ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Verificação de condições de leis de programação em um sistema de re-escritura

Trabalho de Conclusão de Curso

Engenharia da Computação

Gabriel Ramos Falconieri Freitas
Orientador: Prof. Dr. Márcio Lopes Cornélio

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de Sistemas Computacionais

Verificação de condições de leis de programação em um sistema de re-escritura

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Gabriel Ramos Falconieri Freitas
Orientador: Prof. Dr. Márcio Lopes Cornélio

Recife, novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

Gabriel Ramos Falconieri Freitas

Verificação de condições de leis de programação em um sistema de re-escritura

Resumo

Mudanças em softwares são uma constante, tendo em vista manutenção tanto corretiva quanto evolutiva. Contudo, mudanças também podem ocorrer devido a necessidade de melhorar fatores de qualidade como, reuso e legilibilidade. Neste caso, tais mudanças devem manter o comportamento do software inalterado, modificando, contudo, sua estrutura interna, com o intuito de aprimorá-la. A fim de evitar a introdução de erros, à medida que a estrutura é modificada, alterações devem ser realizadas de maneira disciplinada por meio, por exemplo, do uso de leis de programação.

Neste trabalho implementamos a verificação de condições para aplicação de leis de programação, descritas para uma linguagem orientada a objetos semelhante a um subconjunto de Java seqüencial. Esta linguagem possui semântica formal definida, na qual as leis de programação foram provadas.

Para a implementação das condições das leis, utilizamos o sistema de re-escritura Maude. Descrevemos a gramática da nossa linguagem em Maude, assim como um parser que transforma um programa em uma árvore sintática. Utilizando a árvore sintática, realizamos as verificações de condições de leis de comando e de classes.

Abstract

Software changes constantly due to maintenance that leads to correction of fails or just to improve functionalities. However, some changes can take place due to other quality related needs such as code reuse or legibility. In this case, the changes may not alter the software behavior but only its internal structure, thus, making it better. To avoid errors due to modifications, every change has to be done in a following a discipline. Programming laws are a means to change software in a formal way.

In this work we have implemented the verification of conditions that must be satisfied to allow the application of programming laws. These laws are described using an object oriented language similar to a sequential Java subset. Our language has a formal semantics, that is used as a sound basis for proofs of laws.

To implement the verification of law conditions, we used the Maude rewritting system. We have described the grammar of our language in Maude. We use a parser that we have written in Maude that transforms programs into syntactic trees. We check conditions of programming laws of commands and classes by investiganting elements present in syntactic trees.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

ROOL – *Refinement Object-Oriented Language.*
EMC – Existência de método declarado em uma classe e na hierarquia.
EMP – Existência de chamada de método no programa.
EAC – Existência de atributo declarado em uma classe e na hierarquia.
EAP – Existência de chamada de atributo privado em uma classe.

Agradecimentos

Primeiramente gostaria de agradecer a Deus e ao meu guia Sri Sathya Sai Baba que me acompanhou desde o colegial até aqui. Agradeço também a minha mãe Lucia que sempre me aconselhou e me deu forças para estar aqui. A meu pai que com certeza está rezando por mim lá de cima e muito feliz em me ver onde estou hoje. A minha vó que rezou por tantos dias para eu ter forças para alcançar meus objetivos. Ao meu irmão Bruno que sempre esteve lado a lado comigo me ajudando e me dando o suporte necessário para eu ter tempo para minhas pesquisas. Ao meu padrasto Walter. E a minha noiva Gabriela por ter sempre me proporcionado felicidade e tranqulidade.

Gostaria de agradecer muito aos meus chefes Eveline, Genaro e em especial a Flávia por ter me entendido e me concedido dias preciosos para que eu pudesse estudar, realizar minhas pesquisas e concluir este trabalho.

Agradeço também a todos os meus professores do Departamento de Sistemas Computacionais em especial ao professor Carlos Alexandre por ter sido fundamental no meu "renascer" no curso. E principalmente ao meu orientador Márcio Cornélio que foi sempre muito atencioso e companheiro em todos os momentos que precisei. Também agradeço ao professor Luiz por ter contribuído neste trabalho e ao professor José María Álvarez Palomo da Universidade de Málaga que forneceu um auxílio precioso na fase inicial deste trabalho.

Por fim, agradeço aos meus amigos da faculdade que sempre estiverem comigo me acompanhando principalmente nas madrugadas de estudo, em especial o meu amigo Robson. E também a Tássia por ter me ajudado nas revisões deste trabalho.

Capítulo 1

Introdução

Devido à velocidade do processo de desenvolvimento de software, é comum que a estrutura interna do mesmo não satisfaça fatores de qualidade [1] como reuso. Além disso, a falta de qualidade interna do software pode também não favorecer a manutenção e o entendimento dos códigos-fonte de aplicações.

A qualidade da estrutura interna de programas pode decair ao longo do tempo como conseqüência da introdução de novo código que visa atender novos requisitos do usuário ou corrigir falhas encontradas. Alterações que visam à reestruturação e à melhoria da qualidade interna de programas orientados a objetos (POO) e que não modificam o significado dos mesmos constituem o que chamamos de *refactoring* [2]. A aplicação de *refactoring* constitui uma forma disciplinada de alterar o código-fonte de um programa, sem que isto seja percebido por usuários do programa. Quando empregamos *refactoring* em um programa, temos a intenção de melhorar a qualidade do código-fonte após ele já ter sido desenhado e escrito.

Na prática, a garantia de que a reestruturação de programas OO não altera o comportamento de um programa – entenda-se comportamento aqui relativo a um programa seqüencial sem considerações temporais ou espaciais - é feita com base em testes e compilação [2]. Uma mudança no código é precedida pela definição de casos de teste e execução de testes. Se os testes tiverem sucesso, a mudança é realizada e os mesmos testes são realizados novamente. Não havendo falhas, considera-se que não houve mudanças na funcionalidade, ou seja, o comportamento não foi modificado.

Cornélio [3] usa uma abordagem para *refactoring* baseada no uso de leis de programação [4]. Cada pequena mudança é efetuada pela aplicação de uma lei que possui uma parte relativa à transformação e pode ter condições para a aplicação. Essa abordagem não obriga que sejam realizados testes, pois há provas de que as leis de programação não alteram o significado de um programa ao qual são aplicadas, desde que as condições para aplicação sejam satisfeitas. Tanto a verificação, quanto a aplicação dessas leis foram realizadas manualmente, o que possibilita que erros sejam cometidos. No seu trabalho, Cornélio utilizou uma linguagem de programação orientada a objetos [5], baseada em um subconjunto de Java [6], com semântica formal definida, porém com semântica de cópia ao invés de semântica de referência. As provas das leis de programação foram realizadas com base na semântica formal da linguagem.

A aplicação de leis de programação pode ser vista como uma atividade realizada em duas etapas. Na primeira as condições para aplicação da lei devem ser verificadas a fim de determinar se a lei pode ser realmente aplicada. A segunda etapa consiste na transformação do programa

como descrito na lei. Por exemplo, a eliminação de um método público de uma classe requer que seja verificado em todo programa que aquele método não é utilizado (chamado).

Tendo em vista que a aplicação manual de leis de programação é uma tarefa entediante e propensa a erros, é necessário que utilizemos algum suporte computacional que permita tanto a verificação das condições para aplicação da lei, como a transformação em si.

A transformação de programas descrita por leis de programação tem sido explorada com o uso de sistemas de re-escritura. Lira, em seu trabalho [7] utilizou Maude [8] para implementar uma estratégia de redução de linguagens de programação orientadas a objetos. Júnior *et al* [9] propôs o uso de CafeOBJ [10] para mecanizar o processo de aplicação de leis de programação para o desenvolvimento de *refactorings* e para a aplicação destes. Ambos os trabalhos tiveram suas implementações baseadas no texto dos códigos-fonte de especificações descritas na mesma linguagem [5] que utilizamos neste trabalho.

No presente trabalho apresentaremos uma nova abordagem para mecanização de aplicação de leis de programação baseada na meta-representação de programas (em forma de árvores sintáticas) em detrimento do uso do texto do código-fonte de programas. Esse fato torna as implementações, desenvolvidas utilizando esta nova abordagem, mais independentes da linguagem alvo, sendo assim, mais portáveis e passíveis a adaptações para outras linguagens como Java. Apesar de termos implementado apenas as verificações das condições de aplicação das leis de programação, este trabalho serve como uma base sólida para a transformação de programas em si (verificação de condições e aplicação de transformações).

1.1 Objetivos do trabalho

Este trabalho possui os seguintes objetivos.

- Apresentar uma forma de descrever gramáticas de linguagens orientadas a objetos, especificando-as como termos nativos de sistemas de re-escritura.

- Demonstrar o uso de conceitos de meta-programação de Maude na geração de meta-representações genéricas de programas orientados a objetos. Além disso, mostrar como é possível, manipular e alterar essas representações.

- Desenvolver um *parser* capaz de transformar meta-representações de Maude, em árvores sintáticas navegáveis. Utilizar o *parser* construído para gerar árvores sintáticas a partir de programas escritos na nossa linguagem.

- Implementar técnicas de navegação em árvores sintáticas através de algoritmos que utilizam lógica equacional e de re-escritura, em sistemas de re-escritura.

- Codificar uma ferramenta modularizada, capaz de realizar verificações de condições de leis de programação, mais especificamente de leis de comandos e de classes.

- Propor uma nova linha de trabalho (com o tratamento de meta-programas) para futuras implementações de técnicas mecanizadas de transformação de programas por meio da aplicação de leis de programação.

1.2 Estrutura do trabalho

Esta monografia está estruturada da seguinte forma:

- **Capítulo 2 –** Neste capítulo, apresentamos a sintaxe básica da linguagem que utilizamos no desenvolvimento da verificação das condições de leis de programação. Também apresentamos algumas de leis programação, com ênfase nas leis de comandos e de classes.

- **Capítulo 3 –** Descrevemos os conceitos básicos do sistema de re-escritura Maude, utilizado para o desenvolvimento da nossa ferramenta. Além disso, apresentamos a parte do conceito de meta-programação de Maude utilizada na codificação da nossa ferramenta.

- **Capitulo 4 –** Mostramos as implementações construídas em Maude, dos módulos da nossa ferramenta responsáveis por reconhecer programas da nossa linguagem, processar os meta-programas e gerar as árvores sintáticas. Neste capítulo, temos ainda a descrição de técnicas básicas para navegação nessas árvores, utilizando Maude.

- **Capítulo 5 –** Apresentamos como podemos, através da extensão das técnicas desenvolvidas no Capítulo 4, implementar verificações de condições de leis programação (de comandos e classes), definidas em [3].

- **Capítulo 6 –** Neste capítulo apresentamos as conclusões e as contribuições deste trabalho; bem como os trabalhos relacionados e sugestões de trabalhos que poderão ser desenvolvidos no futuro.

Além dos capítulos, o trabalho inclui ainda quatro apêndices:

- **Apêndice A –** Apresentamos a gramática da nossa linguagem com todos os seus construtores.

- **Apêndice B –** Contém a compilação das leis de comandos e de classes, definidas por Cornélio [3], e utilizadas nesta monografia.

- **Apêndice C –** Apresentamos o módulo da nossa ferramenta responsável por processar as meta-representações dos programas especificados na nossa linguagem e gerar as árvores sintáticas.

- **Apêndice D –** Mostramos a nossa implementação para realizar verificações de condições de leis de programação.

Capítulo 2

ROOL e Leis de programação

O estudo apresentado neste trabalho é baseado em uma linguagem que possui semântica formal definida e que foi projetada com o propósito de raciocinar sobre programas orientados a objetos. Esta linguagem é chamada ROOL, um acrônimo para *Refinement Object Oriented Language*.

Neste capítulo é apresentada a sintaxe abstrata de ROOL, com base no relatório técnico que a introduz [5]. E após isso são apresentadas algumas leis de programação descritas utilizando a linguagem ROOL.

2.1 A linguagem

A linguagem ROOL, é um subconjunto de Java seqüencial, com classes, herança, controle de visibilidade de atributos, ligação dinâmica e recursão, porém, utiliza semântica de cópia ao invés de referência. Ela é adequada para se pensar sobre programação orientada a objetos e especificações como as que incluem construtores do cálculo de refinamentos de Morgan [11]. A sintaxe é baseada na linguagem de comandos guardados de Dijkstra [12]. Uma descrição completa sobre a linguagem pode ser encontrada em [5].

2.1.1 Os tipos de dados

Os tipos de dados em ROOL são tanto os nomes de classes (*N*) quanto os tipos básicos encontrados na maioria das linguagens de programação (booleano, inteiro, float e outros). Tipos de dados *T* são os tipos de atributos, de variáveis locais, de parâmetros de métodos e de expressões.

$T \in Type ::= N_{.} \mid$ **bool** | **int** | ... outros tipos primitivos

2.1.2 Expressões

Expressões(**e**) em ROOL incluem expressões típicas de linguagens orientadas a objetos (Tabela 2.1). A expressão **self** tem o mesmo valor semântico do *this* de Java. Escrevemos **self**.*attr* para

Tabela 2.1. Gramática de expressões

e ∈ *Exp* ::=	**self** | **super**	'referências' especiais
	| **null**	'referência' nula
	| **new** *N*	criação de objeto
	| *x*	variável
	| *f (e)*	aplicação de função predefinida
	| *e* **is** *N*	checagem de tipo
	| (*N*) *e*	*cast* de tipo
	| *e.x*	seleção de atributo
	| *(e; x : e)*	atualização de atributo

um atributo *attr* da classe em questão. Utilizamos a expressão **new** *N* para criar uma classe qualquer *N*. É assumido que *x* é um identificador de uma variável qualquer e *f(e)* denota a aplicação de uma função qualquer predefinida da linguagem (por exemplo, +, -, *, concatenação de *strings*, etc). A expressão *e* **is** *N* é similar a *e* **instenceof** *N* de Java. A operação (*N*) *e* resulta um *cast* da expressão *e*. Uma seleção de atributo *e.x* resulta em um erro em tempo de execução quando *e* retornar **null.** A expressão de atualização (*e* ; *x* ; *e'*) denota a cópia de um objeto *e*, porém com o atributo *x* mapeado para uma cópia de *e'*.

As expressões que podem aparecer como alvos de atribuições, e de argumentos de resultado; também como alvos de chamadas de métodos, constituem um subconjunto denominado de expressões à esquerda (ou em Inglês *left-expressions*, que é de onde derivamos a sigla **le**).

le ∈ *Le* ::= *le1* | **self.***le1* | *(N)le.le1*
le1 ∈ *Le1* ::= *x* | *le1.x*

2.1.3 Predicados

Predicados em ROOL, representados pela letra greta Ψ, podem ser expressões do tipo **bool,** verificação de tipos dinâmicos, por meio de **e is N**, e qualquer fórmula lógica da primeira ordem do cálculo de predicados.

2.1.4 Comandos

A parte imperativa de ROOL, usada para definição do comando principal e dos corpos dos métodos, são comandos similares aos do cálculo de refinamento de Morgan.

Os comandos da linguagem ROOL (Tabela 2.2) incluem *specification statement* $\mathbf{x : [\Psi_1, \Psi_2]}$ que é útil para descrevermos um programa que pode modificar apenas as variáveis listadas no *frame x* e quando executado em um estado que satisfaz a pré-condição $\mathbf{\Psi_1}$ termina em um estado que satisfaz a pós-condição $\mathbf{\Psi_2}$. O *frame x* é a lista de variáveis que podem ser modificadas e $\mathbf{\Psi_1}$ e $\mathbf{\Psi_2}$ são fórmulas do cálculo de predicados.

Com o uso do *specification statement* podemos nomear as seguintes abreviações: **abort = x :[false, true]** que indica que o programa não tem garantia que irá terminar em algum momento (pré-condição **false**) e, se terminar, dará um resultado qualquer (pós-condição **true**); **skip** se comporta como um comando que sempre termina e não faz nada **:[true, true]**; escrevemos **miracle** como abreviação da especificação **x :[true, false]**, que é o contrário de **abort** e designa um programa que não pode ser implementado pois representa um programa que sempre termina quando executado em qualquer estado.

Tabela 2.2. Gramática de comandos e comandos parametrizados

c ∈ *Com* ::=	*le* ::= *e* | *c* ; *c*	atribuição múltipla, seqüência
	| *x* : [Ψ,Ψ']	especificação de declaração
	| *pc(e)*	aplicação de comando parametrizado
	| **if** []*i* ● $\Psi_i \rightarrow c_i$ **fi**	condicional
	| **rec** *X* ● *c* **end** | *X*	recursão, chamada recursiva
	| **var** *x* : *T* ● *c* **end**	bloco de variável local
	| **avar** *x* : *T* ● *c* **end**	bloco de variável angélica
pc ∈ *PCom* ::=	*pds* ● *c*	parametrização
	| *le.m* | *((N).le).m*	chamadas de métodos
	| **self**.*m* | **super**.*m*	
pds ∈ *Pds* ::=	| ∅ | *pd* | *pd* ; *pds*	
pd ∈ *Pd* ::=	**val** *x* : *T* | **res** *x* : *T*	declaração de parâmetros

Um comando do tipo **le := e** denota uma atribuição múltipla, onde **le** é uma lista de expressões à esquerda e **e** uma lista de expressões de tamanho igual. Após a execução desse comando, a cada item da lista **le** é atribuído o valor da expressão correspondente da lista de expressões.

O comando condicional é no estilo dos comandos guardados **if** da linguagem de Dijikstra. Ele é formado por um ou vários comandos guardados $\Psi_i \rightarrow c_i$ separados por *[]*, onde Ψ_i é um predicado e c_i é um comando que deverá ser executado se a avaliação do predicado Ψi for verdadeira. O construtor de recursão **rec** *X* ● *c* **end**, define um comando recursivo com o nome local *X*. Dentro do corpo do comando recursivo todas as ocorrências de *X* serão substituídas pelo próprio corpo do comando **rec** em tempo de execução.

Blocos de variáveis podem ser definidos com os comandos **var** *x* : *T* ● *c* **end** e **avar** *x* : *T* ● *c* **end.** Em blocos do tipo **var** o modo de inicialização das variáveis é arbitrária enquanto que em blocos **avar** os valores iniciais das variáveis são escolhidas de forma que se tenha certeza que o comando *c* terá sucesso.

Os comandos parametrizados *pds* ● *c* declaram parâmetros formais *pds* utilizados em um comando *c*. Chamadas de métodos são dadas por *le.m* onde *m* é o método a ser chamado e *le* o objeto alvo da chamada. Os parâmetros são passados através do mecanismo de cópia: valor (**val** *x* : *T*), ou, resultado (**res** *x* : *T*).

2.1.5 Classes e Programas

Um programa em ROOL (*cds* ● *c)* é um conjunto de declarações de classes *cds* seguidas de um comando *c*. A seguir é mostrado um exemplo que demonstra como uma classe pode ser declarada em ROOL.

```
class Pessoa extends object
        pri nome : String;
        pri endereco: String;
        pri telefone: String; ...
        meth getNome ^= (res n : String ● n := self.nome)
        meth setNome ^= (val n : String ● self.nome := n) ...
end
```

Subclasses são declaradas utilizando a cláusula **extends,** que é opcional. Todas as classes declaradas, por definição, têm a classe pré-definida **object** como superclasse. Assim, não é obrigatório que essa relação de herança seja feita explicitamente como por exemplo, foi feita no exemplo citado anteriormente.

A classe *Pessoa* possui três atributos privados: *nome*, *endereco* e *telefone*. Atributos podem ser também públicos (**pub**) e protegidos (**prot**), como em Java.

Os métodos são criados a partir dos comandos parametrizados vistos na seção anterior. Métodos podem ser apenas públicos e possuem parâmetros passados por valor ou resultado. No exemplo citado são encontrados dois métodos: *getNome* e *setNome*. O método *getNome* possui um parâmetro passado por resultado e *setNome* possui um parâmetro passado por valor.

A gramática de programas de ROOL pode ser encontrada no Apêndice A deste trabalho, assim como a BNF completa de todos os seus construtores.

2.2 Leis da programação orientada a objetos

Nesta seção são apresentadas algumas das leis de ROOL. A compilação das leis utilizadas nesta monografia pode ser encontrada no Apêndice B. Estas leis foram apresentadas em [13], e provadas por Cornélio [3].

Nas leis, o símbolo $\rightarrow$, é utilizado para indicar que a verificação das condições de uma lei só é necessária quando a lei é aplicada da esquerda para a direita. O símbolo $\leftarrow$ é utilizado para indicar que as condições apenas devem ser satisfeitas quando a lei é aplicada da direita para a esquerda. Já o símbolo $\leftrightarrow$, indica que as condições devem ser satisfeitas em ambas as direções.

2.2.1 Leis de comandos

Esta seção, é dedicada à apresentação de algumas leis de programação de ROOL que são aplicadas aos comandos da linguagem. Estas leis consideram construções de pequena granularidade. A compilação das leis de comandos utilizadas nesta monografia está disponível no Apêndice B.

Atribuir uma *left-expression* a ela mesma, não causa mudança no estado de um programa, assim isto é equivalente a *skip*. A única condição requerida é que a *left-expression* não seja *error*, caso contrário o programa aborta.

Lei ⟨**:= skip**⟩ Se $le \neq$ **error**, então (*le* **:=** *le*) = **skip**

Um conjunto finito de guardas do tipo ***if*** $[]i \bullet \Psi i \rightarrow c_i$ ***fi***, onde i varia de 1 a n seguido de um comando c qualquer, é equivalente ao mesmo conjunto, porém, com o comando c_i substituído por $c_i ; c$. Tal equivalência pode ser observada na lei abaixo.

Lei ⟨**; — if** *left dist*⟩
Se i está no intervalo 1..n

if *[]i ● Ψi → c_i* ***fi*** *; c* **=** ***if*** *[]i ● Ψi → c_i ; c* ***fi***

Uma variável pode ser renomeada, desde que o novo nome não esteja sendo utilizado.

Lei ⟨**var** *rename*⟩

Se x_2 não é livre em *c*, então

var x_1 *: T ●* *c* **end = var** x_2 *: T ●* *c*[x_2 / x_1] **end**

2.2.2 Leis de classes

As leis que seguem são aplicadas às classes. Tais leis são simples, considerando-se o fato de que leis mais elaboradas e de aplicação prática em transformações de programas, necessitam ser derivadas. O conjunto das leis de classes utilizadas nesta monografia está disponível no Apêndice B.

A notação $cd_1 =_{cds;c} cd_2$ indica uma abreviação para *cds* cd_1 ● *c* = *cds* cd_2 ● *c*, e significa que as declarações de classes cd_1 e cd_2 são equivalentes dado que *cds* é um conjunto de declarações de classes qualquer e *c* é o comando principal do programa. A notação *B.a* refere-se por exemplo, ao uso de um atributo de nome *a* a partir de expressões do tipo *B*. A relação de subclasseamento é indicada pelo símbolo ≤; , assim $B \leq C$ denota que a classe *B* é uma subclasse da classe *C*. Essas notações aparecem nas condições de aplicação das leis de classes.

A lei <*method elimination*> é utilizada quando queremos remover um método de uma classe. Para aplicar a lei da esquerda para a direita, ou seja, para remover um método de uma classe, uma condição precisa ser verificada: o método não pode estar sendo chamado em nenhuma classe em *cds*, em nenhuma parte do comando principal *c*, nem dentro da própria classe *C*. Essa mesma lei tem significado oposto (inserção de um método) quando aplicada da direita para a esquerda. Para se inserir um método *m* em uma classe qualquer *C*, *m* não pode estar declarado em *C*, nem em nenhuma de suas superclasses e subclasses.

Lei ⟨*method elimination*⟩

class *C* **extends** *D* *ads* **meth** *m* ≅ (*pds* ● *c*) *ops* **end**	$=_{cds;c}$	**class** *C* **extends** *D* *ads* *ops* **end**

dado que

(→) *B.m*, não aparece em *cds*, *c* nem em *ops*, para qualquer *B* tal que $B \leq C$;

(←) *m*, não é declarado em *ops* nem em qualquer superclasse ou subclasse de *C* em *cds*.

A lei <*change visibility: from private to public*> diz respeito a mudança de visibilidade de atributos privados para públicos. A aplicação da esquerda para a direita é direta sem que seja necessária a verificação de qualquer condição. Porém, a aplicação da direita para a esquerda só pode ser executada se o atributo não estiver sendo usado em nenhum lugar fora da classe que ele foi declarado.

Lei ⟨*change visibility: from private to public*⟩

class *C* **extends** *D* **pri** *a : T; ads* *ops* **end**	$=_{cds;c}$	**class** *C* **extends** *D* **pub** *a : T; ads* *ops* **end**

dado que
(←) *B.a*, para qualquer $B \leq C$, não aparece em *cds* ou *c*.

Para eliminar uma classe qualquer *C,* ela não pode estar sendo referenciada no comando principal nem em nenhuma outra classe do programa. É o que mostra a lei a seguir. Para a aplicação desta lei da direita para a esquerda é necessária a verificação de três condições: o nome da nova classe não pode estar sendo utilizado; a sua superclasse deve ser **object** ou alguma classe já declarada; atributos e métodos declarados pela classe não podem ter sido declarados por suas superclasses, ao menos que o caso se refira a uma redefinição de método.

Lei ⟨*class elimination*⟩

$cds\ cd_1 \bullet c = cds \bullet c$

dado que
(→) A classe declarada em cd_1 não é referenciada em *cds* ou *c*.
(←) (1) O nome da classe declarada em cd_1 é distinto de todas as classes declaradas em *cds*; (2) a superclasse que aparece em cd_1 ou é **object** ou está declara em *cds*. (3) e os nomes de atributos e métodos declarados em cd_1 não estão declarados por suas superclasses em *cds*, exceto no caso de redefinições de métodos.

2.3 Resumo

Introduzimos neste capítulo a sintaxe abstrata de ROOL, linguagem alvo do nosso trabalho. ROOL foi desenvolvida com o propósito de raciocinar sobre programas orientados a objetos. Ela é um subconjunto de Java seqüencial porém que utiliza semântica de cópia, e possui semântica formal definida. Apresentamos também, algumas das leis de programação de ROOL.

No próximo capítulo apresentaremos o sistema escolhido como plataforma de desenvolvimento da nossa ferramenta.

Capítulo 3

Maude

Maude é um sistema de alto desempenho que suporta lógica equacional [14] e lógica de re-escritura [15]. Além de um sistema, Maude é uma linguagem de programação que nos possibilita modelar sistemas e as ações que ocorrem nesses sistemas e entre eles. Maude é utilizado no mundo todo tanto na área de ensino como de pesquisa sendo especialmente útil na especificação e prototipação de sistemas lógicos, linguagens de programação e sistemas computacionais em geral.

Este capítulo tem como intenção apresentar uma breve introdução sobre Maude, focando nos seus fundamentos e principalmente nos conceitos que foram utilizados na implementação deste trabalho. Primeiramente são apresentados os fundamentos básicos de Maude, em seguida há uma descrição sobre os seus módulos e por fim, apresentamos o conceito de meta-nível. Este capítulo foi baseado principalmente nos dois documentos oficiais de Maude, *Maude Primer* [16] e *Maude Manual* [17].

3.1 Conjuntos e variáveis

Conjuntos (do inglês *sorts*) são tipos de dados. Eles podem ser definidos pelos programadores ou podem ser um dos tipos pré-definidos de Maude. Um *sort* pode ser por exemplo um *Boolean* ou *Integer*. Os *sorts* são introduzidos através da sintaxe: **sort** *Nome* **.**

Os conjuntos podem se relacionar através da definição de subconjuntos(*subsorts*). Um subconjunto tem a mesma conotação de um subtipo. O exemplo abaixo mostra na prática o conceito citado anteriormente:

sorts *Naturais, Inteiros, Irracionais, Racionais, Reais* **.**
subsorts *Irracionais Racionais* < *Reais* .
subsorts *Naturais* < *Inteiros* < *Racionais* **.**

O exemplo acima define Naturais, Inteiros, Irracionais, Racionais, Reais como conjuntos. Irracionais e Racionais são definidos como subconjuntos do conjunto dos Reais e o conjunto dos Naturais é definido como subconjunto dos Inteiros que, por sua vez, é definido como subconjunto dos Racionais.

Variáveis são identificadores que podem assumir qualquer valor de algum tipo definido no módulo. Elas são inseridas através da palavra chave **var**. Um exemplo de declaração de variável pode ser visto abaixo:

var *numero* **:** *Nat* **.**
var *numero2* **:** *Integer* **.**

As variáveis são declaradas com um tipo, porém elas nunca podem assumir um valor constante, como é comum em linguagens de programação como C. Elas são usadas como *placeholders* nas implementações de equações e regras de re-escritura.

3.2 Operações

As operações em Maude definem as ações que ocorrem entre operandos (que pertencem obrigatoriamente a algum *sort*) e os conjuntos. Elas são criadas com o uso de operadores. Os operadores definem a assinatura das ações, as equações e regras de re-escritura descrevem o significado, o comportamento das operações.

Os operadores são declarados com o uso da palavra-chave **op**, onde os caracteres do tipo "_" definem os operandos. Assim, no exemplo abaixo temos

op *succ_ : Nat -> Nat .*
op *_*_ : Nat Nat -> Nat .*

que define o operador *succ_* com um argumento do tipo *Nat* e com resultado do tipo *Nat* e um operador infixo _*_ com dois argumentos do tipo *Nat* resultando em um *Nat*.

Em Maude também é possível definir atributos para as operações. Os atributos mais comuns são comutatividade, associatividade, identidade e idem potência que são representados pelas palavras-chave *comm*, *assoc*, *id* e *idem*, respectivamente. Um outro atributo comum é o que define construtores (*ctor*). Os construtores definem a base de uma álgebra. Por exemplo, construtores típicos para o conjunto *Nat*, seriam *zero* e *s_*, e para o conjunto *Boolean* seriam *true* e *false*.

3.3 Módulos

As especificações em Maude são definidas dentro de módulos. Os módulos contêm definições e operações que interagem com essas definições. Em outras palavras, em um módulo, são declarados os *sorts* e as operações que atuam nesses *sorts*. Além disso toda a informação necessária para a redução e re-escritura das entradas dadas pelos usuários também é descrita dentro do módulo.

Um módulo pode ser importado por outro módulo, ou seja, é possível utilizar-se de operações e conjuntos declarados em um determinado módulo a partir de um outro módulo totalmente diferente. Existem três tipos de importação: *protecting, extending* e *including*. O tipo *protecting* indica que as declarações do módulo importado estarão visíveis, porém permanecerão

inalteradas (protegidas). No tipo *extending* o módulo importado pode ser estendido, mas não alterado. E no tipo *including*, o módulo importado pode ser estendido e também alterado. Por exemplo se estivermos importando o módulo *NAT*, com a cláusula *including*, é possível reimplementar a operação *succ_* para ela passar a retornar o próximo sucessor par. Abaixo são apresentadas as sintaxes que devem ser utilizadas na importação de módulos.

protecting NOMEDOMODULO .
ou
including NOMEDOMODULO .
ou
extending NOMEDOMODULO .

Existem três tipos de módulo: funcional (**fmod**), de sistema (**mod**) e orientado a objetos (**omod**). Abaixo é mostrada a sintaxe de como se deve inserir cada um deles:

fmod *nome* **is** ... **endfm**
mod *nome* **is** ... **endm**
omod *nome* **is** ... **endom**.

onde as reticências simbolizam todas as declarações que estão contidas dentro do módulo. Em módulos do tipo *omod,* podemos definir sistemas orientados a objetos em Maude. Este tipo de módulo faz parte de um nível de Maude, chamado *Full Maude,* que não pertence ao núcleo básico do sistema, estando assim, fora do escopo deste trabalho. Já os módulos funcionais e de sistema, serão tratados em seções subseqüentes.

3.3.1 Módulos nativos

Maude possui alguns módulos pré-definidos que podem ser importados livremente. O módulo *Nat*, por exemplo, define o *sort Nat* que simboliza os números naturais e algumas operações (soma, sucessor, etc.) entre eles. O tipo booleano é descrito no módulo *BOOLEAN*, assim como suas constantes (*true* e *false*) e suas operações básicas (*not, ==, =/=, <, >*, etc.).

Outro módulo nativo da biblioteca padrão de Maude é o módulo *String*. Uma *String* é descrita como qualquer grupo de caracteres circundado por aspas. *Strings* podem ser concatenadas através do operador _+_. A manipulação de *Strings* também é facilitada com o uso de algumas operações pré-definidas. O operador *substr* é utilizado para retornar um pedaço determinado de uma *String*. Um outro operador útil é o *find* que verifica a existência de uma *substring* em uma *String*.

Maude provê outro módulo bastante útil e utilizado. O módulo *Qid* do inglês *QuotedIdentifiers*, descreve o tipo composto por qualquer identificador precedido de um apóstrofo. *'Folder* e *'File* são exemplos *Qids*. Qualquer *String* pode ser transformada em um *Qid* com o uso do operador *qid*. Esse tipo de dado tem papel fundamental nas definições dos módulos de termos (*Term* e *TermList*) e de meta-nível (*Meta-Level*) que serão tratados na seção 4.

3.3.2 Módulos funcionais

Módulos funcionais definem tipos de dados e operações entre eles utilizando equações não condicionais e condicionais. Esta seção fará uma introdução sobre as equações e mostrará exemplos de sua utilização.

Equações

Equações em Maude seguem o mesmo padrão das equações matemáticas. Em Maude uma equação é criada pela cláusula *eq* seguida de duas expressões separadas pelo caractere '=' e um ponto final (.). A idéia das equações é fornecer ao interpretador regras de simplificação de expressões. A especificação abaixo define o módulo funcional *BASIC-NAT*.

```
fmod BASIC-NAT is
      sort Nat .
      op 0 : -> Nat   [ctor] .
      op s_ : Nat -> Nat   [ctor] .
      op _+_ : Nat Nat -> Nat  .
      vars N M : Nat  .
      eq 0 + N = N  .
      eq s(M) + N = s(M + N)  .
endfm
```

O módulo acima representa os números naturais na notação *Peano*. Nele é criado o tipo de dado *Nat*. Na notação *Peano*, nós utilizamos uma constante *0* e uma operação sucessor *s_*, assim evitamos a inviabilidade de definir um conjunto infinito de constantes para representar os números naturais.

O operador de adição _+_ define duas equações que funcionam de fato como regras de simplificação. Uma das regras diz que a expressão "zero mais um número natural" se reduz para o mesmo número. Essa equação provê ao interpretador de Maude a capacidade de reduzir *0 + s(s(0))* para *s(s(0))*, por exemplo. A segunda equação é simples e faz com que uma expressão do tipo *s(s(0)) + s(0)* seja reduzido para *s(s(0) + s(0))*.

Em Maude facilmente criamos especificações modularizadas. O exemplo abaixo demonstra isso e estende o nosso exemplo anterior com novas operações e novas equações.

```
fmod NAT+OPS is
      protecting BOOLEAN .
      protecting BASIC-NAT .
      op _*_ : Nat Nat -> Nat .
      op _-_ : Nat Nat -> Nat .
      op _<=_ : Nat Nat -> Nat  .
      op _>_ : Nat Nat -> Nat  .
      vars N M : NAT  .
      eq 0 * N = 0  .
      eq s(M) * N = (M * N) + N .
      eq 0 – N = 0 .
      eq s(M) – 0 = s(M) .
      eq s(M) – s(N) = M – N .
      eq 0 <= N = true .
      eq s(M) <= 0 = false .
      eq s(M) <= s(N) = M <= N .
      eq M > N = not (M <= N) .
```

```
endfm
```

Equações condicionais

Equações condicionais funcionam como equações não-condicionais, porém acrescentam condições para a execução das simplificações. Elas são introduzidas pela palavra-chave **ceq** seguida de duas expressões separadas pelo caractere "=" e pela condição iniciada pela cláusula **if**. Uma equação condicional só será reduzida se a condição vinculada a ela reduzir-se para verdadeiro (*true*).

No módulo de exemplo *NAT+OPS* poderíamos criar uma equação condicional para garantir que a subtração entre naturais sempre dará um valor natural. A equação condicional abaixo introduz essa garantia,

```
ceq N – M = 0 if M > N .
```

ela impõe que a subtração, mesmo que passível de ter um resultado negativo retorne sempre a constante *0*.

Outra operação que poderia ser acrescentada ao módulo *NAT-OPS,* seria a operação *max*. Basicamente ela retorna o valor máximo entre dois naturais, como podemos acompanhar no trecho de código a seguir:

```
...
op max : Nat Nat -> Nat .
ceq max(N,M) = M if N <= M .
ceq max(N,M) = N if N > M .
```

3.3.3 Módulos de sistema

Módulos de sistema são baseados em lógica de re-escritura. Neles podemos usar tanto regras de re-escritura quanto equações, ambas condicionais ou não-condicionais. A seguir será apresentado o conceito de regras de re-escritura e também exemplos práticos de como utilizar esse conceito em Maude.

Regras de re-escritura

Equações são úteis quando queremos tratar com simplificações e equivalências. Regras de re-escritura são utilizadas para mapear estados e as transições entre estes estados. A equação "`X = Y`" é uma relação e consequentemente não-direcional. Ao contrário, a regra de re-escritura "`X -> Y`", é uma ação direcional: X pode ser substituído por Y, mas não o contrário.

Como já foi dito, regras de re-escritura definem estados e também definem como e quando ocorrem as transições entre eles. Por exemplo, "dia ensolarado" e "dia chuvoso" podem ser considerados dois estados e uma regra pode ser criada para definir a transição entre eles. Suponhamos que uma grande nuvem chuvosa esteja se aproximando e comece a chover, então podemos introduzir a regra [nuvem chuvosa] : dia ensolarado -> dia chuvoso. Ou seja, com a presença de uma nuvem chuvosa e consequentemente de chuva, o dia passa de ensolarado para chuvoso. Note, que se substituíssemos a regra citada, pela equação "dia ensolarado = dia chuvoso", estaríamos instituindo uma informação inverídica e sem sentido lógico.

Em se tratando de Maude, a regra de re-escritura exemplificada anteriormente pode ser criada como abaixo,

```
rl [nuvemchuvosa] : diaensolarado => diachuvoso .
```

onde *nuvemchuvosa* é uma etiqueta associada a regra. Note que a regra é iniciada com a palavra chave **rl**. Escrevemos o nome da regra entre colchetes.

Um módulo de sistema simples, pode ser introduzido para tratar a regra citada acima:

```
mod CLIMA is
      sort condicaoclimatica
      op diaensolarado : -> condicaoclimatica  .
      op diachuvoso : -> condicaoclimatica  .
      rl [nuvemchuvosa] :  diaensolarado => diachuvoso  .
endm
```

Basicamente, o tipo *condicaoclimatica*, é criado para representar as condições climáticas, e as operações *diaensolorado* e *diachuvoso*, representam constantes. A regra etiquetada como *nuvemchuvosa* promove a transição entre as duas constantes.

Regras de re-escritura condicionais

Assim como as equações, as regras de re-escritura também podem ser condicionais. Elas são introduzidas pela palavra-chave **crl** seguida da regra, e da condição iniciada pela cláusula **if**. Uma regra condicional só será aplicada se a condição vinculada a ela reduzir-se para verdadeiro (*true*).

Considere a seguinte versão modificada do módulo *CLIMA*:

```
mod CLIMA is
      sort condicaoclimatica
      op diaensolarado : -> condicaoclimatica  .
      op diachuvoso : -> condicaoclimatica  .
      op dianublado : -> condicaoclimatica  .
      ...
      crl [nuvemchuvosa] :  diaensolarado => diachuvoso  if
      iniciodechuva(nuvem) == true .
      crl [nuvemchuvosa] :  diaensolarado => dianublado  if
      iniciodechuva(nuvem) == false .
endm
```

Temos agora uma nova condição climática, introduzida pela constante *dianublado*. Considerando que a operação *iniciodechuva* nos diz se, dada uma nuvem, começou a chover ou não, a nova versão de *CLIMA* passa a implementar uma condição climática intermediária. Se o dia estiver ensolarado e chegar uma nuvem chuvosa mas ainda não começou a chover, o dia passa a ser nublado. Porém se, na mesma situação começar a chover, o dia passa a ser chuvoso.

3.4 Meta-nível

Os conceitos de meta-programação [17] de Maude, são implementados nos seus meta-módulos. Estes módulos são parte integrante da biblioteca padrão de módulos de Maude. Em resumo nesta biblioteca temos,

- O módulo *META-TERM,* onde termos de Maude são meta-representados como integrantes do tipo de dado *Term* de termos.
- O módulo *META-MODULE,* onde os módulos de Maude são meta-representados como integrantes do módulo *Module* de módulos.
- O módulo *META-LEVEL*, onde as chamadas funções descendentes (*metaReduce, metaApply, metaRewrite,*etc), são definidas.

Nesta seção será apresentado um resumo de algumas das principais funções providas pelos meta-módulos, porém, focando os conceitos que foram utilizados neste trabalho.

3.4.1 O módulo *META-TERM*

No módulo *META-TERM*, termos são meta-representados como elementos do conjunto *Term* de termos. Basicamente, um termo é uma expressão formada de constantes e operações de uma álgebra.

A meta-representação de um termo pode ser expressada como um elemento do conjunto *Term* ou *TermList*. Abaixo segue um fragmento, retirado da biblioteca de Maude, que define esses conjuntos.

```
sort TermList  .
subsort Term < TermList  .
op _,_ : TermList TermList -> TermList  .
op _[_] : Qid TermList -> Term  .
```

O conjunto *Term* é um subconjunto de *TermList*. O operador _,_ define uma lista de termos, sendo o caractere “,” o concatenador de termos.

Um meta-termo pode ser composto pela meta-representação de constantes, variáveis e operações. Uma Constante é um *QuotedIdentifer* que contém o nome da constante e o nome do tipo a qual ela pertence separados por “.”. Por exemplo, a constante 0 do conjunto *Nat* do módulo *BASIC-NAT* da seção 3.3.2 é representada como *‘0.Nat*. Igualmente, uma variável contém seu nome e seu tipo separado por “:”, por exemplo *‘N:Nat*.

O operador _[_] corresponde a meta-representação dos operadores unidos a variáveis e constantes. O primeiro argumento é o nome do operador e o segundo argumento é a lista de termos (*TermList*), representando os operandos envolvidos na operação. Por exemplo, o termo *0 + s(s(0))* do já apresentado, módulo *BASIC-NAT,* é meta-representado como o seguinte elemento do conjunto *Term*:

```
‘_+_[‘0.Nat, ‘s_[‘s_[‘0.Nat]]]
```

Em negrito estão as constantes, nos outros casos temos operadores. Note a recursividade dessa construção. Esta característica será explorada em capítulos posteriores.

Duas operações pré-definidas nesse módulo são bastante úteis, pois elas funcionam como seletores de nomes e tipos em Variáveis e Constantes.

```
op getName : Constant -> Qid  .
op getName : Variable -> Qid  .
op getType : Constant -> Type  .
op getType : Constant -> Type  .
```

O operador *getName* retorna o nome de uma Variável ou Constante e o operador *getType* retorna o tipo. Isto é,

getName (‘0.Nat) = ‘0
getType (‘0.Nat) = ‘Nat

Essas duas operações foram utilizadas na maioria das implementações dos algoritmos que compõem a nossa ferramenta.

3.4.2 O módulo *META-LEVEL*

O módulo *META-LEVEL*, possui três funções descendentes [15] para meta-avaliação, *meta-reduce*, *meta-rewrite* e *meta-apply* que, como os nomes já dizem, realizam a redução e re-escritura de meta-termos, levando em conta as equações e regras dos meta-módulos em questão, no meta-nível. Uma discussão a fundo sobre essas funções fogem do escopo deste trabalho, entretanto uma referência completa sobre elas pode ser encontrada em [17].

Alternando entre os níveis de reflexão: *upTerm e downTerm*

Na seção 3.4.1 foi mostrado como se constrói um meta-termo a partir de um termo composto por operações e constantes. É possível notar que o resultado apresentado não seria uma tarefa tão simples e rápida se fosse feita de maneira manual, além de favorecer o aparecimento de erros. Graças a algumas funções nativas de Maude, a mudança entre os níveis de reflexão, como por exemplo, entre termos e meta-termos é feita de forma automática. Assim, podemos obter um meta-termo a partir de um termo ou obter um termo a partir de um meta-termo.

A função *upTerm* toma um termo como argumento e retorna a meta-representação do termo levando em conta o módulo a qual ele pertence. Como um exemplo simples é possível obter a meta-representação do termo *s(s(s(0))* no módulo *BASIC-NAT,* a partir da função *upTerm*:

```
Maude> reduce in BASIC-NAT : upTerm(s(s(s(0))) )  .
result GroundTerm: 's_['s_['s_['0.Nat]]]
```

A função *downTerm* toma um meta-termo e uma constante como argumentos e executa a redução em relação ao módulo dado. Se houver sucesso na redução ela retorna o termo correspondente, porém se houver falha ela retorna a constante. Podemos verificar essa função em ação em três exemplos. No primeiro obtemos o termo s(*s(s(0)))* a partir de sua meta-representação. No segundo aplicamos a função *upTerm* seguida da função *downTerm*. No terceiro ocorre um problema na redução. Em todos, assumimos a existência de uma constante nomeada *error* declarada no módulo *BASIC-NAT*.

```
Maude> reduce in BASIC-NAT : downTerm('s_['s_['s_['0.Nat]]], error)  .
result Nat: s(s(s(0)))

Maude> reduce in BASIC-NAT : downTerm( upTerm(s(s(s(0)))), error)  .
result Nat: s(s(s(0)))

Maude> reduce in BASIC-NAT : downTerm('s_['s_['s_['1.Nat]]], error)  .
Advisory: could not find a constant 1 of sort Nat in meta-module BASIC-
NAT.
reduce in BASIC-NAT : downTerm('s['s['s['1.Nat]]], error) .
```

```
result Nat: error
```

É importante notar que houve um problema ao tentar obter a representação básica do meta-termo acima. A constante *1* não existe no módulo *BASIC-NAT* assim não foi possível ocorrer a redução do meta-termo para o nível abaixo. Então a função *downTerm* retornou o segundo argumento passado, *error* que foi declarado no módulo *BASIC-NAT*.

3.5 Resumo

Neste capítulo apresentamos uma breve introdução sobre Maude, um sistema de re-escritura de alto desempenho que suporta tanto lógica equacional quanto lógica de re-escritura. Focamos em seus conceitos básicos e nos conceitos utilizados na implementação da ferramenta resultante desse trabalho.

Nos próximos capítulos passaremos a tratar a implementação desenvolvida ao longo desse trabalho.

Capítulo 4

Representação sintática e *parsing*

A primeira fase desse projeto é dedicada à construção da base necessária para a execução das verificações das condições existentes para a aplicação das leis de programação. Este capítulo descreve as atividades desenvolvidas para se obter árvores sintáticas, a partir de especificações de programas em ROOL, e também a técnica implementada para a navegação nessas árvores.

Os módulos da nossa ferramenta que implementam as atividades descritas anteriormente, estão descritos no diagrama da Figura 4.1. As setas indicam uma relação de importação entre os módulos envolvidos. O módulo *LIST-NODE* é composto por operações de manipulação de listas tais como *last* (que retorna o último item de uma lista) e *itemExists* (que verifica se um determinado item existe em uma lista). O módulo *UTIL-MODULE* possui uma série de operações que são utilizadas por todos os outros módulos da nossa ferramenta. As implementações dos módulos restantes do diagrama citado serão tratadas em detalhes ao longo deste capítulo.

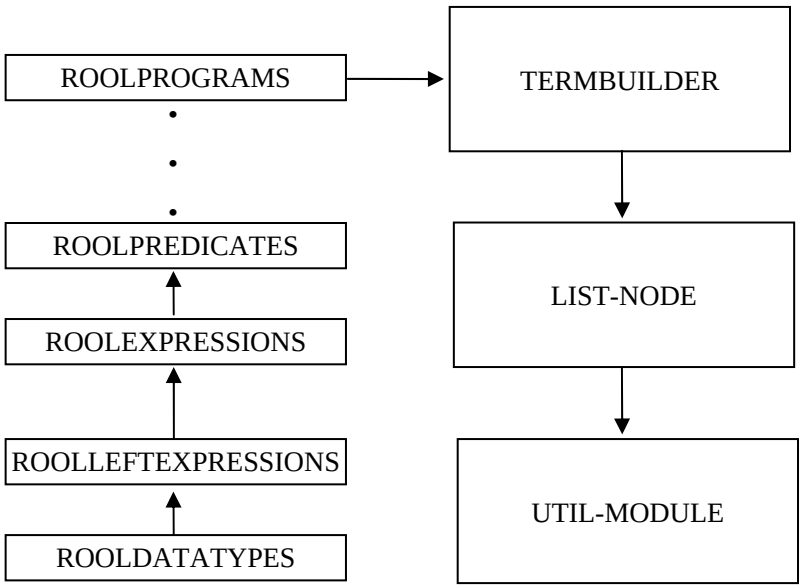


Figura 4.1: Diagrama que mostra o inter-relacionamento dos módulos que especificam os aspectos sintáticos da nossa ferramenta.

Este capítulo está organizado como segue. Primeiro, é tratada a forma como ROOL foi descrita para Maude. Depois disso, é apresentado em detalhes como foi desenvolvido o *parser* para programas escritos em ROOL. Então, são descritos os conceitos básicos necessários para a navegação nas árvores sintáticas resultantes do *parser*. Por fim, é apresentado um resumo do capítulo.

4.1 Descrevendo ROOL em Maude

Para desenvolver um *parser* para a nossa linguagem a primeira atividade a ser desenvolvida é fazer com que o nosso sistema reconheça-a. A forma mais adequada encontrada, foi adaptar a gramática de ROOL e descreve-la diretamente na linguagem nativa de Maude. Essa abordagem foi baseada no trabalho escrito por Lira [7].

A gramática de ROOL foi especificada em Maude por meio de 5 módulos de sistema: *ROOLDATATYPES*, *ROOLLEFTEXPRESSIONS*, *ROOLEXPRESSIONS*, *ROOLPREDICATES*, *ROOLCOMMANDS*, *ROOLPCOMMANDS* e *ROOLPROGRAMS*. Cada um será tratado em detalhes a seguir. A Figura 4.2 exibe um diagrama que demonstra a relação de dependência entre esses módulos. As setas indicam uma relação de importação entre os módulos envolvidos. O módulo *Qid*, que aparece na figura, é um módulo pré-definido de Maude que representa os identificadores prefixados por um apóstrofo.

O módulo de sistema *ROOLDATATYPES* descreve alguns dos tipos primitivos de ROOL como constantes do conjunto *Bool*. Entre eles estão `str`, `int` e `bool`. As palavras reservadas `true` e `false` também são definidas como constantes. As operações entre booleanos são descritas como operações nativas de Maude. Os nomes das classes, que também funcionam como tipos, são criados e referenciados sempre com o prefixo `CLID`. Assim, `CLID 'Conta` é um nome de classe válido para o nosso sistema. Uma exceção a isso é a classe padrão `Object`, que é definida como uma constante especial. O prefixo `CLID` foi criado para que pudéssemos diferenciar nomes de classes, de nomes de atributos e métodos. Outros prefixos, utilizados em outras construções da nossa linguagem, são apresentados na Tabela 4.1.

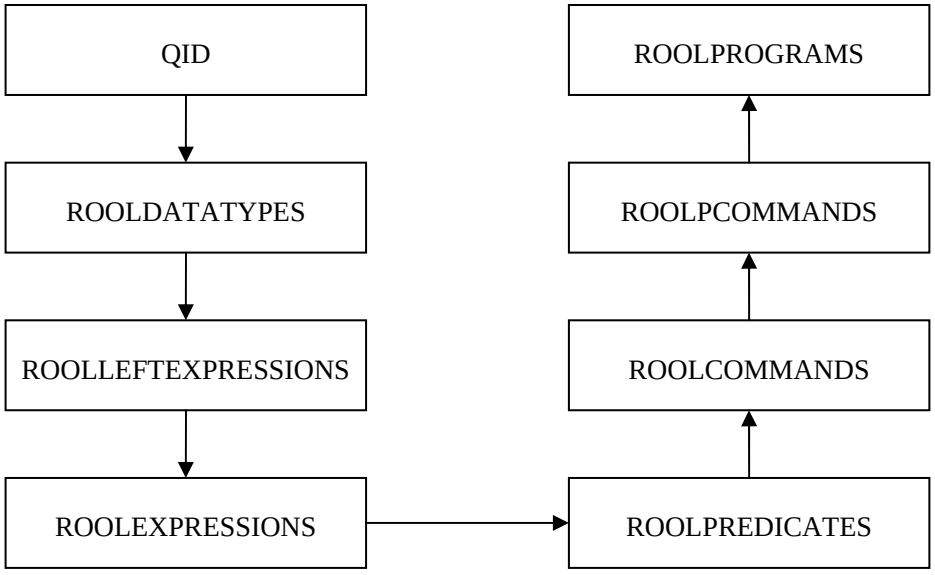


Figura 4.2: Diagrama de dependência entre os módulos que representam as classes sintáticas de ROOL.

Tabela 4.1: Prefixos criados na construção de identificadores

Construtores	**Prefixo**	**Exemplo**
Nome de classes	`CLID`	`CLID'Conta,CLID'Pessoa`
Nome de atributos	Não se aplica	`'nome, 'numero`
Nome de métodos	`MtID`	`'credito, 'getName`
Identificador de comando **rec**	`CID`	`CID 'fatorial`

Após a descrição dos tipos de dados de ROOL, temos a descrição das expressões a esquerda no módulo *ROOLLEFTEXPRESSIONS*. A palavra reservada *self* é definida como uma constante neste módulo. Uma lista de expressões à esquerda é criada com o operador `_:::_`, assim `'var1 ::: 'var2 ::: 'var3` é uma lista válida. Dentro deste módulo há uma cláusula *protecting* apontando para o módulo *ROOLDATATYPES,* que indica a importação do módulo. Em suma, todos os módulos utilizados na descrição de ROOL em Maude são interligados através de cláusulas de importação como já foi observado anteriormente.

As expressões, por sua vez, estão especificadas no módulo *ROOLEXPRESSIONS*. Nele destacamos a especificação das funções pré-definidas da nossa linguagem: `Prod`, `Sum`, `Minus`, `Div` e `Mod` que representam produto, soma, subtração, divisão e módulo, respectivamente. Apesar da simplicidade dessas funções, elas são suficientes para as especificações criadas no nosso trabalho. Outras funções podem ser criadas a partir de derivação das funções já pré-definidas.

Os predicados de ROOL estão definidos no módulo *ROOLPREDICATES*, que importa o módulo *ROOLEXPRESSIONS*. Neste módulo, as expressões booleanas, que são na verdade predicados, são representadas pelo *sort* `ExpBool`, que é um *subsort* do *sort* `Exp`, que representa as expressões, definido no módulo *ROOLEXPRESSIONS*.

O módulo *ROOLCOMMANDS* define os comandos da nossa linguagem. Ele importa o módulo *ROOLPREDICATES*. As abreviações `abort`, `skip` e `miracle` são descritas no módulo como constantes. Os blocos de comandos `var`¸ `avar`, `rec` e `if` também são encontrados neste módulo, assim como os operadores de atribuição (`:=`) e de composição seqüencial (`;`).

A especificação dos comandos parametrizados da nossa linguagem está definida no módulo *ROOLPCOMMANDS,* que importa o módulo *ROOLCOMMANDS*. Nele encontramos entre outras construções, as construções que representam a declaração de parâmetros passados por valor e resultado, `val` e `res`, respectivamente.

O último módulo da nossa descrição que representa um programa completo de ROOL, é o módulo *ROOLPROGRAMS*. Nele estão definidas as construções de orientação a objetos de ROOL. Entre elas estão as declarações de classes, métodos e atributos. Um programa em ROOL é representado aqui pelo operador `_.main<_>` que tem como primeiro argumento um conjunto de declarações de classes e como segundo um comando. Este módulo importa o módulo *ROOLPCOMMANDS*.

Algumas constantes foram criadas para representar a nulidade da declaração de atributos e métodos dentro de declarações de classes. A constante `nullAtribute` indica que uma classe não possui atributos, a constante `nullMethod` indica a ausência de métodos. A classe especial `Object` não possui métodos nem atributos definidos. Ela poderia ser declarada assim:

```
class Object nullAtribute nullMethod end
```

A seguir é mostrado um exemplo de um programa de um banco descrito em ROOL e a sua respectiva versão após feito o mapeamento para Maude. Na Figura 4.3 temos uma classe que

```
class Object nullAttribute nullMethod end

class Conta
      pri numero : String ;
      pri saldo  : int ;

      meth Const ^= val numeroconta : String ; .
            self.numero := numeroconta ;
            self.saldo := 0 end
      meth creditar ^= val valor : int ; .
            self.saldo := self.saldo + valor end
      meth debitar = val valor : int ; .
            if valor <= self.saldo ->
                  self.saldo = self.saldo – valor
            fi
      end
      meth getNumero ^= res tmp : String ; .
            tmp := self.numero end
      meth getSaldo = res tmp : int ; .
            tmp := self. saldo end
end
class Poupanca extends Conta
      meth Const ^= val numeroconta : String ; .
            super.Const(numeroconta) end
      meth renderJuros ^= val taxa : int ; .
            super.creditar(self.saldo * taxa) end
end
class ContaEspecial extends Conta
      pri limite : int ;
      meth Const ^= val numeroconta : String ; val lim : int .
            super.Const (numeroconta) ;
            self.limte := lim ;
      meth debitar ^= val valor : int ; .
            if valor <= (self.saldo + limite) ->
                  self.saldo = self.saldo – valor
            fi
      end
      meth getLimite ^= res tmp : int ; .
            tmp := self.limite end
end

var c1, c2 : Conta .
      c1 := new Conta;
      c1.Const(“11111-1”) ;
      c1.creditar(5000) ;
      c1.debitar(1500) ;
      c2 = new ContaEspecial;
      c2.Const(“12222-2”, 4000);
      ((ContaEspecial)c2).debitar(3000)
end
```

Figura 4.3: Definição do programa BANCO em ROOL.

representa uma conta bancária comum chamada `Conta`, a classe `Poupanca` que representa as cadernetas de poupança e a classe `ContaEspecial` que é um tipo de conta que possui um limite associado. Após as declarações das classes temos o corpo principal do programa.

Agora, vamos transcrever o programa da Figura 4.3 para a versão de entrada para Maude. A nova declaração de `Object` é similar a apresentada anteriormente.

As novas versões das classes `Conta` e `Poupanca`, estão descritas na Figura 4.4. Como pode ser visto, alguns *tokens* novos foram inseridos, mas apenas para garantir a compatibilidade com o nosso sistema. Apesar disso, a obtenção de versões compatíveis é feita de maneira simples e direta, sem dificuldades, porém, esse processo tem de ser feito manualmente.

```
class CLID ‘Conta
      pri ‘numero : str ; |
      pri ‘saldo  : int ;
      meth MtID ‘Const ^= val ‘numeroconta : str ; .#
            self . ‘numero := ‘numeroconta ;
            self . ‘saldo := 0 # end
      meth MtID ‘creditar ^= val ‘valor : int ; .#
            self . ‘saldo := Sum < self . ‘saldo ::: ‘valor > #
      end *
      meth MtID ‘debitar ^= val ‘valor : int ; .#
            if ‘valor <= self . ‘saldo  -> self . ‘saldo :=
               Sum < self . ‘saldo ::: Minus < ‘valor > >
            fi # end *
      meth MtID ‘getNumero ^= res ‘tmp : str ; .#
            ‘tmp := self . ‘numero # end *
      meth MtID ‘getSaldo ^= res ‘tmp : int ; .#
            ‘tmp := self . ‘saldo # end *
end ,

class CLID ‘Poupanca extends CLID CLID ‘Conta
      nullAtribute
      meth MtID ‘Const ^= val ‘numeroconta : str ; .#
            super . MtID ‘Const < ‘numeroconta > # end *
      meth MtID ‘renderJuros ^= val ‘taxa : int ; .#

            super . MtID ‘creditar < Prod < self . ‘saldo :::
‘taxa           > >  # end
end ,
```

Figura 4.4: Especificação das classes `Conta` e `Poupanca` do programa `BANCO` em Maude.

A nova versão da classe `ContaEspecial` está descrita na Figura 4.5. Ela foi obtida a partir da transcrição direta de sua respectiva versão em ROOL.

Por fim, vem a transcrição do comando principal do programa (Figura 4.6). Este comando tem o mesmo significado do método *main* da linguagem Java. A nova versão do bloco *main* do programa `BANCO` pode ser vista na Figura 4.6.

```
      class CLID ‘ContaEspecial extends CLID ‘Conta
            pri ‘limite : int ;
            meth MtID ‘Const ^= val ‘numeroconta : str ; val ‘lim :
      int   .#
                  super . MtID ‘Const < ‘numeroconta > ;
                  self . ‘limite := ‘lim ; # end *
            meth MtID ‘debitar ^= val ‘valor : int ; .#
                  if valor <= ( Sum < self.saldo ::: limite > ) ->
                              Sum < self . ‘saldo ::: Minus < ‘valor >
>
                  fi #  end *
            meth getLimite ^= res ‘tmp : int ; .
                  ‘tmp := self . ‘limite # end
      end
```

Figura 4.5: Especificação da classe `ContaEspecial` do programa `BANCO` em Maude.

```
      .main<
            var ‘c1 : CLID Conta :: ‘c2 : CLID Conta .
                  ‘c1 := new CLID Conta;
                  ‘c1 . MtID ‘Const <”11111-1” > ;
                  ‘c1 . MtID ‘creditar < 5000 > ;
                  ‘c1 . MtID ‘debitar < 1500 > ;
                  ‘c2 = new CLID ‘ContaEspecial ;
                  ‘c2 . MtID ‘Const <“12222-2” ::: 4000 > ;
                  < CLID ‘ContaEspecial > ‘c2 . MtID ‘debitar < 3000 >
            end
      >
```

Figura 4.6: Especificação do bloco *main* do programa `BANCO` em Maude.

Após ter sido feita a exposição de como escrevemos programas especificados em ROOL no nosso sistema, mostraremos na próxima seção, o *parser* desenvolvido para gerar as árvores sintáticas que representam estes programas.

4.2 O *parser*

O desenvolvimento do presente trabalho foi impulsionado pela meta de se conseguir executar as verificações de condições de aplicação de leis de programação em árvores sintáticas de programas e não diretamente nos códigos-fonte dos programas como foi proposto por Lira. A adoção dessa nova abordagem em detrimento da abordagem adota por Lira é justificada pela possibilidade de tornar a implementação construída aqui mais portável para outras linguagens. O ganho em portabilidade se deve ao fato de estarmos tratando, na verdade, uma meta-representação do programa.

A meta-representação definida neste trabalho possui uma base imutável. Essa base é uma estrutura hierárquica fixa (uma árvore), que tem termos como membros, que por sua vez podem representar qualquer construção de qualquer linguagem descrita no nosso sistema. Nesta seção, apresentamos desde o pré-processamento do código-fonte de um programa até a geração da árvore final que o representa.

4.2.1 Pré-processamento

A fase de pré-processamento de um programa está dividida em duas partes: a primeira parte trata da geração do meta-programa e a segunda da geração de marcas especiais em alguns membros do meta-programa. Elas são detalhadas a seguir.

Obtendo o meta-programa

O uso de meta-programação nos remete aos conceitos de Maude que lidam com este assunto. Mais especificamente aos conceitos que regem o conjunto *Term* e a função *upTerm* já discutidos anteriormente. A partir da função *upTerm* obtemos um meta-programa para um programa definido na nossa linguagem.

O comando *upTerm* recebe um conjunto de termos válidos em Maude e gera seu respectivo meta-termo. Isso é feito de maneira recursiva pelo sistema, de modo que o resultado gerado pode ser considerado uma entidade única que representa de forma hierárquica o conjunto de termos passado como entrada. Primeiramente vamos apresentar um exemplo simples na Figura 4.7 que, apesar de sozinho não ter aplicabilidade, nos ajuda a entender o resultado obtido a partir do *upTerm*.

```
Maude> reduce upTerm (
var( 'a : str ) :: ( 'b : str ) .
     'a := "ROOL" ;
     'b := "Maude"
end
) .
reduce in ROOLPROGRAMS : upTerm(var ('a : str).VarExp :: ('b :
str).VarExp .
('a := "ROOL").Comando ; ('b := "Maude").Comando end) .
rewrites: 1 in -2688414721ms cpu (24ms real) (~ rewrites/second)
result GroundTerm:
'var_._end['_::_['_:_[''a.Sort,'str.String],'_:_[''b.Sort,
'str.String]],'_;_['_:=_[''a.Sort,'"ROOL".String],'_:=_[''b.Sort,
    '"Maude".String]]]
```

Figura 4.7: Resultado da execução do comando *upTerm* em um bloco de variáveis exemplo da nossa linguagem.

Neste exemplo podemos perceber como a função *upTerm* atua em um bloco de variáveis simples (grafado em itálico). O bloco possui apenas duas variáveis declaradas, `‘a` e `‘b`, e dois comandos de atribuição dentro dele. A parte que está em negrito representa o resultado (meta-termo) gerado pela função.

Para entender realmente o resultado demonstrado na figura anterior precisamos nos reportar aos módulos que descrevem ROOL em Maude. Neles, encontramos os operadores `var_._end`, `_::_`, `_:_`, `_;_` e `_:=_`. E ainda, a constante *str*. O que o comando *upTerm* faz é agrupar numa única estrutura, operadores e seus respectivos operandos de forma hierárquica. Assim o resultado da aplicação de *upTerm* para a estrutura `‘a := “ROOL”` seria simplesmente `‘_:=_[‘a.Sort, “ROOL”.String]`. Como podemos notar, primeiro vem o operador depois os operandos dentro de dois colchetes. Não importa o tamanho da construção, o *upTerm* vai de maneira recursiva atuando de operador a operador definindo seus operandos. Um outro detalhe a

ser dito, é sobre as constantes e variáveis. Elas são formadas por elas mesmas mais o *sort* em que foram definidas. Quando não é possível descobrir o *sort*, a função classifica-as como do tipo genérico *Sort*.

A partir do conceito da função *upTerm* já discutido, é apresentado na Figura 4.8 o resultado da função *upTerm* aplicada a classe `Poupanca` (Figura 4.4).

```
Maude> reduce upTerm (
class CLID 'Poupanca extends CLID CLID 'Conta
      nullAtribute
      meth MtID 'Const ^= val 'numeroconta : str ; .#
            super. MtID 'Const < 'numeroconta > # end *
      meth MtID 'renderJuros ^= val 'taxa : int ; .#
            super. MtID 'creditar < Prod < self . 'saldo ::: 'taxa >
>     # end
end
) .

reduce in ROOLPROGRAMS : upTerm(
class CLID 'Poupanca extends CLID CLID 'Conta
    nullAtribute meth MtID 'Const ^= val 'numeroconta : str ; .#
super. MtID
    'new < 'numeroconta > # end * meth MtID 'renderJuros ^= val
'taxa : int ;
    .# super. MtID 'creditar < Prod < self . 'saldo ::: 'taxa > > #
end end ) .
rewrites: 1 in -2690638721ms cpu (0ms real) (~ rewrites/second)
result GroundTerm:
'class_extends___end['CLID_[''Poupanca.Sort],'CLID_['CLID_[
''Conta.Sort]],'nullAtribute.Attrs,'_*_['meth_^=_end['MtID_[''Const.
Sort],'_.#_#['val_;['_:_[''numeroconta.Sort,'str.String]],
'super._<_>['MtID_[''Const.Sort],''numeroconta.Sort]]],'meth_^=_end[
'MtID_[ ''renderJuros.Sort],'_.#_#['val_;['_:_[''taxa.Sort,
'int.Int]],'super._<_>['MtID_[''creditar.Sort],
'_<_>['Prod.BuiltinFunction ,'_:::_['_._['self.LeftExp,
''saldo.Sort], ''taxa.Sort]]]]]]
```

Figura 4.8: Resultado da função *upTerm* para a classe `Poupanca` da Figura 4.4.

Classificando identificadores com o uso de etiquetas

A geração de etiquetas em identificadores da nossa linguagem foi motivada pela dificuldade de se discernir entre os significados que cada identificador (`Qid`) possui nos meta-programas. Observando a Figura 4.9, as seguintes perguntas podem ser feitas: Qual a diferença entre a aparição do `Qid ‘Poupanca` na declaração de variáveis, e na atribuição? E mais, qual a diferença dessas duas aparições com as aparições do `Qid ‘Poupanca` como nome de classe? Essas mesmas perguntas podem ser feitas em relação ao meta-programa e o termo `‘’Poupanca.Sort`. A resposta para essas duas perguntas é a seguinte: O que difere entre elas é o operador que as contém. A primeira aparição é “filha” do operador `_:_`, a terceira do operador

_:=_, a segunda e quarta, são filhas do operador CLID_. Agora, imaginemos que todas as vezes que fossemos precisar diferenciar um identificador tivéssemos que varrer todo o nosso meta-programa em busca do operador "pai" dele. Isso acarretaria em um dispendioso trabalho e em um aumento significativo em processamento.

Trecho de código referenciando o programa BANCO:

```
var( 'Poupanca : str ) :: ( 'c1 : CLID 'Poupanca ) .
     'Poupanca := "ROOL" ;
     'c1 := new CLID 'Poupanca
end
```

Meta-programa do código acima:

```
'var_._end['_::_['_:_[''Poupanca.Sort,'str.String],'_:_[
''c1.Sort,'CLID_[''Poupanca.Sort]]],'_;_['_:=_[''Poupanca.Sort,
'"ROOL".String],'_:=_[''c1.Sort,'new_['CLID_[''Poupanca.Sort]]]]]
```

Meta-programa pré-processado pelos geradores de etiquetas:

```
'var_._end['_::_['_:_[''Poupanca.VarDec,'str.String],'_:_[
''c1.Sort,'CLID_[''Poupanca.ClassName]]],'_;_['_:=_[''Poupanca.Sort,
'"ROOL".String],'_:=_[''c1.Sort,'new_['CLID_[''Poupanca.ClassName]]]
]]
```

Figura 4.9: Trecho de código-fonte exemplo, seu meta-programa e seu meta-programa pré-processado.

As etiquetas funcionam como classificadores. Os geradores de etiquetas agem na fase de pré-processamento do meta-programa, ou seja, antes de ser gerada a árvore sintática. Assim, o processo de classificação dos identificadores é feito uma só vez, reduzindo a quantidade de processamento necessária para a classificação dos mesmos. Na Figura 4.9 podemos ver um exemplo de um meta-programa pré-processado. Tratando ainda do identificador 'Poupanca, notamos agora que suas aparições estão marcadas com etiquetas que as classificam como declaração de variável (VarDec) e um nome de classe (ClassName). A outra aparição de 'Poupanca está etiquetada como Sort, o que indica que ela é uma referência à variável 'Poupanca. A seguir serão apresentadas quais são as etiquetas implementadas e o princípio básico de como elas são geradas pelos geradores de etiquetas.

Geradores de etiquetas

Os geradores de etiquetas são responsáveis por etiquetar identificadores classificando-os de diversas formas de modo que possamos diferenciá-los. Eles são operadores com comportamento implementado por regras de re-escritura. Os vários geradores de etiquetas estão listados na Tabela 4.2. Nesta tabela não listamos a etiqueta Sort pois ela não é criada por nenhum gerador de etiquetas. Porém, ela possui um significado como todas as outras etiquetas, indicando que um identificador é uma variável. Os geradores de etiquetas estão descritos no módulo

TERMBUILDER da nossa ferramenta. A implementação do módulo *TERMBUILDER* pode ser vista no Apêndice C.

Tabela 4.2: Lista dos geradores de etiquetas.

Nome do operador	**Descrição**	**Etiquetas**
`preParseClidMtid`	Marca os identificadores filhos dos operadores `CLID` e `MtID`	`ClassName` e `MethName`
`preParsePoint`	Marca os atributos selecionados em *left expressions*	`AttrInside`
`preParseVarDec`	Marca identificadores em declarações de variáveis	`VarDec`
`preParseAttrDec`	Marca identificadores em declarações de atributos	`AttrPri`, `AttrProt` e `AttrPub`
`preParseParDec`	Marca identificadores em declarações de comandos parametrizados	`ResDec` e `ValDec`
`preParseCompParDec`	Marca identificadores no corpo de comandos parametrizados que foram declarados como parâmetros	`ResDecAppear` e `ValDecAppear`
`preParseForAllPred`	Marca identificadores em declaração de variáveis dentro do predicado *ForAll*	`ForAll_VarDec`
`preParseCompForallPred`	Marca identificadores dentro de predicados *ForAll* que foram declarados como variáveis pelo mesmo	`ForAll_Var`

A apresentação em detalhes de todos os geradores de etiquetas neste trabalho seria algo repetitivo e entediante. Diante disso, escolhemos um único gerador, tão importante quanto os outros, e que implementa os conceitos básicos utilizados por todos os outros.

A operação `preParseClidMtid` recebe um meta-termo `Term`, reescreve-o e retorna a sua versão reescrita. A sua implementação é vista na Figura 4.10. Nela encontramos as variáveis livres `o` e `p` do tipo `Qid`, `t1` e `t2` do tipo `Term` e `tl` do tipo `TermList`. Essa operação é recursiva e trata todas as possibilidades de casamento com o meta-termo.

A meta-representação de uma operação aplicada é construída pelo operador `_[_]`. O trecho de código `'MtID_[''Const.Sort]` pode ser manipulado casando-o com a construção *o* `[t1]` onde `o` passa assumir o valor `'MtID_` e `t1` o valor `‘’Const.Sort`. Os temos são agrupados em listas com o auxílio do operador `_,_` também discutido na seção 3.4.1. Assim, o termo `'_:_[''Const.Sort,'str.String]` pode ser manipulado casando-o com a construção `o [t1,t2]` onde `o` passa assumir o valor `‘_:_`, `t1` o valor `‘’Const.Sort` e `t2` o valor `‘str.String`. A cabeça da lista é `t1` e a calda `t2`. Esse conceito é utilizado por praticamente todas as implementações deste trabalho.

Nas linhas 7 e 8 da Figura 4.10 cobrimos a possibilidade de o meta-termo ser formado por um `Qid` e uma respectiva `TermList`. Se a calda for vazia a operação é aplicada recursivamente apenas para a cabeça da lista, porém se a calda não for vazia a operação é aplicada para ambos.

A Figura 4.11 foi retirada da aplicação da operação `preParseClidMtid` no meta-programa da Figura 4.9 no momento em que Maude está aplicando uma lei a um meta-termo específico. Ela ilustra na prática como funciona o interpretador Maude.

As outras operações da Figura 4.10 nomeadas `ctrClid` e `ctrMtID` são operações auxiliares. Elas são chamadas quando, durante a varredura do meta-termo, são encontrados os

termos ‘CLID_ e ‘MtID_. As linhas 3 e 4 contemplam essa situação. Essas duas operações têm como função, reescrever um meta-termo com sua respectiva etiqueta e depois retorná-lo.

```
1. op preParseClidMtid{_} : Term -> Term .
2. rl preParseClidMtid{p} => p .
3. rl preParseClidMtid{'CLID_[p]} => 'CLID_ [ ctrClid p ] .
4. rl preParseClidMtid{'MtID_[p]} => 'MtID_ [ ctrMtid p ] .
5. rl preParseClidMtid{o[p]} => o [ p ]  .
6. rl preParseClidMtid{o[t1]} => o [ preParseClidMtid{t1} ] .
7. crl preParseClidMtid{o[t1,tl]} => o [ preParseClidMtid{t1} ] if
tl == empty .
8. crl preParseClidMtid{o[t1,tl]} => o [ preParseClidMtid{t1} ,
preParseClidMtid{tl} ] if tl =/= empty .
9. rl preParseClidMtid{o[t1,t2]} => o [ preParseClidMtid{t1} ,
preParseClidMtid{t2} ] .
10. crl preParseClidMtid{t1,tl} => preParseClidMtid{t1} if tl ==
empty .
10. crl preParseClidMtid{t1,tl} => preParseClidMtid{t1} ,
preParseClidMtid{tl} if tl =/= empty .
11.
12. op ctrClid_ : Term -> Term .
13. rl ctrClid t1 => qid( string ( getName(t1) ) ++ ".ClassName" ) .
14. op ctrMtid_ : Term -> Term .
15. rl ctrMtid t1 => qid( string ( getName(t1) ) ++ ".MethName" ) .
```

Figura 4.10: Código-fonte do gerador de etiquetas para nomes de classes e métodos.

```
********** rule
rl preParseClidMtid{o[t1,t2]} =>
o[preParseClidMtid{t1},preParseClidMtid{t2}] .
o --> '_::_
t1 --> '_:_[''Poupanca.Sort,'str.String]
t2 --> '_:_[''c1.Sort,'CLID_[''Poupanca.Sort]]
preParseClidMtid{'_::_['_:_[''Poupanca.Sort,'str.String],'_:_[''c1.Sort,'CL
ID_[ ''Poupanca.Sort]]]}
--->
'_::_[preParseClidMtid{'_:_[''Poupanca.Sort,'str.String]},preParseClidMtid{
'_:_[
    ''c1.Sort,'CLID_[''Poupanca.Sort]]}]
```

Figura 4.11: Momento em que Maude aplica uma lei de reescritura da operação `preParseClidMtid`.

4.2.2 Geração das árvores sintáticas

A geração de árvores sintáticas na nossa ferramenta é feita por uma série de operações associadas cujo comportamento é implementado por regras de re-escritura. Essas operações unidas compõem o *parser* do nosso sistema. Todas essas operações estão especificadas no módulo *TERMBUILDER*. A implementação deste módulo pode ser vista no Apêndice C.

Vamos explicar o funcionamento do nosso *parser* através de um exemplo simples, mostrando cada fase do processamento executado pelo interpretador de Maude. A simplicidade

do exemplo se deve a quantidade de re-escrituras e simplificações que o interpretador de Maude faz para gerar as árvores. Qualquer exemplo um pouco mais complexo que este, nos impossibilitaria de mostrá-lo com tanta riqueza de detalhes. Apesar de simples, o exemplo escolhido é completo e serve de base para o entendimento de algo mais extenso. Ao longo desta seção, apresentaremos os trechos de cada fase de processamento provenientes do *console* de Maude. Neles marcamos com itálico a regra que está sendo aplicada e com negrito, o resultado da aplicação.

O exemplo escolhido é o comando de atribuição `'a := "ROOL"`. Para gerar a sua árvore utilizamos a operação *parse* cuja assinatura é mostrada abaixo:

```
op parse_ : Term -> State .
```

Ela recebe um meta-termo (`Term`) e transforma-o em um estado (`State`). O conjunto `State` é um tipo de dado criado para representar os membros da árvore (nós da árvore). Uma árvore é composta por uma lista de elementos do tipo `State`. Aplicando a operação `parse` ao meta-termo do nosso exemplo obtemos o seguinte resultado:

```
lastId 3,
parsed 0 = ('_:=_[1,2])),
parsed 1 = (''a.Sort[0])),
parsed 2 = ('"ROOL".String[0])
```

O resultado acima é a árvore sintática que representa o nosso exemplo. Agora vamos mostrar cada passo executado pela operação `parse`, e no final dessa seção voltaremos a discutir esse resultado.

Abaixo é mostrado o primeiro passo do processamento quando chamamos a operação *parse* passando como parâmetro o meta-programa do nosso exemplo.

```
rewrite in TERMBUILDER :
parse ('_:=_[''a.Sort,'"ROOL".String]) .
*********** rule
rl parse t1 => lastId 1, unParsed 0 = t1 .
t1 --> '_:=_[''a.Sort,'"ROOL".String]
parse ('_:=_[''a.Sort,'"ROOL".String])
--->
lastId 1, unParsed 0 = ('_:=_[''a.Sort,'"ROOL".String])
```

No trecho acima vimos o aparecimento de duas novas operações `lastId` e `unParsed`. Suas assinaturas são:

```
op lastId_ : Nat -> State .
op unParsed_=_ : Nat Term -> State .
```

Ambas resultam em um elemento do tipo `State`. A primeira nos diz o número do identificador do último nó da árvore. A segunda serve como função auxiliar para indicar termos que ainda não passaram pelo *parser*. O próximo passo do processamento é apresentado a seguir.

```
rl lastId N1, unParsed N2 = (o[tl]) => lastId (N1 + sizeof tl),
(parsed N2 = (o[argsof tl N1])), nodesof tl N1 .
N1 --> 1
N2 --> 0
```

```
o --> '_:=_
tl --> ''a.Sort,'"ROOL".String
lastId 1, unParsed 0 = ('_:=_[''a.Sort,'"ROOL".String])
--->
lastId (1 + sizeof (''a.Sort,'"ROOL".String)),
parsed 0 = ('_:=_[argsof (''a.Sort,'"ROOL".String) 1]),
nodesof (''a.Sort,'"ROOL".String) 1
```

A regra apresentada acima monta a estrutura da árvore (em negrito). É possível perceber isso comparando o resultado acima com o resultado final da operação `parse` apresentado anteriormente. A regra cria os itens `lastId` e `parsed 0`, respectivamente, o último identificador e a raíz da árvore. É a partir dessa estrutura que a árvore é construída. As etapas posteriores da construção da árvore são constituídas pela aplicação das regras `sizeof`, `argsof` e `nodesof` que se encontram nas instruções `sizeof (''a.Sort,'"ROOL".String)`, `argsof (''a.Sort,'"ROOL".String) 1` e `nodesof (''a.Sort,'"ROOL".String) 1`, respectivamente. As assinaturas dessas operações estão descritas abaixo.

```
op nodesof__ : TermList Nat -> State .
op sizeof_ : TermList -> Nat .
op argsof__ : TermList Nat -> Args
```

A operação `nodesof` recebe uma lista de termos e um natural e retorna um `State`. O que ela faz é reescrever uma de lista meta-termos como uma lista de elementos `unParsed`. Os elementos `unParsed` se tornam membros finais da nossa árvore. A seguir é mostrado, como a instrução `nodesof (''a.Sort,'"ROOL".String) 1` resulta no restante dos membros (nós) da árvore.

```
crl nodesof (t1,tl) N1 => (unParsed N1 = t1), nodesof tl N1 + 1
if tl =/= empty = true .
t1 --> ''a.Sort
tl --> '"ROOL".String
N1 --> 1
nodesof (''a.Sort,'"ROOL".String) 1
--->
(unParsed 1 = ''a.Sort), nodesof '"ROOL".String 1 + 1
*********** equation (built-in equation for symbol _+_)
1 + 1 ---> 2

rl unParsed N1 = o => parsed N1 = (o[0]) .
N1 --> 1
o --> ''a.Sort
unParsed 1 = ''a.Sort
--->
parsed 1 = (''a.Sort[0])

rl nodesof t1 N1 => unParsed N1 = t1 .
t1 --> '"ROOL".String
N1 --> 2
nodesof '"ROOL".String 2
--->
unParsed 2 = '"ROOL".String
```

```
rl unParsed N1 = o => parsed N1 = (o[0]) .
N1 --> 2
o --> '"ROOL".String
unParsed 2 = '"ROOL".String --->
parsed 2 = ('"ROOL".String[0])
```

A operação `sizeof` retorna o tamanho de uma lista de termos. Abaixo, é mostrado como a instrução `1 + sizeof (''a.Sort,'"ROOL".String)` é processada. O resultado final do processamento é visto na última linha. O natural `2` representa o resultado dado pela instrução `sizeof`. O natural `3` representa o último identificador livre da árvore (`lastId`).

```
crl sizeof (t1,tl) => 1 + sizeof tl if tl =/= empty = true .
t1 --> ''a.Sort
tl --> '"ROOL".String
sizeof (''a.Sort,'"ROOL".String)
--->
1 + sizeof '"ROOL".String
*********** equation
(built-in equation for symbol _+_)
1 + 1 + sizeof '"ROOL".String
--->
sizeof '"ROOL".String + 2

rl sizeof t1 => 1 .
t1 --> '"ROOL".String
sizeof '"ROOL".String
--->
1
*********** equation (built-in equation for symbol _+_)
1 + 2
--->
3
```

A operação `argsof` retorna quais são os identificadores dos nós correspondentes a uma determinada lista de termos, dado o último identificador livre que possa ser utilizado. O conjunto `Args` é um subconjunto dos Naturais que pode ser tanto um único número como uma lista de números. A seguir é mostrado como a instrução `nodesof (''a.Sort,'"ROOL".String) 1` resulta na lista de identificadores filhos da raiz (`1,2`).

```
crl argsof (t1,tl) N1 => N1,argsof tl N1 + 1 if tl =/= empty =
true .
t1 --> ''a.Sort
tl --> '"ROOL".String
N1 --> 1
argsof (''a.Sort,'"ROOL".String) 1
--->
1, argsof '"ROOL".String 1 + 1
*********** equation(built-in equation for symbol _+_)
1 + 1 ---> 2

rl argsof t1 N1 => N1 .
t1 --> '"ROOL".String
```

```
N1 --> 2
argsof '"ROOL".String 2
--->
2
```

Após mostrado o resumo de todo o processamento feito por nosso *parser*, vamos voltar a estrutura final resultante, a árvore sintática. Para fins didáticos, mostramos novamente a seguir, a árvore sintática que representa o nosso exemplo.

```
lastId 3,
parsed 0 = ('_:=_[1,2]),
parsed 1 = (''a.Sort[0]),
parsed 2 = ('"ROOL".String[0])
```

No topo temos a indicação do último identificador livre que pode ser utilizado. Em seguida vem a árvore propriamente dita. Cada membro `parsed` da árvore é composto por seu número identificador e um elemento do tipo `Node`. Isso pode ser constatado na assinatura do operador `parsed`:

```
op parsed_=_ : Nat Node -> State .
```

Um elemento do tipo *Node* é criado a partir do operador que o representa:

```
op _[_] : Qid Args -> Node .
```

Os nós que possuem filhos, guardam seus identificadores dentro dos colchetes. As folhas da árvore são identificadas com 0 dentro dos colchetes.

Para facilitar a visualização da árvore mostrada anteriormente, geramos uma representação gráfica para ela, que pode ser vista na Figura 4.12.

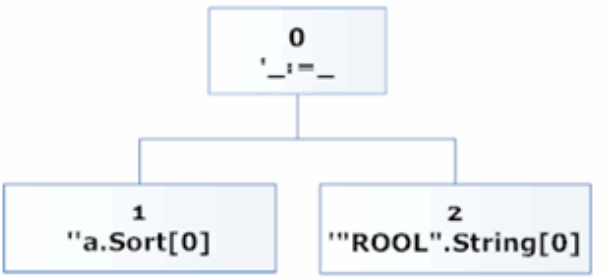


Figura 4.12: Representação gráfica de uma árvore sintática.

Para finalizar, apresentamos na Figura 4.13 a árvore sintática gerada a partir do meta-programa pré-processado referente a classe `Poupanca` do programa `BANCO` da Figura 4.3.

A seguir, apresentaremos a técnica de navegação desenvolvida para tratar a árvore definida nesta subseção.

```
lastId 38,
(parsed 0 = ('class_extends___end[1,(2,(3,4))])),
(parsed 1 = ('CLID_[5])),
(parsed 2 = ('CLID_[6])),
(parsed 3 = ('nullAtribute.Attrs[0])),
(parsed 4 = ('_*_[8,9])),
(parsed 5 = (''Poupanca.ClassName[0])),
(parsed 6 = ('CLID_[7])),
(parsed 7 = (''Conta.ClassName[0])),
(parsed 8 = ('meth_^=_end[10,11])),
(parsed 9 = ('meth_^=_end[18,19])),
(parsed 10 = ('MtID_[12])),
(parsed 11 = ('_.#_#[13,14])),
(parsed 12 = (''Const.MethName[0])),
(parsed 13 = ('val_;[15])),
(parsed 14 = ('super._<_>[26,27])),
(parsed 15 = ('_:_[16,17])),
(parsed 16 = (''numeroconta.ValDec[0])),
(parsed 17 = ('str.String[0])),
(parsed 18 = ('MtID_[20])),
(parsed 19 = ('_.#_#[21,22])),
(parsed 20 = (''renderJuros.MethName[0])),
(parsed 21 = ('val_;[23])),
(parsed 22 = ('super._<_>[29,30])),
(parsed 23 = ('_:_[24,25])),
(parsed 24 = (''taxa.ValDec[0])),
(parsed 25 = ('int.Int[0])),
(parsed 26 = ('MtID_[28])),
(parsed 27 = (''numeroconta.ValDecAppear[0])),
(parsed 28 = (''Const.MethName[0])),
(parsed 29 = ('MtID_[31])),
(parsed 30 = ('_<_>[32,33])),
(parsed 31 = (''creditar.MethName[0])),
(parsed 32 = ('Prod.BuiltinFunction[0])),
(parsed 33 = ('_:::_[34,35])),
(parsed 34 = ('_._[36,37])),
(parsed 35 = (''taxa.ValDecAppear[0])),
(parsed 36 = ('self.LeftExp[0])),
parsed 37 = (''saldo.AttrInside[0])
```

Figura 4.13: Árvore sintática do meta-programa pré-processado referente a classe `Poupanca` do programa `BANCO`.

4.3 Inspecionando árvores sintáticas: Conceito e prática

Tão importante quanto gerar árvores sintáticas de programas é conseguir extrair delas as informações necessárias para executar as verificações de condições de aplicação de leis de

programação. Diante disso, surge a necessidade de elaboração de meios para inspeção dessas árvores. Esta seção abordará as técnicas básicas de inspeção que alicerçam a grande maioria dos algoritmos implementados neste trabalho.

4.3.1 Caminhando em árvores com o algoritmo *preordertcc*

O algoritmo `preordertcc`, elaborado neste trabalho, implementa um processo de *tree traversal* [18], também conhecido como *walking in tree*(caminhando em árvore). Ele serve de base para algoritmos mais complexos que executam navegação em árvores.

Como o nome já diz, o `preordertcc`, é baseado no algoritmo de pré-ordem clássico de travessia em árvores. Resumidamente, este algoritmo recursivo visita cada nó da árvore antes de qualquer um dos seus filhos. Observando a árvore da Figura 4.14, podemos listar a seqüência de nós visitados por um algoritmo de pré-ordem típico. A seqüência seria: A, H, G, I, F, E, B, C, D.

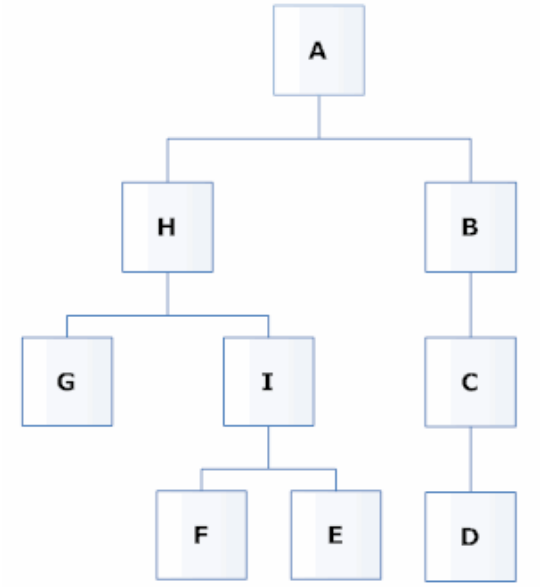


Figura 4.14: Árvore exemplo.

O algoritmo `preordertcc` é implementado por meio de três operações. A entrada de dados é feita pela operação `preordertcc` que recebe uma árvore (`State`) e retorna uma lista de nós (`List`). A navegação propriamente dita é feita pelas operações `basedepthfunc` e `baselookup`. Ambas recebem como entrada uma lista de identificadores (`Args`) e uma árvore, e retornam uma lista de elementos desta árvore. O código-fonte dessas três operações pode ser visto na Figura 4.15. Nela `a1`, `a2` e `ars` são variáveis do tipo `Args`, `s`, do tipo `State` e `q`, do tipo `Qid`. A função `lengthargs` é uma função básica que retorna o tamanho de uma lista de identificadores (`Args`).

A operação `preordertcc`, lista a raiz e chama a operação `basedepthfunc` para os filhos dela (linha 6 da Figura 4.15). A operação `basedepthfunc` recebe uma lista de identificadores (filhos) de um nó e chama a operação `baselookup` para cada elemento dessa lista. Então a operação `baselookup` faz o casamento entre o identificador passado com a árvore, e retorna o elemento correspondente àquele identificador. Se o elemento for uma folha da árvore

(linha 11) ela simplesmente retorna o elemento, porém se for um nó com filhos (linha 12), ela retorna o elemento e chama recursivamente a operação `basedepthfunc` para os filhos.

```
1. op preordertcc : State -> List .
2. op basedepthfunc{_}{_} : Args State -> List .
3. op baselookup : Args State -> List .
4.
5. eq preordertcc (lastId n, s) = preordertcc (s) .
6. eq preordertcc ((parsed n = q [ars]) , s) = {n} q [ars] :
basedepthfunc{ars}{s} .
7.
8. eq basedepthfunc{(a1,a2)}{s} = baselookup (a1, s) ++
basedepthfunc{a2}{s} .
9. ceq basedepthfunc{a1}{s} = baselookup (a1, s) if lengthargs(a1)
== 1 .
10.
11. ceq baselookup (a1, ((parsed a1 = q [ars]), s)) = {a1} q [ars] :
[]  if ars == 0 .
12. ceq baselookup (a1, ((parsed a1 = q [ars]), s)) = {a1} q [ars] :
basedepthfunc{ars}{s}  if ars =/= 0 .
```

Figura 4.15: Código-fonte do algoritmo `preordertcc`.

A Figura 4.16 mostra o resultado da operação `preordertcc` aplicada à árvore da Figura 4.13. Cada elemento da lista é formado por seu identificador (em negrito) seguido do seu conteúdo e seus filhos. Se o elemento não possui filhos ele é marcado com o número 0.

Assim finalizamos os conceitos básicos de navegação em árvores. Na próxima seção mostraremos a implementação de operações básicas de consulta a elementos das árvores. Apesar da simplicidade, essas operações são de muita utilidade na criação de operações mais complexas.

```
{0}'class_extends___end[1,(2,(3,4))] : {1}'CLID_[5] :
{5}''Poupanca.ClassName[0] : {2}'CLID_[6] : {6}'CLID_[7] :
{7}''Conta.ClassName[0] : {3}'nullAtribute.Attrs[0] : {4}'_*_[8,9] :
{8}'meth_^=_end[10,11] : {10}'MtID_[12] : {12}''Const.MethName[0] :
{11}'_.#_#[13,14] : {13}'val_;[15] : {15}'_:_[16,17] :
{16}''numeroconta.ValDec[0] : {17}'str.String[0] :
{14}'super._<_>[26,27] :
{26}'MtID_[28] : {28}''Const.MethName[0] :
{27}''numeroconta.ValDecAppear[0] : {9}'meth_^=_end[18,19] :
{18}'MtID_[20] : {20}''renderJuros.MethName[0] : {19}'_.#_#[21,22] :
{21}'val_;[23] : {23}'_:_[24,25] : {24}''taxa.ValDec[0] :
{25}'int.Int[0] : {22}'super._<_>[29,30] : {29}'MtID_[31] :
{31}''creditar.MethName[0] : {30}'_<_>[32,33] :
{32}'Prod.BuiltinFunction[0] : {33}'_:::_[34,35] : {34}'_._[36,37] :
{36}'self.LeftExp[0] : {37}''saldo.AttrInside[0] :
{35}''taxa.ValDecAppear[0] : ([]).List
```

Figura 4.16: Resultado da aplicação da operação `preordertcc` a árvore sintática da Figura 4.13.

4.3.2 Operações auxiliares

Além de percorrer as árvores sintáticas, é vital que consigamos, através de operações, extrair algumas outras informações úteis, como por exemplo, descobrir a qual elemento pertence um determinado identificador.

A operação `nodeLookup` cuja implementação é encontrada abaixo, retorna o nó correspondente a um determinado identificador.

```
op nodeLookup : Args State -> Node .
eq nodeLookup (a1, ((parsed a1 = q [ars]), s)) = q [ars] .
```

A operação é bastante simples em termos de implementação, porém, isso só é conseguido pelo poder que Maude nos oferece para tratar esse tipo de construção. Como a árvore na verdade é uma lista de elementos, Maude faz o casamento direto entre o identificador dado como argumento e o mesmo identificador dentro da lista.

As operações `qidLookup` e `argLookup` possuem implementação semelhante a operação `nodeLookup`. Ambas têm como operandos um identificador, e uma árvore sintática. A diferença é que na primeira retornamos apenas o *quoted identifier* referente àquele nó e na segunda retornamos os identificadores dos filhos do nó. As implementações dessas operações são como se segue.

```
op qidLookup : Args State -> Qid .
op argLookup : Args State -> Args .
eq qidLookup (a1, ((parsed a1 = q [ars]), s)) = q .
ceq argLookup ( a1, ((parsed a1 = q [ars]), s)) = ars  if ars =/= 0 .
```

4.4 Resumo

Neste capítulo, detalhamos os aspectos sintáticos necessários para a execução das verificações das condições de aplicação de leis de programação. Abordamos desde o reconhecimento de programas de ROOL por Maude até os fundamentos necessárias para a inspeção das árvores sintáticas que simbolizam esses programas.

Como foi visto, um programa em ROOL passa por uma série de etapas até se tornar uma árvore sintática inspecionável. Estas etapas estão agrupadas no diagrama da Figura 4.17.

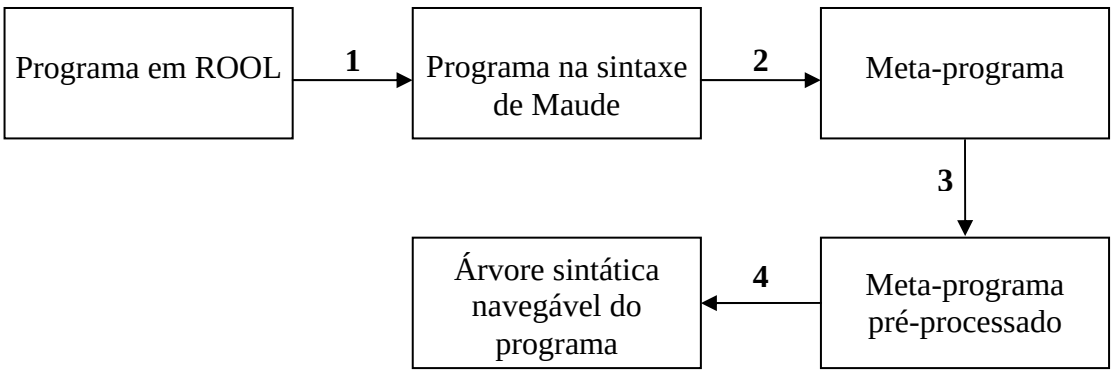


Figura 4.17: Diagrama mostrando as etapas que uma especificação ROOL passa até se tornar uma árvore sintática.

Resumidamente temos:

- Etapa 1: É feita a descrição do programa escrito em ROOL no nosso sistema.
- Etapa 2: Criação do meta-programa através da operação *upTerm*.
- Etapa 3: Pré-processamento do meta-programa para a criação das etiquetas.
- Etapa 4: Geração da árvore sintática do programa por nosso *parser*.

No próximo capítulo, abordaremos a implementação que nos possibilitou fazer as verificações de condições para aplicação de leis de programação.

Capítulo 5

Implementando as verificações

No capítulo anterior apresentamos os fundamentos necessários para a implementação das verificações das condições para aplicação de leis de programação. Neste capítulo apresentaremos os módulos da nossa implementação que descrevem as verificações das condições selecionadas para serem estudadas. Na implementação das operações vamos omitir alguns trechos, que serão indicados por "...". As definições completas das operações apresentadas neste capítulo estão no Apêndice D.

Na Figura 5.1 apresentamos um diagrama mostrando a relação de dependência entre todos os módulos da nossa ferramenta. O bloco denominado "MÓDULOS SINTÁTICOS", representa a união de todos os módulos que especificam as implementações descritas no capítulo anterior. No módulo *COMMAND-LAW-CONDITIONS* estão especificadas as operações implementadas para executar as verificações de condições de leis de comando e no módulo *CLASS-LAW-CONDITIONS* às operações responsáveis por tratar as verificações de leis de classe.

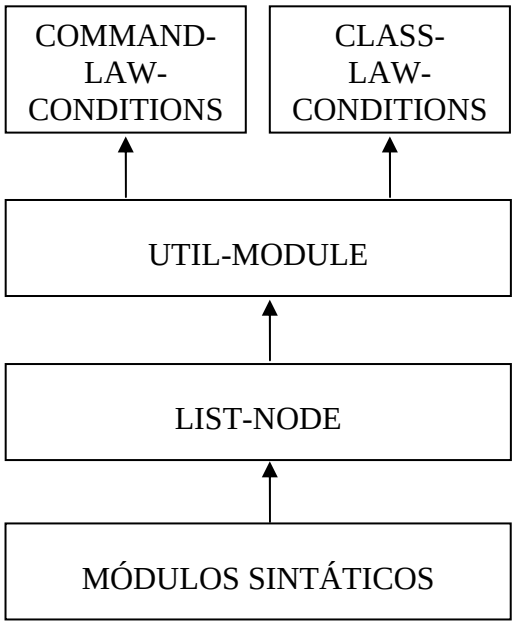


Figura 5.1: Diagrama que mostra o inter-relacionamento entre os módulos da nossa ferramenta.

Este capítulo está dividido em duas partes. Primeiro, são tratadas as condições para a aplicação das leis de comandos. Depois são tratadas as condições relativas às leis de classe.

5.1 Leis de comando

Como vimos na Seção 2.2 desta monografia, as leis de comandos tratam as construções imperativas da nossa linguagem. Estas leis na sua maioria, possuem condições associadas que permitem ou não, que elas possam ser aplicadas.

Algumas leis possuem condições simples, de verificação mais direta, como a Lei 1 <*:= skip*>, que apenas exige que se verifique se uma *left expression* é diferente de *error.* Outras leis não possuem condições para serem aplicadas. Porém, encontramos leis cujas condições de aplicação são complexas e mais difíceis de serem verificadas. Entre essas condições, podemos destacar duas. Elas são nomeadas aqui como “variáveis livres” e “expressões à esquerda”. Elas se destacam por aparecerem na maioria das leis de comando. Introduziremos seus conceitos e suas implementações ao longo desta seção.

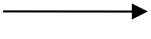
5.1.1 Variáveis livres

Uma variável pode ser livre em um comando, em um predicado ou em uma expressão. Uma variável é dita livre em um comando se alguma referência dela é encontrada dentro do comando, excetuando a situação na qual o próprio comando inclui uma declaração para ela. Por exemplo, no comando `var x:int . y :=x end`, a variável `y` é livre, enquanto a variável `x` não é livre porque surge dentre as variáveis introduzidas no bloco de variáveis. Se considerarmos apenas o comando `y := x`, tanto a variável `x` quanto `y` são livres. Um outro exemplo seria se considerássemos a parte interna do bloco de variáveis do comando principal do programa *Banco* da Figura 4.3. A variável `c2` não é livre, pois ela está referenciada tanto na criação do objeto `ContaEspecial` quanto na chamada do método `debitar` que vem logo abaixo. O mesmo raciocínio é utilizado para predicados e expressões.

A condição de variáveis livres é encontrada em leis como Lei 26 <*var rename*> e a Lei 25 <*var elim*>, ambas descritas no Apêndice B. Esta última está ilustrada através do exemplo da Figura 5.2. Nele, está descrita uma situação na qual inserimos a declaração de uma nova variável `x` do tipo `int`. É possível fazer isso porque `x` não é livre no comando representado no *Quadro 1*. Porém, se quiséssemos declarar uma variável chamada `y`, isso não seria possível , pois `y` é livre no comando mostrado no *Quadro 1*.

Quadro 1

```
var x : str .
     c1 ; c2 ; c3 ;
     y := x
end
```



Quadro 2

```
var x : int
     var x : str .
          c1 ; c2 ; c3 ;
          y := x
     end
end
```

Figura 5.2: Trecho de código representando a aplicação da Lei 25 <*var elim*>.

Apesar de parecer simples, como no exemplo anterior, verificar se uma variável é livre em uma construção qualquer implica algumas dificuldades. Elas estão tratadas na implementação desenvolvida para verificar se uma variável é livre ou não em um determinado comando que será discutida a seguir.

Implementação

A operação `isFree`, que implementa a verificação da condição de variáveis livres, é composta por três outras operações: `vlSearchDepth`, `vlDepthpthFunc` e `vlLookup`. A Figura 5.3 mostra o código-fonte dessas operações. Nela `a1`, `a2` e `ars` são variáveis do tipo `Args`; `s` do tipo `State`; e `q`, `q1` do tipo `Qid`. A operação `isDeclaredIn` é uma operação auxiliar, que diz se uma variável está declarada em um bloco de variáveis ou não.

A operação `isFree` toma como argumentos a variável a ser procurada, e um comando estruturado como uma árvore, onde a procura pela variável será realizada. Sua assinatura está descrita na linha 1 da Figura 5.3.

```
1. op isFree : Qid State -> Bool .
2. op vlSearchDepth : Qid Args State -> Bool .
3. op vlDepthFunc{_}{_}{_} : Qid Args State -> List .
4. op vlLookup : Qid Args State -> List .
5.
6. eq isFree (q, (lastId n, s)) = isFree (q, s) .
7. eq isFree (q, ((parsed n = q1 [ars]) , s)) = vlSearchDepth (q,
ars, s) .
8.
9. ceq vlSearchDepth (q,ars, s) = false if
     length(vlDepthFunc{q}{ars}{s}) == 0 .
10. ceq vlSearchDepth (q,ars, s) = true if
     length(vlDepthFunc{q}{ars}{s}) > 0 .

11. eq vlDepthFunc{q}{0}{s} = [] .
12. eq vlDepthFunc{q}{(a1,a2)}{s} = vlLookup (q, a1, s) ++
     vlDepthFunc{q}{a2}{s} .
13. ceq vlDepthFunc{q}{a1}{s} = vlLookup (q, a1, s) if
     lengthargs(a1) == 1 .
14.
15. ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = q : []  if
     ars == 0 and getName(q) == q1 and getType(q) == 'Sort .
16. ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars
     == 0 and getName(q) == q1 and getType(q) =/= 'Sort .
17. ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars
     == 0 and getName(q) =/= q1 .
18. ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars
     =/= 0 and q == 'var_._end and isDeclaredIn{takeArg 0 from
     ars}{q1}{s} .
19. ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) =
     vlDepthFunc{q1}{ars}{s}  if ars =/= 0 .
```

Figura 5.3: Código-fonte da operação `isFree`.

O fluxo básico do algoritmo `isFree` é mostrado a seguir. Considere que o usuário entrou com o comando `isFree (var, s)`, onde `var` é um `Qid` que representa o nome de uma variável e `s` um `State` representando uma árvore sintática.

1) A primeira ação da operação `isFree` é retirar o identificador `lastId` e chamá-la recursivamente passando a árvore propriamente dita. Depois disso, ela chama a operação `vlSearchDepth` para os identificadores dos nós filhos da raiz da árvore. (linha 7 da Figura 5.3).
2) A operação `vlSearchDepth` então, chama a função `vlDepthFunc` repassando os argumentos de entrada.
3) A operação `vlDepthFunc` recebe uma lista de identificadores de nós (neste ponto, dos filhos da raiz), chama a função `vlBaseLookup` para a cabeça da lista e chama a si mesma para o restante da lista, até que a lista esteja esgotada. Ao final desta etapa teremos uma chamada da operação `vlBaseLookup` para cada filho da raiz.
4) A operação `vlBaseLookup` caminha pela árvore de forma recursiva em busca de nós da árvore que correspondam a referências de variáveis. Se o nó for igual à variável que estamos procurando, ela é inserida no retorno da operação. Esse conceito é baseado na operação `baseLookup` discutida na seção 4.3.1.
5) Ao final da etapa 4 teremos uma lista composta pelas referências da variável procurada encontradas ao longo da árvore sintática. Esta lista é então utilizada pela operação `vlSearchDepthFunc`. Se a lista não for vazia, ou seja, se a variável é livre na árvore, o retorno dela é *true*, senão o retorno é *false*.
6) O retorno da operação `vlSerachDepth` é repassado para a operação `isFreeInDepth` que fornece a resposta para o usuário.

A seguir fecharemos o estudo sobre as condições de leis de comando, com a condição de expressões à esquerda.

5.1.2 Expressões à esquerda

A condição de expressões à esquerda é referente a uma variável e um comando. Uma variável é usada como uma expressão à esquerda em um comando se ela aparece como alvo de atribuição ou alvo de parâmetro passado por resultado ou, ainda, como alvo de chamada de métodos. A Tabela 5.1 lista situações onde uma variável `x` qualquer é usada como expressão à esquerda.

Tabela 5.1: Expressões à esquerda.

Alvo de atribuição	`x := a`
Alvo de parâmetro de resultado	`(res arg : T . c) (x) obj.m(x), sendo m ^= (res i : T . c)`
Alvo de chamada de método	`x.meth()`

Entre as leis que fazem referência a essa condição podemos citar Lei 62 <*var block-val*>, Lei 63 <*var block-res*>, Lei 64 <*pcom elimination-val*> e Lei 65 <*pcom elimination-res*>. A seguir será apresentada a implementação desenvolvida para tratar a condição de expressões à esquerda.

Implementação

A condição de expressões à esquerda está implementada pela operação `isLeftExp` em conjunto com três outras operações: `searchLe`, `depthFuncLe` e `lookupLe`. A Figura 5.4 mostra o código-fonte das operações `isLeftExp`, `searchLe` e `depthFuncLe` e a Figura 5.5 mostra o da operação `lookupLe`. Em ambas, as variáveis `a1`, `a2`, e `ars` são do tipo `Args`; `q` e `q1` do tipo `Qid`; e `s`, `s'` do tipo `State`. As operações `itemPosition` e `resdepthfunc` são operações auxiliares. A primeira verifica se um dado elemento do tipo `Node` pertence a uma lista de elementos; a segunda, lista todos os parâmetros declarados em um determinado comando parametrizado.

```
1. op isLeftExp : Qid State -> Bool .
2. op searchLe : Qid Args State -> Bool .
3. op depthfuncLe{_}{_}{_} : Qid Args State -> List .
4.
5. eq isLeftExp (q, (lastId n, s)) = isLeftExp (q, s) .
6. eq isLeftExp (q, ((parsed n = q1 [ars]) , s)) = searchLe (q, ars,
((parsed n = q1 [ars]) , s)) .
7.
8. ceq searchLe (q,ars, s) = false if length(depthfuncLe{q}{ars}{s})
== 0 .
9. ceq searchLe (q,ars, s) = true if length(depthfuncLe{q}{ars}{s})
> 0 .
10.
11. eq depthfuncLe{q}{0}{s} = [] .
12. eq depthfuncLe{q}{(a1,a2)}{s} = lookupLe (q, a1, s) ++
depthfuncLe{q}{a2}{s} .
13. ceq depth funcLe{q}{a1}{s} = lookupLe (q, a1, s) if
lengthargs(a1) == 1 .
```

Figura 5.4: Código-fonte das operações `isLeftExp`, `searchLe` e `depthfuncLe`.

A operação `isLeftExp` toma como parâmetros a variável a ser verificada, e um comando estruturado como uma árvore, alvo dos testes com a variável. Se for constatado, que em algum ponto da árvore, a variável está sendo usada como uma expressão à esquerda o resultado é *true*, se não, o resultado é *false*. A assinatura da operação `isLeftExp` está descrita na linha 1 da Figura 5.4.

O fluxo básico do algoritmo `isLeftExp` pode ser visto a seguir. Considere que o usuário entrou com o comando `isLeftExp(var, s)`, onde `var` é um identificador qualquer do tipo `Qid` e `s` uma árvore sintática do tipo `State`.

1) A primeira ação da operação `isLeftExp` é retirar o identificador `lastId` e chamá-la recursivamente passando a árvore propriamente dita. Depois disso, ela chama a operação `searchLe` para os identificadores dos nós filhos da raiz da árvore. (linha 6 da Figura 5.4).
2) A operação `searchLe` então, chama a função `depthfuncLe` repassando os argumentos de entrada.

3) A operação `depthfuncLe` recebe uma lista de identificadores de nós (neste ponto, dos filhos da raiz), chama a função `lookupLe` para a cabeça da lista e chama ela mesma para o restante da lista, até que a lista esteja esgotada. Ao final desta etapa teremos uma chamada da operação lookupLe para cada filho da raiz.
4) A operação `lookupLe` (Figura 5.5) caminha pela árvore de forma recursiva em busca das situações que caracterizam uma variável como uma expressão à esquerda. Se durante essa busca, a variável corrente for igual á variável passada como argumento, um `Qid` é acrescentado a lista de retorno.
5) A lista final retornada pela operação `depthfuncLe` é então utilizada pela operação `searchLe`. A operação `searchLe` verifica se essa lista possui elementos. Se possuir, ela retorna *true,* se não retorna *false.*
6) Por fim, a operação `isLeftExp` repassa ao usuário a resposta proveniente da etapa anterior.

```
1. op lookupLe : Qid Args State -> List .
2. ceq lookupLe (q1,a1,((parsed a1 = q [ars]),s)) = [] if ars == 0 .
3. ceq lookupLe (q1, a1, ((parsed a1 = '_:=_ [ars]), s)) = '_:=_ :
[] if head(basedepthfunc{takeArg 0 from ars}{s}) == q1 and
length(basedepthfunc{takeArg 0 from ars}{s}) == 1 .
4. eq lookupLe (q1, a1, ((parsed a1 = '_:=_ [ars]), s)) = [] .
5. ceq lookupLe (q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) =
'<_>_._<_> : [] if head(basedepthfunc{takeArg 1 from ars}{s}) == q1
and length(basedepthfunc{takeArg 1 from ars}{s}) == 1 .
6. ceq lookupLe (q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) =
'<_>_._<_> : [] if ...
7. eq lookupLe (q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) = [] .
8. ceq lookupLe (q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = '_._<_>
: [] if head(basedepthfunc{takeArg 0 from ars}{s}) == q1 and
length(basedepthfunc{takeArg 0 from ars}{s}) == 1 .
9. ceq lookupLe (q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = '_._<_>
: [] if ...
10. eq lookupLe (q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = [] .
11. ceq lookupLe (q1, a1, ((parsed a1 = 'super._<_> [ars]), s)) =
'super._<_> : [] if ...
12. ceq lookupLe (q1, a1, ((parsed a1 = '_.#_#<_> [ars]), s)) =
'_.#_#<_> : [] if ...
13. ceq lookupLe (q1, a1, ((parsed a1 = '_.#_#<_> [ars]), s)) = []
if ...
14. eq lookupLe (q1, a1, ((parsed a1 = q [ars]), s)) =
depthfuncLe{q1}{ars}{((parsed a1 = q [ars]), s)} .
```

Figura 5.5: Código-fonte da operação `lookupLe`.

É importante ressaltar a função da operação `leLookup` no algoritmo. Ela, é a responsável por tratar as situações listadas na Tabela 5.1. Na linha 3 da Figura 5.5 verificamos se a variável é alvo de uma atribuição. Nas linhas 5 e 8 é feita a verificação para saber se a variável é alvo de chamada de método. E nas linhas 6, 9 e 12 verificamos se a variável é um parâmetro passado por resultado.

Assim completamos o estudo das verificações de condições relativas às leis de comandos. Na próxima seção passaremos a tratar as condições relativas às leis de classes.

5.2 Leis de classe

Esta seção é dedicada à apresentação da implementação desenvolvida neste trabalho para tratar as verificações de condições relativas às leis de classe. Essas leis foram previamente introduzidas na Seção 2.2.

Após uma análise no conjunto das leis de classe, constatamos que algumas condições se destacam, pois elas cobrem boa parte das situações encontradas na maioria das leis desse conjunto. Assim, implementando essas condições, estaríamos cobrindo não só as leis que as utilizam por completo, mas também, criando uma base técnica expansível e adaptável para a maioria das outras condições. Essas implementações serão apresentadas em detalhes no decorrer desta seção.

5.2.1 Existência de método declarado em uma classe e na hierarquia (EMC)

Essa condição está vinculada à existência da declaração de um determinado método em uma classe ou em qualquer subclasse ou superclasse dela. Ela está presente na Lei 90 <*method elimination*>. Algumas outras leis necessitam de condições parecidas. A Lei 88 <*introduce method redefinition*>, por exemplo, necessita que o método não esteja declarado apenas no interior da classe.

Agora, apresentaremos a implementação desenvolvida neste trabalho para tratar a verificação da condição EMC.

Implementação

A verificação da existência ou não de um dado método em uma classe ou na hierarquia dela, é feita através da operação `hasMethodList`. Esta, diferentemente das operações `isFree` e `isLeftExp`, não retorna apenas um booleano. Ela retorna uma lista com todas as ocorrências dos métodos encontrados (que possuem o mesmo nome do método buscado), juntamente com as classes a que eles pertencem. Isso faz com que ela seja uma operação mais genérica e mais abrangente. Se quisermos obter uma resposta do tipo Falso ou Verdadeiro, basta criarmos uma operação (por exemplo, `hasMethod`) que chama `hasMethodList` e verifica se a lista retornada possui elementos ou não e então, retornar o booleano correspondente.

A operação `hasMethodList` tem como operandos o nome do método a ser buscado, o nome da classe que será a base das buscas, e uma árvore sintática representando o programa no qual o método e a classe estão contidos. Ela é composta por outras sete operações: `subHasMethod`, `classDepthFunc`, `lookupClass`, `extendsLookupClass`, `sonsLookupClass`, `methodDepthFunc` e `lookupMethod`. As assinaturas de todas elas estão descritas na Figura 5.6.

```
op hasMethodList : Qid Qid State -> List .
op subHasMethod : Qid Qid State -> List .
op classDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupClass : Qid Qid Args State -> List .
op extendsLookupClass{_}{_}{_} : Qid Qid State -> List .
op sonsLookupClass{_}{_}{_} : Qid Qid State -> List .
op methodDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupMethod : Qid Qid Args State -> List .
```

Figura 5.6: Assinaturas das operações que implementam a condição EMC.

O código-fonte da implementação das operações da Figura 5.6 está dividido em duas figuras. Na Figura 5.7 mostramos a implementação das seis primeiras operações que são responsáveis pela varredura na hierarquia das classes. Na Figura 5.8 mostramos as duas últimas que são responsáveis pela varredura no interior das classes.

O fluxo básico do algoritmo `hasMethodList` é descrito a seguir. Considere que o usuário entrou com o comando `hasMethodList(q1, q', s)`, onde `q1` é o nome do método (do tipo `Qid`) a ser buscado, `q'` é o nome da classe (também do tipo `Qid`) e `s` uma árvore sintática (do tipo `State`).

1) A primeira ação da operação `hasMethodList` é retirar o identificador `lastId` e chamá-la recursivamente passando a árvore propriamente dita. Então ela verifica se a raiz da árvore é igual a `'_.main<_>`. Isso significa que a árvore é de um programa completo. Se a verificação for satisfeita ela chama a operação `subHasMethod`.
2) A operação `subHasMethod` chama a operação `classDepthFunc` para os identificadores dos nós que representam todas as declarações de classes do programa.
3) A operação `classDepthFunc` recebe uma lista de identificadores de nós (neste ponto, de todas as declarações de classes), chama a função `lookupClass` para a cabeça da lista e chama a si mesma para o restante da lista, até que a lista esteja esgotada. Ao final desta etapa teremos uma chamada da operação `lookupClass` para cada declaração de classe.
4) A operação `lookupClass` pode seguir por três rotas dependendo de qual condição for satisfeita. Elas estão listadas a seguir. É importante dizer aqui, que todas as rotas têm como objetivo listar possíveis ocorrências válidas de `q1`, o método que procuramos, pois todas se utilizam da operação `methodDepthFunc` para tal fim.
   I) A declaração corrente é do tipo `'class___end` e o nome da classe relativa a esta declaração é igual a `q'`. Então a operação `methodDepthFunc` é chamada para ver se `q1` está declarado dentro de `q'`.
   II) A declaração corrente é do tipo `'class_extends___end` e o nome da classe corrente é igual a `q'`. Então a operação `methodDepthFunc` é chamada para ver se `q1` esta declarado dentro de `q'`. E também a operação `extendsLookupClass` é chamada passando como argumento o nome da classe que aparece como superclasse na declaração corrente. Esta última busca de forma recursiva em todas as superclasses da classe passada como argumento, e para cada uma delas chama a operação `methodDepthFunc` para ver se `q1` está declarado em alguma delas.
   III) A declaração corrente é do tipo `'class_extends___end` e `q'` é igual ao nome da classe que aparece como superclasse nessa declaração. Então a operação `methodDepthFunc` é chamada para ver se `q1` está declarado na classe corrente. E também a operação `sonsLookupClass` é chamada passando como argumento o

nome da classe corrente. Essa última busca de forma recursiva, todas as subclasses da classe passada como argumento, e para cada uma delas chama a operação `methodDepthFunc` para ver se `q1` está declarado em alguma delas.

5) A operação `methodDepthFunc`, procura por `q1` nas declarações de métodos da classe passada como argumento. Ela chama a operação `lookupMethod` para cada declaração de método da classe.
6) Se `q1` estiver contido no conjunto de declarações de métodos da classe passada como argumento, a operação `lookupMethod` retorna um `Qid` formado pela junção de `q1` com a classe passada como argumento.
7) Quando todas as chamadas a `lookupMethod` estiverem concluídas, teremos uma lista formada pela concatenação das listas retornadas por cada uma das chamadas. Então a lista final é mostrada ao usuário como resposta à operação `hasMethodList`.

```
eq hasMethodList (q, q', (lastId n, s)) = hasMethodList (q, q', s) .
ceq hasMethodList (q, q', ((parsed n = q1 [ars]) , s)) =
subHasMethod (q, q', s) if q1 == '_.main<_> .
ceq hasMethodList (q, q', ((parsed n = q1 [ars]) , s)) = [] if ...
eq subHasMethod (q, q', ((parsed n = q1 [ars]) , s)) =
classDepthFunc{q}{q'}{ars}{s} . ...
eq classDepthFunc{q}{q'}{(a1,a2)}{s} = lookupClass (q, q', a1, s) ++
classDepthFunc{q}{q'}{a2}{s} .
ceq classDepthFunc{q}{q'}{a1}{s} = lookupClass (q, q', a1, s) if ...
ceq lookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
methodDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++ []
if q == 'class___end ...
ceq lookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
methodDepthFunc{q1}{q'}{argLookup(takeArg 3 from ars , s)}{s} ++
extendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from
ars}{s})}{s} if q == 'class_extends___end ...
ceq lookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
methodDepthFunc{q1}{head(basedepthfunc{takeArg 0 from ars}{s})}
{argLookup(takeArg 3 from ars,s)}{s} ++ sonsLookupClass{q1}{head
(basedepthfunc{takeArg 0 from ars}{s})}{s}
if q == 'class_extends___end ...
eq lookupClass (q, q', a1, (s,s')) = [] .
ceq extendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
methodDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++ []
if q == 'class___end ...
ceq extendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
methodDepthFunc{q1}{q'}{argLookup(takeArg 3 from ars , s)}{s} ++
extendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from
ars}{s})}{s} if q == 'class_extends___end ...
eq extendsLookupClass {q1}{q'}{(s,s')} = [] .
ceq sonsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} = (
methodDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{argLookup(takeArg 3 from ars , s)}{s} ++
sonsLookupClass{q1}{q'}{s} ) ++ sonsLookupClass{q1}{head(
basedepthfunc{takeArg 0 from ars}{s})}{s} if ...
```

Figura 5.7: Código-fonte das operações: `hasMethodList`, `subHasMethod`, `classDepthFunc`, `lookupClass`, `extendsLookupClass` e `sonsLookupClass`.

```
eq methodDepthFunc{q1}{q'}{0}{s} = [] .
eq methodDepthFunc{q1}{q'}{(a1,a2)}{s} = lookupMethod (q1,q', a1, s)
++ methodDepthFunc{q1}{q'}{a2}{s} .
ceq methodDepthFunc{q1}{q'}{a1}{s} = lookupMethod (q1,q',a1, s) if
lengthargs(a1) == 1 .

ceq lookupMethod (q1, q', a1, ((parsed a1 = q [ars]), s)) =
qid(string(head(basedepthfunc{takeArg 0 from ars}{s})) ++ "." ++ "'"
++ string(q')) : []  if head(basedepthfunc{takeArg 0 from ars}{s})
== q1 .
eq lookupMethod (q1, q', a1, (s,s')) = [] .
```

Figura 5.8: Código-fonte da implementação das operações `methodDepthFunc` e `lookupMethod`.

A seguir apresentaremos uma condição que transcende o escopo relativo as declarações de classes, passando a varrer o programa completo.

5.2.2 Existência de chamada de método no programa (EMP)

A condição EMP, é a mais abrangente e complexa apresentada neste trabalho. Ela inclui tanto o conceito de busca na hierarquia de classes, como o de busca de ocorrências de *Qids*. A condição EMC é aplicada a um método e à classe a que ele pertence. Como exemplo, vamos supor um método qualquer *m* e *C* a classe na qual ele está declarado. Para fazer a verificação da condição EMP para o método *m*, devemos varrer todo o programa (no interior de todas as classes e do comando principal *main*) em busca de chamadas a *m* (do tipo, *B.m*) onde *B* é uma expressão com tipo declarado *C* ou uma subclasse de *C*.

A condição EMP está presente na Lei 90 <*method elimination*>. Algumas outras leis possuem condições que são variações (por exemplo, busca por seleções de atributos ao invés de métodos), ou simplificações da EMP. Entre elas estão as Leis 84 e 85 de mudança de visibilidade de atributos, a Lei 91 <*move original method to superclass*> e a Lei 89 <*move redefined method to superclass*>. Nesta última há uma condição em que é necessária a verificação da existência de chamadas a um determinado método *m* do tipo *super.m* em uma determinada classe. A seguir será apresentada a implementação da condição EMP.

Implementação

A condição EMP está implementada pela operação `listMethodCalls`. Ela tem como operandos o método a ser buscado, a classe na qual o método está declarado e a árvore sintática do programa alvo das buscas. O seu retorno é um booleano indicando a ocorrência ou não de

```
op returnDecType {_}{_} : Args State -> Qid .
op returnAttrListType : Args Qid State -> Qid .
op returnParentClassOfArg{_}{_} : Args State -> Qid .
op isOfOrSubClassOf{_}{_}{_} : Qid Qid State -> Bool .
```

Figura 5.9: Assinatura das operações auxiliares que compõe a implementação da condição EMP.

Tabela 5.2: Principais operações auxiliares da operação `listMethodCalls`.

Nome da operação	**Descrição**
`returnDecType`	Dado um identificador de um nó representando uma variável ou um argumento de um comando parametrizado, retorna o tipo declarado para ele.
`returnAttrListType`	Dada uma lista de identificadores de nós correspondendo a atributos etiquetados com `AttrInside` (por exemplo: `'var.a1.a2.a3`) e uma Classe correspondendo ao tipo declarado do alvo da seleção (no exemplo anterior, `'var`) retorna o tipo declarado do último atributo da lista (neste caso, `a3`).
`returnParentClassOfArg`	Recebe um identificador de um nó e retorna o nome da classe pai dele.
`isOfOrSubClassOf`	Dadas duas classes, verifica se a segunda é subclasse da primeira.

```
op listMethodCalls : Qid Qid State -> Bool .
op empAuxListMethodCalls : Qid Qid Args State -> Bool .
op empDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op empLookupMethodCalls : Qid Qid Args State -> List .
op empCheckSubClassRel{_}{_}{_} : Qid Args State -> Bool .
eq listMethodCalls (qm,q,(lastId n,s)) = listMethodCalls (qm,q,s) .
eq listMethodCalls (qm,q , ((parsed n = q1 [ars]),s)) =
empAuxListMethodCalls (qm , q , ars , ((parsed n = q1 [ars]) , s)) .
ceq empAuxListMethodCalls (qm, q, ars, s) = false if
length(empDepthFunc{qm}{q}{ars}{s}) == 0 .
ceq empAuxListMethodCalls (qm, q, ars, s) = true if
length(empDepthFunc{qm}{q}{ars}{s}) > 0 .
eq empDepthFunc{qm}{q}{0}{s} = [] .
eq empDepthFunc{qm}{q}{(a1,a2)}{s} = empLookupMethodCalls (qm, q,
a1, s) ++ empDepthFunc{qm}{q}{a2}{s} .
ceq empDepthFunc{qm}{q}{a1}{s} = empLookupMethodCalls (qm, q, a1, s)
if lengthargs(a1) == 1 .
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = q [ars]), s)) =
[]  if ars == 0  .
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = '_._<_> [ars]),
s)) = qm : [] if ...
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = 'super._<_>
[ars]), s)) = qm : [] if ...
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = '<_>_._<_>
[ars]), s)) = qm : [] if ...
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = q [ars]), s)) =
empDepthFunc{qm}{q1}{ars}{((parsed a1 = q [ars]), s)} if ars =/= 0 .
eq empCheckSubClassRel{q}{(a1,ars)}{s} = isOfOrSubClassOf{q}{
returnAttrListType(ars, returnDecType{a1}{s}, s)} {s} .
ceq empCheckSubClassRel{q}{a1}{s} = isOfOrSubClassOf{q}{return
ParentClassOfArg{a1}{s}}{s}if qidLookup(a1,s) == 'self.LeftExp .
ceq empCheckSubClassRel{q}{a1}{s} = isOfOrSubClassOf{q}
{returnDecType{a1}{s}}{s} if qidLookup(a1,s) =/= 'self.LeftExp .
```

Figura 5.10: Código-fonte das operações que implementam a condição EMC.

chamadas a esse método no programa. Ela é composta por outras quatro operações (`empAuxListMethodCalls`, `empDepthFunc`, `empLookupMethodCalls` e `empCheckSubClassRel`) e por algumas operações auxiliares. As assinaturas destas últimas podem ser vistas na Figura 5.9 e um resumo descritivo sobre elas é apresentado na Tabela 5.2.

O código-fonte da operação `listMethodCalls` juntamente com os códigos-fonte das outras quatro operações que a compõe estão descritos na Figura 5.10.

O fluxo básico do algoritmo `listMethodCalls`, é mostrado a seguir. Considere que o usuário entrou com o comando `listMethodCalls (qm, q1, s)`. O argumento `qm` é o nome do método a ser buscado, `q1` é o nome da classe na qual o método foi declarado e `s` é a arvore sintática do programa completo (aonde `q1` está declarada) que será alvo das buscas.

1). Primeiro, a operação `listMethodCalls`, desconsidera o identificador `lastId` e chama a si mesma recursivamente passando como argumento a árvore propriamente dita. Após isso, ela chama a operação `empAuxListMethodCalls` passando os filhos da raiz da árvore (`'_.main<_>`) e a árvore.
2). A operação `empAuxListMethodCalls` chama a operação `empDepthFunc` repassando os argumentos de entrada.
3). A operação `empDepthFunc` recebe uma lista de identificadores de nós (neste ponto, dos filhos diretos da raiz), chama a função `empLookupMethodCalls` para a cabeça da lista e também para o restante da lista, até que a lista esteja esgotada.
4). A operação `empLookupMethodCalls` tem como função checar cada nó da árvore para saber se o nó se trata de uma possível chamada do método `qm`. Ela possui cinco diferentes fluxos dependendo do tipo de chamada de método que for detectado. Tais fluxos estão descritos abaixo:
   I) A chamada de método é do tipo `‘_._<>`, como por exemplo `‘banco.’conta1.MtID ‘debitar<10>`. A operação `empCheckSubClassRel` é chamada passando-se `qm`, a lista de identificadores dos nós correspondentes à expressão alvo da chamada do método *qm* e a árvore sintática. Se a *left expression* for a palavra reservada *self* a operação `empCheckSubClassRel` verifica se a classe que é pai da *left expression self* é igual a `q1` ou é uma subclasse de `q1`. Se não for *self*, a operação `empCheckSubClassRel`, verifica se o tipo declarado do identificador da expressão que é alvo da chamada do método (no exemplo citado seria `‘conta1`) é do tipo `q1` ou é um subtipo de `q1`. Em ambos os casos, se o resultado final for *true*, esta etapa resulta em uma lista contendo uma ocorrência de `qm`.
   II) A chamada de método é do tipo `‘super.<_>`, como, por exemplo, `super.MtID ‘debitar<10>`. Primeiramente, a operação `returnParentClassOfArg` é chamada passando-se como argumentos o identificador do nó referente a `‘super.<_>` e a árvore sintática. O retorno dela é o nome da classe pai do identificador passado como argumento. Esse resultado é, então, passado à operação `isOfOrSubClassOf` que verifica se a classe é igual a `q1` ou é uma subclasse de `q1`. Se o resultado de `isOfOrSubClassOf` for *true*, esta etapa resultará em uma lista contendo uma ocorrência de `qm`.
   III) A chamada de método é do tipo `‘<_>_.<_>` que significa uma chamada de método com *cast*, como por exemplo `<CLID ‘ContaEspecial>‘conta1.MtID ‘debitar<100>`. A operação `isOfOrSubClassOf` é chamada passando-se como argumentos `qm`, o nome da classe que está fazendo a operação de *casting* (no

exemplo citado, `'ContaEspecial`) e a árvore sintática. Ela então verifica se a classe é igual a `q1` ou é uma subclasse de `q1`. Caso a verificação seja bem sucedida, esta etapa resultará em uma lista contendo uma ocorrência de `qm`.

IV) O nó corrente, não corresponde a nenhum tipo dos citados acima, e é um nó que possui filhos. Então a operação `empDepthFunc` é chamada recursivamente para os filhos desse nó.

V) O nó corrente, não corresponde a nenhum tipo dos citados acima, e é um nó que não possui filhos. Então o resultado da operação `empLookupMethodCalls` é uma lista vazia.

5). Ao final da etapa 4, após todos os nós da árvore sintática *s* terem sido visitados, todas as possíveis ocorrências válidas de `qm` estarão formando uma única lista. Esta lista então chega à operação `empAuxListMethodCalls` que verifica se ela possui itens. Se possuir, retorna *true*, senão retorna *false*.

6). O resultado da etapa 5 é então retornado à operação principal `listMethodCalls` que o repassa para o usuário.

Com o estudo do fluxo do algoritmo desenvolvido para a verificação da condição EMP, finalizamos esta seção. A seguir apresentaremos as implementações condizentes às condições relativas a atributos.

5.2.3 Existência de atributo declarado em uma classe e na hierarquia (EAC)

A condição EAC está vinculada a existência ou não da declaração de um dado atributo, em uma classe ou em uma de suas subclasses ou superclasses. Esta condição segue a minha linha da EMC porém trata declarações de atributos e não de métodos. A EAC está presente na Lei 83 <*attribute elimination*>. Na Lei 87 <*move attribute to superclasse*> encontramos uma condição similar. Nesta última é necessário que o atributo não seja declarado apenas nas subclasses. A seguir, apresentaremos a implementação da condição EAC.

Implementação

A condição EAC está implementada pela operação `hasAttributeList`. Essa operação é composta por mais sete operações. São elas: `subHasAttributeList`, `eacClassDepthFunc`, `eacLookupClass`, `eacExtendsLookupClass`, `eacSonsLookupClass`, `eacAttributeDepthFunc` e `eacLookupAttributeDec`. As assinaturas de todas elas são mostradas na Figura 5.11.

```
op hasAttributeList : Qid Qid State -> List .
op subHasAttributeList : Qid Qid State -> List .
op eacClassDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op eacLookupClass : Qid Qid Args State -> List .
op eacAttributeDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op eacLookupAttributeDec : Qid Qid Args State -> List .
op eacExtendsLookupClass{_}{_}{_} : Qid Qid State -> List .
op eacSonsLookupClass{_}{_}{_} : Qid Qid State -> List .
```

Figura 5.11: Assinaturas das operações que implementam a condição EAC.

```
eq hasAttributeList (q,q',(lastId n,s)) = hasAttributeList(q,q',s) .
ceq hasAttributeList (q, q', ((parsed n = q1 [ars]) , s)) =
subHasAttributeList (q, q', s) if q1 == '_.main<_> . ...
eq subHasAttributeList (q, q', ((parsed n = q1 [ars]) , s)) =
eacClassDepthFunc{q}{q'}{ars}{s} .
eq eacClassDepthFunc{q}{q'}{0}{s} = [] .
eq eacClassDepthFunc{q}{q'}{(a1,a2)}{s} = eacLookupClass (q, q', a1,
s) ++ eacClassDepthFunc{q}{q'}{a2}{s} .
ceq eacClassDepthFunc{q}{q'}{a1}{s} = eacLookupClass (q,q', a1, s)
if lengthargs(a1) == 1 .
ceq eacLookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 1 from ars , s)}{s}
++ []  if q == 'class___end ...
ceq eacLookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s}
++ eacExtendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from
ars}{s})}{s} if q == 'class_extends___end ...
ceq eacLookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
eacAttributeDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{argLookup(takeArg 2 from ars , s)}{s} ++
eacSonsLookupClass{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{s} if q == 'class_extends___end ...
eq eacLookupClass (q1, q', a1, (s,s')) = [] .
ceq eacExtendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 1 from ars , s)}{s}
++ []  if q == 'class___end ...
ceq eacExtendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s}
++ eacExtendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from
ars}{s})}{s} if q == 'class_extends___end ...
ceq eacSonsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} = (
eacAttributeDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{argLookup(takeArg 2 from ars , s)}{s} ++ eacSonsLookup
Class{q1}{q'}{s} ) ++ eacSonsLookupClass{q1}{head(basedepthfunc
{takeArg 0 from ars}{s})}{s} if q == 'class_extends___end ...
```

Figura 5.12: Código-fonte das operações: `hasAttributeList`, `subHasAttributeList`, `eacClassDepthFunc`, `eacLookupClass`, `eacExtendsLookupClass` e `eacSonsLookupClass`

```
eq eacAttributeDepthFunc{q1}{q'}{0}{s} = [] .
eq eacAttributeDepthFunc{q1}{q'}{(a1,a2)}{s} = eacLookupAttributeDec
(q1,q', a1, s) ++ eacAttributeDepthFunc{q1}{q'}{a2}{s} .
ceq eacAttributeDepthFunc{q1}{q'}{a1}{s} = eacLookupAttributeDec
(q1,q',a1, s) if lengthargs(a1) == 1 .
ceq eacLookupAttributeDec (q1,q',a1,((parsed a1 = q [ars]),s)) = qid
(string(head(basedepthfunc{takeArg 0 from ars}{s})) ++ "." ++ "'" ++
string(q')) : [] if head(basedepthfunc{takeArg 0 from ars}{s}) == q1
eq eacLookupAttributeDec (q1, q', a1, (s,s')) = [] .
```

Figura 5.13: Código-fonte das operações `eacAttributeDepthFunc` e `eacLookupAttributeDec`.

A operação `hasAttributeList` recebe três argumentos. O primeiro é o nome do atributo a ser buscado, o segundo é o nome da classe que será a base da busca, e por último a árvore sintática do programa em que a classe esta declarada. A operação `hasAttributeList` varre a classe passada como argumento e todas as suas superclasses e subclasses, em busca de declarações do atributo também passado como argumento. O retorno dela é dado na forma de uma lista cujos elementos são formados pelo nome do atributo e o nome da classe onde ele foi encontrado.

O código-fonte da implementação das operações da Figura 5.12 foi baseado nos códigos-fonte das Figuras 5.7 e 5.8 que pertencem a condição EMC. Ele está dividido em duas figuras. Na Figura 5.12 mostramos a implementação das seis primeiras operações que são responsáveis pela varredura na hierarquia das classes. Na Figura 5.13 mostramos as duas últimas que são responsáveis pela varredura no interior das classes.

O fluxo básico do algoritmo `hasAttributeList`, é mostrado a seguir. Considere que o usuário entrou com o comando `hasAttributeList(q1, q', s)`. O parâmetro `q1` é o nome do atributo a ser buscado, `q'` o nome da classe a que ele pertence e `s` a árvore sintática que representa o programa no qual a classe está contida.

1) Primeiramente a operação `hasAttributeList` retira o identificador `lastId` e chama a si própria passando a árvore propriamente dita. Então, ela verifica se a raiz da árvore é igual a `'_.main<_>`. Isso significa que a árvore é de um programa completo. Se a verificação for satisfeita ela chama a operação `subHasAttributeList`.
2) A operação `subHasAttributeList` chama a operação `eacClassDepthFunc` para os identificadores dos nós que representam todas as declarações de classes do programa.
3) A operação `eacClassDepthFunc` recebe uma lista de identificadores de nós (neste ponto, de todas as declarações de classes), chama a função `eacLookupClass` para a cabeça da lista e chama a si mesma para o restante da lista, até que a lista esteja esgotada. Ao final desta etapa teremos uma chamada da operação `eacLookupClass` para cada declaração de classe.
4) A operação `eacLokupClass` tem três caminhos a seguir dependendo de qual condição for satisfeita no momento. As condições estão listadas a seguir. É válido dizer aqui que, todos os três caminhos citados abaixo têm como intuito buscar ocorrências válidas para `q1`. Essas ocorrências são descobertas a partir de chamadas à operação `eacAttributeDepthFunc`.
   I) A declaração corrente é do tipo `'class___end` e o nome da classe relativa a esta declaração é igual a `q'`. Então a operação `eacAttributeDepthFunc` é chamada para ver se `q1` esta declarado dentro de `q'`.
   II) A declaração corrente é do tipo `'class_extends___end` e o nome da classe corrente é igual a `q'`. Então a operação `eacAttributeDepthFunc` é chamada para ver se `q1` está declarado dentro de `q'`. E também a operação `extendsLookupClass` é chamada passando como argumento o nome da classe que aparece como superclasse na declaração corrente. Esta última busca de forma recursiva, todas as superclasses da classe passada como argumento, e para cada uma delas chama a operação `eacAttributeDepthFunc` para ver se `q1` está declarado em alguma delas.
   III) A declaração corrente é do tipo `'class_extends___end` e `q'` é igual ao nome da classe que aparece como superclasse nessa declaração. A operação `eacAttributeDepthFunc` é chamada para ver se `q1` esta declarado na classe corrente. E também a operação `sonsLookupClass` é chamada passando como

argumento o nome da classe corrente. Esta última busca de forma recursiva, todas as subclasses da classe passada como argumento, e para cada uma delas chama a operação `eacAttributeDepthFunc` para ver se `q1` está declarado em alguma delas.

5) A operação `eacAttributeDepthFunc` busca `q1` nas declarações de atributos da classe passada como argumento. Ela chama a operação `eacLookupAttributeDec` para cada declaração de atributo da classe.
6) Se `q1` pertencer ao conjunto de declarações de atributos da classe passada como parâmetro, a operação `eacLookupAttributeDec` retorna um `Qid` formado pela junção de `q1` com a classe.
7) No final da etapa 6, cada chamada a `eacLookupAttributeDec` terá retornado uma lista. Essas listas retornarão passo a passo até chegarem novamente na operação `hasAttributeList` que então, mostrará a união de todas essas listas ao usuário.

A próxima condição a ser estudada é um pouco diferente e de certa forma mais simples que a apresentada nesta seção. Nela, não buscaremos a declaração de um atributo, e sim ocorrências dele no corpo dos métodos da classe em que ele está declarado.

5.2.4 Existência de seleção de atributo privado em uma classe (EAP)

A condição EAP, é relativa ao fato de existir ou não, alguma seleção de um determinado atributo privado no interior da classe a que ele pertence. Este fato pode ser constatado, verificando a existência de seleções ao atributo, feitas a partir da palavra chave *self* (ex.: *self.saldo*, *self.limite*).

A condição EAP, está presente na Lei 83 <*attribute elimination*>. Essa condição serve de base para condições mais complexas como as encontradas nas Leis 84 e 85 de mudança de visibilidade de atributos. A implementação dessa condição será apresentada a seguir.

Implementação

A operação `listAttributeInOps` é responsável por verificar se há seleções de um determinado atributo privado na classe a que ele pertence. Seu retorno é uma lista contendo todas as ocorrências do atributo, unido ao método aonde ele foi encontrado. Ela é formada por cinco outras operações. São elas: `subListAttributeInOps`, `laClassDepthFunc`, `lookupInsideClass`, `attributeDepthFunc` e `lookupAttribute`. As assinaturas de todas elas estão descritas na Figura 5.14. E as equações que implementam o comportamento dessas operações podem ser vistas na Figura 5.15.

```
op listAttributeInOps : Qid Qid State -> List .
op subListAttributeInOps : Qid Qid State -> List .
op laClassDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupInsideClass : Qid Qid Args State -> List .
op attributeDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupAttribute : Qid Qid Args State -> List .
```

Figura 5.14: Assinaturas das operações que implementam a condição de existência de chamada de atributo privado em uma classe.

O fluxo básico do algoritmo `listAttributeInOps`, pode ser visto a seguir. Considere que o usuário entrou com o comando `listAttributeInOps(q1, q', s)`, onde `q1` é o nome do atributo; `q'` é o nome da classe a que ele pertence, ambos do tipo `Qid`, e `s` a árvore sintática do programa em que a classe está inserida.

```
eq listAttributeInOps (q1, q', (lastId n, s)) = listAttributeInOps
(q1, q', s) .
ceq listAttributeInOps (q1, q', ((parsed n = q [ars]) , s)) =
subListAttributeInOps (q1, q', s) if q == '_.main<_> .
ceq listAttributeInOps (q1, q', ((parsed n = q [ars]) , s)) = [] if
q =/= '_.main<_> .
eq subListAttributeInOps (q1, q', ((parsed n = q [ars]) , s)) =
laClassDepthFunc{q1}{q'}{ars}{s} .

eq laClassDepthFunc{q1}{q'}{0}{s} = [] .
eq laClassDepthFunc{q1}{q'}{(a1,a2)}{s} = lookupInsideClass (q1, q',
a1, s) ++ laClassDepthFunc{q1}{q'}{a2}{s} .
ceq laClassDepthFunc{q1}{q'}{a1}{s} = lookupInsideClass (q1, q', a1,
s) if lengthargs(a1) == 1 .

ceq lookupInsideClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
attributeDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++
[]  if q == 'class___end and head(basedepthfunc{takeArg 0 from
ars}{s}) == q' .
ceq lookupInsideClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
attributeDepthFunc{q1}{q'}{argLookup(takeArg 3 from ars , s)}{s} ++
[] if q == 'class_extends___end and head(basedepthfunc{takeArg 0
from ars}{s}) == q' .
eq lookupInsideClass (q, q', a1, (s,s')) = [] .
eq attributeDepthFunc{q1}{qm}{0}{s} = [] .
ceq attributeDepthFunc{q1}{qm}{a1}{((parsed a1 = q [ars]), s)} =
lookupAttribute(q1, qm, takeArg 1 from ars , s) if q == '_.#_# .
ceq attributeDepthFunc{q1}{qm}{a1}{((parsed a1 = q [ars]), s)} =
attributeDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{takeArg 1 from ars}{s} if q == 'meth_^=_end
eq attributeDepthFunc{q1}{qm}{a1}{((parsed a1 = q [ars]), s)} =
attributeDepthFunc{q1}{qm}{ars}{s} .
eq attributeDepthFunc{q1}{qm}{(a1,a2)}{s} =
attributeDepthFunc{q1}{qm}{a1}{s} ++
attributeDepthFunc{q1}{qm}{a2}{s} .
ceq lookupAttribute (q1, qm, a1, (( parsed a1 = q2 [ars] ), s)) =
qid(string(q1) ++ "." ++ string(qm)) : [] if q2 == '_._ and
qidLookup((takeArg 0 from ars), s) == 'self.LeftExp and
head(basedepthfunc{takeArg 1 from ars}{s}) == q1 .
eq lookupAttribute (q1, qm, a1, ((parsed a1 = q [0]), s)) = [] .
eq lookupAttribute (q1, qm, a1, ((parsed a1 = q [a2,a3]), s)) =
lookupAttribute (q1, qm, a2, s) ++ lookupAttribute (q1, qm, a3, s) .
eq lookupAttribute (q1, qm, a1, ((parsed a1 = q [a2]), s)) =
lookupAttribute (q1, qm, a2, s) .
```

Figura 5.15: Código fonte das equações que implementam a condição de existência de chamada de atributo privado em uma classe.

1) A primeira ação da operação `listAttributeInOps` é retirar o identificador `lastId` e chamá-la recursivamente passando a árvore propriamente dita. Então ela verifica se a raiz da árvore é igual a `'_.main<_>`. Isso significa que a árvore é de um programa completo. Se a verificação for satisfeita ela chama a operação `subListAttributeInOps`.
2) A operação `subListAttributeInOps` chama a operação `laClassDepthFunc` para os identificadores dos nós que representam todas as declarações de classes do programa.
3) A operação `laClassDepthFunc` recebe uma lista de identificadores de nós (neste ponto, de todas as declarações de classes), chama a função `lookupInsideClass` para a cabeça da lista e chama a si mesma para o restante da lista, até que a lista esteja esgotada. Ao final desta etapa teremos uma chamada à operação `lookupInsideClass` para cada declaração de classe.
4) A operação `lookupInsideClass` verifica se o nome da classe vinculada à declaração de classe corrente é igual a `q'`. Se for igual, ela chama a operação `attributeDepthFunc` passando como argumento, a lista de identificadores (dos nós) correspondentes a todas as declarações de métodos da classe.
5) A operação `attributeDepthFunc` segue o fluxo básico abaixo. Ele ocorre para cada declaração de método pertencente à lista passada como argumento.
   I) O identificador corresponde a um nó do tipo `'meth_^=_end`. Então a operação `attributeDepthFunc`, chama recursivamente ela mesma, passando como parâmetro o identificador do nó correspondente ao corpo do método e também o nome do método.
   II) O identificador corresponde a um nó do tipo `'_.#_#`. Então a operação `attributeDepthFunc`, chama a operação `lookupAttribute` passando como argumento o identificador do nó correspondente ao corpo imperativo do método (os comandos).
6) A operação `lookupAttribute`, então, varre o interior do método em busca de nós do tipo `'_._` (este tipo de nó indica a seleção de atributos) e que possua como filho na extremidade esquerda o `Qid 'self`. Quando encontra um nó que satisfaz as condições de busca, ela retorna o nome do atributo procurado mais o nome do método no qual ele foi encontrado.
7) Quando todas as declarações de métodos da classe tiverem sido percorridas teremos uma lista contendo todas as ocorrências do atributo juntamente com o método aonde elas foram encontradas. Essa lista é retornada operação a operação até chegar de volta a `listAttributeInOps`.

Assim completamos o estudo das verificações de condições relativas às leis de classes. Na próxima seção apresentaremos um resumo deste capítulo.

5.3 Resumo

Neste capítulo definimos e detalhamos as implementações desenvolvidas nesse trabalho para fazer as verificações de algumas das mais comuns e mais importantes condições de aplicação das leis de programação. Foram apresentadas condições relativas as leis de comando (Variáveis livres e Expressões à esquerda) e as leis de classe (EMC, EMP, EAC e EAP).

Na implementação da condição de variáveis livres é verificado se alguma referência de uma dada variável é encontrada dentro de um comando, excetuando a situação na qual o próprio

comando inclui uma declaração para a variável. Já na da condição de expressões à esquerda é verificado se uma dada variável está sendo utilizada em algum ponto de um comando, como uma expressão à esquerda.

Em relação a métodos temos a condição EMC cuja implementação verifica a existência da declaração de um determinado método em uma classe ou em qualquer subclasse ou superclasse dela. E também a condição EMP cuja implementação simplificadamente, verifica a existência de ocorrências de chamadas a um determinado método em todo um programa.

A implementação da condição EAC verifica a existência ou não da declaração de um dado atributo, em uma classe ou em uma de suas subclasses ou superclasses e a condição EAP verifica a existência ou não, de alguma seleção de um determinado atributo privado no interior da classe a que ele pertence. Naturalmente, essas duas últimas condições são relativas a atributos.

No próximo capítulo, faremos uma avaliação final sobre a estratégia adotada neste trabalho para implementar e executar as verificações de condições de aplicação de leis de programação, e também discutiremos sobre outros trabalhos a serem desenvolvidos no futuro.

Capítulo 6

Conclusão

Tendo em vista a necessidade de realizar mudanças no código de programas, sem alteração do comportamento, a disciplina na realização dessas mudanças é fundamental. Leis de programação apresentam-se como um meio de realização de mudanças em programas de forma disciplinada. Nesta monografia apresentamos uma ferramenta capaz de realizar verificações de condições para a aplicação de leis de programação.

O uso de Maude mostrou-nos como podemos especificar linguagens de programação em sistemas de re-escritura. Esta possibilidade mostra que com Maude podemos criar ferramentas que auxiliem no estudo de semântica de linguagens de programação, a partir da especificação de linguagens e implementação da semântica para elas de maneira eficaz.

A implementação que apresentamos está estruturada em módulos. Quanto à sintaxe, cada classe sintática corresponde a um módulo para a descrição da mesma. No módulo *TERMBUILDER*, mostramos como é possível caminhar em meta-termos de Maude além de demonstrar como alterá-los através de regras de re-escritura. Isso foi descrito através da implementação dos pré-processadores. Também mostramos como um *parser* pode ser desenvolvido para tratar os meta-termos e criar estruturas hierárquicas genéricas para representá-los. Juntas, as implementações desses módulos, formam um exemplo sólido, que pode ser utilizado pela comunidade de Maude, como auxílio no aprendizado teórico e prático do módulo de meta-níveis de Maude.

A técnica apresentada nesse trabalho, para navegação em árvores sintáticas, também é considerada uma base bastante útil, para o estudo e construção de algoritmos de busca e navegação em árvores utilizando sistemas de re-escritura. Visto que, a bibliografia sobre esse assunto com base nesses sistemas é bastante escassa, a apresentação dessa técnica constitui uma contribuição para a comunidade acadêmica.

Através da implementação dos módulos *COMMAND-LAW-CONDITIONS* e *CLASS-LAW-CONDITIONS*, foi possível mostrar como algoritmos complexos e extensos de busca em árvores sintáticas podem ser desenvolvidos utilizando apenas conceitos nativos de Maude e recursão. Apesar de possuírem um alto grau de complexidade, esses algoritmos foram implementados de maneira bem mais inteligível quando comparados a linguagens como Java e C.

Na seção a seguir apresentaremos um pouco das dificuldades encontradas ao longo do desenvolvimento deste trabalho.

6.1 Dificuldades encontradas

O sistema Maude, com seus recursos técnicos nativos, conseguiu atender as necessidades requeridas para a criação da nossa ferramenta de verificações de condições de aplicação de leis de programação. Porém, Maude peca no quesito de interface com o usuário. A interface é feita através de *prompt* de comandos o que dificultou a entrada de dados e tornou a coleta de resultados uma tarefa difícil e entediante. Além disso, Maude não provê uma forma automatizada para aplicar regras de re-escritura em seqüência. Ou seja, não há como fazer com que a saída de uma re-escritura seja a entrada para uma outra. Isso tornou a fase de pré-processamento dos meta-programas bastante lenta, pois somos obrigados a aplicar os pré-processadores um a um, copiando o resultado de um e colando na entrada do próximo.

Outra dificuldade encontrada foi em relação à obtenção de um ambiente de desenvolvimento amigável. A SRI Internacional, desenvolvedora oficial do Maude, não disponibiliza nenhum aplicativo do tipo IDE para o desenvolvimento de implementações para Maude. Isso fez com que a produtividade do período inicial de desenvolvimento ficasse bem prejudicada. Unido a isso, Maude foi desenvolvido para sistemas operacionais baseados no Linux, o que nos obrigou a migrar de sistema operacional. Entretanto, conseguimos através do guia disponível no site oficial do Maude, juntamente com o auxílio precioso do Professor José María Álvarez Palomo (Universidade de Málaga, Espanha), instalar o Maude no Windows utilizando o emulador Cygwin [19]. Esta tarefa não foi fácil e demandou alguns dias de dedicação.

Posteriormente, através de pesquisas, conseguimos identificar a existência de um *plugin* para Eclipse que, apesar de simples, implementa funções bastante úteis como coloração de sintaxe e envio de entradas para um *console* rodando Maude. Este *plugin*, nomeado *Maude Simple GUI,* foi criado pelo grupo de desenvolvimento do projeto *MOMENT* [20] e pode ser obtido em [21]. Este grupo além de criar este *plugin*, desenvolveu um pacote de instalação que contém uma versão de Maude para Windows. O uso do *plugin* citado anteriormente, determinou um aumento substancial na produtividade do desenvolvimento da nossa ferramenta. Além disso, tornou muito mais simples e prática a interface com Maude, principalmente no tocante à entrada de dados e coleta de resultados, através das funcionalidades que norteiam o *console* disponível.

Na próxima seção apresentaremos alguns trabalhos relacionados, fazendo algumas observações e comparações.

6.2 Trabalhos relacionados

Nesse trabalho, mostramos como sistemas de re-escritura, em particular Maude, podem ser utilizados na mecanização da aplicação de leis de programação. Trabalhos nesta linha foram realizados anteriormente por Júnior [9], que utilizou CafeOBJ [10] para automatizar a aplicação de leis de *refactoring,* e por Lira [7], que com o uso de Maude, implementou uma estratégia de redução de linguagens orientadas a objetos. As implementações desenvolvidas em ambos os trabalhos, se baseiam no texto dos programas escritos na linguagem ROOL.

O trabalho desenvolvido por Júnior, Silva e Cornélio, utilizando CafeOBJ, apresentou a derivação de uma regra de *refactoring* (Extract/Inline Method) e a implementação da regra *Self Encapsulate Field*. A respeito das transformações decorrentes da aplicação de leis de programação, o trabalho de Júnior, Silva e Cornélio está à frente do nosso. Contudo, não são verificadas condições para aplicação das leis de programação Neste sentido, nosso trabalho constitui um avanço.

Lira automatizou o processo de obtenção de programas em ROOL que estejam de acordo com uma forma normal. Além disso, apresentou uma proposta para a derivação de regras de *refactoring*. Porém, a abordagem adotada por Lira é diferente da que adotamos, uma vez que não manipulamos o texto do programa, mas a árvore sintática do mesmo. Em outras palavras, realizamos verificações de condições de leis de programação por investigação de elementos da árvore sintática.

Mesmo estando a nossa implementação restrita às verificações das condições de aplicação de leis, conseguimos desenvolver uma nova linha de pesquisa, baseada no tratamento de meta-representações de programas de ROOL, ao invés do tratamento do texto do código-fonte dos programas. Isso fez com que pudéssemos constatar, um grande ganho em portabilidade para a nossa implementação, visto que as meta-representações geradas possuem uma forma genérica e imutável. Assim, especificações de programas escritos em outra linguagem OO, podem ser meta-representadas de forma idêntica àquelas escritas em ROOL. Este fato nos leva a crer que, o impacto nas adaptações da nossa implementação, para atender novas linguagens, seria menor que o obtido, se fossemos fazer a mesma coisa, em implementações baseadas nos trabalhos de Júnior e Lira.

6.3 Trabalhos futuros

No presente trabalho apresentamos uma ferramenta que tem como intuito principal, realizar verificações de condições para aplicação de leis de programação. Muito embora tenhamos abordado as verificações das condições mais comuns às leis de classes e comandos, é necessário que sejam criadas implementações para todas as outras condições restantes. Esta tarefa poderá ser feita estendendo ou adaptando os algoritmos desenvolvidos na nossa ferramenta.

Um outro trabalho é realizar um estudo sobre o desempenho da aplicação das operações responsáveis pela execução das verificações das condições. Visto que os testes desenvolvidos ao longo do trabalho foram feitos com pequenos programas (árvores com no máximo 400 nós), é necessário que testes com programas maiores sejam executados para medir o desempenho de Maude e dos algoritmos implementados, visando futuras aplicações reais para a nossa ferramenta.

Um outro trabalho a ser realizado é o desenvolvimento de novos módulos na nossa ferramenta para a execução das transformações, propriamente ditas, dos programas. Além da criação de um *plugin* para Eclipse para fornecer a interação entre o desenvolvedor, o código-fonte dos programas e Maude. Este *plugin* pode ser uma extensão para o já citado, *Maude Simple GUI*. Assim teríamos uma ferramenta mais completa e de aplicabilidade prática maior.

No módulo *CLASS-LAW-CONDITIONS*, a operação `listMethodCalls` possui uma limitação quando tratamos chamadas de métodos sem *cast* (do tipo `'var . MtID 'm <'arg>`) feitas a partir de variáveis ou argumentos de comandos parametrizados. O que acontece é que verificamos apenas o tipo declarado da variável ou argumento, de forma estática. Em situações onde a variável ou argumento teve seu tipo mudado dinamicamente o algoritmo pode produzir resultados inesperados. Assim, uma possível extensão para a nossa ferramenta é a criação de um verificador de tipos. Para lidar com tipo dinâmico, o verificador precisa fazer inferência de tipos a partir do programa analisado. Isto permitiria um aumento do escopo de atuação de algoritmos derivados da operação `listMethodCalls`.

Há também a possibilidade de desenvolvermos uma outra linha de trabalho junto com o grupo de trabalho do Projeto SAAR [22], que criou uma ferramenta de verificação de condições para aplicações de leis de programação utilizando Prolog que trata programas em ROOL como fatos em Prolog. Nesta linha de trabalho visualizamos a possibilidade de utilização de Prolog e

Maude em conjunto na aplicação de leis de programação, já que a base de fatos de Prolog é estruturalmente semelhante às meta-representações (árvores sintáticas) trabalhadas pela nossa ferramenta. As verificações das condições podem ser realizadas com Prolog (com a implementação já existente); usando Maude realizaríamos as transformações (utilizando os conceitos nativos de re-escritura de Maude), ambos atuando na base de fatos dos programas.

Bibliografia

[1] MEYER, Bertrand: “Object-Oriented Software Construction”, 2 ed, Ed. Addison-Wesley, 1997.

[2] FOWLER, Martin: “Refactoring: Improving the Design of Existing Code”, 1 ed, Ed. Addison-Wesley, 1999.

[3] CORNÉLIO, Márcio: “Refactorings as Formal Refinements”. Tese de Doutorado, Centro de Informática - UFPE, 2004.

[4] BORBA, Paulo, et al: “Algebraic Reasoning for Object-Oriented Programming”. Science of Computer Programming, (52):53 – 100, 2004.

[5] CAVALCANTI, Ana e NAUMMAN, Davi: “A Weakest Precondition Semantics for an Object-oriented Language of Refinement”. Em Jeannete M. Wing, Jim Woodcock and Jim Davies, editors, FM´99 – Formal Methods, LNCS, 1709:1439-1459, Ed. Springer-Verlag, 1999.

[6] Java Technology. Sun Microsystems. Disponível em <http://java.sun.com/>. Acessado em: 10 Nov. 2006.

[7] LIRA, Bruno: “Automação de regras para a programação orientada a objetos”. Dissertação de Mestrado, Centro de Informática - UFPE, 2002.

[8] The Maude System. Disponível em <http://maude.cs.uiuc.edu/> Acessado em: 10 Nov. 2006.

[9] CARVALHO JÚNIOR, Antônio; SILVA, Leila e CORNÉLIO, Márcio : ”Using CafeOBJ to Mechanise Refactoring Proofs and Applications. In: Brazilian Symposium on Formal Methods, 2005, Porto Alegre”. Brazilian Symposium on Formal Methods, 2005. p. 32-46.

[10] NAKAGAWA, Ataru; SAWADA, Toshimi e FUTATSUGI, Kokichi: “CafeOBJ User’s Manual – ver. 1.4, 1998”. Disponível em http://citeseer.ist.psu.edu/nakagawa98cafeobj.html. Acessado em 10 de Nov. 2006.

[11] MORGAN, Carrol: “Programming from Specifications”, 2 ed, Ed. Prentice Hall, 1994.

[12] DIJKSTRA, Edsger: “A Discipline of Programming”, 1 ed, Ed. Prentice Hall, 1976.

[13] BORBA, Paulo e SAMPAIO, Augusto: “Basic Laws of ROOL an Object-Oriented Language”. 3rd Brazilian WorkShop of Formal Methods, 33-44, 2000. Revista de Informática Téorica e Aplicada, 7(1): 49-68, 2000.

[14] *An introduction to teaching logic as a tool* – David Gries Home Page. http://www.cs.cornell.edu/Info/People/Gries/Logic/Equational.html. Acessado em: 10 de Nov. 2006.

[15] CLAVEL, Manuel; MARTÍ-OLIET, Narciso; MESEGUER, José *et al*: “A Maude Tutorial”. European Joint Conference on Theory and Practice of Software (ETAPS'00), 2000. Disponível em. <http://maude.cs.uiuc.edu/papers/abstract/CDELMMQtutorial_2000.html>. Acessado em 10 de Nov. 2006.

[16] MCCOMBS, Theodore: “Maude 2.0 Primer – ver. 1.0. 2003”. Disponível em <http://maude.cs.uiuc.edu/primer/maude-primer.pdf>. Acessado em 10 de Nov. 2006.

[17] CLAVEL, Manuel; MESEGUER, José *et al*: “Maude Manual – ver. 2.2. 2005”. Disponível em <http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf>. Acessado em 10 de Nov. 2006.

[18] Dictionary.com. Disponível em <http://dictionary.reference.com/browse/traversal>. Acessado em 10 de Nov. 2006.

[19] Cygwin home. Disponível em <http://cygwin.com/>. Acessado em 10 de Nov. 2006.

[20] The Moment Project. Disponível em <http://moment.dsic.upv.es/>. Acessado em 10 de Nov. 2006.

[21] The Moment Project Download. http://moment.dsic.upv.es/Default.aspx?tabid=60. Acessado em 10 de Nov. 2006.

[22] Currículo do Sistema de Currículos Lattes (Márcio Lopes Cornélio). http://lattes.cnpq.br/3776948564582273#LinhaPesquisa. Acessado em 10 de Nov. 2006.

Apêndice A

Gramática de ROOL

Neste apêndice apresentamos a gramática de ROOL com todos os seus construtores.

Tipos de dados

$T \in Type$ ::= N | **bool** | **int** | ...

Expressões

$e \in Exp$ ::=	**self** | **super**	'referências' especiais
	| **null**	'referência' nula
	| **new** N	criação de objeto
	| x	variável
	| $f\ (e)$	aplicação de função predefinida
	| e **is** N	checagem de tipo
	| $(N)\ e$	*cast* de tipo
	| $e.x$	seleção de atributo
	| $(e; x : e)$	atualização de atributo

Predicados

$\Psi \in Pred$ ::=	e	expressão booleana
	$\mid \Psi \Rightarrow \Psi$	
	$\mid (\vee i \bullet \Psi_i)$	
	$\mid \forall x : T \bullet \Psi$	
	$\mid e$ **isExaclty** N	teste estrito de tipo

Expressões à esquerda

$le \in Le$	::=	$le1$ | **self.**$le1$ | $(N)le.le1$
$le1 \in Le1$	::=	x | $le1.x$

Comandos e comandos parametrizados

$c \in Com$	::=	le ::= e | c ; c	atribuição mútipla, sequência
		| $x : [\Psi,\Psi']$	especificação de declaração
		| $pc(e)$	aplicação de comando parametrizado
		| **if** $[]i \bullet \Psi_i \rightarrow c_i$ **fi**	condicional
		| **rec** $X \bullet c$ **end** | X	recursão, chamada recursiva
		| **var** $x : T \bullet c$ **end**	bloco de variável local
		| **avar** $x : T \bullet c$ **end**	bloco de variável angélica
$pc \in PCom$	::=	$pds \bullet c$	parametrização
		| $le.m$ | $((N).le).m$	chamadas de métodos
		| **self**.m | **super**.m	
$pds \in Pds$	::=	| $\varnothing$ | pd | pd ; pds	
$pd \in Pd$	::=	**val** $x : T$ | **res** $x : T$	declaração de parâmetros

Programas

$Program$	::=	$cds \bullet c$
$cds \in Cds$	::=	$\varnothing$ | $cd\ cds$
$cd \in Cd$	::=	**class** N_1 **extends** N_2
		pri $x_1 : T_1$;
		prot $x_2 : T_2$;
		{ **meth** m ^= ($pds \bullet c$) **end** }*
		end

Apêndice B

Leis de comandos e classes

Neste apêndice apresentamos a compilação das leis de comandos e classes utilizadas ao longo desta monografia. A compilação completa das leis pode ser encontrada em [3].

B1. Leis de comandos

Lei ⟨**:= skip**⟩ Se *le* ≠ **error**, então (*le* **:=** *le*) = **skip**

Lei ⟨**; — if** *left dist*⟩
Se *i* está no intervalo 1..*n*
if *[]i* ● $\Psi_i \rightarrow c_i$ ***fi*** **=** ***if*** *[]i* ● $\Psi_i \rightarrow c_i$ *; c* ***fi***

Lei ⟨**var** *elim*⟩
Se *x* não é livre em *c*, então **var** *x : T* ● *c* **end = c**

Lei ⟨**var** *rename*⟩
Se x_2 não é livre em *c*, então
var x_1 *: T* ● *c* **end = var** x_2 *: T* ● $c[x_2 / x_1]$ **end**

Lei ⟨*var block*-**val**⟩

$$c = \textbf{var } l\text{: } T \bullet l := x;\ c[l / x] \textbf{ end}$$

dado que *l* é limpa–não é livre em *c*–; *x* não está no lado esquerdo de atribuições, não é um argumento passado por resultado, x não ocorre no *frame* de *specification statements* em *c*, *x* não é um alvo de chamada de método, e *x*≠ **error**.

Lei ⟨*var block*-**res**⟩

$$c = \textbf{var}\ l{:}\ T \bullet l := x;\ c[l\,/\,x]\ \textbf{end}$$

dado que l é limpa–não é livre em c–; x não está no lado direito de atribuições e não é um argumento passado por valor nem é alvo de chamada de método, x não é usada em seleção de atributo nem em expressão de atualização.

Lei ⟨*pcom elimination*-**val**⟩

$$(\textbf{val}\ vl : T \bullet c)(x) = \textbf{val}\ l : T \bullet l := x;\ c[l\,/\,vl]\ \textbf{end}$$

dado que as variáveis de l são limpas: não livres em c, x, e vl. Variáveis em vl não aparecem no lado esquerdo de atribuições, não são usadas como argumentos passado por resultado, não ocorrem no *frame* de *specification statements*, nem são alvos de chamada de método, e não são iguais a **error**.

Lei ⟨*pcom elimination*-**res**⟩

$$(\textbf{res}\ vl : T \bullet c)(x) = \textbf{res}\ l : T \bullet l := x;\ c[l\,/\,vl]\ \textbf{end}$$

dado que as variáveis de l são limpas: não livres em c, x, e vl. Variáveis em vl não aparecem no lado direito de atribuições, não são usadas como argumentos passado por valor, e não são usadas em seleção de atributo nem em expressão de atualização.

B2. Leis de classes

Lei ⟨*class elimination*⟩

$$cds\ cd_1 \bullet c = cds \bullet c$$

dado que

(→) A classe declarada em cd_1 não é referenciada em *cds* ou *c*,

(←) (1) O nome da classe declarada em cd_1 é distinto de todas as classes declaradas em *cds*; (2) a superclasse que aparece em cd_1 ou é **object** ou está declara em *cds*. (3) e os nomes de atributos e métodos declarados em cd_1 não estão declarados por suas superclasses em *cds*, exceto no caso de redefinições de métodos.

Lei ⟨*attribute elimination*⟩

class *B* **extends** *A* **pri** *a* : *T*; *ads* *ops* **end**	$=_{cds;c}$	**class** *B* **extends** *A* *ads*; *ops* **end**

dado que

(→) *B.a*, não aparece em *ops*;

(←) *a* não aparece em *ads* e não está declarado como um atributo por uma superclasse ou subclasse de *B* em *cds*.

Lei ⟨*change visibility: from protected to public*⟩

class *C* **extends** *D* **prot** *a* : *T*; *ads* *ops* **end**	$=_{cds;c}$	**class** *C* **extends** *D* **pub** *a* : *T*; *ads* *ops* **end**

dado que

(←) *B.a*, para qualquer $B \leq C$, aparece apenas em *ops* e nas subclasses de *C* em *cds*.

Lei ⟨*change visibility: from private to public*⟩

class *C* **extends** *D* **prot** *a* : *T*; *ads* *ops* **end**	$=_{cds;c}$	**class** *C* **extends** *D* **pub** *a* : *T*; *ads* *ops* **end**

dado que

(←) *B.a*, para qualquer $B \leq C$, não aparece em *cds* ou *c*.

Lei ⟨*move attribute to superclass*⟩

class *B* **extends** *A* *ads* *ops* **end class** *C* **extends** *B* **pub** *a* : *T*; *ads'* *ops'* **end**	$=_{cds;c}$	**class** *B* **extends** *A* **pub** *a* : *T*; *ads* *ops* **end class** *C* **extends** *B* *ads'* *ops'* **end**

dado que

(→) O nome do atributo *a* não está declarado pelas subclasses de *B* em *cds*;

(←) *D.a*, para qualquer $D \leq B$ e $D \not\leq B$, não aparece em *cds*, *c*, *ops*, ou *ops'*.

Lei ⟨*move redefined method to superclass*⟩

```
class B extends A
  ads
  meth m ≙ (pds ● b)
  ops
end
class C extends B
  ads'
  meth m ≙ (pds ● b')
  ops'
end
```

$=_{cds;c}$

```
class B extends A
  ads
  meth m ≙ (pds ●
    if ¬(self is C) → b
    [] self is C → b'
    fi)
  ops
end
class C extends B
  ads'
  ops'
end
```

dado que

(↔) (1) **super** e atributos privados não aparecem em *b'*; (2) **super**.*m* não aparece em *ops';*

(→) (1) *b'* não contém ocorrências sem *cast* de **self** nem expressões na forma ((*C*)**self**.*a* para qualquer atributo privado *a* em *ads'*.

(←) *m* não está declarado em *ops'*.

Lei ⟨*method elimination*⟩

```
class C extends D
  ads
  meth m ≙ (pds ● c)
  ops
end
```

$=_{cds;c}$

```
class C extends D
  ads
  ops
end
```

dado que

(→) *B.m*, não aparece em cds, *c* nem em ***ops*, para** qualquer *B* tal que $B \leq C$;

(←) *m*, não é declarado em *ops* nem em qualquer superclasse ou subclasse de *C* em *cds*.

Lei ⟨*move original method to superclass*⟩

class *B* **extends** *A* *ads* *ops* **end** **class** *C* **extends** *B* *ads'* **meth** *m* ≙ *pc* *ops'* **end**	$=_{cds;c}$	**class** *B* **extends** *A* *ads* **meth** *m* ≙ *pc* *ops* **end** **class** *C* **extends** *B* *ads'* *ops'* **end**

dado que

(↔) (1) **super** e atributos privados não aparecem em *pc*; (2) *m* não está declarado em qualquer superclasse de *B* em *cds*;

(→) (1) *m* não está declarado em *ops*, e apenas pode estar declarado em uma classe *D*, para qualquer $D \leq B$ e $D \not\leq C$, se ele possui os mesmos parâmetros de *pc*; (2) *pc* não contém ocorrências sem *cast* de **self** nem expressões na forma ((*C*)**self**).*a* para qualquer atributo privado *a* em *ads'*.

(←) (1) *m* não está declarado em *ops'*.

Apêndice C

O módulo *TERMBUILDER*

Neste apêndice apresentamos o código-fonte do módulo *TERMBUILDER*, responsável por processar as meta-representações dos programas escritos em ROOL e gerar as árvores sintáticas.

```
mod TERMBUILDER is

protecting META-LEVEL .
protecting UTIL-MODULE .

sorts Node NodeList Tree Args .
sorts State .
subsort Term < Node .
subsort Nat < Args .
subsort Qid < Node .

op _,_ : State State -> State [assoc comm format ( d d n d ) ] .
op unParsed_=_ : Nat Term -> State .
op parsed_=_ : Nat Node -> State .
op _[_] : Qid Args -> Node .
op {_}_[_] : Args Qid Args -> Node .
op _,_ : Args Args -> Args [ assoc ] .
op lastId_ : Nat -> State .
op inc : -> State .

var N1 N2 : Nat .
var o p u : Qid .
var s : String .
vars t1 t2 t3 t4 t5 t6 : Term .
vars tl : TermList .
vars tlist1 tlist2 tlist3 tlist4 : TermList .

rl lastId N1 , inc => lastId(N1 + 1) .

rl unParsed N1 = o => parsed N1 = o[0] .

rl lastId N1 , unParsed N2 = (o[tl])  =>
   ( lastId (N1 + sizeof tl) ),
 (parsed N2 = (o[(argsof tl N1)])) ,
   nodesof tl N1 .
```

```
op sizeof_ : TermList -> Nat .
crl sizeof (t1,tl) => 1 if tl == empty .
crl sizeof (t1,tl) => 1 + sizeof tl if tl =/= empty .
rl sizeof t1 => 1 .

op argsof__ : TermList Nat -> Args .
crl argsof (t1,tl) N1 => (N1) if tl == empty .
crl argsof (t1,tl) N1 => (N1, argsof tl (N1 + 1)) if tl =/= empty .
rl argsof t1 N1 => N1 .

op nodesof__ : TermList Nat -> State .
crl nodesof (t1,tl) N1 => (unParsed N1 = t1) if tl == empty .
crl nodesof (t1,tl) N1 => (unParsed N1 = t1) , (nodesof tl (N1 + 1)) if tl =/=
empty .
rl nodesof t1 N1 => unParsed N1 = t1 .

op parse_ : Term -> State .
rl parse t1 => lastId 1 , unParsed 0 = t1 .

op preParseVarDec{_} : Term -> Term .
crl preParseVarDec{o[tlist1,tlist2]} => o [ tlist1 , preParseVarDec{tlist2} ]
if o =/= 'var_._end .
crl preParseVarDec{t1,p[tlist3]} => t1, p [ preParseVarDec{tlist3} ] if p =/=
'var_._end .
crl preParseVarDec{p[tlist3],t1} => p [ preParseVarDec{tlist3} ], t1 if p =/=
'var_._end .
crl preParseVarDec{p[tlist1,tlist2]} => p[ parseVars{tlist1} ,
preParseVarDec{tlist2} ] if p == 'var_._end .
crl preParseVarDec{t1,p[t2,tlist4]} => t1 , p[ parseVars{t2},
preParseVarDec{tlist4} ] if p == 'var_._end .
crl preParseVarDec{p[t2,tlist4],t1} => p[ parseVars{t2},
preParseVarDec{tlist4} ],t1 if p == 'var_._end .
rl preParseVarDec{o[tlist1,tlist2],p[tlist3,tlist4]} => preParseVarDec{o [
tlist1 , tlist2 ]} , preParseVarDec{p [ tlist3, tlist4 ]} .
rl preParseVarDec{t1,t2} => t1,t2 .
rl preParseVarDec{t1} => t1 .

op parseVars{_} : Term -> Term .
crl parseVars{p} => ctrVar p if find("'" ++ string(p), "'", 0) =/= notFound .
crl parseVars{p} => p if find("'" ++ string(p), "'", 0) == notFound
crl parseVars{o[p]} => o [ ctrVar p ]  if o =/= 'CLID_ .
crl parseVars{o[p]} => o [ p ]  if o == 'CLID_ .
rl parseVars{o[t1]} => o [ parseVars{t1} ] .
rl parseVars{o[t1,tl]} => o [ parseVars{t1} , parseVars{tl} ] .
rl parseVars{o[t1,t2]} => o [ parseVars{t1} , parseVars{t2} ] .
crl parseVars{t1,tl} => parseVars{t1} if tl == empty .
crl parseVars{t1,tl} => parseVars{t1} , parseVars{tl} if tl =/= empty .
op ctrVar_ : Term -> Term .
rl ctrVar t1 => qid( string ( getName(t1) ) ++ ".VarDec" ) .

op preParseClidMtid{_} : Term -> Term .
rl preParseClidMtid{p} => p .
rl preParseClidMtid{'CLID_[p]} => 'CLID_ [ ctrClid p ] .
rl preParseClidMtid{'MtID_[p]} => 'MtID_ [ ctrMtid p ] .
rl preParseClidMtid{o[p]} => o [ p ]  .
rl preParseClidMtid{o[t1]} => o [ preParseClidMtid{t1} ] .
crl preParseClidMtid{o[t1,tl]} => o [ preParseClidMtid{t1} ] if tl == empty .
crl preParseClidMtid{o[t1,tl]} => o [ preParseClidMtid{t1} ,
preParseClidMtid{tl} ] if tl =/= empty .
```

```
rl preParseClidMtid{o[t1,t2]} => o [ preParseClidMtid{t1} ,
preParseClidMtid{t2} ] .
crl preParseClidMtid{t1,tl} => preParseClidMtid{t1} if tl == empty .
crl preParseClidMtid{t1,tl} => preParseClidMtid{t1},preParseClidMtid{tl} if tl
=/= empty .
op ctrClid_ : Term -> Term .
rl ctrClid t1 => qid( string ( getName(t1) ) ++ ".ClassName" ) .
op ctrMtid_ : Term -> Term .
rl ctrMtid t1 => qid( string ( getName(t1) ) ++ ".MethName" ) .

op preParsePoint{_} : Term -> Term .
rl preParsePoint{p} => p .
rl preParsePoint{o[p]} => o [ p ]  .
rl preParsePoint{o[t1]} => o [ preParsePoint{t1} ]  .
crl preParsePoint{o[t1,tl]} => o [ preParsePoint{t1} ] if o =/= '_._ and tl ==
empty .
crl preParsePoint{o[t1,tl]} => o [ preParsePoint{t1} , preParsePoint{tl} ] if
o =/= '_._ and tl =/= empty .
crl preParsePoint{o[t1,t2]} => o [ preParsePoint{t1} , preParsePoint{t2} ] if
o =/= '_._ .
crl preParsePoint{o[t1,tl]} => o [ preParsePoint{t1} , ctrAttr{tl} ] if o ==
'_._ .
crl preParsePoint{o[t1,t2]} => o [ preParsePoint{t1} , ctrAttr{t2} ] if o ==
'_._ .
crl preParsePoint{t1,tl} => preParsePoint{t1} if tl == empty .
crl preParsePoint{t1,tl} => preParsePoint{t1} , preParsePoint{tl} if tl =/=
empty .
op ctrAttr{_} : Term -> Term .
crl ctrAttr{o[t1]} => o[t1] if o == 'MtID_ .
rl ctrAttr{p} => qid( string ( getName(p) ) ++ ".AttrInside" ) .

op parseVars{_}{_} : Term String -> Term .
crl parseVars{p}{s} => ctrParDec p s if find("'" ++ string(p), "'", 0) =/=
notFound .
crl parseVars{p}{s} => p if find("'" ++ string(p), "'", 0) == notFound .
crl parseVars{o[p]}{s} => o [ ctrParDec p s ]  if o =/= 'CLID_ .
crl parseVars{o[p]}{s} => o [ p ]  if o == 'CLID_ .
rl parseVars{o[t1]}{s} => o [ parseVars{t1}{s} ] .
rl parseVars{o[t1,tl]}{s} => o [ parseVars{t1}{s} , parseVars{tl}{s} ] .
rl parseVars{o[t1,t2]}{s} => o [ parseVars{t1}{s} , parseVars{t2}{s} ] .
crl parseVars{t1,tl}{s} => parseVars{t1}{s} if tl == empty .
crl parseVars{t1,tl}{s} => parseVars{t1}{s},parseVars{tl}{s} if tl =/= empty .

op preParseParDec{_} : Term -> Term .
rl preParseParDec{p} => p .
rl preParseParDec{'res_;[t1]} => 'res_;[ parseVars{t1}{".ResDec"} ] .
rl preParseParDec{'val_;[t1]} => 'val_;[ parseVars{t1}{".ValDec"} ] .
rl preParseParDec{'vres_;[t1]} => 'vres_;[ parseVars{t1}{".VResDec"} ] .
rl preParseParDec{o[p]} => o [ p ]  .
rl preParseParDec{o[t1]} => o [ preParseParDec{t1} ] .
crl preParseParDec{o[t1,tl]} => o [ preParseParDec{t1} ]  if tl == empty .
crl preParseParDec{o[t1,tl]} => o [ preParseParDec{t1} , preParseParDec{tl} ]
if tl =/= empty .
rl preParseParDec{o[t1,t2]} => o [ preParseParDec{t1} , preParseParDec{t2} ] .
crl preParseParDec{t1,tl} => preParseParDec{t1} if tl == empty .
crl preParseParDec{t1,tl} => preParseParDec{t1} , preParseParDec{tl} if tl =/=
empty .
op ctrParDec__ : Term String -> Term .
rl ctrParDec t1 s => qid( string ( getName(t1) ) ++ s ) .
```

```
op preParseAttrDec{_} : Term -> Term .
rl preParseAttrDec{p} => p .
rl preParseAttrDec{'pri_;[t1]} => 'pri_;[ parseVars{t1}{".AttrPri"} ] .
rl preParseAttrDec{'prot_;[t1]} => 'prot_;[ parseVars{t1}{".AttrProt"} ] .
rl preParseAttrDec{'pub_;[t1]} => 'pub_;[ parseVars{t1}{".AttrPub"} ] .
rl preParseAttrDec{o[p]} => o [ p ]  .
rl preParseAttrDec{o[t1]} => o [ preParseAttrDec{t1} ] .
crl preParseAttrDec{o[t1,tl]} => o [ preParseAttrDec{t1} ] if tl == empty .
crl preParseAttrDec{o[t1,tl]} => o [ preParseAttrDec{t1} , preParseAttrDec{tl}
] if tl =/= empty .
rl preParseAttrDec{o[t1,t2]} => o [ preParseAttrDec{t1} , preParseAttrDec{t2}
] .
crl preParseAttrDec{t1,tl} => preParseAttrDec{t1} if tl == empty .
crl preParseAttrDec{t1,tl} => preParseAttrDec{t1} , preParseAttrDec{tl} if tl
=/= empty .

op preParseForAllPred{_} : Term -> Term .
rl preParseForAllPred{p} => p .
rl preParseForAllPred{o[p]} => o [ p ]  .
rl preParseForAllPred{o[t1]} => o [ preParseForAllPred{t1} ] .
crl preParseForAllPred{o[t1,tl]} => o [ parseVars{t1}{".ForAll_Pred"},tl]  if
o == 'ForAll_:_._ .
crl preParseForAllPred{o[t1,tl]} => o [ preParseForAllPred{t1} ,
preParseForAllPred{tl} ] if o =/= 'ForAll_:_._ and tl =/= empty .
crl preParseForAllPred{o[t1,tl]} => o [ preParseForAllPred{t1} ] if o =/=
'ForAll_:_._ and tl == empty .
crl preParseForAllPred{o[t1,t2]} => o [ parseVars{t1}{".ForAll_Pred"},t2]  if
o == 'ForAll_:_._ .
crl preParseForAllPred{o[t1,t2]} => o [ preParseForAllPred{t1} ,
preParseForAllPred{t2} ] if o =/= 'ForAll_:_._ .
crl preParseForAllPred{t1,tl} => preParseForAllPred{t1} if tl == empty .
crl preParseForAllPred{t1,tl} => preParseForAllPred{t1} ,
preParseForAllPred{tl} if tl =/= empty .

op preParseCompForAllPred{_} : Term -> Term .
rl preParseCompForAllPred{p} => p .
rl preParseCompForAllPred{o[p]} => o [ p ]  .
rl preParseCompForAllPred{o[t1]} => o [ preParseCompForAllPred{t1} ] .
crl preParseCompForAllPred{o[t1,t2,t3]} => o [ t1,t2,t3 ]  if o =/=
'ForAll_:_._ .
crl preParseCompForAllPred{o[t1,t2,t3]} => 'ForAll_:_._[ t1, t2,
parseForAllPredVars{t3}{t1} ] if o == 'ForAll_:_._ .
crl preParseCompForAllPred{o[t1,tl]} => o [ preParseCompForAllPred{t1} ] if tl
== empty .
crl preParseCompForAllPred{o[t1,tl]} => o [ preParseCompForAllPred{t1} ,
preParseCompForAllPred{tl} ] if tl =/= empty .
rl preParseCompForAllPred{o[t1,t2]} => o [ preParseCompForAllPred{t1} ,
preParseCompForAllPred{t2} ] .
crl preParseCompForAllPred{t1,tl} => preParseCompForAllPred{t1} if tl == empty
.
crl preParseCompForAllPred{t1,tl} => preParseCompForAllPred{t1} ,
preParseCompForAllPred{tl} if tl =/= empty .

op parseForAllPredVars{_}{_} : Term Term -> Term .
crl parseForAllPredVars{p}{t1} => qid ( string( getName(p) ) ++
".ForAll_Pred_Appear")  if getType(p) == 'Sort and ( makeParDecList{t1}{p} ==
'ForAll_Pred ) .
rl parseForAllPredVars{p}{t1} => p .
rl parseForAllPredVars{o[t1]}{t2} => o [ parseForAllPredVars{t1}{t2} ] .
```

```
rl parseForAllPredVars{o[t1,tl]}{t3} => o [ parseForAllPredVars{t1}{t3} ,
parseForAllPredVars{tl}{t3} ] .
rl parseForAllPredVars{o[t1,t2]}{t3} => o [ parseForAllPredVars{t1}{t3} ,
parseForAllPredVars{t2}{t3} ] .
crl parseForAllPredVars{t1,tl}{t2} => parseForAllPredVars{t1}{t2} if tl ==
empty .
crl parseForAllPredVars{t1,tl}{t2} => parseForAllPredVars{t1}{t2} ,
parseForAllPredVars{tl}{t2} if tl =/= empty .

op parseParDecVars{_}{_} : Term Term -> Term .
crl parseParDecVars{p}{t1} => qid ( string( getName(p) ) ++ ".ResDecAppear")
if getType(p) == 'Sort and ( makeParDecList{t1}{p} == 'ResDec ) .
crl parseParDecVars{p}{t1} => qid ( string( getName(p) ) ++ ".ValDecAppear")
if getType(p) == 'Sort and ( makeParDecList{t1}{p} == 'ValDec ) .
crl parseParDecVars{p}{t1} => qid ( string( getName(p) ) ++ ".VResDecAppear")
if getType(p) == 'Sort and ( makeParDecList{t1}{p} == 'VResDec ) .
rl parseParDecVars{p}{t1} => p .
rl parseParDecVars{o[t1]}{t2} => o [ parseParDecVars{t1}{t2} ] .
rl parseParDecVars{o[t1,tl]}{t3} => o [ parseParDecVars{t1}{t3} ,
parseParDecVars{tl}{t3} ] .
rl parseParDecVars{o[t1,t2]}{t3} => o [ parseParDecVars{t1}{t3} ,
parseParDecVars{t2}{t3} ] .
crl parseParDecVars{t1,tl}{t2} => parseParDecVars{t1}{t2} if tl == empty .
crl parseParDecVars{t1,tl}{t2} => parseParDecVars{t1}{t2} ,
parseParDecVars{tl}{t2} if tl =/= empty .

op makeParDecList{_}{_} : Term Term -> TermList .
ceq makeParDecList{p}{u} = getType(p) if getName(p) == getName(u) .
ceq makeParDecList{p}{u} = empty if getName(p) =/= getName(u) .
eq makeParDecList{o[t1]}{u} = makeParDecList{t1}{u} .
ceq makeParDecList{o[t1,tl]}{u} = makeParDecList{t1}{u} if tl == empty .
ceq makeParDecList{o[t1,tl]}{u} = makeParDecList{t1}{u} ,
makeParDecList{tl}{u} if tl =/= empty .
eq makeParDecList{o[t1,t2]}{u} = makeParDecList{t1}{u} , makeParDecList{t2}{u}
.
ceq makeParDecList{t1,tl}{u} = makeParDecList{t1}{u} if tl == empty .
ceq makeParDecList{t1,tl}{u} = makeParDecList{t1}{u} , makeParDecList{tl}{u}
if tl =/= empty .

op preParseCompParDec{_} : Term -> Term .
rl preParseCompParDec{p} => p .
rl preParseCompParDec{o[p]} => o [ p ]  .
rl preParseCompParDec{o[t1]} => o [ preParseCompParDec{t1} ] .
crl preParseCompParDec{o[t1,t2,t3]} => '_.#_#<_>[ t1, parseParDecVars{t2}{t1},
t3 ] if o == '_.#_#<_> .
crl preParseCompParDec{o[t1,tl]} => '_.#_#[ t1, parseParDecVars{tl}{t1} ] if o
== '_.#_# .
crl preParseCompParDec{o[t1,tl]} => o [ preParseCompParDec{t1} ] if tl ==
empty and o =/= '_.#_# .
crl preParseCompParDec{o[t1,tl]} => o [ preParseCompParDec{t1} ,
preParseCompParDec{tl} ] if tl =/= empty and o =/= '_.#_# .
crl preParseCompParDec{o[t1,t2]} => o [ t1 , parseParDecVars{t2}{t1} ] if o ==
'_.#_# .
crl preParseCompParDec{o[t1,t2]} => o [ preParseCompParDec{t1} ,
preParseCompParDec{t2} ] if o =/= '_.#_# .
crl preParseCompParDec{t1,tl} => preParseCompParDec{t1} if tl == empty .
crl preParseCompParDec{t1,tl} => preParseCompParDec{t1} ,
preParseCompParDec{tl} if tl =/= empty .

endm
```

Apêndice D

Os módulos *COMMAND-LAW-CONDITIONS* e *CLASS-LAW-CONDITIONS*

Neste apêndice apresentamos os módulos responsáveis por fazer as verificações de condições de leis de programacão. O módulo *COMMAND-LAW-CONDITIONS* implementa as verificações das condições de leis de comando, e o módulo *CLASS-LAW-CONDITIONS*, de leis de classes.

```
mod COMMAND-LAWS-CONDITION is

protecting UTIL-MODULE .

op isFree : Qid State -> Bool .
op vlSearchDepth : Qid Args State -> Bool .
op vlDepthFunc{_}{_}{_} : Qid Args State -> List .
op vlLookup : Qid Args State -> List .

op isLeftExp : Qid State -> Bool .
op searchLe : Qid Args State -> Bool .
op depthfuncLe{_}{_}{_} : Qid Args State -> List .
op lookupLe : Qid Args State -> List .

var s : State .
var s' : State .
var n : Nat .
var ars : Args .
var q, q' : Qid .
var q1 : Qid .
var a1 : Args .
var a2 : Args .

eq isFree (q, (lastId n, s)) = isFree (q, s) .
eq isFree (q, ((parsed n = q1 [ars]) , s)) = vlSearchDepth (q, ars, s) .

ceq vlSearchDepth (q,ars, s) = false if length(vlDepthFunc{q}{ars}{s}) == 0 .
ceq vlSearchDepth (q,ars, s) = true if length(vlDepthFunc{q}{ars}{s}) > 0 .
```

```
eq vlDepthFunc{q}{0}{s} = [] .
eq vlDepthFunc{q}{(a1,a2)}{s} = vlLookup (q, a1, s) ++ vlDepthFunc{q}{a2}{s} .
ceq vlDepthFunc{q}{a1}{s} = vlLookup (q, a1, s) if lengthargs(a1) == 1 .

ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = q : []  if ars == 0 and
getName(q) == q1 and getType(q) == 'Sort .
ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars == 0 and
getName(q) == q1 and getType(q) =/= 'Sort .
ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars == 0 and
getName(q) =/= q1 .
ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars =/= 0 and q ==
'var_._end and isDeclaredIn{takeArg 0 from ars}{q1}{s} .
ceq vlLookup (q1, a1, ((parsed a1 = q [ars]), s)) = vlDepthFunc{q1}{ars}{s}
if ars =/= 0 .

eq isLeftExp (q, (lastId n, s)) = isLeftExp (q, s) .
eq isLeftExp (q, ((parsed n = q1 [ars]) , s)) = searchLe (q, ars, ((parsed n =
q1 [ars]) , s)) .

ceq searchLe (q,ars, s) = false if length(depthfuncLe{q}{ars}{s}) == 0 .
ceq searchLe (q,ars, s) = true if length(depthfuncLe{q}{ars}{s}) > 0 .

eq depthfuncLe{q}{0}{s} = [] .
eq depthfuncLe{q}{(a1,a2)}{s} = lookupLe (q, a1, s) ++ depthfuncLe{q}{a2}{s} .
ceq depthfuncLe{q}{a1}{s} = lookupLe (q, a1, s) if lengthargs(a1) == 1 .

ceq lookupLe (q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars == 0 .
ceq lookupLe (q1, a1, ((parsed a1 = '_:=_ [ars]), s)) = '_:=_ : [] if
head(basedepthfunc{takeArg 0 from ars}{s}) == q1 and
length(basedepthfunc{takeArg 0 from ars}{s}) == 1 .
eq lookupLe (q1, a1, ((parsed a1 = '_:=_ [ars]), s)) = [] .

ceq lookupLe (q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) = '<_>_._<_> : []
if head(basedepthfunc{takeArg 1 from ars}{s}) == q1 and
length(basedepthfunc{takeArg 1 from ars}{s}) == 1 .

ceq lookupLe (q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) = '<_>_._<_> : []
      if itemExists{q1}{(basedepthfunc{takeArg 3 from ars}{s})} == true
            and getType ( select itemPosition {q1}{(basedepthfunc{takeArg 3
            from ars}{((parsed a1 = '<_>_._<_> [ars]), s)})} from
            (resdepthfunc{takeArg 0 from
            sonsSelector{head(returnParDecInside(head(basedepthfunc{takeArg 2
            from ars}{((parsed a1 = '<_>_._<_> [ars]), s)}),
            head(basedepthfunc{takeArg 0 from ars}{s})  ,((parsed a1 =
            '<_>_._<_> [ars]), s)))}}{((parsed a1 = '<_>_._<_> [ars]), s)})  )
            == 'ResDec .

eq lookupLe (q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) = [] .

ceq lookupLe (q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = '_._<_> : [] if
head(basedepthfunc{takeArg 0 from ars}{s}) == q1 and
length(basedepthfunc{takeArg 0 from ars}{s}) == 1 .

ceq lookupLe (q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = '_._<_> : []
     if itemExists{q1}{(basedepthfunc{takeArg 2 from ars}{s})} == true
            and getType ( select itemPosition {q1}{(basedepthfunc{takeArg 2
            from ars}{((parsed a1 = '_._<_> [ars]), s)})} from
            (resdepthfunc{takeArg 0 from
            sonsSelector{head(returnParDecInside(head(basedepthfunc{takeArg 1
            from ars}{((parsed a1 = '_._<_> [ars]), s)}), varDecParent
```

```
            {baseargsdepthfunc{takeArg 0 from ars}{((parsed a1 = '_._<_>
            [ars]), s)}}{((parsed a1 = '_._<_> [ars]), s)},((parsed a1 =
            '_._<_> [ars]), s))}}{((parsed a1 = '_._<_> [ars]), s)})  ) ==
            'ResDec .

eq lookupLe (q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = [] .


ceq lookupLe (q1, a1, ((parsed a1 = 'super._<_> [ars]), s)) = 'super._<_> : []
     if itemExists{q1}{(basedepthfunc{takeArg 1 from ars}{s})} == true
            and getType ( select itemPosition {q1}{(basedepthfunc{takeArg 1
            from ars}{((parsed a1 = 'super._<_> [ars]), s)})} from
            (resdepthfunc{takeArg 0 from
            sonsSelector{head(returnParDecInside(head(basedepthfunc{takeArg 0
            from ars}{((parsed a1 = 'super._<_> [ars]), s)}),
            returnParentClassOfArg{a1}{((parsed a1 = 'super._<_> [ars]), s)}
            ,((parsed a1 = 'super._<_> [ars]), s))}}{((parsed a1 =
            'super._<_> [ars]), s)})  ) == 'ResDec .

ceq lookupLe (q1, a1, ((parsed a1 = '_.#_#<_> [ars]), s)) = '_.#_#<_> : []
      if    itemExists{q1}{(basedepthfunc{takeArg 2 from ars}{s})} == true
            and   getType(select itemPosition{q1}{(basedepthfunc{takeArg 2
            from  ars}{s})} from (resdepthfunc{takeArg 0 from ars}{s})) ==
            'ResDec .

ceq lookupLe (q1, a1, ((parsed a1 = '_.#_#<_> [ars]), s)) = []
      if itemExists{q1}{(basedepthfunc{takeArg 2 from ars}{s})} == false .

eq lookupLe (q1, a1, ((parsed a1 = q [ars]), s)) =
depthfuncLe{q1}{ars}{((parsed a1 = q [ars]), s)} .


---------------------------------------------------------------------------

mod CLASS-LAWS-CONDITION is

protecting TERMBUILDER .

--- EMC ---------------------------------------------------------------------
op hasMethodList : Qid Qid State -> List .
op subHasMethod : Qid Qid State -> List .
op classDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupClass : Qid Qid Args State -> List .
op extendsLookupClass{_}{_}{_} : Qid Qid State -> List .
op sonsLookupClass{_}{_}{_} : Qid Qid State -> List .
op methodDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupMethod : Qid Qid Args State -> List .
---------------------------------------------------------------------------

--- EAP ---------------------------------------------------------------------
op listAttributeInOps : Qid Qid State -> List .
op subListAttributeInOps : Qid Qid State -> List .

op laClassDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupInsideClass : Qid Qid Args State -> List .

op attributeDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op lookupAttribute : Qid Qid Args State -> List .
---------------------------------------------------------------------------
```

```
--- EAC ------------------------------------------------------------------------
op hasAttributeList : Qid Qid State -> List .
op subHasAttributeList : Qid Qid State -> List .
op eacClassDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op eacLookupClass : Qid Qid Args State -> List .

op eacAttributeDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op eacLookupAttributeDec : Qid Qid Args State -> List .
op eacExtendsLookupClass{_}{_}{_} : Qid Qid State -> List .
op eacSonsLookupClass{_}{_}{_} : Qid Qid State -> List .
------------------------------------------------------------------------------

--- EMP ------------------------------------------------------------------------
op listMethodCalls : Qid Qid State -> Bool .
op empAuxListMethodCalls : Qid Qid Args State -> Bool .
op empDepthFunc{_}{_}{_}{_} : Qid Qid Args State -> List .
op empLookupMethodCalls : Qid Qid Args State -> List .
op empCheckSubClassRel{_}{_}{_} : Qid Args State -> Bool .
------------------------------------------------------------------------------

var s : State .
var s' : State .
var n : Nat .
var ars : Args .
var q, q' : Qid .
var q1, q2 : Qid .
var qm : Qid .
var a1 : Args .
var a2 : Args .
var a3 : Args .

--- EMC ------------------------------------------------------------------------
eq hasMethodList (q, q', (lastId n, s)) = hasMethodList (q, q', s) .
ceq hasMethodList (q, q', ((parsed n = q1 [ars]) , s)) = subHasMethod (q, q',
s) if q1 == '_.main<_> .
ceq hasMethodList (q, q', ((parsed n = q1 [ars]) , s)) = [] if q1 =/=
'_.main<_> .

eq subHasMethod (q, q', ((parsed n = q1 [ars]) , s)) =
classDepthFunc{q}{q'}{ars}{s} .

eq classDepthFunc{q}{q'}{0}{s} = [] .
eq classDepthFunc{q}{q'}{(a1,a2)}{s} = lookupClass (q, q', a1, s) ++
classDepthFunc{q}{q'}{a2}{s} .
ceq classDepthFunc{q}{q'}{a1}{s} = lookupClass (q, q', a1, s) if
lengthargs(a1) == 1 .

---q' == nome da classe
---q1 == nome do metodo procurado
ceq lookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
methodDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++ []  if q ==
'class___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

ceq lookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
methodDepthFunc{q1}{q'}{argLookup(takeArg 3 from ars , s)}{s} ++
extendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from ars}{s})}{s} if q ==
'class_extends___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

ceq lookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
methodDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
```

```
ars}{s})}{argLookup(takeArg 3 from ars , s)}{s} ++
sonsLookupClass{q1}{head(basedepthfunc{takeArg 0 from ars}{s})}{s} if q ==
'class_extends___end and head(basedepthfunc{takeArg 1 from ars}{s}) == q' .

eq lookupClass (q, q', a1, (s,s')) = [] .

ceq extendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
methodDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++ []  if q ==
'class___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

ceq extendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
methodDepthFunc{q1}{q'}{argLookup(takeArg 3 from ars , s)}{s} ++
extendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from ars}{s})}{s} if q ==
'class_extends___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

eq extendsLookupClass {q1}{q'}{(s,s')} = [] .

ceq sonsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} = (
methodDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{argLookup(takeArg 3 from ars , s)}{s} ++ sonsLookupClass{q1}{q'}{s}
) ++ sonsLookupClass{q1}{head(basedepthfunc{takeArg 0 from ars}{s})}{s} if q
== 'class_extends___end and head(basedepthfunc{takeArg 1 from ars}{s}) == q' .
eq sonsLookupClass {q1}{q'}{(s,s')} = [] .

eq methodDepthFunc{q1}{q'}{0}{s} = [] .
eq methodDepthFunc{q1}{q'}{(a1,a2)}{s} = lookupMethod (q1,q', a1, s) ++
methodDepthFunc{q1}{q'}{a2}{s} .
ceq methodDepthFunc{q1}{q'}{a1}{s} = lookupMethod (q1,q',a1, s) if
lengthargs(a1) == 1 .

ceq lookupMethod (q1, q', a1, ((parsed a1 = q [ars]), s)) =
qid(string(head(basedepthfunc{takeArg 0 from ars}{s})) ++ "." ++ "'" ++
string(q')) : []  if head(basedepthfunc{takeArg 0 from ars}{s}) == q1 .
eq lookupMethod (q1, q', a1, (s,s')) = [] .
-------------------------------------------------------------------------------


--- EAP -----------------------------------------------------------------------
eq listAttributeInOps (q, q', (lastId n, s)) = listAttributeInOps (q, q', s) .
ceq listAttributeInOps (q, q', ((parsed n = q1 [ars]) , s)) =
subListAttributeInOps (q, q', s) if q1 == '_.main<_> .
ceq listAttributeInOps (q, q', ((parsed n = q1 [ars]) , s)) = [] if q1 =/=
'_.main<_> .

eq subListAttributeInOps (q, q', ((parsed n = q1 [ars]) , s)) =
laClassDepthFunc{q}{q'}{ars}{s} .

eq laClassDepthFunc{q}{q'}{0}{s} = [] .
eq laClassDepthFunc{q}{q'}{(a1,a2)}{s} = lookupInsideClass (q, q', a1, s) ++
laClassDepthFunc{q}{q'}{a2}{s} .
ceq laClassDepthFunc{q}{q'}{a1}{s} = lookupInsideClass (q, q', a1, s) if
lengthargs(a1) == 1 .

---q' == nome da classe
---q1 == nome do atributo procurado
ceq lookupInsideClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
attributeDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++ []  if q
== 'class___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .
```

```
ceq lookupInsideClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
attributeDepthFunc{q1}{q'}{argLookup(takeArg 3 from ars , s)}{s} ++ [] if q ==
'class_extends___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .
eq lookupInsideClass (q, q', a1, (s,s')) = [] .

eq attributeDepthFunc{q1}{qm}{0}{s} = [] .
ceq attributeDepthFunc{q1}{qm}{a1}{((parsed a1 = q [ars]), s)} =
lookupAttribute(q1, qm, takeArg 1 from ars , s) if q == '_.#_# .
ceq attributeDepthFunc{q1}{qm}{a1}{((parsed a1 = q [ars]), s)} =
attributeDepthFunc{q1}{head(basedepthfunc{takeArg 0 from ars}{s})}{takeArg 1
from ars}{s} if q == 'meth_^=_end .

eq attributeDepthFunc{q1}{qm}{a1}{((parsed a1 = q [ars]), s)} =
attributeDepthFunc{q1}{qm}{ars}{s} .

eq attributeDepthFunc{q1}{qm}{(a1,a2)}{s} = attributeDepthFunc{q1}{qm}{a1}{s}
++ attributeDepthFunc{q1}{qm}{a2}{s} .

ceq lookupAttribute (q1, qm, a1, (( parsed a1 = q2 [ars] ), s)) =
qid(string(q1) ++ "." ++ string(qm)) : [] if q2 == '_._ and qidLookup((takeArg
0 from ars), s) == 'self.LeftExp and head(basedepthfunc{takeArg 1 from
ars}{s}) == q1 .

eq lookupAttribute (q1, qm, a1, ((parsed a1 = q [0]), s)) = [] .

eq lookupAttribute (q1, qm, a1, ((parsed a1 = q [a2,a3]), s)) =
lookupAttribute (q1, qm, a2, s) ++ lookupAttribute (q1, qm, a3, s) .

eq lookupAttribute (q1, qm, a1, ((parsed a1 = q [a2]), s)) = lookupAttribute
(q1, qm, a2, s) .
------------------------------------------------------------------------------

--- EAC ----------------------------------------------------------------------
eq hasAttributeList (q, q', (lastId n, s)) = hasAttributeList (q, q', s) .
ceq hasAttributeList (q, q', ((parsed n = q1 [ars]) , s)) =
subHasAttributeList (q, q', s) if q1 == '_.main<_> .
ceq hasAttributeList (q, q', ((parsed n = q1 [ars]) , s)) = [] if q1 =/=
'_.main<_> .

eq subHasAttributeList (q, q', ((parsed n = q1 [ars]) , s)) =
eacClassDepthFunc{q}{q'}{ars}{s} .

eq eacClassDepthFunc{q}{q'}{0}{s} = [] .
eq eacClassDepthFunc{q}{q'}{(a1,a2)}{s} = eacLookupClass (q, q', a1, s) ++
eacClassDepthFunc{q}{q'}{a2}{s} .
ceq eacClassDepthFunc{q}{q'}{a1}{s} = eacLookupClass (q, q', a1, s) if
lengthargs(a1) == 1 .

---q' == nome da classe
---q1 == nome do metodo procurado
ceq eacLookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 1 from ars , s)}{s} ++ []  if
q == 'class___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

ceq eacLookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++
eacExtendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from ars}{s})}{s} if q
== 'class_extends___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .
```

```
ceq eacLookupClass (q1, q', a1, ((parsed a1 = q [ars]), s)) =
eacAttributeDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{argLookup(takeArg 2 from ars , s)}{s} ++
eacSonsLookupClass{q1}{head(basedepthfunc{takeArg 0 from ars}{s})}{s} if q ==
'class_extends___end and head(basedepthfunc{takeArg 1 from ars}{s}) == q' .

eq eacLookupClass (q1, q', a1, (s,s')) = [] .

ceq eacExtendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 1 from ars , s)}{s} ++ []  if
q == 'class___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

ceq eacExtendsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} =
eacAttributeDepthFunc{q1}{q'}{argLookup(takeArg 2 from ars , s)}{s} ++
eacExtendsLookupClass{q1}{head(basedepthfunc{takeArg 1 from ars}{s})}{s} if q
== 'class_extends___end and head(basedepthfunc{takeArg 0 from ars}{s}) == q' .

eq eacExtendsLookupClass {q}{q'}{(s,s')} = [] .

ceq eacSonsLookupClass {q1}{q'}{((parsed a1 = q [ars]), s)} = (
eacAttributeDepthFunc{q1}{head(basedepthfunc{takeArg 0 from
ars}{s})}{argLookup(takeArg 2 from ars , s)}{s} ++
eacSonsLookupClass{q1}{q'}{s} ) ++
eacSonsLookupClass{q1}{head(basedepthfunc{takeArg 0 from ars}{s})}{s} if q ==
'class_extends___end and head(basedepthfunc{takeArg 1 from ars}{s}) == q' .

eq eacSonsLookupClass {q1}{q'}{(s,s')} = [] .

eq eacAttributeDepthFunc{q1}{q'}{0}{s} = [] .
eq eacAttributeDepthFunc{q1}{q'}{(a1,a2)}{s} = eacLookupAttributeDec (q1,q',
a1, s) ++ eacAttributeDepthFunc{q1}{q'}{a2}{s} .
ceq eacAttributeDepthFunc{q1}{q'}{a1}{s} = eacLookupAttributeDec (q1,q',a1, s)
if lengthargs(a1) == 1 .

ceq eacLookupAttributeDec (q1, q', a1, ((parsed a1 = q [ars]), s)) =
qid(string(head(basedepthfunc{takeArg 0 from ars}{s})) ++ "." ++ "'" ++
string(q')) : []  if head(basedepthfunc{takeArg 0 from ars}{s}) == q1 .
eq eacLookupAttributeDec (q1, q', a1, (s,s')) = [] .
------------------------------------------------------------------------

--- EMP ------------------------------------------------------------------
eq listMethodCalls (qm, q , (lastId n, s)) = listMethodCalls (qm, q, s) .
eq listMethodCalls (qm,q , ((parsed n = q1 [ars]),s)) = empAuxListMethodCalls
(qm , q , ars , ((parsed n = q1 [ars]) , s)) .

ceq empAuxListMethodCalls (qm, q, ars, s) = false if
length(empDepthFunc{qm}{q}{ars}{s}) == 0 .
ceq empAuxListMethodCalls (qm, q, ars, s) = true if
length(empDepthFunc{qm}{q}{ars}{s}) > 0 .

eq empDepthFunc{qm}{q}{0}{s} = [] .
eq empDepthFunc{qm}{q}{(a1,a2)}{s} = empLookupMethodCalls (qm, q, a1, s) ++
empDepthFunc{qm}{q}{a2}{s} .
ceq empDepthFunc{qm}{q}{a1}{s} = empLookupMethodCalls (qm, q, a1, s) if
lengthargs(a1) == 1 .

ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = q [ars]), s)) = []  if ars
== 0  .
```

```
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = '_._<_> [ars]), s)) = qm :
[] if head(basedepthfunc{takeArg 1 from ars}{s}) == qm and
      empCheckSubClassRel{q1}{baseargsdepthfunc{takeArg 0 from ars}{((parsed
      a1 = '_._<_> [ars]), s)}}{((parsed a1 = '_._<_> [ars]), s)} == true .

ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = 'super._<_> [ars]), s)) =
qm : [] if head(basedepthfunc{takeArg 0 from ars}{s}) == qm and
           isOfOrSubClassOf{q1}{returnParentClassOfArg{a1}{((parsed a1 =
           'super._<_> [ars]), s)}}{((parsed a1 = 'super._<_> [ars]), s)} ==
           true .
ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = '<_>_._<_> [ars]), s)) =
qm : [] if  head(basedepthfunc{takeArg 2 from ars}{s}) == qm and
           isOfOrSubClassOf{q1}{head(basedepthfunc{takeArg 0 from
           ars}{((parsed a1 = '<_>_._<_> [ars]), s)})}{((parsed a1 =
           '<_>_._<_> [ars]), s)} == true .

ceq empLookupMethodCalls (qm, q1, a1, ((parsed a1 = q [ars]), s)) =
empDepthFunc{qm}{q1}{ars}{((parsed a1 = q [ars]), s)} if ars =/= 0 .

eq empCheckSubClassRel{q}{(a1,ars)}{s} = isOfOrSubClassOf{q}{
returnAttrListType(ars, returnDecType{a1}{s}, s)} {s} .

ceq empCheckSubClassRel{q}{a1}{s} = isOfOrSubClassOf{q}{
returnParentClassOfArg{a1}{s} }{s} if qidLookup(a1,s) == 'self.LeftExp .

ceq empCheckSubClassRel{q}{a1}{s} =
isOfOrSubClassOf{q}{returnDecType{a1}{s}}{s} if qidLookup(a1,s) =/=
'self.LeftExp .
--------------------------------------------------------------------------------


endm
```