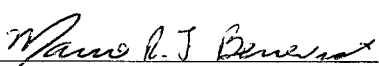


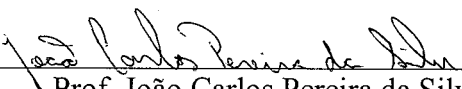
PROCALCULUS
PROGRAMAÇÃO EM LÓGICA E ÁLGEBRA DE PROCESSOS

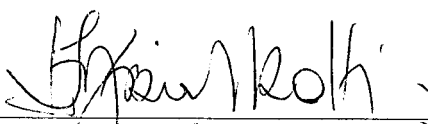
Ricardo Pires Mesquita

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:


Prof. Mário Roberto Folhadela Benevides, Ph.D.


Prof. João Carlos Pereira da Silva, D.Sc.


- Prof. Fábio Pratti, D.Sc.

MESQUITA, RICARDO PIRES

ProCalculus – Programação em Lógica e
Álgebra de Processos [Rio de Janeiro] 2002.

X, 100 p., 29,7 cm (COPPE/UFRJ, M.Sc.,
Engenharia de Sistemas e Computação,
2002.

Tese – Universidade Federal do Rio de
Janeiro, COPPE.

1 – Programação em Lógica (PROLOG)

2 – CCS

3 – Linguagem envolvendo PROLOG e CCS

I. COPPE/UFRJ

II. Título (série)

*“À pequena estrela que surgiu na noite
escura e que me mostrou com sua simples
presença que a vida pode valer a pena...”*

Agradecimentos

A CAPES, pela concessão da bolsa de estudos sem a qual não teria sido possível a realização deste trabalho.

A Priscilla, para quem dedico toda a minha motivação e que me proporciona o apoio e incentivo necessário em todos os momentos, e sem a qual eu não seria o homem que sou hoje.

Aos amigos Rogério L. Salvini e Iuri Wickert que me acompanharam de perto por essa jornada.

A minha família e a todos que de alguma forma contribuíram para o meu bem-estar especialmente nos momentos mais difíceis.

E, em especial, a Mario Benevides, meu orientador, por ter me guiado pelos tortuosos caminhos do conhecimento, com observações e críticas fundamentais para o bom andamento do trabalho, sempre com bom humor, paciência e atenção, preocupado não só com a tese em si, mas com meu crescimento intelectual como um todo, acreditando, sempre, em minha capacidade.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROCALCULUS
PROGRAMAÇÃO EM LÓGICA E ÁLGEBRA DE PROCESSOS

Ricardo Pires Mesquita

Março/2003

Orientador: Mario Roberto Folhadela Benevides

Programa de Engenharia de Sistemas e Computação

Neste trabalho apresentamos o ProCalculus, uma nova linguagem que, em seu escopo, mescla os conceitos do Prolog e do CCS com o intuito de se construir um formalismo capaz de bem representar estruturas lógicas e também simular estruturas cujos elementos, ou agentes, evoluem no tempo, dadas as suas definições.

No ProCalculus, toda a estrutura do Prolog clássico é preservada de modo que este pode ser usado para representar estruturas lógicas estáticas. No entanto, nessa linguagem foram inseridas as idéias de ação e comunicação, fazendo com que predicados comportem-se como os agentes estudados no CCS.

Este trabalho enfoca a simulação de sistemas que evoluem no tempo e que, portanto, tem a estrutura formal do CCS.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROCALCULUS
PROGRAMMING IN LOGIC AND PROCESS ALGEBRA

Ricardo Pires Mesquita

March, 2003

Advisor: Mario Roberto Folhadela Benevides

Department: Computing and Systems Engineering

In this work we present the ProCalculus, a new language that, in its purpose, it blend the concepts of Prolog and CCS, having the intention of to build a formalism that can represent logical structures and to simulate structures which elements, or agents, evolve in the time, given their definitions.

In the ProCalculus, all of the structure of the classical Prolog is preserved so that it can be used to represent statical logical structures. However, in this language was inserted the idea of action and communication, doing that predicates behave like studied agents of CCS.

This work skip the simulation of systems that evolve in the time, therefore, they have the formal structure of CCS.

Índice

1. Introdução	1
2. Noções de PROLOG	4
2.1. Programação em Lógica.	4
2.1.1. Descrição Informal	5
2.1.2. Descrição Formal.	6
2.1.3. Unificação	11
2.1.4. Resolução LSD.	14
2.2. Linguagem PROLOG Básica	16
2.2.1. Sintaxe	18
2.2.2. Semântica.	24
3. Noções de CCS	31
3.1. Definições Básicas.	31
3.2. Sincronismo.	35
3.3. Ação e Transição.	35
3.4. O Poder das Ações Internas.	40
3.5. A Linguagem Básica.	42
3.6. Semântica Transicional.	43
3.7. Um Exemplo: o <i>Jobshop</i>	47
3.8. Bissimulação Forte.	50
3.9. Equivalência por Observação e Bissimulação Fraca.	52
3.10. Mostrando a Correção de Sistema <i>Jobshop</i>	55
4. Uma Nova Linguagem – ProCalculus	60

4.1. Introdução.	60
4.2. A Linguagem ProCalculus.	61
4.2.1. ProCalculus – Definições da Linguagem.	62
4.2.2. Semântica Operacional.	66
4.2.3. Semântica Transicional.	69
4.3. Relação entre CCS e ProCalculus.	72
4.3.1. Mapeando Programas CCS em ProCalculus.	73
4.3.2. ProCalculus <i>versus</i> CCS.	80
4.4. Exemplos de Programas ProCalculus.	81
4.5. Bissimulação e ProCalculus.	94
5. Conclusão	99
Referências	101

Capítulo 1

Introdução

Este trabalho tem por finalidade construir uma linguagem para especificação de elementos da álgebra de processos com base na linguagem Prolog. Na verdade, o que faremos é ampliar o escopo do Prolog para que, além das operações lógicas padrões, possamos também implementar ações, agentes e suas transições. Para que isso seja feito e compreendido, revisaremos os conceitos nos quais fundamentamos nossa teoria, a saber, o Prolog e o CCS.

O PROLOG (PROgramming in LOGic), é uma implementação das idéias de programação em lógica para o subconjunto das cláusulas definidas. Esta linguagem foi projetada e implementada por Colmerauer e seu Grupo de Inteligência Artificial, na Universidade de Marselha, onde foi escrito o primeiro interpretador Prolog na linguagem ALGOL-W [3]. Depois, Battani e Méloni [1] implementaram uma versão mais eficiente deste interpretador, escrita parcialmente em FORTRAN e Roberts [16] implementou na, Universidade de Waterloo, uma versão que recebeu o nome desta cidade, escrita totalmente em linguagem de máquina. No entanto, a linguagem Prolog só passou a atrair maior interesse quando Pereira, Pereira e Warren [15] implementaram, na Universidade de Edimburgo, uma versão muito eficiente, conhecida como DEC-10

Prolog, ou Edinburgh Prolog [5, 7], que incluiu o primeiro compilador Prolog escrito em Prolog. A seguir, o anúncio feito pelo Japão a respeito de um projeto de um computador de quinta geração [14] focalizou grande atenção sobre o Prolog por escolher esta linguagem como o núcleo de programação deste empreendimento.

O Prolog é uma linguagem interativa que permite resolver problemas que envolvem representação simbólica de objetos e seus relacionamentos. O Prolog reforçou a idéia de que a lógica é um formalismo conveniente para representar e processar conhecimento. No Prolog não são descritos procedimentos para se obter a solução de algum problema sendo permitido que se expresse declarativamente apenas a estrutura lógica do problema através de termos, fórmulas e cláusulas.

O CCS (*Calculus for Communicating Systems*) define comunicação e concorrência como noções complementares essenciais para a compreensão de sistemas distribuídos complexos. Por um lado, tal sistema tem diversidade, sendo composto por várias partes, cada uma atuando simultaneamente com, e independentemente de outras partes; por outro lado, um sistema complexo tem unidade que é alcançada através da comunicação entre suas partes.

A base dessas noções está no fato de que se supõe que cada uma das várias partes de tal sistema, chamadas *agentes*, têm identidade própria, que persiste ao longo do tempo. De fato, sem esta suposição, seríamos capazes de fazer apenas distinções entre os eventos de comportamento do sistema. Se desejarmos identificar um evento particular, temos de identificar os agentes que participam deste evento, o que significa determinar *onde*, isto é, em que parte ou partes o evento ocorre.

Cada ação de um agente ou é uma interação com os seus agentes vizinhos e, portanto, uma *comunicação*, ou ocorre de forma independente deles podendo ocorrer *simultaneamente* com suas ações. Frequentemente as ações independentes de um agente não são nada além de comunicações entre os componentes desse mesmo agente.

Nesse trabalho foi feito um estudo dos conceitos e definições básicos do Prolog e do CCS, tendo por objetivo mesclá-los numa única linguagem na qual tanto as propriedades lógicas do Prolog quanto os conceitos de comunicação, transição e mudança de estado do CCS estejam presentes. Tal linguagem chamaremos ProCalculus.

A idéia é a de termos, em princípio, a estrutura padrão da linguagem Prolog, a qual, em seguida, serão acrescidas *ações* que, como no CCS, levarão os agentes (predicados) a mudarem de estado. O ProCalculus gerará sistemas capazes, portanto, de evoluírem no tempo e não apenas manipular estruturas estáticas predefinidas como

acontece no Prolog padrão. Assim, a linguagem apresentará todas as características desejáveis da estrutura de programação declarativa, sendo possível a construção de programas de lógica semelhantes aos do Prolog, e ainda contando com elementos sintáticos extraídos da álgebra de processos que tornarão certos predicados passíveis de mudança de estado, passando a desempenhar novas atribuições, de acordo com suas respectivas definições.

Veremos que o ProCalculus tem uma estrutura que possibilitará a simulação tanto de programas em CCS como programas em Prolog, sem perdas das características dos mesmos, ou seja, sistemas representados em CCS serão bem modelados em ProCalculus e o mesmo ocorrerá com programas em Prolog. Entretanto, o que nos interessará neste trabalho é, principalmente, o tratamento de programas com ações, isto é, programas com a estrutura semelhante à do CCS.

A primeira parte dessa tese contém uma revisão bibliográfica acerca dos principais conceitos utilizados no trabalho.

No Capítulo 2, as definições básicas da linguagem Prolog são apresentadas com o intuito de estruturar o formalismo sobre o qual se baseará a nova linguagem proposta.

No Capítulo 3 é feita uma introdução aos conceitos de CCS, que posteriormente serão modelados pelo ProCalculus.

O Capítulo 4 contém a definição da linguagem ProCalculus, com sua sintaxe e semântica, juntamente com a apresentação de uma máquina abstrata para essa linguagem e também alguns exemplos ilustrativos.

O Capítulo 5 encerra o texto com algumas conclusões obtidas e efetuando propostas para melhor entender o trabalho realizado.

Capítulo 2

Noções de PROLOG

2.1. Programação em Lógica

Um programa em lógica é um modelo de um certo problema ou situação expresso através de um conjunto finito de sentenças lógicas, não sendo, dessa forma, a descrição de procedimentos para se obter a solução de um problema. A pesquisa de soluções fica, portanto, totalmente sob responsabilidade do interpretador utilizado, de modo que um programa em lógica assemelha-se muito a um banco de dados exceto pelo fato de que as afirmações em um banco de dados descrevem apenas observações tais como “Maria é mãe de Pedro”, enquanto as sentenças em um programa em lógica podem também ter um escopo mais genérico como “o chefe de um funcionário é superior hierárquico do funcionário” ou “o chefe de um superior hierárquico de um funcionário é superior hierárquico do funcionário”.

Dessa forma, a programação em lógica ilustra um estilo mais fundamental que pode ser chamado de *programação declarativa* (ou *não-procedimental*) que contrasta com a programação procedimental das linguagens tradicionais. A noção de

programação declarativa engloba a de programação funcional, como no LISP e quase todas as linguagens para consulta a bancos de dados como no SQL.

Os conceitos de consulta e de resposta também diferem das noções tradicionais. Uma consulta a um programa em lógica é uma afirmação exprimindo as condições a serem satisfeitas por uma resposta correta em presença da informação descrita pelo programa. Porém, o ponto fundamental de programação em lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, a maioria dos sistemas para programação em lógica reduzem a busca de respostas corretas à pesquisa de refutações a partir das sentenças do programa e da negação da consulta.

Assim, a resposta de uma consulta a um programa em lógica não se limita apenas a indicar que uma suposição acerca da informação contida no programa é falsa ou verdadeira. A resposta efetivamente exhibe uma informação extraída do programa e pode vir acompanhada de uma explicação sobre como foi obtida, expressa em termos da refutação que a gerou.

A seguir, apresentamos descrições da linguagem segundo Casanova [2].

2.1.1. *Descrição Informal*

Vamos supor que queiramos descrever um conjunto de doces em uma confeitaria, indicando o material com que são feitos e que toda a informação que temos acerca dos doces seja descrita pelas seguintes afirmações:

- I₁. *A* é uma trufa
- I₂. *B* é um quindim
- I₃. Toda trufa é feita de chocolate
- I₄. Todo quindim é feito de ovos

Observe que as duas primeiras informações são fatuais e que as outras duas capturam informação mais genérica. Essas quatro afirmações em conjunto constituem um programa em lógica.

A afirmação

- C. *x* é feito de chocolate

descreve uma consulta ao programa dado.

Na maioria dos sistemas para programação em lógica, a execução do programa anterior a partir de C corresponde à pesquisa de uma refutação a partir da negação de C e das afirmações no programa. Por exemplo, podemos ter a seguinte seqüência de afirmações:

R_1 . A é uma trufa

R_2 . Toda trufa é feita de chocolate

R_3 . x não é feito de chocolate

R_4 . x não é uma trufa

R_5 . Contradição

Note que R_1 e R_2 são afirmações do programa, R_3 é a negação da consulta C , R_4 segue de R_2 e R_3 e, finalmente, R_5 segue de R_4 e R_1 , substituindo-se x por A em R_4 .

Na verdade, a substituição de x por A na refutação indica que A é um objeto que satisfaz às condições da consulta C em presença das afirmações no programa, ou seja, que A é uma resposta correta.

2.1.2. *Descrição Formal*

Apresentaremos aqui uma formalização do exemplo anterior nos seguintes contextos:

- Programação em linguagens de primeira ordem.
- Programação em cláusulas.
- Programação em cláusulas definidas.

A formalização do exemplo começa com a escolha de uma linguagem apropriada. As linguagens de primeira ordem são definidas da seguinte forma. O alfabeto consiste de uma série de *símbolos lógicos*, incluindo *variáveis*, *conectivos lógicos* e *quantificadores*, e uma série de *símbolos não-lógicos*, incluindo *constantes* para denotar objetos específicos, *símbolos predicativos* para denotar relacionamentos entre objetos e

símbolos funcionais para denotar funções sobre objetos. Apenas os símbolos não-lógicos variam de linguagem para linguagem, já que os símbolos lógicos são sempre parte do alfabeto por definição.

Na sua forma mais geral, as sentenças de uma linguagem de primeira ordem são os *termos* e as *fórmulas*. Um termo é uma variável, uma constante ou uma expressão da forma $f(t_1, \dots, t_m)$, onde f é um símbolo funcional admitindo m argumentos e t_1, \dots, t_m são termos. Uma fórmula é ou uma *fórmula atômica* da forma $p(t_1, \dots, t_m)$, onde p é um símbolo predicativo admitindo n argumentos e t_1, \dots, t_m são termos, ou é uma expressão obtida compondo-se fórmulas através dos conectivos lógicos ou prefixando-se fórmulas com quantificadores.

Intuitivamente, os termos permitem denotar objetos do domínio do problema, diretamente através de seus nomes ou de variáveis, ou indiretamente através de funções. As fórmulas atômicas permitem expressar relações existentes entre objetos do domínio do problema. Finalmente, as fórmulas (não-atômicas) permitem expressar propriedades das relações sobre o domínio do problema. Por exemplo, uma fórmula da forma “ $\forall x(P \rightarrow Q)$ ” lê-se “para todo objeto x , se a propriedade P vale para x , então a propriedade Q também vale para x ”.

No caso específico do exemplo anterior, escolheremos um alfabeto de primeira ordem cujos símbolos não-lógicos incluem apenas constantes para denotar os doces da confeitaria e o tipo do doce, um símbolo predicativo, *doce*, admitindo dois argumentos, e dois símbolos predicativos, *chocolate* e *ovos*, admitindo apenas um argumento, de forma que:

- $doce(x, y)$ lê-se “ x é uma doce do tipo y ”.
- $chocolate(x)$ lê-se “ x é feito de chocolate”.
- $ovos(x)$ lê-se “ x é feito de ovos”.

Partindo das linguagens de primeira ordem, há várias possibilidades para o desenvolvimento de programação em lógica.

Em primeiro lugar, temos *programação em linguagens de primeira ordem*, onde um programa é qualquer coleção finita de fórmulas de primeira ordem e uma consulta é qualquer fórmula de primeira ordem exprimindo as condições a serem satisfeitas por uma resposta correta.

Nesse contexto, o seguinte conjunto de fórmulas define o programa apresentado anteriormente.

$F_1. \text{doce}(A, \text{trufa})$

$F_2. \text{doce}(B, \text{quindim})$

$F_3. \forall x(\text{doce}(x, \text{trufa}) \rightarrow \text{chocolate}(x))$

$F_4. \forall x(\text{doce}(B, \text{quindim}) \rightarrow \text{ovos}(x))$

Observe que, usando a leitura intuitiva indicada anteriormente, essas fórmulas possuem a mesma leitura que as afirmações do exemplo informal. Por exemplo, a fórmula F_4 tem o mesmo significado da afirmação I_4 .

A fórmula a seguir ilustra a consulta C:

$P. \text{chocolate}(x)$

Embora conceitualmente importante, esta abordagem não será levada adiante neste texto pois a maioria dos sistemas para programação em lógica não a segue.

Para possibilitar o desenvolvimento de programação em lógica, temos uma Segunda abordagem, qualificada de *programação em cláusulas genéricas*, na qual assumimos que um programa é um conjunto finito de cláusulas mantendo a definição de consulta como qualquer fórmula de primeira ordem. Expressar programas apenas através de cláusulas foi um passo importante para a construção de sistemas práticos para programação em lógica.

A notação em cláusulas, embora muito mais simples, tem o mesmo poder de expressão que a notação das linguagens de primeira ordem. Porém, esta família de linguagens não deve ser ignorada, pois, em certos casos, torna-se mais conveniente modelar primeiro o programa em lógica através de fórmulas, para depois mapeá-lo na notação de cláusulas.

Uma cláusula é uma lista $L_1 \dots L_n$ onde, para cada i no intervalo de números naturais $[1, \dots, n]$, L_i é ou uma fórmula atômica ou a negação de uma fórmula atômica. A lista vazia é chamada de *cláusula vazia* e denotada por “ ”.

Uma cláusula da forma $L_1 \dots L_n$, cujas variáveis são x_1, \dots, x_k , deve ser lida como:

para todo x_1, \dots, x_k , L_1 ou ... ou L_n valem

Em particular, a cláusula vazia é sempre falsa, ou seja, representa uma contradição.

Uma refutação neste contexto reduz-se a uma seqüência de cláusulas terminando na cláusula vazia de forma que cada cláusula ou pertence ao conjunto inicial dado ou é inferida a partir das cláusulas anteriores.

No contexto de programação em cláusulas genéricas, temos

$C_1. \text{doce}(A, \text{trufa})$

$C_1. \text{doce}(B, \text{quindim})$

$C_1. \neg \text{doce}(x, \text{trufa}) \text{chocolate}(x)$

$C_1. \neg \text{doce}(x, \text{quindim}) \text{ovos}(x)$

Esta versão do programa é exatamente equivalente à anterior, pois uma cláusula da forma “ $\neg p(x) q(x)$ ” é equivalente à fórmula “ $\forall x(p(x) \rightarrow q(x))$ ”.

A terceira possibilidade para o desenvolvimento de programação em lógica, chamada de *programação em cláusulas definidas*, trabalha apenas com uma classe especial de cláusulas e adota uma notação própria. Esta variante facilita ainda mais a construção de sistemas para programação em lógica e forma a base da linguagem PROLOG. Porém, sob alguns aspectos, perde poder de expressão quando comparada com programação em cláusulas genéricas.

Mais precisamente, um programa neste enfoque é um conjunto finito de cláusulas definidas e uma consulta é uma conjunção de fórmulas atômicas.

Uma *cláusula definida* por sua vez é uma expressão da forma $A \leftarrow B_1 \dots B_n$, onde A, B_1, \dots, B_n são fórmulas atômicas. Quando $n = 0$, a cláusula se reduz a expressão $A \leftarrow$.

Se x_1, \dots, x_k são as variáveis ocorrendo na cláusula definida, então ela deve ser interpretada como:

para todo x_1, \dots, x_k, A vale se B_1 e ... e B_n valerem

Se $n = 0$, então a interpretação se resume a

para todo x_1, \dots, x_k, A vale

Uma *cláusula objetivo*, é uma expressão da forma $\leftarrow B_1 \dots B_n$, onde B_1, \dots, B_n são fórmulas atômicas e $n > 0$. Se x_1, \dots, x_k são as variáveis ocorrendo na cláusula objetivo, então ela deve ser interpretada como:

para nenhum x_1, \dots, x_k , B_1 e ... e B_n valem

ou seja, a cláusula nega a existência de objetos satisfazendo simultaneamente as condições B_1, \dots, B_n . Em outras palavras, a cláusula representa a negação da consulta expressa pela fórmula $B_1 \wedge \dots \wedge B_n$.

A cláusula vazia, “ ”, também é considerada como cláusula objetivo, sendo novamente sempre falsa.

Assim, temos o seguinte programa em cláusulas definidas para o exemplo das vestimentas:

D₁. *doce*(A, trufa) \leftarrow

D₂. *doce*(B, quindim) \leftarrow

D₃. *tecido*(x) \leftarrow *doce*(x, trufa)

D₄. *ovos*(x) \leftarrow *doce*(x, quindim)

Novamente, esta versão do programa é exatamente equivalente às anteriores, pois uma cláusula definida da forma “ $q(x) \leftarrow p(x)$ ” é equivalente à fórmula “ $\forall x(p(x) \rightarrow q(x))$ ”. Porém, nem sempre é possível obter um conjunto de cláusulas definidas que seja equivalente a uma dada fórmula.

Este programa possui ainda a seguinte interpretação intuitiva. As cláusulas D₁ e D₂ correspondem às afirmações I₁ e I₂ e capturam informação diretamente conhecida; as cláusulas D₃ e D₄ correspondem a I₃ e I₄ e reduzem o problema de provar que x é feito de tecido ao problema de provar que x é uma camisa.

Recorde que a execução de um programa para uma dada consulta corresponde à pesquisa de uma refutação a partir das cláusulas do programa e de cláusulas representando a negação da consulta. No caso do exemplo, uma particular refutação seria:

R₁. *doce*(A, trufa) \leftarrow

R₂. *chocolate*(x) \leftarrow *doce*(x, trufa)

$R_3. \leftarrow chocolate(x)$

$R_4. \leftarrow doce(x, trufa)$

$R_5.$

Esta refutação é idêntica à apresentada informalmente. As cláusulas definidas em R_1 e R_2 pertencem ao programa e a cláusula objetivo em R_3 representa a negação da consulta P . A cláusula em R_4 segue de R_2 e R_3 essencialmente porque:

- R_3 afirma que não há doces feitos de chocolate.
- R_2 afirma que toda trufa é feita de chocolate.
- Logo, R_2 e R_3 implicam em que não haja doces feitos de chocolate, o que é expresso pela cláusula em R_4 .

A cláusula em R_5 segue de R_1 e R_4 porque

- R_4 afirma que não há doces feitos de chocolate.
- R_1 afirma que A é feito de chocolate.
- Logo, R_1 e R_4 levam a uma contradição, o que é expresso pela cláusula vazia em R_5 .

2.1.3. Unificação

Definição 2.1:

- a. Um par (x, t) é uma *substituição simples* (lê-se “ x substituído por t ”) se e somente se x é uma variável e t é um termo.
- b. Um conjunto finito β de substituições simples é uma *substituição* se e somente se duas substituições simples em β não coincidem no primeiro elemento.
- c. Uma substituição β é uma *substituição básica* se e somente se, para todo (x, t) em β , t é um termo sem ocorrências de variáveis.
- d. β é a *substituição vazia* se e somente se β for o conjunto vazio.
- e. Uma substituição β é uma *renomeação de variáveis* ou, simplesmente, uma *renomeação*, se e somente se cada par (x, t) em β for tal que t é uma variável e não existirem dois pares (x, u) e (y, v) em β tais que $x \neq y$ e $u = v$.

A expressão “ x/t ” denotará uma substituição simples (x, t). Letras gregas denotarão substituições e, em especial, “ ε ” denotará a substituição vazia.

De acordo com essas definições, $\beta = \{x/a, y/b, z/y\}$ e $\theta = \{x/f(y), y/z\}$ são substituições. Porém, $\varphi = \{x/f(x), x/a\}$ não é uma substituição pois possui dois pares com o mesmo primeiro elemento.

Uma *expressão* é qualquer seqüência de símbolos de um alfabeto de primeira ordem, e uma *expressão simples* é qualquer literal ou termo sobre o alfabeto.

Definição 2.2:

- a. Sejam E uma expressão e $\beta = \{x_1/t_1, \dots, x_n/t_n\}$ uma substituição. A *instanciação* de E por β ou, simplesmente, uma *instância* de E é a expressão $E\beta$ obtida substituindo-se simultaneamente em E cada ocorrência de x_i por $t_i, i = 1, \dots, n$.
- b. Sejam \mathbf{E} um conjunto de expressões e $\beta = \{x_1/t_1, \dots, x_n/t_n\}$ uma substituição. O conjunto de expressões $\mathbf{E}\beta = \{E\beta \mid E \in \mathbf{E}\}$ é a *instanciação* de \mathbf{E} por β ou, simplesmente, uma *instância* de \mathbf{E} .
- c. Seja C uma cláusula e β uma substituição. A *instanciação* de C por β , denotada por $C\beta$, é a cláusula obtida instanciando-se C por β e eliminando-se as ocorrências repetidas do mesmo literal, exceto a ocorrência mais à esquerda.

Exemplo:

Se $\mathbf{E} = \{p(x, y), \neg q(f(x))\}$ e $\theta = \{x/f(y), y/z\}$, então $\mathbf{E}\theta = \{p(f(y), z), \neg q(f(f(x)))\}$ é a instanciação de \mathbf{E} por θ .

Definição 2.3:

A *composição* de substituições é a função, denotada por “ \circ ”, que mapeia pares de substituições em uma substituição e é definida da seguinte forma. Para todo par de substituições

$$\beta = \{x_1/t_1, \dots, x_n/t_n, y_1/s_1, \dots, y_k/s_k\}$$

$$\theta = \{y_1/r_1, \dots, y_k/r_k, z_1/q_1, \dots, z_m/q_m\}$$

onde $x_1, \dots, x_n, y_1, \dots, y_k$ e z_1, \dots, z_m , são variáveis distintas, a composição de β com θ será a substituição:

$$\beta \circ \theta = \{x_1/(t_1)\theta, \dots, x_n/(t_n)\theta, y_1/(s_1)\theta, \dots, y_k/(s_k)\theta, z_1/q_1, \dots, z_m/q_m\}$$

Exemplo:

A composição de $\beta = \{x/f(y), y/z\}$ com $\theta = \{x/a, y/b, z/y\}$ é a substituição $\beta \circ \theta = \{x/f(b), y/y, z/y\}$ obtida aplicando-se θ aos termos das substituições simples em β , formando o conjunto $\{x/f(b), y/y\}$ e acrescentando-se a esse conjunto a única substituição simples de θ , “z/y”, cujo primeiro elemento não coincide com o primeiro elemento de uma substituição simples em β .

As substituições possuem as seguintes propriedades:

Proposição 1:

Sejam θ, β e φ substituições e ε a substituição vazia.

- $\theta \circ \varepsilon = \varepsilon \circ \theta = \theta$
- $(E\theta)\beta = E(\theta \circ \beta)$, para toda expressão E
- Se $E\theta = E\beta$ para toda variável E , então $\theta = \beta$
- $(\theta \circ \beta) \circ \varphi = \theta \circ (\beta \circ \varphi)$

Definição 2.4:

Seja $\mathbf{E} = \{E_1, \dots, E_n\}$ um conjunto de expressões simples e β uma substituição.

- β é um *unificador* de \mathbf{E} se e somente se $(E_1)\beta = \dots = (E_n)\beta$.
- β é um *unificador mais geral* (u.m.g.) de \mathbf{E} se e somente se β é um unificador de \mathbf{E} e, para todo unificador θ de \mathbf{E} , existe uma substituição φ tal que $\theta = \beta \circ \varphi$.
- O conjunto \mathbf{E} é *unificável* se e somente se existe um unificador para \mathbf{E} .

Exemplo:

Se $\mathbf{E} = \{p(a, f(x)), p(y, z)\}$, então $\theta = \{y/a, x/b, z/f(b)\}$ é um unificador de \mathbf{E} pois $\mathbf{E}\theta = \{p(a, f(b))\}$. Mas $\beta = \{y/a, z/f(x)\}$ também é um unificador de \mathbf{E} pois $\mathbf{E}\beta = \{p(a, f(x))\}$. Porém, β é mais geral do que θ pois evita a substituição de x por b . De fato, $\theta = \beta \circ \varphi$, onde $\varphi = \{x/b\}$. Na verdade, β é um u.m.g. de \mathbf{E} .

Já o conjunto $\mathbf{F} = \{p(a, x), p(b, y)\}$ não é unificável, essencialmente porque os seus dois literais diferem em duas constantes e não podemos substituir uma constante por outra no processo de unificação.

Observamos também que o unificador mais geral para um dado conjunto de expressões simples \mathbf{E} não é único. Porém, o seguinte resultado mostra que dois unificadores mais gerais diferem apenas por renomeações de variáveis.

2.1.4. Resolução-LSD

A Resolução-LSD funciona com conjuntos compostos de uma ou mais cláusulas com exatamente um literal positivo e de uma cláusula apenas com literais negativos.

Definição 2.5:

Seja A um alfabeto de primeira ordem.

- a. Uma *cláusula de Horn* sobre A é ou uma cláusula definida ou uma cláusula objetivo sobre A .
- b. Um *conjunto quase definido* é um conjunto de cláusulas definidas acrescido exatamente de uma cláusula objetivo.

Definição 2.6:

A *linguagem das cláusulas de Horn* sobre um alfabeto de primeira ordem A é o conjunto de todas as cláusulas de Horn sobre A .

Definição 2.7:

- a. Uma função f é uma *função de seleção* para cláusulas objetivo, ou simplesmente uma *função de seleção*, se e somente se f mapeia cada cláusula objetivo C em um literal de C .
- b. Uma função f é a *função de seleção padrão* se e somente se f mapeia cada cláusula objetivo C no literal mais à esquerda de C .

Definição 2.8:

Seja f uma função de seleção. O sistema formal da *resolução com função de seleção f para conjuntos quase-definidos*, $RSD(f)$, consiste de:

- a. Uma classe de linguagens: linguagens das cláusulas de Horn.
- b. Uma regra de inferência: f -extensão.

Se A' é uma cláusula objetivo, A'' é uma cláusula definida e A é uma f -extensão de A' por A'' , então derive A de A' e A'' .

Com isso, podemos agora introduzir o conceito de resolução-LSD.

Definição 2.9:

Seja f uma função de seleção. O *método da resolução linear com função de seleção f para conjuntos quase-definidos* ou *resolução-LSD(f)*, consiste do par $(RSD(f), DD(f))$, onde $RSD(f)$ é o sistema formal de seleção f para conjuntos quase-definidos e $DD(f)$ é o conjunto das LSD(f)-deduções.

Quando não for relevante mencionar a função de seleção, nos referimos ao método como resolução-LSD.

Como consequência das restrições sobre a aplicação da regra de f -extensão e sobre a construção das LSD(f)-deduções, o método da resolução-LSD(f) possui as seguintes características básicas:

- O conjunto de entrada deverá ser um conjunto de cláusulas definidas, acrescido de uma cláusula objetivo.
- As refutações são lineares de entrada e têm o seguinte formato:
 - uma lista de cláusulas definidas de entrada, seguida da
 - cláusula objetivo de entrada, seguida de
 - uma lista de cláusulas objetivo.
- A fatoração é desnecessária.
- A cláusula auxiliar em uma aplicação da regra de f -extensão é sempre uma cláusula definida do conjunto de entrada e o resolvente é sempre uma cláusula objetivo.
- A estratégia de seleção de literais, incorporada na definição da regra de f -extensão, dita que
 - o literal da cláusula pai selecionado é indicado pela função de seleção f ;
 - o literal da cláusula auxiliar selecionado é sempre a cabeça da cláusula.

As *árvores de refutação por resolução-LSD(f)* ou, simplesmente, *árvores de LSD(f)-refutação*, também são consideravelmente mais simples já que os rótulos dos nós serão sempre cláusulas objetivo e os filhos de um nó serão obtidos apenas por f -extensão, usando como cláusula auxiliar cada uma das cláusulas definidas de entrada cuja cabeça é unificável com o literal selecionado por f da cláusula que rotula o nó. Note que nenhum outro literal além do selecionado é considerado, o que não é o caso na construção das árvores de refutação por resolução linear. Veremos um exemplo mais adiante.

2.2. Linguagem PROLOG Básica

A idéia de se usar a Lógica como um formalismo executável em computador explorada principalmente por Kowalski [6, 7] e Hayes [4], recebeu grande impulso com o advento da linguagem PROLOG (PROgramming in LOGic), que é uma implementação das idéias de programação em lógica para o subconjunto das cláusulas definidas. Esta linguagem foi projetada e implementada por Colmerauer e seu Grupo de Inteligência Artificial, na Universidade de Marselha, onde foi escrito o primeiro interpretador Prolog na linguagem ALGOL-W [3]. Depois, Battani e Méloni [1] implementaram uma versão

mais eficiente deste interpretador, escrita parcialmente em FORTRAN e Roberts [16] implementou na, Universidade de Waterloo, uma versão que recebeu o nome desta cidade, escrita totalmente em linguagem de máquina. No entanto, a linguagem Prolog só passou a atrair maior interesse quando Pereira, Pereira e Warren implementaram, na Universidade de Edimburgo, uma versão muito eficiente, conhecida como DEC-10 Prolog, ou Edinburgh Prolog [5, 7], que incluiu o primeiro compilador Prolog escrito em Prolog. A seguir, o anúncio feito pelo Japão a respeito de um projeto de um computador de quinta geração [14] focalizou grande atenção sobre o Prolog por escolher esta linguagem como o núcleo de programação deste empreendimento.

O Prolog é uma linguagem interativa que permite resolver problema que envolvem representação simbólica de objetos e seus relacionamentos. O Prolog reforçou a idéia de que a Lógica é um formalismo conveniente para representar e processar conhecimento. No Prolog não são descritos procedimentos para se obter a solução de algum problema sendo permitido que se expresse declarativamente apenas a estrutura lógica do problema através de termos, fórmulas e cláusulas.

Um programa em Prolog possui três interpretações semânticas básicas. Na interpretação declarativa, as cláusulas que definem o programa descrevem uma teoria de primeira ordem. Na procedimental, as cláusulas são vistas como entrada para um método de refutação. E finalmente, na interpretação operacional, as cláusulas são vistas como comandos para um procedimento de refutação particular.

Estas alternativas para a semântica são valiosas em termos de entendimento e codificação de um programa Prolog. A interpretação declarativa permite que o programador modele um dado problema através de assertivas acerca dos objetos do domínio de discurso. A interpretação procedimental permite que o programador identifique e descreva o problema pela redução do mesmo a subproblemas, através da definição de uma série de chamadas de procedimentos. Por fim, a semântica operacional reintroduz a idéia de controle de execução, que é irrelevante do ponto de vista da semântica declarativa, através da ordem das cláusulas em um programa Prolog e das fórmulas atômicas em uma cláusula Prolog. Ou seja, esta interpretação é semelhante à semântica operacional de muitas linguagens de programação tradicionais e deve ser considerada, principalmente, em grandes programas, por questões de eficiência de execução.

As principais características da linguagem Prolog são:

- Orientado para processamento simbólico.

- Representa uma implementação da Lógica como linguagem de programação.
- Apresenta uma semântica declarativa inerente à Lógica.
- Permite a definição de programas invertíveis, ou seja, programas que não distinguem entre argumentos de entrada e de saída. Como consequência, permite a definição de programas com mais de uma finalidade, que podem ser chamados com formas diferentes de entrada/saída.
- Permite a obtenção de respostas alternativas.
- Incorpora um mecanismo uniforme para passagem, análise, seleção e criação de estruturas de dados.
- Suporta uma estrutura de dados que permita simular registros ou listas.
- Permite a recuperação dedutiva de informações.
- Suporta codificações recursivas para descrição de processos e problemas dispensando os mecanismos tradicionais de controle, como o comando “goto” e os laços “do”, “for” e “while”.
- Permite aproximar o processo de especificação do processo de codificação de programas.
- Representa programas e dados através do mesmo formalismo (cláusulas).
- Incorpora facilidades computacionais extra e metalógicas.

2.2.1. *Sintaxe*

A descrição da sintaxe da linguagem vista a seguir é baseada em Casanova [2], Montague [13] e Kamp [5].

Definição 2.10:

O alfabeto Prolog consiste dos seguintes símbolos:

- Pontuação: (,) , . , ‘ , “
- Conectivos: & (conjunção)
:- (implicação)
- Letras: $a, b, \dots, z, A, B, \dots, Z$
- Dígitos: 0, 1, ..., 9

- Especiais: *, _, +, -, /, ...

Definição 2.11:

a. *Átomo:*

- Uma cadeia de letras e dígitos, iniciando com uma letra minúscula.
- Uma cadeia de símbolos do alfabeto básico Prolog (podendo incluir o branco delimitada por aspas.

Exemplo:

a, d, permitido, leOA, “a b”, “João Silva”, “Pedro I”, “:-”

b. *Constante Prolog:*

- Um átomo.
- Uma cadeia de dígitos com no máximo uma ocorrência do símbolo “.”.
- Uma cadeia de símbolos do alfabeto básico Prolog (podendo incluir o branco) delimitada por apóstrofes.

Exemplo:

a, d, permitido, leOA, “a b”, “João Silva”, “Pedro I”, “:-”

15, 123, 15.5, 123.32

‘a b’, ‘João Silva’, ‘Pedro I’, ‘:-’

c. *Variável Prolog:*

- Uma cadeia de letras e dígitos iniciando com uma letra maiúscula.
- O símbolo “*”, chamado de *variável anônima*.

Exemplo:

$X, Y, Z, W, Ma, Pa, X523x, Xyz, Nome, RESPOSTA, A11, *$

Definição 2.12:

O conjunto dos *termos Prolog*, ou simplesmente dos *termos*, é o menor conjunto satisfazendo às seguintes condições:

- i. Toda variável Prolog é um termo Prolog.
- ii. Toda constante Prolog é um termo Prolog.
- iii. Se t_1, \dots, t_n são termos Prolog e f é um átomo, então $f(t_1, \dots, t_n)$ é um termo Prolog, onde o átomo f tem o papel de um símbolo funcional n -ário. Diz-se ainda que a expressão $f(t_1, \dots, t_n)$ é um *termo funcional Prolog*.

Exemplo:

$x, a, "a\ b", fortran, 'João\ Silva', 15.5, X, livro(Autor, Editora, Ano)$

Definição 2.13:

Uma *fórmula atômica Prolog* é uma cadeia da forma $p(t_1, \dots, t_n)$, para $n \geq 0$, onde t_1, \dots, t_n são termos Prolog e p é um átomo *no papel de um símbolo predicativo* n -ário.

Quando n for igual a zero, por abuso de linguagem, escreve-se $p()$ como p .

É interessante observar que um átomo pode representar simultaneamente o papel de um ou mais símbolos funcionais ou predicativos de aridades diferentes. Este uso múltiplo do mesmo átomo não causa ambigüidades, pois o contexto de um programa indicará sempre o papel que o mesmo representa.

Exemplo:

$x(a, "a\ b", fortran, 'João\ Silva', 15.5, X)$
 $livro(Autor, Editora, Ano)$

Observe que uma mesma expressão pode ser um termo ou uma fórmula atômica, dependendo do contexto na qual a mesma se encontre inserida.

Definição 2.14:

O conjunto das *cláusulas Prolog*, ou simplesmente *cláusulas*, é o menor conjunto satisfazendo as seguintes condições:

- i. Se A é uma fórmula atômica Prolog, então “ A .” é uma cláusula Prolog, chamada de *cláusula unitária Prolog* (ou *cláusula terminal Prolog*).
- ii. Se A e B_1, \dots, B_n são fórmulas atômicas Prolog, então a expressão “ $A :- B_1 \& \dots \& B_n$.” é uma cláusula Prolog, chamada de *cláusula não-unitária Prolog*, onde A é a *cabeça* e “ $B_1 \& \dots \& B_n$ ” é o *corpo* da cláusula.
- iii. Se C_1, \dots, C_n são fórmulas atômicas Prolog, então “ $:- C_1 \& \dots \& C_n$.” é uma cláusula Prolog, chamada de *cláusula objetivo Prolog*.

Exemplo:

a. Cláusula unitária:

Uma cláusula unitária básica tem a forma:

$$p(t_1, t_2, \dots, t_q).$$

onde “ p ” é um átomo e $t_k, k \in [1, q]$, são constantes. A cláusula unitária básica representa um fato relativo a um determinado domínio de problema. Por exemplo:

$$\text{programa}(a, \text{fortran}).$$

Lê-se: “ a é um programa fortran”.

$$\text{arquivo}(d, \text{sentencial}).$$

Lê-se: “ d é um arquivo sentencial”.

b. Cláusula não-unitária:

Uma cláusula não-unitária permite expressar uma implicação entre relações existentes no domínio do problema. De um modo geral, implicações podem representar associações como regras, definições, dependências de causa-e-efeito entre cláusulas unitárias, heurísticas, conhecimento, restrições de integridade, meta-regras definindo métodos de arbitragem entre regras e meta-regras definindo regras de aquisição de novas regras. Cláusulas não-unitárias permitem estender uma base de fatos, transformando-a em uma base de dados dedutiva na qual:

- Átomos representam nomes de relações.
- Cláusulas unitárias representam n -uplas das relações.
- Cláusulas não-unitárias representam regras de dedução.
- Cláusulas objetivo representam consultas que permitem recuperar valores de uma ou mais relações.

Por exemplo:

$$\textit{depende}(X, Y) \textit{:} \textit{-} \textit{chama}(X, Y) \ \& \ \textit{depende}(Z, Y)$$

Lê-se: “para todo X, Y e Z, X depende de Y se X chama Z e Z depende de Y ”.

Note que esta cláusula é sobre o átomo “depende” no papel de um símbolo predicativo binário. Note também que esta cláusula não-unitária é recursiva, no sentido de que a cabeça da cláusula é referenciada no corpo da mesma.

c. Cláusula objetivo:

Uma cláusula objetivo Prolog corresponde a uma consulta atômica ou conjuntiva e força a execução de um programa Prolog, permitindo:

- Certificar-se a respeito de relacionamentos entre objetos do domínio do problema.
- Recuperar dedutivamente informações armazenadas.

A máquina Prolog responderá a uma cláusula objetivo sem variáveis livres, supondo a inexistência de laços infinitos, com mensagens de *sucesso* e *falha*. Uma mensagem de *sucesso* indica que uma cláusula objetivo pode ser provada e uma mensagem de *falha* indica o oposto. Por exemplo:

:- *depende(a, e)*.

Lê-se: “*a* depende de *e*?”.

:- *depende(X, b) & programa(X, fortran)*.

Lê-se: “existe algum *X* que depende de *b* e é um programa fortran?”

Definição 2.15:

Uma *cláusula definida Prolog* é uma cláusula unitária ou uma cláusula não-unitária Prolog.

Definição 1.16:

Uma cláusula definida *C* é *sobre* um átomo *p*, no papel de um símbolo predicativo *n*-ário, se e somente se a cabeça de *C* for da forma “*p(t*₁, ... , *t*_{*n*})” ou *C* for uma cláusula unitária da forma “*p(t*₁, ... , *t*_{*n*})”. Dizemos que *p* é o átomo da cláusula definida.

Definição 2.17:

- a. Um *programa Prolog* é uma seqüência de cláusulas definidas Prolog.
- b. Uma *consulta Prolog* é uma cláusula objetivo Prolog.

A ordem das fórmulas atômicas em uma cláusula Prolog, bem como a ordem das cláusulas em um programa Prolog é importante. Assim, por exemplo, “*A :- B*₁ & *B*₂.” e “*A :- B*₂ & *B*₁.” são cláusulas não-unitárias Prolog distintas.

Um átomo pode possuir, em um mesmo programa, diferentes papéis, identificados por diferentes aridades e contextos, ou seja, diferentes ocorrências em termos ou fórmulas atômicas. O processo conhecido como unificação distingue ocorrências do mesmo átomo em diferentes papéis.

Definição 2.18:

Dado um programa Prolog \mathbf{P} , a subsequência de cláusulas em \mathbf{P} sobre um mesmo átomo p , no mesmo papel, é o *procedimento* p definido em \mathbf{P} .

2.2.2. Semântica

A semântica da linguagem foi descrita com base em Casanova [2], Kamp [5] e Montague [13].

Um programa Prolog, com pequenas diferenças sintáticas, nada mais é do que um programa em cláusulas definidas. Portanto, todos os conceitos e resultados sobre as semânticas declarativa e procedimental para programas em cláusulas definidas transfere-se para o Prolog de forma quase imediata.

É um ponto importante explicitar o significado da variável anônima e do uso do mesmo átomo em papéis diferentes. Intuitivamente, essas facilidades sintáticas devem ser entendidas da seguinte forma:

- Cada ocorrência da variável anônima representa a ocorrência de uma nova variável.
- Um mesmo átomo em dois papéis diferentes representa símbolos funcionais ou predicativos diferentes.

Definição 2.19:

Sejam \mathbf{P} e \mathbf{C} um programa e uma consulta Prolog, respectivamente. A *regularização* de \mathbf{P} e \mathbf{C} são o programa \mathbf{R} e a consulta \mathbf{D} resultantes de se aplicar as seguintes transformações a \mathbf{P} e \mathbf{C} :

- i. Substitua cada ocorrência da variável anônima por uma nova variável.

- ii. Se p é um átomo que ocorre em mais de um papel, substitua todas as ocorrências de p no mesmo papel por um novo átomo. Repita este passo até que todos os átomos ocorram em um único papel.

Note que **R** e **D** não são evidentemente únicos, mas é imediato mostrar que:

- **R** e **D** são realmente um programa e uma consulta Prolog, respectivamente.
- Nenhuma variável anônima ocorre em **R** ou **D**.
- Nenhum átomo ocorre em **R** ou **D** em mais de um papel.

Exemplo:

Seja **P** o seguinte programa Prolog:

1. chama(a, b).
2. usa(b, e).
3. depende(X, Y) :- chama(X, Y).
4. depende(X, Y) :- usa(X, Y).
5. depende(X, Y) :- chama(X, Y) & depende(X, Y).
6. depende(X, Y, Z) :- chama(X, Z) & depende(Z, Y).

e **C** a seguinte consulta Prolog:

7. :- depende(X, b, e).

A regularização de **P** e **C** resulta no programa **R**:

1. chama(a, b).
2. usa(b, e).
3. dep1(X, Y) :- chama(X, Y).
4. dep1(X, Y) :- usa(X, Y).
5. dep1(X, Y) :- chama(X, Z) & dep1(Z, Y).
6. dep2(X, Y, Z) :- chama(X, Z) & dep1(Z, Y).

e na consulta D:

7. :- dep2(X, b, e).

Note que **R** e D forma obtidos de **P** e C substituindo todas as ocorrências de “depende” no papel de um símbolo predicativo binário por “dep1” e todas as ocorrências de “depende” no papel de um símbolo predicativo ternário por “dep2” (“dep1” e “dep2” foram escolhidos arbitrariamente). Note ainda que esta segunda substituição afeta tanto a cláusula 6 do programa quanto a cláusula 7 exprimindo a consulta.

Definição 2.20:

Sejam **P** e C um programa e uma consulta Prolog, respectivamente, e **R** e D uma regularização de **P** e C. Um programa em cláusulas definidas **T** e uma consulta E a **T** são *equivalentes* a **P** e C se e somente se:

- i. Uma cláusula unitária Prolog “A.” ocorre em **R** se e somente se existe uma cláusula “A :- ” em **T**.
- ii. Uma cláusula não-unitária Prolog “A :- B₁ & ... & B_n.” ocorre em **R** se e somente se existe uma cláusula “A :- B₁ ... B_n” em **T**.
- iii. D é da forma “:- C₁ & ... & C_m.” se e somente se E for da forma “C₁ ∧ ... ∧ C_m”.

Note que:

- A definição exige uma regularização prévia de **P** e C para eliminar variáveis e átomos em mais de um papel.
- **T** é uma simples transformação sintática de **R**.
- D é a representação clausal da negação de E.

Sob a ótica operacional, a máquina Prolog é um procedimento de refutação particular baseado em resolução-LSD tal que:

- O literal selecionado de cada cláusula é sempre o mais à esquerda, ou seja, a função de seleção utilizada é a padrão.
- As cláusulas do programa são selecionadas na ordem em que ocorrem no programa.

Os compiladores para o Prolog que seguem estas duas estratégias de seleção são chamados de *padrão*.

A máquina aceita como entrada um programa Prolog **P** e uma consulta Prolog **C**, e produz como saída *falha* ou *sucesso*. No segundo caso, a saída também inclui uma substituição θ para **C**. A máquina comporta-se de tal forma que, se a saída for *falha*, não há respostas corretas de **C** a **P** e, se a saída for *sucesso*, θ é uma resposta correta de **C** e **P**. Mas a máquina poderá divergir e não produzir qualquer saída, tanto no caso de haver uma resposta correta de **C** a **P**, quanto no caso contrário. No entanto a máquina pode, também, ser modificada para produzir respostas corretas alternativas.

A máquina inicialmente modifica o programa **P** e a consulta **C**, eliminando as variáveis anônimas e os átomos ocorrendo em mais de um papel. Como visto anteriormente, esse passo é chamado de regularização. Portanto, ignorando pequenas variações sintáticas, a resolução-LSD pode ser aplicada diretamente ao conjunto de cláusulas Prolog resultante da regularização. Em seguida, a máquina simula o caminho em pré-ordem em uma árvore de refutação particular para as cláusulas resultantes da regularização, conforme ilustrado a seguir.

Exemplo:

Seja **P** o seguinte programa Prolog:

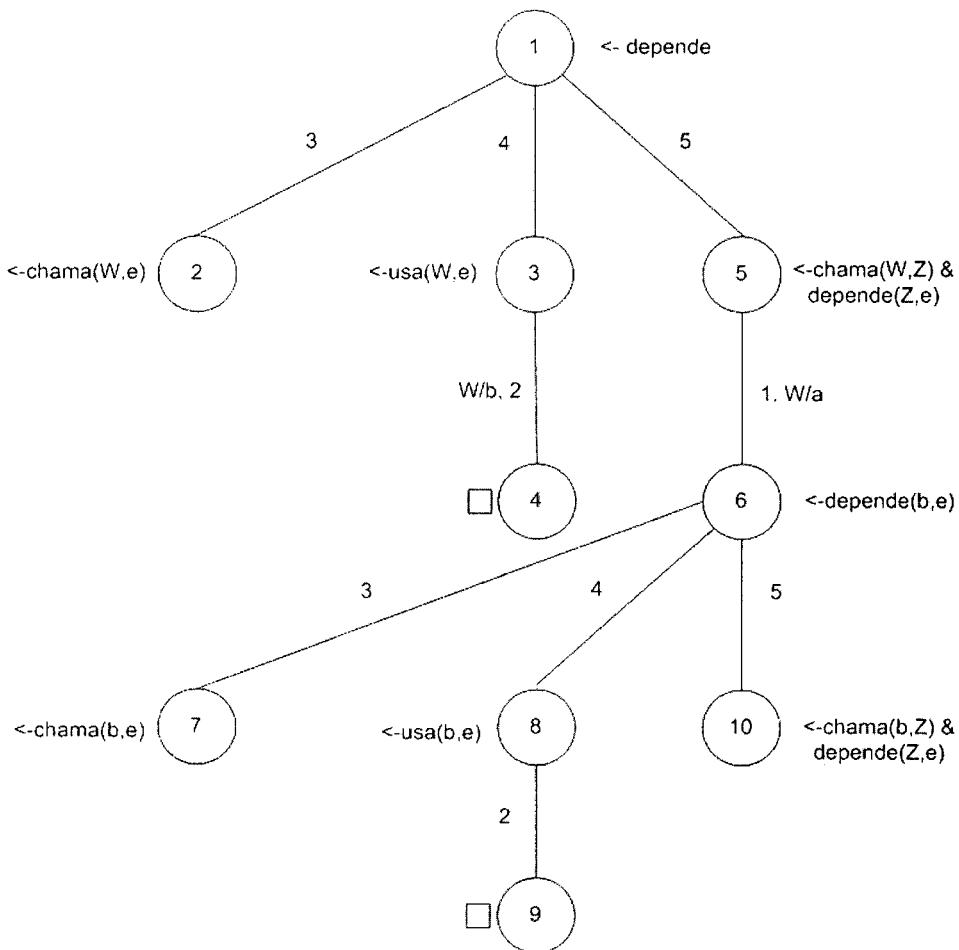
1. chama(a, b).
2. usa(b, e).
3. depende(X, Y) :- chama(X, Y).
4. depende(X, Y) :- usa(X, Y).
5. depende(X, Y) :- chama(X, Y) & depende(Z, Y).

e **C** a seguinte consulta Prolog:

6. :- depende(W, e).

Note que **P** e **C** já estão regularizados.

A árvore de refutação cujo caminhamento a máquina Prolog simula será a seguinte:



Esta é uma árvore de refutação por resolução-LSD construída de acordo com a seguinte estratégia:

- rotula-se a raiz com a consulta C ;
- para cada nó N da árvore, com rótulo “:- L_1 & ... & L_m .”:
 - seleciona-se o literal L_1 mais a esquerda;
 - para cada cláusula do programa “ M :- M_1 & ... & M_n .”, na ordem em que ocorre no programa e renomeando preliminarmente as variáveis, se necessário, se a cabeça M unifica com L_1 , tendo θ como unificador mais geral, gera-se um novo filho de N rotulado com a cláusula objetivo “:- (M_1 & ... & M_n & L_1 & ... & L_m) θ .”

Para facilitar o entendimento desta estratégia, os rótulos das arestas indicam as cláusulas do programa que geraram cada filho. Por exemplo, a cláusula 3 gerou o nó 2.

A máquina simula, então, o caminho em pré-ordem desta árvore. Novamente, para facilitar o entendimento, os rótulos dos nós indicam a ordem em que a máquina os visita.

A máquina retorna *sucesso* assim que detecta o primeiro (em pré-ordem) nó de sucesso, que, neste exemplo, é o nó 4. Como a consulta possui a variável W , a saída inclui também a substituição $\{W/b\}$. Note que, de fato, esta substituição é uma resposta correta de C a \mathbf{P} , pois \mathbf{P} implica logicamente “depende(b, e)”.

A máquina pode ser modificada para continuar a pesquisa por outro nó de sucesso, que, neste caso, será o nó 9, retornando a saída *sucesso* acompanhada da substituição $\{W/a\}$. De novo, note que esta substituição é uma resposta correta de C a \mathbf{P} , pois implica logicamente “depende(a, e)”.

A máquina mantém, essencialmente:

- A *cláusula objetivo corrente* (O).
- O *número da cláusula de entrada corrente* (N), que é o número da última cláusula de entrada resolvida contra a cláusula objetivo corrente.
- A *substituição corrente* (θ), que é a composição de todas as substituições efetuadas para gerar a cláusula objetivo corrente.

A especificação da máquina Prolog segue, intuitivamente, a seguinte estratégia:

1. Inicialmente, O é igual à consulta C ; N é 0, indicando que nenhuma cláusula do programa foi resolvida contra C ; e θ é a lista de substituições simples da forma X/X , onde X é uma variável que ocorre em C .
2. Selecionando para a unificação o literal L mais à esquerda de O , a máquina deriva, através de um *passo de avaliação*, um novo valor para O . Esta derivação se dá pela unificação de L com a cabeça M da primeira cláusula D que ocorre em \mathbf{P} e que não foi utilizada até então para este valor de O . Ou seja, D é a primeira cláusula que ocorre em \mathbf{P} após a cláusula de ordem N . Note que uma renomeação preliminar das variáveis de D pode ser necessária. Os valores correntes de O , N e θ são

armazenados para utilização posterior no passo de retrocesso. O novo valor de O é a cláusula formada substituindo-se L pelo corpo de D e aplicando-se o unificador mais geral utilizado, digamos, φ ; o novo valor de N será 0 ; e o novo valor de θ é a composição do valor anterior com φ .

3. Se o novo valor de O for a cláusula vazia, então a máquina Prolog retorna *sucesso*, junto com a composição de todas as substituições efetuadas sobre as variáveis de C , que é o valor de θ .
4. Se não for possível unificar L com a cabeça de alguma cláusula de \mathbf{P} , o *passo de retrocesso* é automaticamente ativado. Retorna-se ao passo (2), considerando-se como valor de O , N e θ os seus valores anteriores.
5. Se o passo de retrocesso atingiu novamente a consulta C e o literal não puder ser unificado com a cabeça de mais nenhuma cláusula de \mathbf{P} , a máquina Prolog pára e retorna *falha*.

Capítulo 3

Noções de CCS

3.1. Definições Básicas

As definições do CCS foram escritas baseadas no trabalho de Milner [8, 9, 12]. O CCS (*Calculus for Communicating Systems*) define comunicação e concorrência como noções complementares essenciais para a compreensão de sistemas distribuídos complexos. Por um lado, tal sistema tem diversidade, sendo composto por várias partes, cada uma atuando simultaneamente com, e independentemente de outras partes; por outro lado, um sistema complexo tem unidade que é alcançada através da comunicação entre suas partes.

A base dessas noções está no fato de que se supõe que cada uma das várias partes de tal sistema, chamadas *agentes*, têm identidade própria, que persiste ao longo do tempo. De fato, sem esta suposição, seríamos capazes de fazer apenas distinções entre os eventos de comportamento do sistema. Se desejarmos identificar um evento particular, temos de identificar os agentes que participam deste evento, o que significa

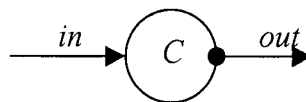
determinar *onde*, isto é, em que parte ou partes o evento ocorre.

Cada ação de um agente ou é uma interação com os seus agentes vizinhos e, portanto, uma *comunicação*, ou ocorre de forma independente deles podendo ocorrer *simultaneamente* com suas ações. Frequentemente as ações independentes de um agente não são nada além de comunicações entre os componentes desse mesmo agente.

Uma parte essencial de uma teoria de sistemas complexos é uma noção precisa de *comportamento*, que é a capacidade total de comunicação do sistema. Ou seja, o comportamento de um sistema é exatamente o que é observável, e observar um sistema é comunicar-se com ele.

A comunicação entre os agentes do CCS é feita por meio de portas que funcionam como canais de comunicação. Assim, existem portas de comunicação com o meio através das quais dados são transmitidos do meio para o sistema (portas de entrada) ou do sistema para o meio (portas de saída). Também existem portas de comunicação entre os agentes, ou seja, que fazem a comunicação interna do sistema.

Vamos agora ilustrar o funcionamento do CCS através de exemplos simples. Considere um agente C como uma célula que pode guardar um certo dado:



O diagrama mostra que a célula tem duas *portas*, mas não define o comportamento da célula. Supomos então que, quando vazia, C pode aceitar um valor na porta de entrada in e, quando armazenado um valor, pode transmiti-lo pela porta out . Dessa forma, podemos definir o comportamento de C da seguinte forma:

$$C =_{def} in(x).C'(x)$$
$$C' =_{def} \overline{out(x)}.C$$

É preciso notar aqui três coisas importantes:

- O prefixo “ $in(x)$.” significa um *handshake* em que um valor é recebido na porta in e é assumido como valor da variável x .

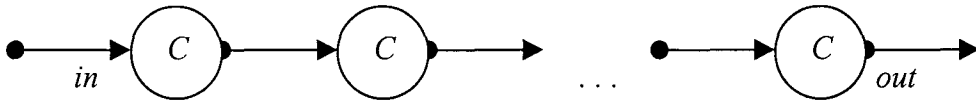
- $in(x).C'(x)$ é uma expressão de agente e seu comportamento é realizar o *handshake* descrito e então proceder conforme a definição de C' .
- $\overline{out}(x).C$ é uma expressão de agente e seu comportamento é fazer a saída do valor x na porta \overline{out} e então proceder de acordo com a definição de C .

Note que a definição auxiliar de C' é apenas uma conveniência, de forma que o comportamento de C pode ser definido por uma única equação:

$$C =_{def} in(x).\overline{out}(x).C$$

Podemos entender as expressões de agentes C e C' como representações de diferentes possibilidades de *estados* de um agente. Naturalmente, um agente poderá assumir muitos estados diferentes.

Tomemos agora n cópias de C



Pode-se definir esse diagrama como

$$C^{(n)} =_{def} C \wedge C \wedge \dots \wedge C$$

Observe que $C^{(n)}$ tem somente duas portas externas, sendo todas as demais internas.

É fácil notar que $C^{(n)}$ comporta-se como um *buffer* de capacidade n , isto é, definimos $Buff_n$ como uma especificação e concluímos que $Buff_n = C^{(n)}$. O símbolo de igualdade nessa expressão quer dizer “comportamento semelhante”.

Definimos $Buff_n(s)$, onde o parâmetro s pode ser qualquer seqüência $\langle v_1, \dots, v_k \rangle$ de valores k , $0 \leq k \leq n$, como a seguir

$$Buff_n \langle \rangle =_{def} in(x).Buff_n \langle n \rangle$$

$$Buff_n \langle v_1, \dots, v_k \rangle =_{def} \overline{out(v_n)}.Buff_n \langle v_1, \dots, v_{n-1} \rangle$$

$$Buff_n \langle v_1, \dots, v_k \rangle =_{def} in(x).Buff_n \langle x, v_1, \dots, v_k \rangle + \overline{out(v_k)}.Buff_n \langle v_1, \dots, v_{k-1} \rangle$$

com $0 < k < n$.

Nessa definição usamos um outro combinador básico, “+”, chamado *Soma*. O agente $P + Q$ comporta-se como P ou como Q . Tão logo ocorra a primeira ação de um, o outro é descartado.

Vamos considerar agora os agentes A e B ,



onde a, b e c são nomes distintos.

Os agentes A e B são definidos da seguinte forma:

$$A =_{def} a.A'$$

$$A' =_{def} \bar{c}.A$$

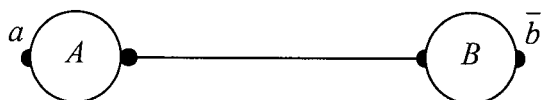
$$B =_{def} c.B'$$

$$B' =_{def} \bar{b}.B$$

Podemos construir a *Composição* $A|B$



Se tornarmos as portas c e \bar{c} exclusivas da composição $A|B$, podemos omitir seus nomes e chamamos a isso de *Restrição* de c com relação a $A|B$. Este é outro importante operador do CCS que é representado por $\backslash x$. No exemplo dado, a restrição em c é representada por $A|B \backslash c$. Podemos representar graficamente da seguinte maneira:



onde podemos notar que a comunicação entre A e B é feita por um link de comunicação exclusivo, no caso a porta c . Observe que o nome da porta “ c ”, não aparece mais discriminado na representação gráfica, pois agora trata-se de um link de comunicação *interno*.

3.2. Sincronismo

A descrição de sincronismo em CCS foi escrita baseada em Milner [10, 12]. A idéia básica de sincronismo apresentada no CCS é baseada em *handshakes*, uma ação indivisível em que um dado é simultaneamente emitido por um agente e recebido por outro.

Por outro lado, existem também aplicações para uma comunicação em que nenhum dado é transmitido, ou seja, apenas o sinal de que alguma comunicação foi feita é transmitido (*ackin*, *ackout*).

No entanto, essa primeira visão de sincronismo não parece suficiente, uma vez que desejamos descrever sistemas cujo comportamento futuro depende da informação que foi recebida. Uma forma apropriada de expressar essa dependência pode ser através de uma expressão condicional, cuja condição contém variáveis que esperam por valores de entrada. Isso que nos leva a concluir que tal variável deve aparecer anteriormente como um parâmetro em um prefixo de entrada, representando a comunicação do tipo “passagem de valores”.

3.3. Ação e Transição

Considere um conjunto infinito A de *nomes* e use a, b, c, \dots para referenciar elementos de A . Exemplos de nomes podem ser *geth*, *ackin* etc., mas muitas vezes utilizaremos nomes genéricos como a, b, c, \dots . Denotamos por \bar{A} o conjunto de *co-nomes* como $\overline{geth}, \overline{ackin}, \bar{a}, \bar{b}, \bar{c}, \dots$ para referenciar elementos de \bar{A} . Então, temos $L = A \cup \bar{A}$ o conjunto de *rótulos* (rótulos de portas) e usamos l, l' para referenciar elementos de L . Entendemos que $\bar{\bar{a}} = a$.

O conjunto L compreende quase todas (mas não todas!) as ações que os agentes podem desempenhar.

Uma vez tendo declarado que os agentes são identificados com estados, e uma vez que a transição de um estado para outro é alcançada por uma ação, é razoável escrever tal transição como

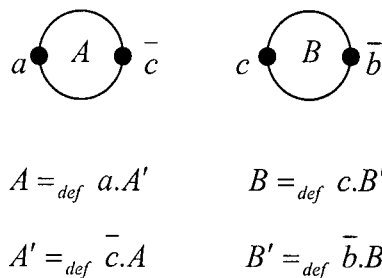
$$P \xrightarrow{l} Q$$

Por exemplo, se escrevemos

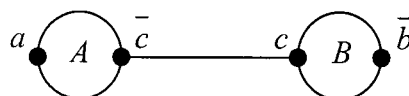
$$\text{martelo} \xrightarrow{\text{pegar}} \text{martelo_em_uso}$$

entendemos que a ação *pegar* altera o estado do agente *martelo* (livre) para *martelo_em_uso* (ocupado), definindo, portanto, o comportamento de *martelo*. De fato, podemos dizer que as transições definem o significado do combinador Prefixo ‘pegar.’ e nosso objetivo é definir o significado de todos os combinadores em termos de transições.

Para definirmos o significado de Composição, precisaremos determinar quais as transições possíveis para um agente composto $P \mid Q$, em termos das possíveis transições de P e Q separadamente. Por exemplo, considere os agentes A e B dados da seguinte maneira:



Agora, considere o agente composto



Nossa primeira regra de transição diz que se A pode fazer uma ação sozinho, então ele pode fazer essa ação no contexto $A \mid B$, sem interferir em B (igualmente, B executa uma ação sem interferir em A). Assim,

Uma vez que $A \xrightarrow{a} A'$, inferimos $A | B \xrightarrow{a} A' | B$

Também, uma vez que a porta \bar{c} de A é ligada à porta c de B , nossa regra nos diz que

Uma vez que $A' \xrightarrow{\bar{c}} A$, inferimos $A' | B \xrightarrow{\bar{c}} A | B$

Isso *não* representa uma comunicação entre A' e B ; em vez disso, representa a possibilidade de A' se comunicar com um terceiro agente – ainda não fornecido – através da porta \bar{c} , ainda sem interferir em B .

Então, como podemos representar a comunicação *handshake* que deveria ser inferida das ações $A' \xrightarrow{\bar{c}} A$ e $B \xrightarrow{c} B'$? Lembrando que uma comunicação *handshake* consiste de ações simultâneas de ambas as partes, esperamos que exista outra regra de transição, que, nesse caso particular, deveria dizer:

Uma vez que $A' \xrightarrow{\bar{c}} A$ e $B \xrightarrow{c} B'$, inferimos $A' | B \xrightarrow{?} A | B'$

Isso incorpora a idéia de que A e B mudam de estado simultaneamente – a Composição sendo preservada – mas o que escreveremos no lugar de “?” ?

A resposta é uma das mais importantes decisões no projeto do *calculus*. Temos de perceber que “?” representa uma ação *completa* ou *perfeita* para o agente composto $A' | B$ e, mais que isso – já que a ação é interna para o agente composto – a mesma ação perfeita surge de qualquer par (b, \bar{b}) de ações complementares dos componentes do agente composto. Denotaremos essa ação perfeita por τ , que representará todos os *handshakes* internos. Afirmamos anteriormente que o conjunto L de rótulos não compreende todas as ações que um agente pode executar; de fato, τ é a única ação extra que precisávamos. Portanto, temos que $Ações = L \cup \{\tau\}$, o conjunto de todas as ações possíveis, e usaremos α, β, \dots para referenciar elementos de $Ações$. (A ação τ não tem complemento.) Assim, no nosso exemplo, deduziremos de nossa segunda regra que

Uma vez que $A' \xrightarrow{\bar{c}} A$ e $B \xrightarrow{c} B'$, inferimos $A' | B \xrightarrow{\tau} A | B'$

Nosso objetivo, ao analisar o comportamento de sistemas compostos, é ignorar, tanto quanto possível, suas ações internas (perfeitas); τ (não tendo complemento) não representa uma comunicação em potencial e, portanto, não é diretamente observável. Queremos considerar dois sistemas equivalentes se eles apresentarem o mesmo (em um certo sentido) padrão de ações *externas*. Isso equivale a abstrair de um sistema apenas aspectos externos de seu comportamento, o que é relevante quando ele ocorre como um componente de um sistema ainda maior, fazendo com que a seqüência

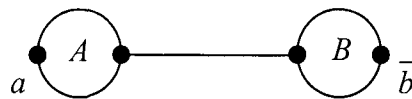
$$P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} P_n$$

de ações internas seja equivalente a uma única ação interna

$$P \xrightarrow{\tau} P_n$$

permitindo-nos uma considerável simplificação, via equações algébricas apropriadas, da expressão para o agente P .

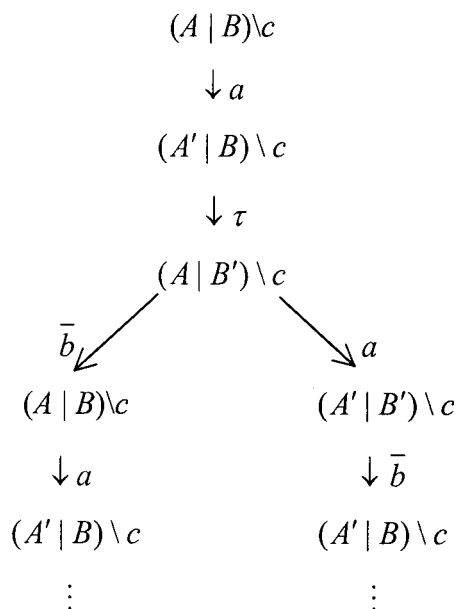
Retornando ao nosso exemplo, vamos discutir o efeito de se impor a restrição $\setminus c$ sobre $A \mid B$. (Abreviaremos a restrição única $\setminus \{c\}$ para $\setminus c$.) O grafo de fluxo para $(A \mid B) \setminus c$ é



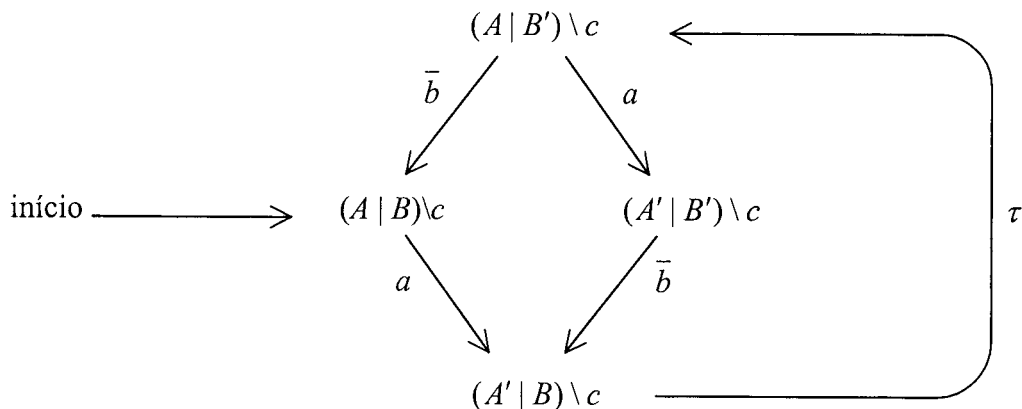
onde as portas c e \bar{c} foram omitidas, significando que o agente composto $(A \mid B) \setminus c$ não executa ações em c e \bar{c} , embora – crucialmente – ele possa executar uma ação τ que resulta de uma comunicação (c, \bar{c}) entre seus componentes. De fato, a regra geral para restrição é

$$\text{Se } P \xrightarrow{\alpha} P', \text{ então } P \setminus L \xrightarrow{\alpha} P' \setminus L \text{ nos diz que } \alpha, \bar{\alpha} \notin L$$

Podemos agora estabelecer todas as transições que podem ocorrer em um sistema $(A \mid B) \setminus c$, o que nos permite construir as *árvores de transição* ou *árvores de derivação* [11, 12]:



Podemos ver que a árvore repete seus passos. De fato, podemos resumir o comportamento do sistema através do grafo de transição



A partir disso, tomando as definições dos agentes C_0, \dots, C_3 a seguir, podemos ver que $(A \mid B) \setminus c$ tem comportamento idêntico ao do agente C_1 ,

$$\begin{aligned}
C_0 &=_{def} \bar{b}.C_1 + a.C_2 \\
C_1 &=_{def} a.C_3 \\
C_2 &=_{def} \bar{b}.C_3 \\
C_3 &=_{def} \tau.C_0
\end{aligned}$$

Alternativamente, podemos dizer que

$$(A \mid B) \setminus c = a.\tau.C, \text{ onde } C =_{def} a.\bar{b}.\tau.C + \bar{b}.a.\tau.C$$

Essa argumentação não está ainda apropriada para suportar definições precisas; estamos apenas descrevendo informalmente o significado dos combinadores, faltando dizer qual o significado do comportamento com respeito à igualdade. Nossa suposição de que $(A \mid B) \setminus c = C_1$ será apenas justificada se o significado do comportamento de um agente for determinado por sua árvore de derivação, ignorando a natureza sintática das expressões nos nós da árvore. Mas o nosso exemplo ilustra algo que frequentemente é útil – a saber, equiparar o comportamento de um sistema composto a um comportamento definido sem o uso do operador de Composição ‘|’ ou o de Restrição que frequentemente acompanha o primeiro.

Quando tivermos definido o comportamento de igualdade, nós estaremos aptos a provar uma importante lei equacional, $\alpha.\tau.P = \alpha.P$, que permite muitas ocorrências de τ serem eliminadas. Indicamos anteriormente que desejamos abstrair dos detalhes das comunicações internas e essa lei é um meio pelo qual faremos essa abstração. Com isso, podemos deduzir finalmente que

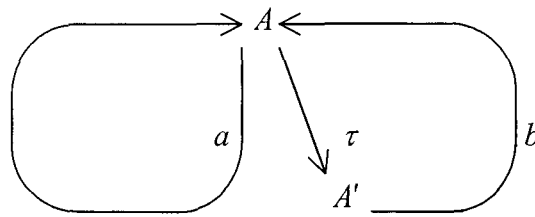
$$(A \mid B) \setminus c = a.D, \text{ onde } D =_{def} a.\bar{b}.D + \bar{b}.a.D$$

3.4. O Poder das Ações Internas

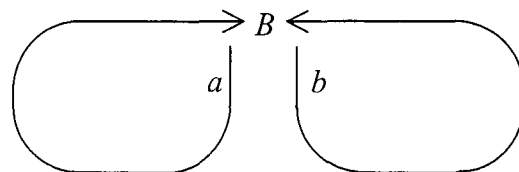
Vamos agora mostrar outras propriedades das ações τ , nas quais tais ações não podem ser sempre ‘ocultadas’. Considere um agente simples definido recursivamente por

$$A =_{def} a.A + \tau.b.A$$

cujo grafo de transição é



Se permitirmos ‘ocultar’ a ação τ – isto é, se a equação $\tau.P = P$ for sempre válida – nós estaríamos admitindo que $A = B$, onde $B =_{def} a.B + b.B$:



Entretanto, assumindo que $a \neq b$, temos fortes bases intuitivas para não admitirmos $A = B$! B é um agente que pode executar cada uma das ações a e b , qualquer que seja o estado que ele atinja. A , por outro lado, pode atingir (via τ) um estado A' em que a ação b é possível, enquanto a é impossível. Uma vez que τ é uma ação interna de A – o resultado de uma comunicação entre dois de seus componentes se analisarmos essa composição – essa ocorrência é autônoma no sentido de que não precisa de participação externa; assim, A é incontrolável, ou não-determinístico, enquanto B é perfeitamente controlável e determinístico.

Se acreditamos que essa intuição captura uma propriedade que está presente em sistemas de comunicação real, então devemos admitir que a equação $\tau.P = P$ é, em geral, inválida. Também temos de admitir que não podemos concluir que dois agentes são iguais em um sentido comportamental meramente a partir do fato de que eles podem desempenhar a mesma seqüência de ações externas (tal como A e B no exemplo anterior). Assim, somos forçados a abandonar – ou pelo menos a refinar – a teoria de autômatos clássica, já que, naquela teoria, duas máquinas são tidas como equivalentes se e somente se elas podem executar a mesma seqüência de ações.

Estamos preparando o caminho para uma definição precisa de nosso calculus, em termos de transições de estados. Como pudemos perceber, é importante se Ter um

bom tratamento da natureza interna da comunicação entre os componentes de um sistema.

3.5. A Linguagem Básica

Nesta seção descreveremos a sintaxe de nosso calculus básico. Na Seção 3.3, introduzimos os *nomes* A , os *co-nomes* \bar{A} e os *rótulos* $L = A \cup \bar{A}$. Lembre-se que a, b, c, \dots referem-se a elementos de A , $\bar{a}, \bar{b}, \bar{c}, \dots$ referem-se a elementos de \bar{A} e que l, l' referem-se a elementos de L . Também introduzimos a *ação silenciosa* ou *perfeita* τ , e definimos $Ação = L \cup \{\tau\}$ como o conjunto das ações, com α, β, \dots referenciando elementos deste conjunto.

Usaremos R para representar subconjuntos de L e chamaremos \bar{R} o conjunto dos complementos dos rótulos em R . Uma *função de renomeação* f , com $f: R \rightarrow R$, tal que $f(\bar{l}) = \overline{f(l)}$; também estendemos f para o conjunto $Ação$ afirmando que $f(\tau) = \tau$.

Além disso, temos um conjunto de *agentes*, representados por A, B, \dots , que entenderemos como predicados.

Definimos um conjunto de expressões de agentes da seguinte forma:

1. $\alpha(x).E(x)$, um *Prefixo* ($\alpha \in Ação$)
2. $\sum_{i \in I} E_i(x)$, uma *Soma* (I um conjunto de índices)
3. $E_1(x) | E_2(x)$, uma *Composição*
4. $E(x) \setminus L$, uma *Restrição* ($R \subseteq L$)
5. $E(x)[f]$, uma *Renomeação* (f uma função de renomeação)

A expressão (2) quer dizer a soma de todos os E_i , com $i \in I$, e podemos também escrever como $\sum\{E_i(x): i \in I\}$, ou, abreviadamente, $\sum_i E_i$ quando I é conhecido. No lugar de (2), poderíamos ter escrito uma Soma como $E_1(x) + E_2(x)$; de fato, isso é como escreveríamos a Soma quando $I = \{1, 2\}$. Mais frequentemente usamos Soma binária, mas, por razões teóricas, algumas vezes necessitaremos usar uma soma infinita – isto é, o caso em que I é infinito. Outro caso especial é quando $I = \emptyset$, o conjunto vazio; isso

nos dá um agente inativo, incapaz de executar qualquer ação. Isso é tão importante que daremos um nome especial, 0, que definimos por

$$0 =_{def} \sum_{i \in \emptyset} E_i(x)$$

Para evitar muitos parênteses, adotaremos a convenção de que os combinadores têm poder de ligação decrescente, da seguinte forma: Restrição e Renomeação, Prefixação, Composição, Soma. Assim, por exemplo,

$$R + a.P | b.QR \quad \text{significa} \quad R + ((a.P) | (b.(QR)))$$

Uma Constante é um agente cujo comportamento é dado por uma equação de definição. De fato, assumiremos que para *toda* constante A existe uma equação de definição da forma

$$A =_{def} P$$

Considere, por exemplo, as duas equações de definição $A =_{def} a.A'$ e $A' =_{def} \bar{c}.A$; elas ilustram que as Constantes podem ser definidas em termos umas das outras – em outras palavras, por recursão mútua. Este mecanismo de definição é o único com que agentes com comportamento infinito podem ser definidos no calculo como temos apresentado.

Usaremos sempre $E_1 \equiv E_2$ para mostrar que as expressões E_1 e E_2 são *sintaticamente idênticas*.

3.6. Semântica Transicional

Dado o significado de nossa linguagem básica, usaremos a noção geral de um *sistema de transição rotulado*

$$(S, T, \{\xrightarrow{t} : t \in T\})$$

que consiste de um conjunto S de estados, um conjunto T de transições e uma *relação de transição* $\xrightarrow{t} \subseteq S \times S$ para cada $t \in T$.

Definiremos as transições de cada agente composto em termos das transições de seu(s) agente(s) componente(s). Anteriormente, indicamos que $A \xrightarrow{\alpha} A'$ inferia $A | B \xrightarrow{\alpha} A' | B$; a regra geral que permite essa inferência é

$$\text{De } E \xrightarrow{\alpha} E' \text{ inferimos } E | F \xrightarrow{\alpha} E' | F$$

E escreveremos isso da seguinte maneira:

$$\frac{E \xrightarrow{\alpha} E'}{E | F \xrightarrow{\alpha} E' | F}$$

Existirão uma ou mais regras de transição associadas a cada combinador e uma associada às constantes. Cada regra terá uma *conclusão* e zero ou mais *hipóteses*. Em uma regra associada com um combinador, a conclusão será uma transição de uma expressão de agente consistindo de um combinador aplicado a um ou mais componentes, e a hipótese será a transição de algum(ns) dos componentes. O conjunto de regras associadas com cada combinador será compreendida como tendo o significado do combinador; a regra para constantes afirma que cada constante tem as mesmas transições de suas expressões de definição.

Veremos agora um conjunto completo de regras de transição; os nomes Ação, Soma, Comp, Rest, Reno e Const, indicam que as regra são associadas com Prefixação, Soma, Composição, Restrição, Renomeação e Constantes, respectivamente.

$$\text{Ação} \quad \frac{}{\alpha.E \xrightarrow{\alpha} E}$$

$$\text{Soma}_j \quad \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'_j} \quad (j \in I)$$

$$\text{Comp}_1 \quad \frac{E \xrightarrow{\alpha} E'}{E | F \xrightarrow{\alpha} E' | F}$$

$$Comp_2 \frac{F \xrightarrow{\alpha} F'}{E | F \xrightarrow{\alpha} E | F'}$$

$$Comp_3 \frac{E \xrightarrow{i} E' \quad F \xrightarrow{i} F'}{E | F \xrightarrow{\tau} E' | F'}$$

$$Rest \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L} \quad (\alpha, \bar{\alpha} \notin L)$$

$$Reno \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$Const \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A =_{def} P)$$

A soma finita, que é suficiente para muitos propósitos práticos, pode ser apresentada de uma forma mais conveniente. Se $I = \{1, 2\}$, então obtemos duas regras para $E_1 + E_2$, com $j = 1, 2$:

$$\frac{E_1 \xrightarrow{\alpha} E'_1}{E_1 + E_2 \xrightarrow{\alpha} E'_1} \quad \frac{E_2 \xrightarrow{\alpha} E'_2}{E_1 + E_2 \xrightarrow{\alpha} E'_2}$$

Também, lembremos que $0 =_{def} \sum_{i \in \emptyset} E_i$; já que $I = \emptyset$ neste caso, não existem regras para 0, e isso reflete o fato de 0 não ter transições.

Dissemos que nosso conjunto de regras está completo; com isso queremos dizer que não existem transições exceto aquelas que podem ser inferidas pelas regras. Podemos agora expor a justificativa para uma transição de qualquer expressão de agente na forma de um diagrama de inferência onde anotamos cada inferência com o nome da regra que a justifica. Por exemplo, a justificativa da transição

$$((a.E + b.0) | \bar{a}.F) \setminus a \xrightarrow{\tau} (E | F) \setminus a$$

é dada por

$$\begin{array}{c}
\text{Ação} \frac{}{\alpha.E \xrightarrow{\alpha} E} \\
| \\
\text{Soma}_1 \frac{}{\alpha.E + b.0 \xrightarrow{\alpha} E} \qquad \text{Ação} \frac{}{\bar{a}.F \xrightarrow{\bar{a}} F} \\
\swarrow \qquad \searrow \\
\text{Comp}_3 \frac{}{(a.E + b.0) | \bar{a}.F \xrightarrow{\tau} E | F} \\
| \\
\text{Rest} \frac{}{((a.E + b.0) | \bar{a}.F) \setminus a \xrightarrow{\tau} (E | F) \setminus a}
\end{array}$$

De fato, temos dadas agora as duas únicas transições possíveis para $((a.E + b.0) | \bar{a}.F) \setminus a$; podemos checar isso tentando encontrar todos os diagramas de inferência possíveis da seguinte forma

$$\frac{}{(a.E + b.0) | \bar{a}.F \setminus a \xrightarrow{\tau} ?}$$

O último passo só pode ser *Rest* e, portanto, procederemos da seguinte forma

$$\begin{array}{c}
\vdots \\
\frac{}{(a.E + b.0) | \bar{a}.F \xrightarrow{\alpha} G} \\
| \\
\text{Rest} \frac{}{((a.E + b.0) | \bar{a}.F) \setminus a \xrightarrow{\alpha} G \setminus a}
\end{array}$$

onde α e G são ainda desconhecidos, mas α não pode ser a ou \bar{a} . Assim, das transições possíveis, temos

$$(a.E + b.0) | \bar{a}.F \xrightarrow{\tau} E | F$$

o que nos leva ao nosso primeiro diagrama de inferência. Por este exemplo, mostramos como a restrição e a composição trabalham juntas para representar a comunicação interna via rótulos (a, \bar{a}) .

Como outro exemplo, considere como inferir a ação

$$(A | B) \setminus c \xrightarrow{a} (A' | B) \setminus c$$

Com as definições de A e B dadas na Seção 2. A inferência é

$$\begin{array}{c}
 \text{Ação} \quad \frac{}{a.A' \xrightarrow{a} A'} \\
 | \\
 \text{Const} \quad \frac{}{A \xrightarrow{a} A'} \\
 | \\
 \text{Comp} \quad \frac{}{A | B \xrightarrow{a} A' | B} \\
 | \\
 \text{Rest} \quad \frac{}{(A | B) \setminus c \xrightarrow{a} (A' | B) \setminus c}
 \end{array}$$

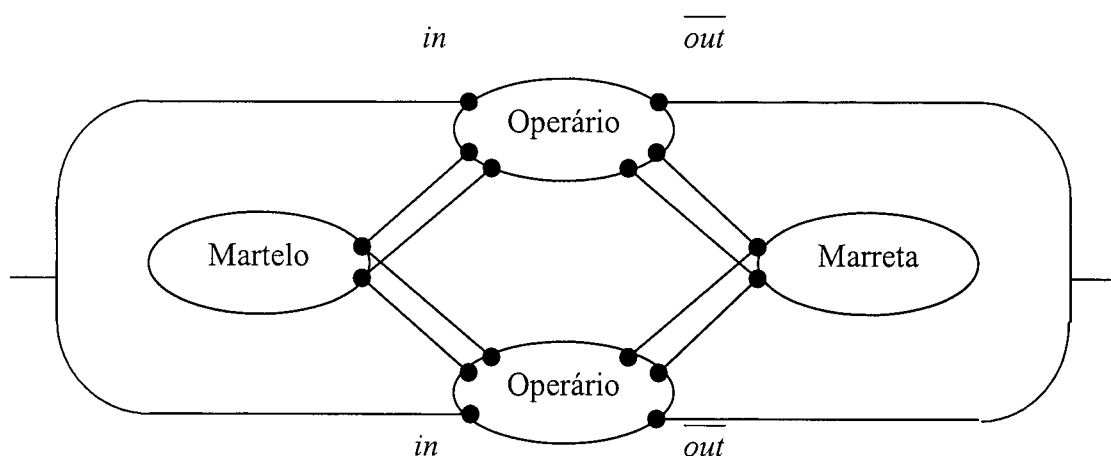
3.7. Um Exemplo: o *Jobshop*

Mostraremos agora como modelar uma linha de produção simples.

Suponha que duas pessoas compartilhem o uso de duas ferramentas – um martelo e uma marreta – para manufaturar objetos a partir de componentes simples. Cada *objeto* é feito introduzindo-se um pino de madeira em um bloco. Chamaremos o par constituído pelo pino e pelo bloco de *tarefa*. As tarefas chegam seqüencialmente

sobre uma esteira rolante e objetos prontos seguem adiante também sobre uma esteira rolante. O *jobshop* pode envolver qualquer número de pessoas, que chamaremos de *operários*, compartilhando mais ou menos ferramentas. Esses – os operários e as ferramentas – serão os agentes de nosso sistema e a forma que iremos modelar o sistema não dependerá de suas quantidades. Em contraste, as tarefas e os objetos serão os dados que entram e saem do sistema.

O sistema *jobshop* com dois operários, um martelo e uma marreta, pode ser ilustrado da seguinte maneira:



Primeiro descreveremos o comportamento de *Martelo*.

$$\begin{aligned} \text{Martelo} &=_{def} \text{pegar_martelo.Martelo_Ocupado} \\ \text{Martelo_Ocupado} &=_{def} \text{liberar_martelo.Martelo} \end{aligned}$$

Assim, o comportamento de *Martelo* é uma seqüência alternada infinita de ações de pegar (*pegar_martelo*) e de largar a ferramenta (*liberar_martelo*).

Para a marreta, temos:

$$\begin{aligned} \text{Marreta} &=_{def} \text{pegar_marreta.Marreta_Ocupada} \\ \text{Marreta_Ocupada} &=_{def} \text{liberar_marreta.Marreta} \end{aligned}$$

É óbvio que o comportamento de *Marreta* é idêntico ao de *Martelo*, apenas mudando-se *pegar/liberar_martelo* por *pegar/liberar_marreta*.

Vamos agora descrever o comportamento de *Operário* em termos de vários estados, por conveniência:

<i>Início(tarefa)</i>	O operário recebeu a tarefa
<i>Useferramenta(tarefa)</i>	Ele escolhe uma ferramenta para fazer a tarefa
<i>Usemartelo(tarefa)</i>	Ele escolhe o martelo para fazer o seu trabalho
<i>Usemarreta(tarefa)</i>	Ele escolhe a marreta para fazer o seu trabalho
<i>Fim(tarefa)</i>	A tarefa está pronta

A descrição de *Operário* é a seguinte:

$$\begin{aligned}
 \text{Operário} &=_{\text{def}} \text{in}(tarefa).\text{Início}(tarefa) \\
 \text{Início}(tarefa) &=_{\text{def}} \text{Useferramenta}(tarefa) \\
 \text{Useferramenta}(tarefa) &=_{\text{def}} \text{Usemartelo}(tarefa) + \text{Usemarreta}(tarefa) \\
 \text{Usemartelo}(tarefa) &=_{\text{def}} \overline{\text{pegar_martelo}}.\overline{\text{liberar_martelo}}.\text{Fim}(tarefa) \\
 \text{Usemarreta}(tarefa) &=_{\text{def}} \overline{\text{pegar_marreta}}.\overline{\text{liberar_marreta}}.\text{Fim}(tarefa) \\
 \text{Fim}(tarefa) &=_{\text{def}} \overline{\text{out_feita}(tarefa)}.\text{Operário}
 \end{aligned}$$

Vamos analisar um sistema simples constituído por um operário e uma ferramenta, por exemplo, o martelo. Teremos então a composição

$$\text{Operário} \mid \text{Martelo}$$

que evoluiria da seguinte forma:

$$\begin{aligned}
 &\text{Operário} \mid \text{Martelo} \\
 &\text{in}(tarefa).\text{Início}(tarefa) \mid \text{Martelo} \\
 &\text{Useferramenta}(tarefa) \mid \text{Martelo} \\
 &\text{Usemartelo}(tarefa) + \text{Usemarreta}(tarefa) \mid \text{Martelo} \\
 &\overline{\text{pegar_martelo}}.\overline{\text{liberar_martelo}}.\text{Fim}(tarefa) \mid \text{pegar_martelo}.\text{Martelo_Ocupado} \\
 &\overline{\text{liberar_martelo}}.\text{Fim}(tarefa) \mid \text{Martelo_Ocupado} \\
 &\overline{\text{liberar_martelo}}.\text{Fim}(tarefa) \mid \text{liberar_martelo}.\text{Martelo}
 \end{aligned}$$

