v < e2) = v < e2' **when** step(Venv, VenvB, e2) = e2'
[12d] step(Venv, VenvB, v <= e2) = v <= e2' **when** step(Venv, VenvB, e2) = e2'

[13a] step(Venv, VenvB, b1 & b2) = b1' & b2 **when** step(Venv, VenvB, b1)= b1'
[13b] step(Venv, VenvB, b1 | b2) = b1' | b2 **when** step(Venv, VenvB, b1)= b1'

[14a] step(Venv, VenvB, bv1 & b2) = bv1 & b2' **when** step(Venv, VenvB, b2)= b2'
[14b] step(Venv, VenvB, bv1 | b2) = bv1 | b2' **when** step(Venv, VenvB, b2) = b2'

[15] step(Venv, VenvB, not (b)) = not(b') **when** step(Venv, VenvB, b) = b'

[05a] step(Venv, VenvB, v1 + v2) = v **when** plus(v1, v2) = v
[05b] step(Venv, VenvB, v1 - v2) = v **when** sub(v1, v2) = v
[05c] step(Venv, VenvB, v1 * v2) = v **when** mult(v1, v2)= v

[06a] step(Venv, VenvB, bv1 & bv2) = bv **when** and(bv1, bv2) = bv
[06b] step(Venv, VenvB, bv1 | bv2) = bv **when** or(bv1, bv2) = bv

[07a] step(Venv, VenvB, v1 < v2) = bv **when** lt(v1, v2) = bv
[07b] step(Venv, VenvB, v1 <= v2) = bv **when** le(v1, v2) = bv
[07c] step(Venv, VenvB, v1 > v2) = bv **when** gt(v1, v2) = bv
[07d] step(Venv, VenvB, v1 >= v2) = bv **when** ge(v1, v2) = bv

[08] step(Venv, VenvB, not (bv)) = bv' **when** neg(bv) = bv'
[n1] isnumber(IB) = false
[n2] isnumber(n) = true

[ev1] evalExp(Venv, VenvB, n) = n
[ev2] evalExp(Venv, VenvB, IB) = true **when** IB = true
[ev3] evalExp(Venv, VenvB, IB) = false **when** IB =false
[ev4] evalExp(Venv, VenvB, IB) = IB" **when**
IB != false, IB != true, isnumber(IB) = false,
step(Venv, VenvB, IB) = IB',
evalExp(Venv, VenvB, IB') = IB"

A.1.9 Value-EnvBool.sdf2

module Value-EnvBool

imports Identifiers Booleans

exports

sorts VENVB VPAIRB

context-free syntax

"(" Bid ":" Bool ")" → VPAIRB
"[" {VPAIRB "," }* "]" → VENVB

VPAIRB "+" VENVB → VENVB

"lookupB" "(" Bid "," VENVB ")" → Bool

"updateB" "(" Bid "," Bool "," VENVB ")" → VENVB

variables

"VPairB" [0-9]* → VPAIRB

"VPairB*"[0-9\']* → {VPAIRB ","}*

"VenvB" [0-9\']* → VENVB

A.1.10 Value-EnvBool.eqs

equations

[veb1] VPairB + [VPairB*] = [VPairB, VPairB*]

[veb2a] lookupB(bid, [(bid:bv), VPairB*]) = bv

[veb2b] lookupB(bid, [(bid':bv), VPairB*]) = lookupB(bid, [VPairB*]) **when** bid != bid'

[veb2c] lookupB(bid, []) = nil-value

[veb3a] updateB(bid, bv', [(bid:bv),VPairB*]) = [(bid:bv'), VPairB*]

[veb3b] bid != bid'

==========

updateB(bid, bv', [(bid':bv), VPairB*]) = (bid':bv) + updateB(bid,bv',[VPairB*])

[veb3c] updateB(bid,bv,[]) = [(bid:bv)]

A.1.11 Value-EnvInt.sdf2

module Value-EnvInt

imports Identifiers Integers

exports

sorts VENV VPAIR

context-free syntax

"(" Id ":" Num ")" → VPAIR

"[" {VPAIR "," }* "]" → VENV

VPAIR "+" VENV → VENV

"lookup" "(" Id "," VENV ")" → Num

"update" "(" Id "," Num "," VENV ")" → VENV

variables

"VPair" [0-9]* → VPAIR

"VPair*"[0-9\']* → {VPAIR ","}*

"Venv" [0-9\']* → VENV

A.1.12 Value-EnvInt.eqs

equations

```
[ve1]   VPair + [VPair*]                 =  [VPair, VPair*]
[ve2a]  lookup(id, [(id:n), VPair*])     =  n
[ve2b]  lookup(id, [(id':n), VPair*])    =  lookup(id, [VPair*]) when
                                            id != id'
[ve2c]  lookup(id, [])                   =  nil-value
[ve3a]  update(id, n', [(id:n),VPair*])  =  [(id:n'), VPair*]
[ve3b]  id != id'
        ========================================
        update(id, n', [(id':n), VPair*]) = (id':n) + update(id,n',[VPair*])
[ve3c]  update(id,n,[]) = [(id:n)]
```

A.2 Capítulo 5 - Linguagem Prolog

A.2.1 Booleans.sdf2

module Booleans

imports Layout

exports

sorts BOOL

context-free syntax

```
"true"                  →  BOOL
"false"                 →  BOOL

BOOL "|" BOOL           →  BOOL {left}
BOOL "&" BOOL           →  BOOL {left}
"not" "(" BOOL ")"      →  BOOL

"(" BOOL ")"            →  BOOL {bracket}
```

variables

```
"B"[\ ']*  →  BOOL
```

context-free priorities

```
BOOL "&" BOOL  →  BOOL  >
BOOL "|" BOOL  →  BOOL
```

A.2.2 Booleans.eqs

equations

[o1] true | B = true

[o2] false | B = B

[a1] true & B = B

[a2] false & B = false

[n1] not(false) = true

[n2] not(true) = false

A.2.3 Equations.sdf2

module Equations

imports Substitution

exports

sorts EQ EQS

context-free syntax

TERM "equivalente" TERM → EQ

"{" {EQ ","}* "}" → EQS

EQ "[e" SUBS "]" → EQ

EQS "[es" SUBS "]" → EQS

variables

"E"[\']* → EQ

"E+"[\']* → {EQ ","}+

"E*"[\']* → {EQ ","}*

"EQS"[\']* → EQS

A.2.4 Equations.eqs

equations

[eq1] T equivalente T'[e S] = T[t S] equivalente T'[t S]

[es1] {}[es S] = {}

[es2] {E}[es S] = {E[e S]}

[es3] {E+, E+'}[es S] = {E+", E+"'} **when**
{E+}[es S] = {E+"}, {E+'}[es S] = {E+"'}

A.2.5 EvalProlog.sdf2

module EvalProlog

imports VariantClause

exports

sorts LISTSUBS

context-free syntax

"eval-K" "(" CLAUSE* "," SUBS "," PROGRAM ")" → LISTSUBS
"eval" "(" PROGRAM ")" → LISTSUBS

"[" {SUBS ","}* "]" → LISTSUBS
"sucess" → LISTSUBS
"fail" → LISTSUBS
"yes" → LISTSUBS

"verify-LS" "(" LISTSUBS "," TERM-SET ")" → LISTSUBS
"union-LS" "(" LISTSUBS "," LISTSUBS ")" → LISTSUBS
"insertSinLS" "(" SUBS "," LISTSUBS ")" → LISTSUBS
"conc" "(" LISTSUBS "," LISTSUBS ")" → LISTSUBS
"append" "(" TERMLIST ";" TERMLIST ")" → TERMLIST

"headO" "(" OBJECTIVE ")" → TERM
"tailC" "(" CLAUSE ")" → TERMLIST

"var" "(" TERMLIST ")" → TERM-SET
"verify-var" "(" TERMLIST ";" TERM-SET ")" → TERM-SET
"listvar" "(" OBJECTIVE "," TERM-SET ")" → TERM-SET
"find" "(" TERM-SET "," LISTSUBS ")" → LISTSUBS
"findS" "(" TERM-SET "," SUBS "," SUBS ")" → SUBS

variables

"LS"[\']* → LISTSUBS
"LS*"[\']* → {SUBS ","}*
"x"[\']* → CHAR
"x+"[\']* → CHAR+

A.2.6 EvalProlog.eqs

equations

%% Evalution Function

[ev1] eval(C* O) = LS' **when**
listvar(O,{}) = TS,
variant-P(C* O, {}) = C*' O',
eval-K(C*', {}, C*' O')= LS,
verify-LS(LS,TS) = LS'

%% Evalution Auxiliary Function

[evk1a] eval-K(C*,S,C' C*' ?-) = sucess

[evk1b] eval-K(C*,S,?-) = sucess

[evk2a] eval-K(C*, S, ?- T.) = [] **when** T != []
[evk2b] eval-K(C*, S, ?- T, T+.) = [] **when** T != []

[evk3a] eval-K(C*,S, ?-[].) = [S]
[evk3b] eval-K(C*, S, C' C*' ?-[].) = [S]

[evk4a] eval-K(C*,S,C' C*' O) = LS **when**
head(C') = T',
headO(O) = T,
mgu-nv(T',T) = S', S'= fail,
eval-K(C*,S, C*' O) = LS
[evk4b] eval-K(C*,S, C' C*' ?- T, T+.) = LS"' **when**
head(C') = T',
mgu-nv(T',T) = S', S' != fail,
S simples S' = S",
tailC(C') = T+',
append(T+';T+) = T+",
T+"[tl S"] = T+"',
eval-K(C*,{},C* ?-T+"'.) = LS,
insertSinLS(S', LS) = LS',
eval-K(C*,S,C*' ?- T, T+.) = LS",
conc(LS',LS") = LS"'
[evk4c] eval-K(C*,S, C' C*' ?- T.) = LS"' **when**
head(C') = T',
mgu-nv(T',T) = S', S' != fail,
S simples S' = S",
tailC(C') = T+',
T+'[tl S"] = T+",
eval-K(C*,{},C* ?-T+".) = LS,
insertSinLS(S', LS) = LS',
eval-K(C*,S,C*' ?- T.) = LS",
conc(LS', LS") = LS"'

%% Insert Substitution in LS

[isls1a] insertSinLS(S,[S']) = [S"] **when**
S simples S' = S"

[isls1b] insertSinLS(S, [S',S", LS*]) = LS' **when**
S simples S' = S"',
insertSinLS(S,[S", LS*]) = LS,
conc([S"'],LS) = LS'

[isls1c] insertSinLS(S,[]) = []

%% Answer Verify Function

[vls1a] verify-LS(LS,TS) = fail **when**
LS= []

[vls1b] verify-LS(LS,TS) = yes **when**
LS=[{}]

[vls1c] verify-LS(LS,TS) = yes **when**
LS!=[], LS != [{}], TS={}

[vls1d] verify-LS(LS,TS) = LS' **when**
LS!=[], LS != [{}], TS!={},
find(TS,LS) = LS'

[f1a] find(TS,[]) = []

[f1b] find(TS,[S,LS*]) = LS" **when**
findS(TS,S,{}) = S',
find(TS,[LS*]) = LS',
conc([S'],LS') = LS"

[fs1a] findS(TS, {}, S) = S

[fs1b] findS({T*, T', T*'} , {(V |- > T),S*}, S) = S' **when**
T' != V,
findS({T*, T', T*'}, {S*},S)= S'

[fs1c] findS({T*, T', T*'}, {(V |- > T),S*}, S) = S"' **when**
T' = V,
S simples {(V |- > T)} = S",
findS({T*, T', T*'},{S*},S") = S"'

[lo1a] listvar(?-T.,TS) = TS" **when**
var(T) = TS',
TS uniao TS'= TS"

[lo1b] listvar(?-T, T+.,TS) = TS"' **when**
var(T) = TS',
listvar(?- T+.,TS') = TS",
TS uniao TS" = TS"'

[v1a] var(A(T)) = verify-var(T;{})

[v1b] var(A(T, T+)) = verify-var(T, T+;{})

[v1c] var(V) = {V}

[v1d] var(I) = {}

[v1e] var(A) = {}

[vv1a] verify-var(T;TS) = TS **when** isVar(T) = false

[vv1b] verify-var(T;TS) = TS' **when** isVar(T) = true,
TS uniao {T} = TS'

[vv1c] verify-var(T, T+;TS) = TS" **when**
verify-var(T;TS) = TS',
verify-var(T+;TS')= TS"

%% Auxiliary Function

[ho1a] headO(?-T.) = T

[ho1b] headO(?-T, T+.) = T

[tc1a] tailC(T.) = []

[tc1b] tailC(T:- T+.) = T+

[ap1a] append(T+' ;T+) = T+ **when** T+' =[], T+ != []

[ap1b] append(T+';T+) = T+' **when** T+ =[], T+'!=[]

[ap1c] append(T+; T+') = T+, T+' **when** T+ !=[], T+'!=[]

[ap1d] append(T+';T+) = [] **when** T+' =[], T+ =[]

[co1a] conc(LS, LS') = LS **when** LS =[], LS'=[]

[co1b] conc(LS, LS') = LS' **when** LS =[], LS'!=[]

[co1c] conc(LS, LS') = LS **when** LS!=[], LS'=[]

[co1d] conc(LS, LS') = LS' **when** LS=[{}], LS'!=[]

[co1e] conc(LS, LS') = LS **when** LS!=[], LS'=[{}]

[co1f] conc(LS, LS') = LS" **when**
LS !=[], LS != [{}],
LS'!=[], LS'!= [{}],
union-LS(LS, LS') = LS"

[un1s] union-LS([LS*],[LS*']) = [LS*, LS*']

A.2.7 Layout.sdf2

Idem A.1.6.

A.2.8 NormalisationFunction.sdf2

module NormalisationFunction

imports PrologTerms

exports

lexical syntax

"." → ATOM

context-free syntax

"norm" "(" TERM ")" → TERM

A.2.9 NormalisationFunction.eqs

equations

%%-- Transforma todos os termos listas para uma forma normal

[no1]	norm([])	=	[]
[no2]	norm([T])	=	..(norm(T),[])
[no3]	norm([T,T+])	=	..(norm(T), norm([T+]))
[no4]	norm([T|L])	=	..(norm(T), norm(L))
[no5]	norm([T,T+|L])	=	..(norm(T),norm([T+|L]))
[no6]	norm([T|V])	=	..(norm(T),V)
[no7]	norm([T,T+|V])	=	..(norm(T),norm([T+|V]))
[no8]	norm(T A T')	=	A(norm(T),norm(T'))
[no9]	norm(A(T))	=	A(norm(T))
[no10]	norm(A(T+,T+'))	=	A(T+", T+"') **when** norm(A(T+)) = T+", norm(A(T+')) = T+"'
[no]	norm(T)	=	T

A.2.10 PrologFunctions.sdf2

module PrologFunctions

imports PrologProgram NormalisationFunction Booleans-Prolog

exports

context-free syntax

"isVar" "(" TERM ")"	→	BOOL
"isInteger" "(" TERM ")"	→	BOOL
"isEmpty" "(" LIST ")"	→	BOOL
VARIABLE "contem" TERM	→	BOOL
"isBodyTerm" "(" TERM "," CLAUSE ")"	→	BOOL
"head" "(" CLAUSE ")"	→	TERM

hiddens

context-free syntax

VARIABLE "contemh" TERM → BOOL

A.2.11 PrologFunctions.eqs

equations

[iv1] isVar(V) = true
[iv] isVar(T) = false

[ii1] isInteger(I) = true
[ii] isInteger(T) = false
[ie1] isEmpty([]) = true
[ie] isEmpty(L) = false
[oc1] V contem T = V contemh norm(T)
[oh1] V contemh A(T*, V, T*') = true
[oh2] V contemh A(T*, T, T*') = true **when** V contemh T = true
[oh] V contemh T = false

[ib1] isBodyTerm(T, T' :- T*, T, T*' .) = true
[ib] isBodyTerm(T, C) = false

[p1] [T+ | []] = [T+]

[h1] head(T .) = T
[h2] head(T :- T+ .) = T

A.2.12 PrologProgram.sdf2

module PrologProgram

imports PrologTerms

exports

sorts PROGRAM CLAUSE OBJECTIVE

context-free syntax

TERM "." → CLAUSE
TERM ":-" TERMLIST "." → CLAUSE
"?-" TERMLIST "." → OBJECTIVE
"?-" → OBJECTIVE
CLAUSE* OBJECTIVE → PROGRAM

variables

"C"[\']* → CLAUSE
"C*"[\']* → CLAUSE*
"P"[\']* → PROGRAM
"O"[\']* → OBJECTIVE

A.2.13 PrologTerms.sdf2

module PrologTerms

imports Layout

exports

sorts TERM TERMLIST INTEGER VARIABLE ATOM OPSYM LIST LIST1

lexical syntax

[0-9]+ → INTEGER
[\-][0-9]+ → INTEGER
[A-Z_][a-zA-Z0-9_]* → VARIABLE
[a-z][a-zA-Z0-9_]* → ATOM
"!" → ATOM
" ' " ~[\ \' \ n]+ " ' " → ATOM
OPSYM+ → ATOM
"+" → OPSYM
"\-" → OPSYM
"*" → OPSYM
"/" → OPSYM
">" → OPSYM
"<" → OPSYM
"=" → OPSYM

context-free syntax

INTEGER → TERM
VARIABLE → TERM
ATOM → TERM
ATOM "(" TERMLIST ")" → TERM
TERM ATOM TERM → TERM
LIST → TERM

LIST → LIST1
"[" "]" → LIST
"[" TERMLIST "]" → LIST
"[" TERMLIST "|" VARIABLE "]" → LIST
"[" TERMLIST "|" LIST1 "]" → LIST

{TERM ","}+ → TERMLIST

"(" TERM ")" → TERM **{bracket}**

"(" TERMLIST ")" → TERMLIST **{bracket}**

context-free restrictions

INTEGER −/− [0-9]

VARIABLE −/− [A-Z_]

ATOM −/− [a-z]

variables

"V"[\']* → VARIABLE

"I"[\']* → INTEGER

"T"[\']* → TERM

"T+"[\']* → {TERM ","}+

"T*"[\']* → {TERM ","}*

"L"[\']* → LIST

"A"[\']* → ATOM

A.2.14 Substitution.sdf2

module Substitution

imports PrologFunctions

exports

sorts SUBS SUB

context-free syntax

"(" VARIABLE "|− >" TERM ")" → SUB

"{" {SUB ,"}"*"}" → SUBS

"fail" → SUBS

"yes" → SUBS

SUBS "inteligente" SUBS → SUBS **{left}**

SUBS "simples" SUBS → SUBS **{left}**

"isMember" "(" VARIABLE "," SUBS ")" → BOOL

"getTerm" "(" VARIABLE "," SUBS ")" → TERM

CLAUSE "[c" SUBS "]" → CLAUSE

TERMLIST "[tl" SUBS "]" → TERMLIST

TERM "[t" SUBS "]" → TERM

variables

"S"[\']* → SUBS

"S*"[\']* → {SUB ","}*

A.2.15 Substitution.eqs

equations

[sj1]	fail inteligente S	=	fail
[sj2]	S inteligente fail	=	fail
[sj3]	{(V |- > T),S*} inteligente S	=	{S*} inteligente S **when** isMember(V, S) = true, getTerm(V, S) = T
[sj4]	{(V |- > T), S*} inteligente S	=	fail **when** isMember(V,S) = true, getTerm(V, S) != T
[sj5]	{(V |- > T), S*} inteligente S	=	{(V |- > T[t S]), S*'} **when** isMember(V,S) = false, {S*} inteligente S = {S*'}
[sj6]	{} inteligente S	=	S
[sj7]	S inteligente yes	=	S
[ij1]	S simples fail	=	S
[ij2]	fail simples S	=	S
[ij3]	{S*} simples {S*'}	=	{S*, S*'}
[m]	isMember(V, S)	=	false
[m1]	isMember(V, {S*, (V' |- > T), S*'})	=	true **when** V = V'
[g]	getTerm(V, S)	=	V
[g1]	getTerm(V, {S*, (V' |- > T), S*'})	=	T **when** V = V'
[cl1]	T . [c S]	=	T[t S].
[cl2]	T:- T+ . [c S]	=	T [t S] :- T+ [tl S].
[tl1]	T[tl S]	=	T' **when** T[t S] = T'
[tl2]	T+, T+'[tl S]	=	T+", T+"' **when** T+[tl S] = T+", T+' [tl S] = T+"'
[at1]	A[t S]	=	A
[at2]	A(T+)[t S]	=	A(T+[tl S])
[at3]	T A(T'[t S])	=	T[t S] A(T'[t S])
[va1]	V[t S]	=	getTerm(V, S)

[in1] I[t S] = I

[li1] [] [t S] = []
[li2] [T+] [t S] = [T+[tl S]]
[li3] [T+ | L][t S] = [T+[tl S]| L'] **when** L[t S] = L'
[li4] [T+ | V][t S] = [T+[tl S]|V'] **when** V[t S] = V'
[li5] [T+ | V][t S] = [T+[tl S]|L] **when** V[t S] = L

A.2.16 TermSets.sdf2

module TermSets

imports Substitution

exports

sorts TERM-SET

context-free syntax

"{" {TERM ","}* "}" → TERM-SET
TERM-SET "uniao" TERM-SET → TERM-SET **{assoc}**
TERM-SET "intersecao" TERM-SET → TERM-SET **{assoc}**
TERM-SET "diferenca" TERM-SET → TERM-SET
TERM "pertence" TERM-SET → BOOL
TERM-SET "[ts" SUBS "]" → TERM-SET

"(" TERM-SET ")" → TERM-SET **{bracket}**

variables

"TS"[\']* → TERM-SET

A.2.17 TermSets.eqs

equations

[se1] {T*, T, T*', T, T*"} = {T*, T, T*',T*"}

[un1] {T*} uniao {T*'} = {T*, T*'}

[is1] {T*, T, T*'} intersecao {T*", T, T*"'} = TS' **when**
T = T',
{T*, T*'} intersecao {T*", T*"'} = TS,
{T} uniao TS = TS'

[is] TS intersecao TS' = {}

[di1] {T*, T, T*'} diferenca {T*", T, T*"'} = {T*, T*'} diferenca {T*", T, T*"'}

[di] TS diferenca TS' = TS

[me] T pertence {T*} = false
[me1] T pertence {T*, T', T*'} = true **when** T = T'

[ts1] {}[ts S] = {}
[ts2] {T} [ts S] = {T[t S]}
[ts3] {T+, T+'}[ts S] = {T+", T+"'} **when**
{T+}[ts S] = {T+"}, {T+'}[ts S] = {T+"'}

A.2.18 Unification.sdf2

module Unification

imports Equations

exports

context-free syntax

"mgu" "(" EQS ")" → SUBS
"mgu-nv" "(" TERM "," TERM ")" → SUBS

hiddens

context-free syntax

"mgu" "(" EQS "," SUBS ")" → SUBS

A.2.19 Unification.eqs

equations

[mgu1] mgu(EQS) = mgu(EQS,{})

[m1] mgu({T equivalente T', E*}, S) = mgu({E*}[es S'], S inteligente S') **when**
isVar(T) = false, isVar(T') = false,
mgu-nv(T, T') = S', S' != fail
[m2] mgu({T equivalente T', E*}, S) = fail **when**
isVar(T) = false, isVar(T') = false,
mgu-nv(T, T') = S', S' = fail
[m3] mgu({V equivalente V, E*}, S) = mgu({E*}, S)
[m4] mgu({T equivalente V, E*},S) = mgu({V equivalente T, E*}, S) **when**
isVar(T) = false
[m5] mgu({V equivalente T, E*},S) = mgu({E*}[es{(V |- > T)}],
S inteligente {(V |- > T)}) **when**
V != T, V contem T = false

[m6]	mgu({V equivalente T, E*}, S)	=	fail **when**
			V != T, V contem T = true
[m7]	mgu(EQS, S)	=	S
[nv1]	mgu-nv(I, I)	=	{}
[nv2]	mgu-nv(A, A)	=	{}
[nv3]	mgu-nv(A(T), A(T'))	=	mgu({T equivalente T'})
[nv4]	mgu-nv(A(T, T+), A(T', T+'))	=	S inteligente mgu-nv(A(T+[tl S]),
			A(T+' [tl S])) **when**
			mgu({T equivalente T'}) = S
[nv5]	mgu-nv(T A T', T" A T"')	=	mgu({T equivalente T", T' equivalente T"'})
[nv6]	mgu-nv([],[])	=	{}
[nv7]	mgu-nv(L, L')	=	mgu-nv(norm(L), norm(L')) **when**
			isEmpty(L) = false, isEmpty(L') = false
[nv]	mgu-nv(T, T')	=	fail

A.2.20 VariantClause.sdf2

module VariantClause

imports Unification TermSets

exports

context-free syntax

"variant-P" "(" PROGRAM "," TERM-SET ")" → PROGRAM

"variant" "(" PROGRAM "," TERM-SET ")" → PROGRAM

hiddens

context-free syntax

"newvar" "(" VARIABLE "," TERM-SET ")" → TERM

"prime" "(" VARIABLE ")" → VARIABLE

"varset" "(" CLAUSE ")" → TERM-SET

"varset-O" "(" OBJECTIVE ")" → TERM-SET

"tvarset" "(" {TERM ","}+ ")" → TERM-SET

"varsub" "(" TERM-SET "," TERM-SET ")" → SUBS

"add-C" "(" CLAUSE "," PROGRAM ")" → PROGRAM

variables

"c" → CHAR+

A.2.21 VariantClause.eqs

equations

[va] variant-P(C* O, TS) = P' **when**
varset-O(O) = TS',
variant(C* O, TS') = P'

[va1] variant(C C' C* O, TS) = P' **when**
varset(C) = TS',
varset(C') = TS",
TS'uniao TS" = TS"',
C[c varsub(TS uniao TS"', TS uniao TS"')] = C",
varset(C") = TS"",
TS uniao TS"" = TS""',
variant(C' C* O, TS""') = P,
add-C(C", P) = P'

[va2] variant(C O, TS) = C" O **when**
varset(C) = TS',
varset-O(O) = TS",
TS'uniao TS" = TS"',
C[c varsub(TS uniao TS"', TS uniao TS"')] = C"

[sv1] varset(T.) = tvarset(T)

[sv2] varset(T :- T+.) = tvarset(T) uniao tvarset(T+)

[sv3] varset-O(?- T.) = tvarset(T)

[sv4] varset-O(?- T, T+.) = tvarset(T) uniao tvarset(T+)

[st1] tvarset(I) = {}

[st2] tvarset(V) = {V}

[st3] tvarset(A) = {}

[st4] tvarset(A(T)) = tvarset(T)

[st5] tvarset(A(T+)) = tvarset(T+)

[st6] tvarset(T A T') = tvarset(T) uniao tvarset(T')

[st7] tvarset(L) = {} **when** isEmpty(L) = true

[st8] tvarset(L) = tvarset(norm(L)) **when** isEmpty(L) = false

[st9] tvarset(T+, T+') = tvarset(T+) uniao tvarset(T+')

[vs1] varsub({}, TS) = {}

[vs2] varsub({V, T*}, TS) = {(V |– > V')} inteligente varsub({T*},TS) **when** newvar(V, TS) = V'

[nv1] newvar(V, TS) = prime(V) **when** prime(V) pertence TS = false

[nv2] newvar(V, TS) = newvar(prime(V), TS) **when** prime(V) pertence TS = true

[pr1] prime(variable(c)) = variable(c "_" "n")

[add1] add-C(C, C* O) = C C* O

A.3 Capítulo 6 - Linguagem ParaLog

A.3.1 Booleans.sdf2

module Booleans

imports Layout

exports

sorts BOOL

context-free syntax

"true" → BOOL

"false" → BOOL

BOOL "|" BOOL → BOOL {**left**}

BOOL "∧" BOOL → BOOL {**left**}

"neg" "(" BOOL ")" → BOOL

"(" BOOL ")" → BOOL {**bracket**}

variables

"B"[\ ']* → BOOL

context-free priorities

BOOL "∧" BOOL → BOOL >

BOOL "|" BOOL → BOOL

A.3.2 Booleans.eqs

equations

[o1] true | B = true

[o2] false | B = B

[a1] true ∧ B = B

[a2] false ∧ B = false

[n1] neg(false) = true

[n2] neg(true) = false

A.3.3 Closed.sdf2

module Closed

imports VariantClause

exports

sorts CLAUSE-UNION

context-free syntax

"closed" "(" PROGRAM ")" → PROGRAM

"lub" "(" CONST-ANOT "," CONST-ANOT ")" → CONST-ANOT

hiddens

context-free syntax

"belong-lub" "(" CONST-ANOT ","
"{" CONST-ANOT "," CONST-ANOT "}" ")" → BOOL

"closed-P" "(" CLAUSE-UNION ")" → CLAUSE-UNION

"closed-C" "(" CLAUSE "," CLAUSE ")" → CLAUSE

"closed-F" "(" FORMULA "," FORMULA ")" → FORMULA

"closed-UC" "(" CLAUSE "," CLAUSE-UNION ")" → CLAUSE-UNION

"mutant" "(" CLAUSE "," CLAUSE-UNION ")" → BOOL

"mutant-T" "(" TERM "," TERM ")" → BOOL

"mutant-T+" "(" TERMLIST ";" TERMLIST ";" BOOL ")" → BOOL

"mutant-F" "(" FORMULA "," FORMULA ")" → BOOL

"mutant-F+" "(" FORMBODY "," FORMBODY "," BOOL ")" → BOOL

"mutant-C" "(" CLAUSE "," CLAUSE ")" → BOOL

"[" "]" → FORMULA

"fail" → FORMULA

"fail" → CLAUSE

"fail" → CLAUSE-UNION

CLAUSE* → CLAUSE-UNION

"body-C" "(" CLAUSE "," CLAUSE ")" → FORMBODY

"conc-C" "(" CLAUSE-UNION "," CLAUSE-UNION ")" → CLAUSE-UNION

"comp-C" "(" CLAUSE-UNION "," CLAUSE-UNION ")" → CLAUSE-UNION

"insere-C" "(" CLAUSE-UNION "," CLAUSE-UNION ")" → CLAUSE-UNION

"insert-fail" "(" BOOL ")" → CLAUSE

"verify-CP" "(" CLAUSE "," CLAUSE-UNION
"," CLAUSE-UNION ")" → CLAUSE-UNION

"verify-mutant" "(" CLAUSE-UNION "," CLAUSE-UNION
"," BOOL ")" → CLAUSE-UNION

"isEmpty-ClauseUnion" "(" CLAUSE-UNION ")" → BOOL

"variablesub" "(" TERM-SET "," TERM-SET ")" → SUBS

"newTS" "(" TERM-SET ")" → TERM-SET

"replace-C" "(" CLAUSE ")" → CLAUSE

"newvar" "(" VARIABLE ")" → VARIABLE

"underscore?" "(" TERM ")" → BOOL

"verify-variable" "(" TERM ")" → TERM

"equal-var-name" "(" TERM "," TERM ")" → BOOL

variables

"x"[\']* → CHAR

"x+"[\']* → CHAR+

A.3.4 Closed.eqs

equations

%% lub - least upper bound

[su1a] lub(*, *) = *

[su1b] lub(lt, lt) = lt

[su1c] lub(lf, lf) = lf

[su1d] lub(t, t) = t

[su1e] lub(f, f) = f

[su1f] lub(tf, tf) = tf

[su2a] lub(*, lt) = lt

[su2b] lub(*, lf) = lf

[su2c] lub(*, t) = t

[su2d] lub(*, f) = f

[su2e] lub(*, tf) = tf

[su3a] lub(lt, t) = t

[su3b] lub(lt, tf) = tf

[su3c] lub(lt, lf) = tf

[su3d] lub(lt, f) = tf

[su4a] lub(lf, f) = f

[su4b] lub(lf, tf) = tf

[su4c] lub(lf, t) = tf

[su5a] lub(t, tf) = tf

[su5b] lub(f, tf) = tf

[su6a] lub(t, f) = tf

%% Inverse of rule [su2...]

[su21a] lub(lt, *) = lt

[su21b] lub(lf, *) = lf

[su21c] lub(t, *) = t

[su21d] lub(f, *) = f

[su21e] lub(tf, *) = tf

%% Inverse of rule [su3...]

[su31a] lub(t, lt) = t

[su31b] lub(tf, lt) = tf

[su31c] lub(lf, lt) = tf

[su31d] lub(f, lt) = tf

%% Inverse of rule [su4...]

[su41a] lub(f, lf) = f

[su41b] lub(tf, lf) = tf

[su41c] lub(t, lf) = tf

%% Inverse of rule [su5...]

[su51a] lub(tf, t) = tf

[su51b] lub(tf, f) = tf

%% Inverse of rule [su6...]

[su61a] lub(f, t) = tf

%% Verify if a constant anotacional belong a lub

[bl1] belong-lub(CA, {CA', CA"}) = true **when** CA = CA'

[bl2] belong-lub(CA, {CA', CA"}) = true **when** CA = CA"

[bl3] belong-lub(CA, {CA', CA"}) = false **when** CA != CA', CA != CA"

%% Closed

[cl1a] closed(C O) = C O

[cl1b] closed(C C' C* O) = C' C* O **when** C = fail

[c1b] closed(C C' C* O) = C*"' O **when**
C != fail,
closed-P(C C' C*) = C" C*',
verify-CP(C", C*', C C' C*) = C"' C*",
closed(C"' C*" O) = C*"' O

[vcp1a] verify-CP(C, C*, C*') = C C* C*' **when** C = fail

[vcp1b] verify-CP(C, C*, C*') = C" C*" **when**
C != fail,
verify-mutant(C C*, C*', true) = C" C*"

[cp1a] closed-P(C C') = closed-UC(C, C')

[cp1b] closed-P(C C' C" C*) = C*"' **when**
closed-UC(C, C' C" C*) = C*',
closed-P(C' C" C*) = C*",
conc-C(C*', C*") = C*"'

[uc1a] closed-UC(C, C' C" C*) = C*" **when**
closed-C(C, C') = C"',
closed-UC(C, C" C*) = C*',
conc-C(C"', C*') = C*"

[uc1b] closed-UC(C, C') = closed-C(C, C')

[cc1a] closed-C(C, C') = fail **when**
head(C) = F,
head(C') = F',
closed-F(F, F') = F", F" = fail

[cc1b] closed-C(C, C') = F". **when**
head(C) = F,
head(C') = F',
closed-F(F, F') = F", F" != fail,
body-C(C, C') = F+, F+ = []

[cc1c] closed-C(C, C') = F" <-- F+. **when**
head(C) = F,
head(C') = F',
closed-F(F, F') = F", F" != fail,
body-C(C, C') = F+, F+ !=[]

[cf1a] closed-F(A(T+):CA, A'(T+'):CA') = fail **when**
lub(CA, CA') = CA",
belong-lub(CA", {CA, CA'}) = true

[cf1b] closed-F(A(T+):CA, A'(T+'):CA') = fail when
lub(CA, CA') = CA",
belong-lub(CA", {CA, CA'}) = false,
mgu-f(A(T+):CA, A'(T+'):CA') = S, S = fail

[cf1c] closed-F(A(T+):CA, A'(T+'):CA') = F **when**
lub(CA, CA') = CA",
belong-lub(CA", {CA, CA'}) = false,
mgu-f(A(T+):CA, A'(T+'):CA') = S, S != fail,
A(T+):CA" [f S] = F

%% Create new clausule

[rc1] replace-C(C) = C' **when**
varset(C) = TS,
C[c variablesub(TS,TS)] = C'

[vs1a] variablesub({},TS) = {}

[vs1b] variablesub({V},{V'}) = {(V' |- > V")} **when**
newvar(V)= V"

[vs1c] variablesub({V, T*},{V', V", T*'}) =
{(V' |- > V"')} inteligente variablesub(newTS({T*}), {V", T*'}) **when**
newvar(V) = V"'

[nts1a] newTS({V, T*}) = {V', T*'} **when**
newvar(V) = V',
newTS({T*}) = {T*'}

[nts1b] newTS({}) = {}

[nv1a] newvar(variable(c)) = variable(c "_" "N")

[uv1a] underscore?(variable(x)) = false

[uv1b] underscore?(variable("_")) = true

[vv1a] verify-variable(variable(x x+)) = variable(x x+) **when**
underscore?(variable(x)) = B, B = true

[vv1b] verify-variable(variable(x x+)) = T **when**
underscore?(variable(x))= B, B = false,
verify-variable(variable(x+)) = T

[ev1a] equal-var-name(variable(x), variable(x')) = false

[ev1b] equal-var-name(variable(x), variable(x)) = true

[ev1c] equal-var-name(variable(x x+), variable(x' x+')) = false

[ev1d] equal-var-name(variable(x x+), variable(x x+')) = true ∧ B when
equal-var-name(variable(x+), variable(x+')) = B

[ev1e] equal-var-name(variable(x), variable(x' x+)) = false

[ev1f] equal-var-name(variable(x x+), variable(x')) = false

%% Mutant

[mu1a] mutant-T(T, T') = B" **when**
isVar(T) = B, B = true,
isVar(T') = B', B'= true,
verify-variable(T) = T",
verify-variable(T')= T"',
equal-var-name(T", T"') = B"

[mu1b] mutant-T(T, T') = B' **when**
isVar(T) = B, B = true,
isVar(T') = B', B' = false

[mu1c] mutant-T(T, T') = B **when**
isVar(T) = B, B = false

[mu1d] mutant-T(T, T') = true **when**
isVar(T) = B, B = false,
isVar(T') = B', B'= false,
mgu-nv(T, T') = {}

[mu1e] mutant-T(T, T') = false **when**
isVar(T) = B, B = false,
isVar(T') = B', B'= false,
mgu-nv(T, T') != {}

[mut1a] mutant-T+(T ; T'; B) = B ∧ B' **when**
mutant-T(T, T') = B'

[mut1b] mutant-T+(T, T+ ; T', T+' ; B) = B ∧ B" **when**
mutant-T(T, T') = B',
mutant-T+(T+; T+'; B ∧ B') = B"

[mut1c] mutant-T+(T ; T',T+; B) = false

[mut1d] mutant-T+(T, T+; T'; B) = false

[muf1a] mutant-F(A(T+):CA, A'(T+'):CA') = false **when**
mgu-nv(A, A') != {}

[muf1b] mutant-F(A(T+):CA, A'(T+'):CA') = false **when**
mgu-nv(A, A') = {},
CA != CA'

[muf1c] mutant-F(A(T+):CA, A'(T+'):CA') = B **when**
mgu-nv(A, A') = {},
CA = CA',
mutant-T+(T+; T+'; true) = B

[muff1a] mutant-F+(F & F+, F' & F+', B) = B ∧ B" **when**
mutant-F(F, F') = B',
mutant-F+(F+, F+', B ∧ B') = B"

[muff1b] mutant-F+(F, F', B) = B ∧ B' **when**
mutant-F(F, F') = B'

[muff1c] mutant-F+(F, F' & F+, B) = false

[muff1d] mutant-F+(F & F+, F', B) = false

[muc1a] mutant-C(F <-- F+., F'.) = false

[muc1b] mutant-C(F. , F' <-- F+.) = false

[muc1c] mutant-C(F., F'.) = B **when**
replace-C(F.) = F".,
replace-C(F'.) = F"'.,
mutant-F(F", F"') = B

[muc1d] mutant-C(F <-- F+., F' <-- F+'.) = B ∧ B' when
replace-C(F <-- F+.) = F" <-- F+".,
replace-C(F' <-- F+'.) = F"' <-- F+"'.,
mutant-F(F", F"') = B,
mutant-F+(F+", F+"', true) = B'

[mut1a] mutant(C, C') = B **when**
mutant-C(C, C') = B

[mut1b] mutant(C, C' C" C*) = B | B' when
mutant-C(C, C') = B,
mutant(C, C" C*) = B'

[vmut1c] verify-mutant(C C*, C*', B) = C*" **when**
mutant(C, C*') = B', B' = true,
verify-mutant(C*, C*', B) = C*"

[vmut1d] verify-mutant(C C*, C*', B) = C*"' **when**
mutant(C, C*') = B', B' = false,
insere-C(C, C*') = C*",
verify-mutant(C*, C*", B') = C*"'

[vmut1e] verify-mutant(, C*, B) = C* **when** B = false

[vmut1f] verify-mutant(, C*, B) = C*" **when**
B = true,
insert-fail(B) = C,
insere-C(C, C*) = C*"

[if1a] insert-fail(B) = fail

%% Verify if a clause-joint is empty or not empty

[vcc1a] isEmpty-ClauseUnion(C C*) = false

[vcc1b] isEmpty-ClauseUnion() = true

%% Withdraw body-clause

[b1] body-C(F., F'.) = []

[b2] body-C(F <-- F+., F'.) = F+

[b3] body-C(F. , F' <-- F+.) = F+

[b4] body-C(F <-- F+. , F' <-- F+'.) = F+ & F+'

%% Insert clause

[ic1a] insere-C(C, C*) = C C*

[ic1b] insere-C(C*, C) = C* C

[ic1c] insere-C(C*, C*') = C* C*'

[ic1d] insere-C(fail, C*) = fail C*

%% Together clause

[cc1] conc-C(C*, C) = C* C **when** C* != fail, C != fail

[cc2] conc-C(C*, C) = C* **when** C = fail, C* != fail

[cc3] conc-C(C*, C) = C **when** C != fail, C* = fail

[cc4] conc-C(C*, C) = fail **when** C = fail, C* = fail

[cc5] conc-C(C*, C*') = C* C*' **when** C* != fail, C*' != fail

[cc6] conc-C(C*, C*') = C*' **when** C* = fail, C*' != fail

[cc7] conc-C(C*, C*') = C* **when** C* != fail, C*' = fail

[cc8] conc-C(C*, C*') = fail **when** C* = fail, C*' = fail

[cc9] conc-C(C, C*) = C C* **when** C != fail, C* != fail

[cc10] conc-C(C, C*) = C* **when** C = fail, C* != fail

[cc11] conc-C(C, C*) = C **when** C != fail, C* = fail

[cc12] conc-C(C, C*) = fail **when** C = fail, C* = fail

[cc13] conc-C(C, C') = C **when** C != fail , C' = fail

A.3.5 EliminationOfNegation.sdf2

module EliminationOfNegation

imports ParaLogSyntax

exports

context-free syntax

"∼" "(" CONST-ANOT ")"	→	CONST-ANOT
"verify-form" "(" FORMULA ")"	→	FORMULA
"verify-body" "(" FORMBODY ")"	→	FORMBODY
"elimination" "(" PROGRAM ")"	→	PROGRAM
"elimination-C" "(" CLAUSE ")"	→	CLAUSE
"elimination-O" "(" OBJECTIVE ")"	→	FORMBODY

A.3.6 EliminationOfNegation.eqs

equations

%% Unitary Operator: ∼ | CONST-ANT | → | CONST-ANOT |

[op1] ∼ (t) = f

[op2] ∼ (f) = t

[op3] ∼ (lf) = lt

[op4] ∼ (lt) = lf

[op5] ∼ (*) = *

[op6] ∼ (tf) = tf

%% Elimination of *not* Operator

[e1] elimination(O) = <-- F+. **when** elimination-O(O) = F+

[e2] elimination(C C* O) = C' C*' O' **when**
elimination-C(C) = C',
elimination(C* O) = C*' O'

[ec1] elimination-C (F.) = F'. **when** verify-form(F) = F'

[ec2] elimination-C (F <-- F+.) = F' <-- F+'. when
verify-form(F) = F',
verify-body(F+)= F+'

[eo1] elimination-O (<-- F.) = F' **when** verify-form(F) = F'

[eo2] elimination-O (<-- F & F+.) = F' & F+' **when**
verify-form(F) = F',
verify-body(F+)= F+'

[vf1] verify-form(A(T+):CA) = A(T+): CA

[vf2] verify-form(not A(T+):CA) = A(T+):CA' **when**
∼(CA) = CA'

[vb1] verify-body(F) = verify-form(F)

[vb2] verify-body(F & F+) = F' & F+' **when**
verify-form(F) = F',
verify-body(F+) = F+'

A.3.7 Equations.sdf2

Idem A.2.3.

A.3.8 Equations.eqs

Idem A.2.4.

A.3.9 EvalParaLog.sdf2

module EvalParaLog

imports Closed EliminationOfNegation

exports

sorts LISTSUBS LISTSC

context-free syntax

"eval-K" "(" CLAUSE* "," SUBS "," PROGRAM "," BOOL ")"	→	LISTSUBS
"eval" "(" PROGRAM ")"	→	LISTSUBS
SUBS ":" CONST-ANOT	→	LISTSC
CONST-ANOT	→	LISTSC
{SUBS ","}*	→	LISTSC
"[" {LISTSC ","}* "]"	→	LISTSUBS
LISTSC	→	LISTSUBS
"verify-LS" "(" LISTSUBS "," TERM-SET ")"	→	LISTSUBS
"union-LS" "(" LISTSUBS "," LISTSUBS ")"	→	LISTSUBS
"insertSinLS" "(" SUBS "," LISTSUBS ")"	→	LISTSUBS
"conc" "(" LISTSUBS "," LISTSUBS ")"	→	LISTSUBS
"[" "]"	→	FORMULA
"append" "(" FORMBODY ";" FORMBODY ")"	→	FORMBODY
"headO" "(" OBJECTIVE ")"	→	FORMULA
"tailC" "(" CLAUSE ")"	→	FORMBODY
"var" "(" FORMULA ")"	→	TERM-SET
"verify-var" "(" TERMLIST ";" TERM-SET ")"	→	TERM-SET

"listvar" "(" OBJECTIVE "," TERM-SET ")" → TERM-SET

"delete-{}" "(" LISTSUBS ")" → LISTSUBS

"find" "(" TERM-SET "," LISTSUBS ")" → LISTSUBS

"findS" "(" TERM-SET "," SUBS "," SUBS ")" → SUBS

"find-CA" "(" CLAUSE ")" → CONST-ANOT

"insertCAinLS" "(" LISTSUBS "," CONST-ANOT ")" → CONST-ANOT

variables

"LS"[\']* → LISTSUBS

"LS*"[\']* → {LISTSC ","}*

"x"[\']* → CHAR

"x+"[\']* → CHAR+

A.3.10 EvalParaLog.eqs

equations

%% Evalution Function

[ev1] eval(C* O) = LS' **when**
listvar(O,{}) = TS,
elimination(C* O) = C*' O',
closed(C*' O') = C*" O",
variant-P(C*" O", {}) = C*"' O"',
eval-K(C*"', {}, C*"' O"' "," false)= LS,
verify-LS(LS,TS) = LS'

%% Evalution Auxiliary Function

[evk1a] eval-K(C*, S, <-- F., B) = [] **when** F != []

[evk1b] eval-K(C*, S, <-- F & F+., B) = [] **when** F != []

[evk2a] eval-K(C*,S, <-- []., B) = [S]

[evk2b] eval-K(C*, S, C' C*' <-- []., B) = [S]

[evk3a] eval-K(C*,S,C' C*' O, B) = LS **when**
find-CA(C') = CA,
headO(O) = F,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" != CA,
eval-K(C*,S, C*' O, B) = LS

[evk3b] eval-K(C*,S,C' C*' O, B) = LS bf when
head(C') = F',
headO(O) = F,
mgu-f(F',F) = S', S'= fail,
eval-K(C*,S, C*' O, B) = LS

[evk3c1] eval-K(C*,S, F'. C*' <-- F., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" != CA',
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
eval-K(C*,{},C* <-- []., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F., true) = LS"',
conc(LS", LS"') = LS""

[evk3c2] eval-K(C*,S, F'. C*' <-- F., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" = CA', B = false,
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
eval-K(C*,{},C* <-- []., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F., B) = LS"',
conc(LS", LS"') = LS""

[evk3d1] eval-K(C*,S, F' <-- F+. C*' <-- F., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" != CA',
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
F+[fb S"] = F+',
eval-K(C*,{},C* <-- F+'., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F., true) = LS"',
conc(LS", LS"') = LS""

[evk3d2] eval-K(C*,S, F' <-- F+. C*' <-- F., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" = CA', B = false,
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
F+[fb S"] = F+',
eval-K(C*,{},C* <-- F+'., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F., B) = LS"',
conc(LS", LS"') = LS""

[evk3e1] eval-K(C*,S, F'. C*' <-- F & F+., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" != CA',
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
F+[fb S"] = F+',
eval-K(C*,{},C* <--F+'., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F & F+., true) = LS"',
conc(LS", LS"') = LS""

[evk3e2] eval-K(C*,S, F'. C*' <-- F & F+., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" = CA', B = false,
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
F+[fb S"] = F+',
eval-K(C*,{},C* <--F+'., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",

eval-K(C*,S,C*' <-- F & F+., B) = LS"',
conc(LS", LS"') = LS""

[evk3f1] eval-K(C*,S, F' <-- F+'. C*' <-- F & F+., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" != CA',
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
append(F+'; F+) = F+",
F+"[fb S"] = F+"',
eval-K(C*,{},C* <-- F+"'., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F & F+., true) = LS"',
conc(LS", LS"') = LS""

[evk3f2] eval-K(C*,S, F' <-- F+'. C*' <-- F & F+., B) = LS"" **when**
find-CA(F'.) = CA,
find-CA(F.) = CA',
lub(CA, CA') = CA",
CA" = CA, CA" = CA', B = false,
mgu-f(F',F) = S', S' != fail,
S simples S' = S",
append(F+'; F+) = F+",
F+"[fb S"] = F+"',
eval-K(C*,{},C* <-- F+"'., B) = LS,
insertSinLS(S', LS) = LS',
insertCAinLS(LS', CA") = LS",
eval-K(C*,S,C*' <-- F & F+., B) = LS"',
conc(LS", LS"') = LS""

%% Insert S in LS

[isls1a] insertSinLS(S,[S']) = [S"] **when**
S simples S' = S"

[isls1b] insertSinLS(S, [S',S", LS*]) = LS' **when**
S simples S' = S"',
insertSinLS(S,[S", LS*]) = LS,
conc([S"'],LS) = LS'

[isls1c] insertSinLS(S, [S':CA, S":CA', LS*]) = LS'**when**
S simples S' = S"',
insertSinLS(S, [S":CA, LS*]) = LS,
conc([S"':CA],LS) = LS'

[isls1d] insertSinLS(S, [S':CA]) = [S":CA] **when**
S simples S' = S"

[isls1e] insertSinLS(S,[]) = []

%% Find CA

[fca1] find-CA(A(T+):CA.) = CA

%% Insert CA in LS

[icals1a] insertCAinLS([S], CA) = [S:CA]

[icals1b] insertCAinLS([S:CA], CA') = [S:CA]

[icals1c] insertCAinLS([S:CA, LS*], CA') = [S:CA, LS*]

[icals1d] insertCAinLS([], CA) = []

%% Answer Verify Function

[vls1a] verify-LS(LS,TS) = [*] **when**
LS= []

[vls1b] verify-LS(LS,TS) = LS' **when**
LS!=[], TS={},
delete-{}(LS) = LS'

[vls1c] verify-LS(LS,TS) = LS' **when**
LS!=[], TS!={},
find(TS,LS) = LS'

[dsv1a] delete-{}([{}:CA]) = [CA]

[dsv1a] delete-{}([{}:CA, {}:CA', LS*]) = LS **when**
delete-([{}:CA', LS*]) = [LS*'],
conc([CA], [LS*']) = LS

[fl1a] find(TS,[]) = []

[fl1b] find(TS,[S:CA,LS*]) = LS" **when**
findS(TS,S,{}) = S',
find(TS,[LS*]) = LS',
conc([S':CA],LS') = LS"

[fs1a] findS(TS, {}, S) = S

[fs1b] findS({T*, T', T*'} , {(V |- > T),S*}, S) = S' **when**
T' != V,
findS({T*, T', T*'}, {S*},S)= S'

[fs1c] findS({T*, T', T*'}, {(V |− > T),S*}, S) = S"' **when**
T' = V,
S simples {(V |− > *T*)} = S",
findS({T*, T', T*'},{S*},S") = S"'

[lo1a] listvar(<-- F.,TS) = TS" **when**
var(F) = TS',
TS uniao TS'= TS"

[lo1b] listvar(<-- F & F+.,TS) = TS"' **when**
var(F) = TS',
listvar(<-- F+.,TS') = TS",
TS uniao TS" = TS"'

[v1a] var(A(T):CA) = verify-var(T;{})

[v1b] var(A(T, T+):CA) = verify-var(T, T+;{})

[vv1a] verify-var(T;TS) = TS **when** isVar(T) = false

[vv1b] verify-var(T;TS) = TS' **when** isVar(T) = true,
TS uniao {T} = TS'

[vv1c] verify-var(T, T+;TS) = TS" **when**
verify-var(T;TS) = TS',
verify-var(T+;TS')= TS"

%% Auxiliary Function

[ho1a] head*O*(<-- F.) = F

[ho1b] headO(<-- F & F+.) = F

[tc1a] tailC(F.) = []

[tc1b] tailC(F <-- F+.) = F+

[ap1a] append([] ;F+) = F+ **when** F+!= []

[ap1b] append(F+';[]) = F+' **when** F+'!=[]

[ap1c] append(F+; F+') = F+ & F+' **when** F+ !=[], F+'!=[]

[ap1d] append([];[]) = []

[co1a] conc([], LS) = LS **when** LS!=[]

[co1b] conc(LS, []) = LS **when** LS!=[]

[co1c] conc(LS, LS') = LS" **when**
LS !=[], LS' != [],
union-LS(LS, LS') = LS"

[co1d] conc([], []) = []

[unls] union-LS([LS*],[LS*']) = [LS*, LS*']

A.3.11 Layout.sdf2

Idem A.1.6.

A.3.12 NormalisationFunction.sdf2

module NormalisationFunction

imports ParaLogSyntax

exports

lexical syntax

"." → ATOM

context-free syntax

"norm" "(" TERM ")" → TERM

A.3.13 NormalisationFunction.eqs

equations

[no1]	norm([])	=	[]
[no2]	norm([T])	=	..(norm(T),[])
[no3]	norm([T,T+])	=	..(norm(T), norm([T+]))
[no4]	norm([T|L])	=	..(norm(T), norm(L))
[no5]	norm([T,T+|L])	=	..(norm(T),norm([T+|L]))
[no6]	norm([T|V])	=	..(norm(T),V)
[no7]	norm([T,T+|V])	=	..(norm(T),norm([T+|V]))
[no8]	norm(A(T))	=	A(norm(T))
[no9]	norm(A(T+,T+'))	=	A(T+", T+"') **when**
			norm(A(T+)) = T+",
			norm(A(T+')) = T+"'
[no]	norm(T)	=	T

A.3.14 ParaLogFunctions.sdf2

module ParaLogFunctions

imports ParaLogSyntax NormalisationFunction Booleans

exports

context-free syntax

"isVar" "(" TERM ")"	→	BOOL
"isInteger" "(" TERM ")"	→	BOOL
"isEmpty" "(" LIST ")"	→	BOOL
VARIABLE "contem" TERM	→	BOOL
head "(" CLAUSE ")"	→	FORMULA

hiddens

context-free syntax

VARIABLE "contemh" TERM → BOOL

A.3.15 ParaLogFunctions.eqs

equations

[iv1] isVar(V) = true

[iv] isVar(T) = false

[ii1] isInteger(I) = true

[ii] isInteger(T) = false

[ie1] isEmpty([]) = true

[ie] isEmpty(L) = false

[oc1] V contem T = V contemh norm(T)

[oh1] V contemh A(T*, V, T*') = true

[oh2] V contemh A(T*, T, T*') = true **when** V contemh T = true

[oh] V contemh T = false

[p1] [T+ | []] = [T+]

[h1] head(F .) = F

[h2] head(F <-- F+ .) = F

A.3.16 ParaLogSyntax.sdf2

module ParaLogSyntax

imports Layout

exports

sorts TERM TERMLIST INTEGER VARIABLE ATOM OPSYM LIST LIST1 CONST-ANOT PROGRAM OBJECTIVE CLAUSE FORMBODY FORMULA

lexical syntax

[0-9]+ → INTEGER

[\-][0-9]+ → INTEGER

"+" → OPSYM

"\-" → OPSYM

"*" → OPSYM

"/" → OPSYM

">" → OPSYM
"<" → OPSYM
"=" → OPSYM

[A-Z_][a-zA-Z0-9_]* → VARIABLE

[a-z][a-zA-Z0-9_]* → ATOM
"!" → ATOM
"'" ~[\ \'\n]+ "'" → ATOM
OPSYM+ → ATOM

"t" → CONST-ANOT
"f" → CONST-ANOT
"lt" → CONST-ANOT
"lf" → CONST-ANOT
"tf" → CONST-ANOT
"*" → CONST-ANOT

context-free syntax

INTEGER → TERM
VARIABLE → TERM
ATOM → TERM
LIST → TERM
ATOM "(" TERMLIST ")" → TERM
LIST → LIST1

"[" "]" → LIST
"[" TERMLIST "]" → LIST
"[" TERMLIST "|" VARIABLE "]" → LIST
"[" TERMLIST "|" LIST1 "]" → LIST

{TERM ","}+ → TERMLIST

ATOM "(" TERMLIST ")" ":" CONST-ANOT → FORMULA
"not" ATOM "(" TERMLIST ")" ":" CONST-ANOT → FORMULA

FORMULA "." → CLAUSE
FORMULA "<--" FORMBODY "." → CLAUSE

"<--" FORMBODY "." → OBJECTIVE

{FORMULA "&"}+ → FORMBODY

CLAUSE* OBJECTIVE → PROGRAM

context-free restrictions

INTEGER -/- [0-9]

VARIABLE -/- [A-Z_]

ATOM -/- [a-z]

variables

"V"[\']* → VARIABLE

"I"[\']* → INTEGER

"T"[\']* → TERM

"T+"[\']* → {TERM ","}+

"T*"[\']* → {TERM ","}*

"L"[\']* → LIST

"A"[\']* → ATOM

"CA"[\']* → CONST-ANOT

"C"[\']* → CLAUSE

"C*"[\']* → CLAUSE*

"P"[\']* → PROGRAM

"O"[\']* → OBJECTIVE

"F"[\']* → FORMULA

"F+"[\']* → {FORMULA "&"}+

"F*"[\']* → {FORMULA "&"}*

A.3.17 Substitution.sdf2

module Substitution

imports ParaLogFunctions

exports

sorts SUBS SUB

context-free syntax

"(" VARIABLE "|- >" TERM ")" → SUB

"{" {SUB ","}*"}" → SUBS

"fail" → SUBS

"yes" → SUBS

SUBS "inteligente" SUBS → SUBS {**left**}

SUBS "simples" SUBS → SUBS {**left**}

"isMember" "(" VARIABLE "," SUBS ")" → BOOL

"getTerm" "(" VARIABLE "," SUBS ")" → TERM

CLAUSE "[c" SUBS "]" → CLAUSE

TERMLIST "[tl" SUBS "]" → TERMLIST

TERM "[t" SUBS "]" → TERM

FORMULA "[f" SUBS "]" → FORMULA

FORMBODY "[fb" SUBS "]" → FORMBODY

"[" "]" → FORMULA

"[" "]" → FORMBODY

variables

"S"[\']* → SUBS

"S*"[\']* → {SUB ","}*

A.3.18 Substitution.eqs

equations

[sj1] fail inteligente S = fail

[sj2] S inteligente fail = fail

[sj3] {(V |− > T),S*} inteligente S = {S*} inteligente S **when** isMember(V, S) = true, getTerm(V, S) = T

[sj4] {(V |− > T), S*} inteligente S = fail **when** isMember(V,S) = true, getTerm(V, S) != T

[sj5] {(V |− > T), S*} inteligente S = {(V |− > T[t S]), S*'} **when** isMember(V,S) = false, {S*} inteligente S = {S*'}

[sj6] {} inteligente S = S

[sj7] S inteligente yes = S

[ij1] S simples fail = S

[ij2] fail simples S = S

[ij3] {S*} simples {S*'} = {S*, S*'}

[m] isMember(V, S) = false

[m1] isMember(V, {S*, (V' |− > T), S*'}) = true **when** V = V'

[g] getTerm(V, S) = V

[g1] getTerm(V, {S*, (V' |− > T), S*'}) = T **when** V = V'

[cl1] F. [c S] = F'. **when**
F [f S] = F'

[cl2] F <-- F+. [c S] = F' <-- F+'. **when**
F [f S] = F',
F+ [fb S] = F+'

[af1] A(T+):CA [f S] = A(T+'):CA **when**
T+[tl S] = T+'

[af2] not A(T+):CA [f S] = not A(T+'):CA **when**
T+ [tl S] = T+'

[fb1] F[fb S] = F' **when**
F[f S] = F'

[fb2] F & F+[fb S] = F' & F+' **when**
F [f S] = F',
F+ [fb S] = F+'

[tl1] T[tl S] = T' **when**
T[t S] = T'

[tl2] T+, T+'[tl S] = T+", T+"' **when**
T+[tl S] = T+", T+' [tl S] = T+"'

[at1] A[t S] = A

[at2] A(T+)[t S] = A(T+[tl S])

[va1] V[t S] = getTerm(V, S)

[in1] I[t S] = I

[li1] [] [t S] = []

[li11] [] [f S] = []

[li12] [] [fb S] = []

[li2] [T+] [t S] = [T+[tl S]]

[li3] [T+ | L][t S] = [T+[tl S]| L'] **when**
L[t S] = L'

[li4] [T+ | V][t S] = [T+[tl S]|V'] **when**
V[t S] = V'

[li5] [T+ | V][t S] = [T+[tl S]|L] **when**
V[t S] = L

A.3.19 TermSets.sdf2

Idem A.2.16.

A.3.20 TermSets.eqs

Idem A.2.17.

A.3.21 Unification.sdf2

module Unification

imports Equations

exports

context-free syntax

"mgu" "(" EQS ")" → SUBS

"mgu-nv" "(" TERM "," TERM ")" → SUBS

"mgu-f" "(" FORMULA "," FORMULA ")" → SUBS

hiddens

context-free syntax

"mgu" "(" EQS "," SUBS ")" → SUBS

A.3.22 Unification.eqs

equations

[mgu1] mgu(EQS) = mgu(EQS,{})

[m1] mgu({T equivalente T', E*}, S) = mgu({E*}[es S'], S inteligente S') **when**
isVar(T) = false, isVar(T') = false,
mgu-nv(T, T') = S', S' != fail

[m2] mgu({T equivalente T', E*}, S) = fail **when**
isVar(T) = false, isVar(T') = false,
mgu-nv(T, T') = S', S' = fail

[m3] mgu({V equivalente V, E*}, S) = mgu({E*}, S)

[m4] mgu({T equivalente V, E*},S) = mgu({V equivalente T, E*}, S) **when**
isVar(T) = false

[m5] mgu(V equivalente T, E*,S) =
mgu({E*}[es{(V |− > T)}], S inteligente {(V |− > T)}) **when**
V != T, V contem T = false

[m6] mgu({V equivalente T, E*}, S) = fail **when**
V != T, V contem T = true

[m7] mgu(EQS, S) = S

[nv1] mgu-nv(I, I) = {}

[nv2] mgu-nv(A, A) = {}

[nv3] mgu-nv(A(T), A(T')) = mgu({T equivalente T'})

[nv4] mgu-nv(A(T, T+), A(T', T+')) =
S inteligente mgu-nv(A(T+[tl S]), A(T+' [tl S])) **when**
mgu({T equivalente T'}) = S

[nv6] mgu-nv([],[]) = {}

[nv7] mgu-nv(L, L') = mgu-nv(norm(L), norm(L')) **when**
isEmpty(L) = false, isEmpty(L') = false

[nv] mgu-nv(T, T') = fail

[nf1] mgu-f(A(T):CA, A(T'):CA') = mgu({T equivalente T'})

[nf2] mgu-f(not A(T):CA, not A(T'):CA') = mgu({T equivalente T'})

[nf3] mgu-f(A(T, T+):CA, A(T', T+'):CA') =
S inteligente mgu-f(A(T+[tl S]):CA, A(T+' [tl S]):CA') **when**
mgu({T equivalente T'}) = S

[nf] mgu-f(F, F') = fail

A.3.23 VariantClause.sdf2

module VariantClause

imports Unification TermSets

exports

context-free syntax

"variant-P" "(" PROGRAM "," TERM-SET ")" → PROGRAM

"variant" "(" PROGRAM "," TERM-SET ")" → PROGRAM

"varset" "(" CLAUSE ")" → TERM-SET

variables

"c" → CHAR+

hiddens

context-free syntax

"newvar" "(" VARIABLE "," TERM-SET ")" → TERM

"prime" "(" VARIABLE ")" → VARIABLE

"varset-O" "(" OBJECTIVE ")" → TERM-SET

"varset-F" "(" FORMBODY ")" → TERM-SET

"tvarset" "(" {TERM ","}+ ")" → TERM-SET

"tvarset-F" "(" FORMULA ")" → TERM-SET

"varsub" "(" TERM-SET "," TERM-SET ")" → SUBS

"add-C" "(" CLAUSE "," PROGRAM ")" → PROGRAM

A.3.24 VariantClause.eqs

equations

[va] variant-P(C* O, TS) = P' **when**
varset-O(O) = TS',
variant(C* O, TS') = P'

[va1] variant(C C' C* O, TS) = P' **when**
varset(C)= TS',
varset(C')= TS",
TS' uniao TS" = TS"',
C[c varsub(TS uniao TS"', TS uniao TS"')] = C",
varset(C") = TS"",
TS uniao TS"" = TS""',
variant(C' C* O, TS""')= P,
add-C(C",P) = P'

[va2] variant(C O ,TS) = C" O **when**
varset(C)= TS',
varset-O(O)= TS",
TS' uniao TS" = TS"',
C[c varsub(TS uniao TS"', TS uniao TS"')] = C"

[sv1] varset(F.) = varset-F(F)

[sv2] varset(F <-- F+.) = varset-F(F) uniao varset-F(F+)

[sv3] varset-O(<-- F.) = varset-F(F)

[sv4] varset-O(<-- F & F+.) = varset-F(F) uniao varset-F(F+)

[sv5] varset-F (F) = tvarset-F(F)

[sv6] varset-F (F & F+) = tvarset-F(F) uniao varset-F(F+)

[st1] tvarset-F(A(T+):CA) = tvarset(T+)

[st2] tvarset-F(not A(T+):CA) = tvarset(T+)

[st3]	tvarset(I)	=	{}
[st4]	tvarset(V)	=	{V}
[st5]	tvarset(A)	=	{}
[st6]	tvarset(A(T))	=	tvarset(T)
[st7]	tvarset(A(T+))	=	tvarset(T+)
[st8]	tvarset(L)	=	{} **when** isEmpty(L) = true
[st9]	tvarset(L)	=	tvarset(norm(L)) **when** isEmpty(L) = false
[st10]	tvarset(T+, T+')	=	tvarset(T+) uniao tvarset(T+')
[vs1]	varsub({}, TS)	=	{}
[vs2]	varsub({V, T*}, TS)	=	{(V |- > V')} inteligente varsub({T*},TS) **when** newvar(V, TS) = V'
[nv1]	newvar(V, TS)	=	prime(V) **when** prime(V) pertence TS = false
[nv2]	newvar(V, TS)	=	newvar(prime(V), TS) **when** prime(V) pertence TS = true
[pr1]	prime(variable(c))	=	variable(c "_" "n")
[add1]	add-C(C, C* O)	=	C C* O